**Unit -V**

**Data and Time Data Types and Tools, Time Series Basics, Time Zone Handling**

Time series data is an important form of structured data in many different fields, such as finance, economics, ecology, neuroscience, or physics. Anything that is observed or measured at many points in time forms a time series. Many time series are *fixed frequency*, which is to say that data points occur at regular intervals according to some rule, such as every 15 seconds, every 5 minutes, or once per month. Time series can also be *irregular* without a fixed unit or time or offset between units. How you mark and refer to time series data depends on the application and you may have one of the following:

• *Timestamps*, specific instants in time

• Fixed *periods*, such as the month January 2007 or the full year 2010

• *Intervals* of time, indicated by a start and end timestamp. Periods can be thought of as special cases of intervals

• Experiment or elapsed time; each timestamp is a measure of time relative to a particular start time. For example, the diameter of a cookie baking each second since being placed in the oven

In this chapter, I am mainly concerned with time series in the first 3 categories, though many of the techniques can be applied to experimental time series where the index may be an integer or floating point number indicating elapsed time from the start of the experiment. The simplest and most widely used kind of time series are those indexed by timestamp.

pandas provides a standard set of time series tools and data algorithms. With this, you can efficiently work with very large time series and easily slice and dice, aggregate, and resample irregular and fixed frequency time series. As you might guess, many of these tools are especially useful for financial and economics applications, but you could certainly use them to analyze server log data, too.

**Date and Time Data Types and Tools**

The Python standard library includes data types for date and time data, as well as calendar-related functionality. The datetime, time, and calendar modules are the main places to start. The datetime. datetime type, or simply datetime, is widely used:

In [317]: from datetime import datetime

In [318]: now = datetime.now()

In [319]: now

Out[319]: datetime.datetime(2012, 8, 4, 17, 9, 21, 832092)

In [320]: now.year, now.month, now.day

Out[320]: (2012, 8, 4)

datetime stores both the date and time down to the microsecond. datetime.time

delta represents the temporal difference between two datetime objects:

In [321]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

In [322]: delta

Out[322]: datetime.timedelta(926, 56700)

In [323]: delta.days In [324]: delta.seconds

Out[323]: 926 Out[324]: 56700

You can add (or subtract) a timedelta or multiple thereof to a datetime object to yield

a new shifted object:

In [325]: from datetime import timedelta

In [326]: start = datetime(2011, 1, 7)

In [327]: start + timedelta(12)

Out[327]: datetime.datetime(2011, 1, 19, 0, 0)

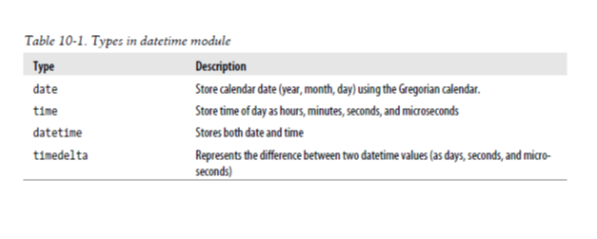
In [328]: start - 2 \* timedelta(12)

Out[328]: datetime.datetime(2010, 12, 14, 0, 0)

The data types in the datetime module are summarized in Table 10-1. While this chapter

is mainly concerned with the data types in pandas and higher level time series manipulation,

you will undoubtedly encounter the datetime-based types in many other places in Python the wild.



**Converting between string and datetime**

datetime objects and pandas Timestamp objects, which I’ll introduce later, can be formatted as strings using str or the strftime method, passing a format specification:

In [329]: stamp = datetime(2011, 1, 3)

In [330]: str(stamp)

In [331]: stamp.strftime('%Y-%m-%d')

Out[330]: '2011-01-03 00:00:00'

Out[331]: '2011-01-03'

See Table 10-2 for a complete list of the format codes. These same format codes can be used to convert strings to dates using datetime.strptime:

In [332]: value = '2011-01-03'

In [333]: datetime.strptime(value, '%Y-%m-%d')

Out[333]: datetime.datetime(2011, 1, 3, 0, 0)

In [334]: datestrs = ['7/6/2011', '8/6/2011']

In [335]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]

