

10 Pandas IV: Time Series

Lab Objective: *Many real world data sets—stock market measurements, ocean tide levels, website traffic, seismograph data, audio signals, fluid simulations, quarterly dividends, and so on—are time series, meaning they come with time-based labels. There is no universal format for such labels, and indexing by time is often difficult with raw data. Fortunately, pandas has tools for cleaning and analyzing time series. In this lab we use pandas to clean and manipulate time-stamped data and introduce some basic tools for time series analysis.*

Working with Dates and Times

The `datetime` module in the standard library provides a few tools for representing and operating on dates and times. The `datetime.datetime` object represents a *time stamp*: a specific time of day on a certain day. Its constructor accepts a four-digit year, a month (starting at 1 for January), a day, and, optionally, an hour, minute, second, and microsecond. Each of these arguments must be an integer, with the hour ranging from 0 to 23 (military time).

```
>>> from datetime import datetime

# Represent November 18th, 1991, at 2:01 PM.
>>> bday = datetime(1991, 11, 18, 14, 1)
>>> print(bday)
1991-11-18 14:01:00

# Find the number of days between 11/18/1991 and 11/9/2017.
>>> dt = datetime(2017, 11, 9) - bday
>>> dt.days
9487
```

The `datetime.datetime` object has a parser method, `strptime()`, that converts a string into a new `datetime.datetime` object. The parser is flexible because the user must specify the format that the dates are in. For example, if the dates are in the format "**Month/Day//Year::Hour**", specify `format="%m/%d//%Y::%H"` to parse the string appropriately. See Table 10.1 for formatting options.

Pattern	Description
%Y	4-digit year
%y	2-digit year
%m	1- or 2-digit month
%d	1- or 2-digit day
%H	Hour (24-hour)
%I	Hour (12-hour)
%M	2-digit minute
%S	2-digit second

Table 10.1: Formats recognized by `datetime.strptime()`

```
>>> print(datetime.strptime("1991-11-18 / 14:01", "%Y-%m-%d / %H:%M"),
...       datetime.strptime("1/22/1996", "%m/%d/%Y"),
...       datetime.strptime("19-8, 1998", "%d-%m, %Y"), sep='\n')
1991-11-18 14:01:00      # The date formats are now standardized.
1996-01-22 00:00:00      # If no hour/minute/seconds data is given,
1998-08-19 00:00:00      # the default is midnight.
```

Converting Dates to an Index

The `TimeStamp` class is the pandas equivalent to a `datetime.datetime` object. A pandas index composed of `TimeStamp` objects is a `DatetimeIndex`, and a `Series` or `DataFrame` with such an index is called a *time series*. The function `pd.to_datetime()` converts a collection of dates in a parsable format to a `DatetimeIndex`. The format of the dates is inferred if possible, but it can be specified explicitly with the same syntax as `datetime.strptime()`.

```
>>> import pandas as pd

# Convert some dates (as strings) into a DatetimeIndex.
>>> dates = ["2010-1-1", "2010-2-1", "2012-1-1", "2012-1-2"]
>>> pd.to_datetime(dates)
DatetimeIndex(['2010-01-01', '2010-02-01', '2012-01-01', '2012-01-02'],
              dtype='datetime64[ns]', freq=None)

# Create a time series, specifying the format for the dates.
>>> dates = ["1/1, 2010", "1/2, 2010", "1/1, 2012", "1/2, 2012"]
>>> date_index = pd.to_datetime(dates, format="%m/%d, %Y")
>>> pd.Series([x**2 for x in range(4)], index=date_index)
2010-01-01    0
2010-01-02    1
2012-01-01    4
2012-01-02    9
dtype: int64
```

Problem 1. The file `DJIA.csv` contains daily closing values of the Dow Jones Industrial Average from 2006–2016. Read the data into a `Series` or `DataFrame` with a `DatetimeIndex` as the index. Drop rows with missing values, cast the **"VALUES"** column to floats, then plot the data. (Hint: Use `lw=.5` to make the line thin enough for the data.)

Generating Time-based Indices

Some time series datasets come without explicit labels, but still have instructions for deriving timestamps. For example, a list of bank account balances might have records from the beginning of every month, or heart rate readings could be recorded by an app every 10 minutes. Use `pd.date_range()` to generate a `DatetimeIndex` where the timestamps are equally spaced. The function is analogous to `np.arange()` and has the following parameters.