Out[335]: [datetime.datetime(2011, 7, 6, 0, 0), datetime.datetime(2011, 8, 6, 0, 0)]

datetime.strptime is the best way to parse a date with a known format. However, it can be a bit annoying to have to write a format spec each time, especially for common date formats. In this case, you can use the parser.parse method in the third party dateutil package:

In [336]: from dateutil.parser import parse

In [337]: parse('2011-01-03')

Out[337]: datetime.datetime(2011, 1, 3, 0, 0)

dateutil is capable of parsing almost any human-intelligible date representation:

In [338]: parse('Jan 31, 1997 10:45 PM')

Out[338]: datetime.datetime(1997, 1, 31, 22, 45)

In international locales, day appearing before month is very common, so you can pass

dayfirst=True to indicate this:

In [339]: parse('6/12/2011', dayfirst=True)

Out[339]: datetime.datetime(2011, 12, 6, 0, 0)

pandas is generally oriented toward working with arrays of dates, whether used as an axis index or a column in a DataFrame. The to\_datetime method parses many different kinds of date representations. Standard date formats like ISO8601 can be parsed very quickly.

In [340]: datestrs

Out[340]: ['7/6/2011', '8/6/2011']

In [341]: pd.to\_datetime(datestrs)

Out[341]: <class 'pandas.tseries.index.DatetimeIndex'>

[2011-07-06 00:00:00, 2011-08-06 00:00:00]

Length: 2, Freq: None, Timezone: None

It also handles values that should be considered missing (None, empty string, etc.):

In [342]: idx = pd.to\_datetime(datestrs + [None])

In [343]: idx

Out[343]:<class 'pandas.tseries.index.DatetimeIndex'>

[2011-07-06 00:00:00, ..., NaT]

Length: 3, Freq: None, Timezone: None

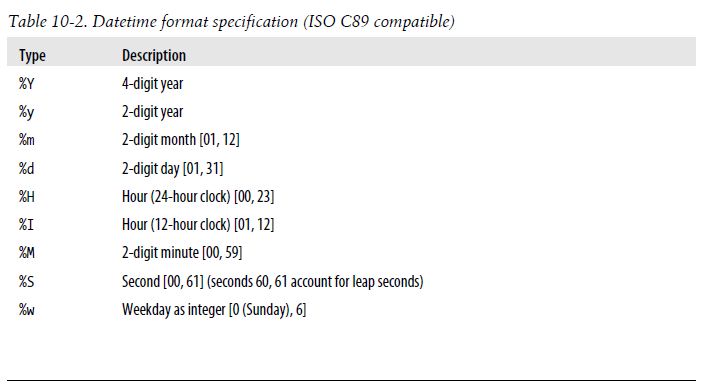
In [344]: idx[2]

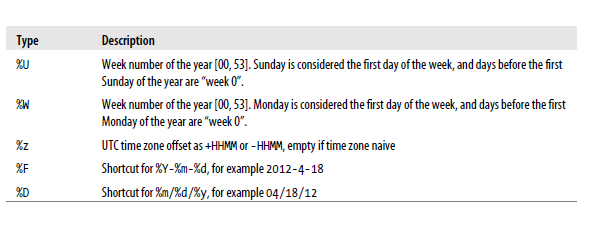
Out[344]: NaT

In [345]: pd.isnull(idx)

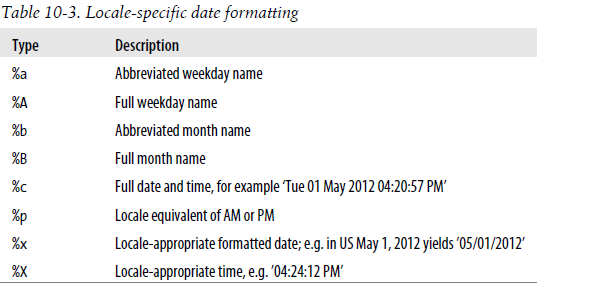
Out[345]: array([False, False, True], dtype=bool)

NaT (Not a Time) is pandas’s NA value for timestamp data.





datetime objects also have a number of locale-specific formatting options for systems in other countries or languages. For example, the abbreviated month names will be different on German or French systems compared with English systems.



**Time Series Basics**

The most basic kind of time series object in pandas is a Series indexed by timestamps, which is often represented external to pandas as Python strings or datetime objects:

In [346]: from datetime import datetime

In [347]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5), datetime(2011, 1, 7),

.....: datetime(2011, 1, 8), datetime(2011, 1, 10), datetime(2011, 1, 12)]

In [348]: ts = Series(np.random.randn(6), index=dates)

In [349]: ts

Out[349]:

2011-01-02 0.690002

2011-01-05 1.001543

2011-01-07 -0.503087

2011-01-08 -0.622274

2011-01-10 -0.921169

2011-01-12 -0.726213

Under the hood, these datetime objects have been put in a DatetimeIndex, and the variable ts is now of type TimeSeries:

In [350]: type(ts)

Out[350]: pandas.core.series.TimeSeries

In [351]: ts.index

Out[351]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2011-01-02 00:00:00, ..., 2011-01-12 00:00:00]

Length: 6, Freq: None, Timezone: None

Like other Series, arithmetic operations between differently-indexed time series automatically

align on the dates:

In [352]: ts + ts[::2]

Out[352]:

2011-01-02 1.380004

2011-01-05 NaN

2011-01-07 -1.006175

2011-01-08 NaN

2011-01-10 -1.842337

2011-01-12 NaN

pandas stores timestamps using NumPy’s datetime64 data type at the nanosecond resolution:

In [353]: ts.index.dtype

Out[353]: dtype('datetime64[ns]')

Scalar values from a DatetimeIndex are pandas Timestamp objects

In [354]: stamp = ts.index[0]

In [355]: stamp

Out[355]: <Timestamp: 2011-01-02 00:00:00>

A Timestamp can be substituted anywhere you would use a datetime object. Additionally,

it can store frequency information (if any) and understands how to do time zone conversions and other kinds of manipulations. More on both of these things later.

**Indexing, Selection, Subsetting**

TimeSeries is a subclass of Series and thus behaves in the same way with regard to indexing and selecting data based on label:

In [356]: stamp = ts.index[2]

In [357]: ts[stamp]

Out[357]: -0.50308739136034464

As a convenience, you can also pass a string that is interpretable as a date:

In [358]: ts['1/10/2011'] In [359]: ts['20110110']

Out[358]: -0.92116860801301081 Out[359]: -0.92116860801301081

For longer time series, a year or only a year and month can be passed to easily select slices of data:

In [360]: longer\_ts = Series(np.random.randn(1000),

.....: index=pd.date\_range('1/1/2000', periods=1000))

In [361]: longer\_ts

Out[361]:

2000-01-01 0.222896

2000-01-02 0.051316

2000-01-03 -1.157719

2000-01-04 0.816707

...

2002-09-23 -0.395813

2002-09-24 -0.180737

2002-09-25 1.337508

2002-09-26 -0.416584

Freq: D, Length: 1000

In [362]: longer\_ts['2001'] In [363]: longer\_ts['2001-05']

Out[362]: Out[363]:

2001-01-01 -1.499503 2001-05-01 1.662014

2001-01-02 0.545154 2001-05-02 -1.189203

2001-01-03 0.400823 2001-05-03 0.093597

2001-01-04 -1.946230 2001-05-04 -0.539164

... ...

2001-12-28 -1.568139 2001-05-28 -0.683066

2001-12-29 -0.900887 2001-05-29 -0.950313

2001-12-30 0.652346 2001-05-30 0.400710

2001-12-31 0.871600 2001-05-31 -0.126072

Freq: D, Length: 365 Freq: D, Length: 31

Slicing with dates works just like with a regular Series:

In [364]: ts[datetime(2011, 1, 7):]

Out[364]:

2011-01-07 -0.503087

2011-01-08 -0.622274

2011-01-10 -0.921169

2011-01-12 -0.726213

Because most time series data is ordered chronologically, you can slice with timestamps

not contained in a time series to perform a range query:

In [365]: ts In [366]: ts['1/6/2011':'1/11/2011']

Out[365]: Out[366]:

2011-01-02 0.690002 2011-01-07 -0.503087

2011-01-05 1.001543 2011-01-08 -0.622274

2011-01-07 -0.503087 2011-01-10 -0.921169

2011-01-08 -0.622274

2011-01-10 -0.921169

2011-01-12 -0.726213

As before you can pass either a string date, datetime, or Timestamp. Remember that slicing in this manner produces views on the source time series just like slicing NumPy arrays. There is an equivalent instance method truncate which slices a TimeSeries between two dates:

In [367]: ts.truncate(after='1/9/2011')

Out[367]:

2011-01-02 0.690002

2011-01-05 1.001543

2011-01-07 -0.503087

2011-01-08 -0.622274

All of the above holds true for DataFrame as well, indexing on its rows:

In [368]: dates = pd.date\_range('1/1/2000', periods=100, freq='W-WED')

In [369]: long\_df = DataFrame(np.random.randn(100, 4),

.....: index=dates,

.....: columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [370]: long\_df.ix['5-2001']

Out[370]:

Colorado Texas New York Ohio

2001-05-02 0.943479 -0.349366 0.530412 -0.508724

2001-05-09 0.230643 -0.065569 -0.248717 -0.587136

2001-05-16 -1.022324 1.060661 0.954768 -0.511824

2001-05-23 -1.387680 0.767902 -1.164490 1.527070

2001-05-30 0.287542 0.715359 -0.345805 0.470886

**Time Series with Duplicate Indices**

In some applications, there may be multiple data observations falling on a particular timestamp. Here is an example:

In [371]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/2000',

.....: '1/3/2000'])

In [372]: dup\_ts = Series(np.arange(5), index=dates)

In [373]: dup\_ts

Out[373]:

2000-01-01 0

2000-01-02 1

2000-01-02 2

2000-01-02 3

2000-01-03 4

We can tell that the index is not unique by checking its is\_unique property:

In [374]: dup\_ts.index.is\_unique

Out[374]: False

Indexing into this time series will now either produce scalar values or slices depending on whether a timestamp is duplicated:

In [375]: dup\_ts['1/3/2000'] # not duplicated

Out[375]: 4

In [376]: dup\_ts['1/2/2000'] # duplicated

Out[376]:

2000-01-02 1

2000-01-02 2

2000-01-02 3

Suppose you wanted to aggregate the data having non-unique timestamps. One way

to do this is to use groupby and pass level=0 (the only level of indexing!):

In [377]: grouped = dup\_ts.groupby(level=0)

In [378]: grouped.mean() In [379]: grouped.count()

Out[378]: Out[379]:

2000-01-01 0 2000-01-01 1

2000-01-02 2 2000-01-02 3

2000-01-03 4 2000-01-03 1

**Date Ranges, Frequencies, and Shifting**

Generic time series in pandas are assumed to be irregular; that is, they have no fixed frequency. For many applications this is sufficient. However, it’s often desirable to work relative to a fixed frequency, such as daily, monthly, or every 15 minutes, even if that means introducing missing values into a time series. Fortunately pandas has a full suite of standard time series frequencies and tools for resampling, inferring frequencies, and generating fixed frequency date ranges. For example, in the example time series, converting it to be fixed daily frequency can be accomplished by calling resample:

In [380]: ts In [381]: ts.resample('D')

Out[380]: Out[381]:

2011-01-02 0.690002 2011-01-02 0.690002

2011-01-05 1.001543 2011-01-03 NaN

2011-01-07 -0.503087 2011-01-04 NaN

2011-01-08 -0.622274 2011-01-05 1.001543

2011-01-10 -0.921169 2011-01-06 NaN

2011-01-12 -0.726213 2011-01-07 -0.503087

2011-01-08 -0.622274

2011-01-09 NaN

2011-01-10 -0.921169

2011-01-11 NaN

2011-01-12 -0.726213

Freq: D

Conversion between frequencies or *resampling* is a big enough topic to have its own

section later. Here I’ll show you how to use the base frequencies and multiples thereof.

**Generating Date Ranges**

While I used it previously without explanation, you may have guessed that pan das.date\_range is responsible for generating a DatetimeIndex with an indicated length according to a particular frequency:

In [382]: index = pd.date\_range('4/1/2012', '6/1/2012')

In [383]: index

Out[383]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2012-04-01 00:00:00, ..., 2012-06-01 00:00:00]

Length: 62, Freq: D, Timezone: None

By default, date\_range generates daily timestamps. If you pass only a start or end date,

you must pass a number of periods to generate:

In [384]: pd.date\_range(start='4/1/2012', periods=20)

Out[384]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2012-04-01 00:00:00, ..., 2012-04-20 00:00:00]

Length: 20, Freq: D, Timezone: None

In [385]: pd.date\_range(end='6/1/2012', periods=20)