Parameter	Description
<code>start</code>	Starting date
<code>end</code>	End date
<code>periods</code>	Number of dates to include
<code>freq</code>	Amount of time between consecutive dates
<code>normalize</code>	Whether or not to trim the time to midnight

Table 10.2: Parameters for `pd.date_range()`.

Exactly two of the parameters `start`, `end`, and `periods` must be specified to generate a range of dates. The `freq` parameter accepts a variety of string representations, referred to as *offset aliases*. See Table 10.3 for a sampling of some of the options. For a complete list of the options, see <http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases>.

Parameter	Description
"D"	calendar daily (default)
"B"	business daily
"H"	hourly
"T"	minutely
"S"	secondly
"MS"	first day of the month
"BMS"	first weekday of the month
"W-MON"	every Monday
"WOM-3FRI"	every 3rd Friday of the month

Table 10.3: Options for the `freq` parameter to `pd.date_range()`.

```
# 5 consecutive days starting with September 28, 2016.
>>> pd.date_range(start='9/28/2016 16:00', periods=5)
DatetimeIndex(['2016-09-28 16:00:00', '2016-09-29 16:00:00',
               '2016-09-30 16:00:00', '2016-10-01 16:00:00',
               '2016-10-02 16:00:00'],
              dtype='datetime64[ns]', freq='D')
```

```

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

# The first weekday of every other month in 2016.
>>> pd.date_range(start='1/1/2016', end='1/1/2017', freq="2BMS" )
DatetimeIndex(['2016-01-01', '2016-03-01', '2016-05-02', '2016-07-01',
              '2016-09-01', '2016-11-01'],
              dtype='datetime64[ns]', freq='2BMS')

# 10 minute intervals between 4:00 PM and 4:30 PM on September 9, 2016.
>>> pd.date_range(start='9/28/2016 16:00',
                  end='9/28/2016 16:30', freq="10T")
DatetimeIndex(['2016-09-28 16:00:00', '2016-09-28 16:10:00',
              '2016-09-28 16:20:00', '2016-09-28 16:30:00'],
              dtype='datetime64[ns]', freq='10T')

```

The `freq` parameter also supports more flexible string representations.

```

>>> pd.date_range(start='9/28/2016 16:30', periods=5, freq="2h30min")
DatetimeIndex(['2016-09-28 16:30:00', '2016-09-28 19:00:00',
              '2016-09-28 21:30:00', '2016-09-29 00:00:00',
              '2016-09-29 02:30:00'],
              dtype='datetime64[ns]', freq='150T')

```

Problem 2. The file `paychecks.csv` contains values of an hourly employee's last 93 paychecks. He started working March 13, 2008. This company hands out paychecks on the first and third Fridays of the month.

Read in the data, using `pd.date_range()` to generate the `DatetimeIndex`. Plot the data. (Hint: use the `union()` method of `DatetimeIndex` class.)

Periods

The pandas `Timestamp` object represents a precise moment in time on a given day. Some data, however, is recorded over a time interval, and it wouldn't make sense to place an exact timestamp on any of the measurements. For example, a record of the number of steps walked in a day, box office earnings per week, quarterly earnings, and so on. This kind of data is better represented with the pandas `Period` object and the corresponding `PeriodIndex`.

The constructor of the `Period` accepts a `value` and a `freq`. The `value` parameter indicates the label for a given `Period`. This label is tied to the **end** of the defined `Period`. The `freq` indicates the length of the `Period` and also (in some cases) indicates the offset of the `Period`. The `freq` parameter accepts the majority, but not all, of frequencies listed in Table 10.3.

```

# The default value for 'freq' is "M" for months.
>>> p1 = pd.Period("2016-10")
>>> p1.start_time          # The start and end times of the period
Timestamp('2016-10-01 00:00:00') # are recorded as Timestamps.