Out[385]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2012-05-13 00:00:00, ..., 2012-06-01 00:00:00]

Length: 20, Freq: D, Timezone: None

The start and end dates define strict boundaries for the generated date index. For example, if you wanted a date index containing the last business day of each month, you would pass the 'BM' frequency (business end of month) and only dates falling on or inside the date interval will be included:

In [386]: pd.date\_range('1/1/2000', '12/1/2000', freq='BM')

Out[386]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2000-01-31 00:00:00, ..., 2000-11-30 00:00:00]

Length: 11, Freq: BM, Timezone: None

date\_range by default preserves the time (if any) of the start or end timestamp:

In [387]: pd.date\_range('5/2/2012 12:56:31', periods=5)

Out[387]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2012-05-02 12:56:31, ..., 2012-05-06 12:56:31]

Length: 5, Freq: D, Timezone: None

Sometimes you will have start or end dates with time information but want to generate a set of timestamps *normalized* to midnight as a convention. To do this, there is a normalize option:

In [388]: pd.date\_range('5/2/2012 12:56:31', periods=5, normalize=True)

Out[388]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2012-05-02 00:00:00, ..., 2012-05-06 00:00:00]

Length: 5, Freq: D, Timezone: None

**Frequencies and Date Offsets**

Frequencies in pandas are composed of a *base frequency* and a multiplier. Base frequencies

are typically referred to by a string alias, like 'M' for monthly or 'H' for hourly.

For each base frequency, there is an object defined generally referred to as a *date offset*.

For example, hourly frequency can be represented with the Hour class:

In [389]: from pandas.tseries.offsets import Hour, Minute

In [390]: hour = Hour()

In [391]: hour

Out[391]: <1 Hour>

You can define a multiple of an offset by passing an integer:

In [392]: four\_hours = Hour(4)

In [393]: four\_hours

Out[393]: <4 Hours>

In most applications, you would never need to explicitly create one of these objects, instead using a string alias like 'H' or '4H'. Putting an integer before the base frequency creates a multiple:

In [394]: pd.date\_range('1/1/2000', '1/3/2000 23:59', freq='4h')

Out[394]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2000-01-01 00:00:00, ..., 2000-01-03 20:00:00]

Length: 18, Freq: 4H, Timezone: None

Many offsets can be combined together by addition:

In [395]: Hour(2) + Minute(30)

Out[395]: <150 Minutes>

Similarly, you can pass frequency strings like '2h30min' which will effectively be parsed

to the same expression:

In [396]: pd.date\_range('1/1/2000', periods=10, freq='1h30min')

Out[396]:

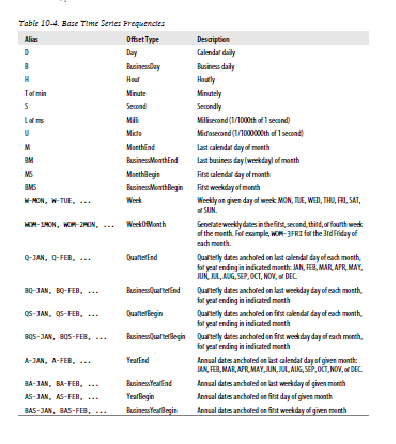
<class 'pandas.tseries.index.DatetimeIndex'>

[2000-01-01 00:00:00, ..., 2000-01-01 13:30:00]

Length: 10, Freq: 90T, Timezone: None

Some frequencies describe points in time that are not evenly spaced. For example, 'M' (calendar month end) and 'BM' (last business/weekday of month) depend on the number of days in a month and, in the latter case, whether the month ends on a weekend or not. For lack of a better term, I call these *anchored* offsets.

See Table 10-4 for a listing of frequency codes and date offset classes available in pandas.



**Week of month dates**

One useful frequency class is “week of month”, starting with WOM. This enables you to

get dates like the third Friday of each month:

In [397]: rng = pd.date\_range('1/1/2012', '9/1/2012', freq='WOM-3FRI')

In [398]: list(rng)

Out[398]:

[<Timestamp: 2012-01-20 00:00:00>,

<Timestamp: 2012-02-17 00:00:00>,

<Timestamp: 2012-03-16 00:00:00>,

<Timestamp: 2012-04-20 00:00:00>,

<Timestamp: 2012-05-18 00:00:00>,

<Timestamp: 2012-06-15 00:00:00>,

<Timestamp: 2012-07-20 00:00:00>,

<Timestamp: 2012-08-17 00:00:00>]

Traders of US equity options will recognize these dates as the standard dates of monthly

expiry.

**Shifting (Leading and Lagging) Data**

“Shifting” refers to moving data backward and forward through time. Both Series and

DataFrame have a shift method for doing naive shifts forward or backward, leaving

the index unmodified:

In [399]: ts = Series(np.random.randn(4),

.....: index=pd.date\_range('1/1/2000', periods=4, freq='M'))

In [400]: ts In [401]: ts.shift(2) In [402]: ts.shift(-2)

Out[400]: Out[401]: Out[402]:

2000-01-31 0.575283 2000-01-31 NaN 2000-01-31 1.814582

2000-02-29 0.304205 2000-02-29 NaN 2000-02-29 1.634858

2000-03-31 1.814582 2000-03-31 0.575283 2000-03-31 NaN

2000-04-30 1.634858 2000-04-30 0.304205 2000-04-30 NaN

Freq: M Freq: M Freq: M

A common use of shift is computing percent changes in a time series or multiple time

series as DataFrame columns. This is expressed as

ts / ts.shift(1) – 1

Because naive shifts leave the index unmodified, some data is discarded. Thus if the frequency is known, it can be passed to shift to advance the timestamps instead of

simply the data:

In [403]: ts.shift(2, freq='M')

Out[403]:

2000-03-31 0.575283

2000-04-30 0.304205

2000-05-31 1.814582

2000-06-30 1.634858

Freq: M

Other frequencies can be passed, too, giving you a lot of flexibility in how to lead and

lag the data:

In [404]: ts.shift(3, freq='D') In [405]: ts.shift(1, freq='3D')

Out[404]: Out[405]:

2000-02-03 0.575283 2000-02-03 0.575283

2000-03-03 0.304205 2000-03-03 0.304205

2000-04-03 1.814582 2000-04-03 1.814582

2000-05-03 1.634858 2000-05-03 1.634858

In [406]: ts.shift(1, freq='90T')

Out[406]:

2000-01-31 01:30:00 0.575283

2000-02-29 01:30:00 0.304205

2000-03-31 01:30:00 1.814582

2000-04-30 01:30:00 1.634858

**Shifting dates with offsets**

The pandas date offsets can also be used with datetime or Timestamp objects:

In [407]: from pandas.tseries.offsets import Day, MonthEnd

In [408]: now = datetime(2011, 11, 17)

In [409]: now + 3 \* Day()

Out[409]: datetime.datetime(2011, 11, 20, 0, 0)

If you add an anchored offset like MonthEnd, the first increment will roll forward a date

to the next date according to the frequency rule:

In [410]: now + MonthEnd()

Out[410]: datetime.datetime(2011, 11, 30, 0, 0)

In [411]: now + MonthEnd(2)

Out[411]: datetime.datetime(2011, 12, 31, 0, 0)

Anchored offsets can explicitly “roll” dates forward or backward using their roll forward and rollback methods, respectively:

In [412]: offset = MonthEnd()

In [413]: offset.rollforward(now)

Out[413]: datetime.datetime(2011, 11, 30, 0, 0)

In [414]: offset.rollback(now)

Out[414]: datetime.datetime(2011, 10, 31, 0, 0)

A clever use of date offsets is to use these methods with groupby:

In [415]: ts = Series(np.random.randn(20),

.....: index=pd.date\_range('1/15/2000', periods=20, freq='4d'))

In [416]: ts.groupby(offset.rollforward).mean()

Out[416]:

2000-01-31 -0.448874

2000-02-29 -0.683663

2000-03-31 0.251920

Of course, an easier and faster way to do this is using resample (much more on this later):

In [417]: ts.resample('M', how='mean')

Out[417]:

2000-01-31 -0.448874

2000-02-29 -0.683663

2000-03-31 0.251920

Freq: M

**Time Zone Handling**

Working with time zones is generally considered one of the most unpleasant parts of time series manipulation. In particular, daylight savings time (DST) transitions are a common source of complication. As such, many time series users choose to work with time series in *coordinated universal time* or *UTC*, which is the successor to Greenwich Mean Time and is the current international standard. Time zones are expressed as offsets from UTC; for example, New York is four hours behind UTC during daylight savings time and 5 hours the rest of the year.

In Python, time zone information comes from the 3rd party pytz library, which exposes the *Olson database*, a compilation of world time zone information. This is especially important for historical data because the DST transition dates (and even UTC offsets) have been changed numerous times depending on the whims of local governments. In the United States, the DST transition times have been changed many times since 1900!

For detailed information about pytz library, you’ll need to look at that library’s documentation. As far as this book is concerned, pandas wraps pytz’s functionality so you can ignore its API outside of the time zone names. Time zone names can be found interactively and in the docs:

In [418]: import pytz

In [419]: pytz.common\_timezones[-5:]

Out[419]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']

To get a time zone object from pytz, use pytz.timezone:

In [420]: tz = pytz.timezone('US/Eastern')

In [421]: tz

Out[421]: <DstTzInfo 'US/Eastern' EST-1 day, 19:00:00 STD>

Methods in pandas will accept either time zone names or these objects. I recommend just using the names.

**Localization and Conversion**

By default, time series in pandas are *time zone naive*. Consider the following time series:

rng = pd.date\_range('3/9/2012 9:30', periods=6, freq='D')

ts = Series(np.random.randn(len(rng)), index=rng)

The index’s tz field is None:

In [423]: print(ts.index.tz)

None

Date ranges can be generated with a time zone set:

In [424]: pd.date\_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')

Out[424]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2012-03-09 09:30:00, ..., 2012-03-18 09:30:00]

Length: 10, Freq: D, Timezone: UTC

Conversion from naive to *localized* is handled by the tz\_localize method:

In [425]: ts\_utc = ts.tz\_localize('UTC')

In [426]: ts\_utc

Out[426]:

2012-03-09 09:30:00+00:00 0.414615

2012-03-10 09:30:00+00:00 0.427185

2012-03-11 09:30:00+00:00 1.172557

2012-03-12 09:30:00+00:00 -0.351572

2012-03-13 09:30:00+00:00 1.454593

2012-03-14 09:30:00+00:00 2.043319

Freq: D

In [427]: ts\_utc.index

Out[427]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2012-03-09 09:30:00, ..., 2012-03-14 09:30:00]

Length: 6, Freq: D, Timezone: UTC

Once a time series has been localized to a particular time zone, it can be converted to

another time zone using tz\_convert:

In [428]: ts\_utc.tz\_convert('US/Eastern')

Out[428]:

2012-03-09 04:30:00-05:00 0.414615

2012-03-10 04:30:00-05:00 0.427185

2012-03-11 05:30:00-04:00 1.172557

2012-03-12 05:30:00-04:00 -0.351572

2012-03-13 05:30:00-04:00 1.454593

2012-03-14 05:30:00-04:00 2.043319

Freq: D

In the case of the above time series, which straddles a DST transition in the US/Eastern

time zone, we could localize to EST and convert to, say, UTC or Berlin time:

In [429]: ts\_eastern = ts.tz\_localize('US/Eastern')

In [430]: ts\_eastern.tz\_convert('UTC')

Out[430]:

2012-03-09 14:30:00+00:00 0.414615

2012-03-10 14:30:00+00:00 0.427185

2012-03-11 13:30:00+00:00 1.172557

2012-03-12 13:30:00+00:00 -0.351572

2012-03-13 13:30:00+00:00 1.454593

2012-03-14 13:30:00+00:00 2.043319

Freq: D

In [431]: ts\_eastern.tz\_convert('Europe/Berlin')

Out[431]:

2012-03-09 15:30:00+01:00 0.414615

2012-03-10 15:30:00+01:00 0.427185

2012-03-11 14:30:00+01:00 1.172557

2012-03-12 14:30:00+01:00 -0.351572

2012-03-13 14:30:00+01:00 1.454593

2012-03-14 14:30:00+01:00 2.043319

Freq: D

tz\_localize and tz\_convert are also instance methods on DatetimeIndex:

In [432]: ts.index.tz\_localize('Asia/Shanghai')

Out[432]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2012-03-09 09:30:00, ..., 2012-03-14 09:30:00]

Length: 6, Freq: D, Timezone: Asia/Shanghai