```

```
>>> p1.end_time
Timestamp('2016-10-31 23:59:59.999999999')

# Represent the annual period ending in December that includes 10/03/2016.
>>> p2 = pd.Period("2016-10-03", freq="A-DEC")
>>> p2.start_time
Timestamp('2007-01-01 00:00:00')
>>> p2.end_time
Timestamp('2007-12-31 23:59:59.999999999')

# Get the weekly period ending on a Saturday that includes 10/03/2016.
>>> print(pd.Period("2016-10-03", freq="W-SAT"))
2016-10-02/2016-10-08
```

Like the `pd.date_range()` method, the `pd.period_range()` method is useful for generating a `PeriodIndex` for unindexed data. The syntax is essentially identical to that of `pd.date_range()`. When using `pd.period_range()`, remember that the `freq` parameter marks the end of the period.

```
# Represent quarters from 2008 to 2010, with Q4 ending in December.
>>> pd.period_range(start="2008", end="2010-12", freq="Q-DEC")
PeriodIndex(['2008Q1', '2008Q2', '2008Q3', '2008Q4', '2009Q1', '2009Q2',
            '2009Q3', '2009Q4', '2010Q1', '2010Q2', '2010Q3', '2010Q4'],
            dtype='period[Q-DEC]', freq='Q-DEC')
```

After creating a `PeriodIndex`, the `freq` parameter can be changed via the `asfreq()` method.

```
# Get every three months form March 2010 to the start of 2011.
>>> p = pd.period_range("2010-03", "2011", freq="3M")
>>> p
PeriodIndex(['2010-03', '2010-06', '2010-09', '2010-12'],
            dtype='period[3M]', freq='3M')

# Change frequency to be quarterly.
>>> p.asfreq("Q-DEC")
PeriodIndex(['2010Q2', '2010Q3', '2010Q4', '2011Q1'],
            dtype='period[Q-DEC]', freq='Q-DEC')
```

Say you have created a `PeriodIndex`, but the bounds are not exactly where you expected they would be. You can actually shift `PeriodIndex` objects by adding or subtracting an integer, n . The resulting `PeriodIndex` will be shifted by $n \times \text{freq}$.

```
# Shift index by 1
>>> p -= 1
>>> p
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'],
            dtype='int64', freq='Q-DEC')
```

If for any reason you need to switch from periods to timestamps, pandas provides a very simple method to do so.

```
# Convert to timestamp (last day of each quarter)
>>> p = p.to_timestamp(how='end')
>>> p
DatetimeIndex(['2010-03-31', '2010-06-30', '2010-09-30', '2010-12-31'],
              dtype='datetime64[ns]', freq='Q-DEC')
```

Similarly, you can switch from timestamps to periods.

```
>>> p.to_period("Q-DEC")
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'],
           dtype='int64', freq='Q-DEC')
```

Problem 3. The file `finances.csv` contains a list of simulated quarterly earnings and expense totals from a fictional company. Load the data into a `Series` or `DataFrame` with a `PeriodIndex` with a quarterly frequency. Assume the fiscal year starts at the beginning of September and that the data begins in September 1978. Plot the data.

Operations on Time Series

There are certain operations only available to `Series` and `DataFrames` that have a `DatetimeIndex`. A sampling of this functionality is described throughout the remainder of this lab.

Slicing

Slicing is much more flexible in pandas for time series. We can slice by year, by month, or even use traditional slicing syntax to select a range of dates.

```
# Select all rows in a given year
>>> df["2010"]
              0          1
2010-01-01  0.566694  1.093125
2010-02-01 -0.219856  0.852917
2010-03-01  1.511347 -1.324036

# Select all rows in a given month of a given year
>>> df["2012-01"]
              0          1
2012-01-01  0.212141  0.859555
2012-01-02  1.483123 -0.520873
2012-01-03  1.436843  0.596143

# Select a range of dates using traditional slicing syntax
>>> df["2010-1-2":"2011-12-31"]
              0          1
2010-02-01 -0.219856  0.852917
```

2010-03-01	1.511347	-1.324036
2011-01-01	0.300766	0.934895

Resampling

Imagine you have a dataset that does not have datapoints at a fixed frequency. For example, a dataset of website traffic would take on this form. Because the datapoints occur at irregular intervals, it may be more difficult to procure any meaningful insight. In situations like these, *resampling* your data is worth considering.

The two main forms of resampling are *downsampling* (aggregating data into fewer intervals) and *upsampling* (adding more intervals).

To downsample, use the `resample()` method of the `Series` or `DataFrame`. This method is similar to `groupby()` in that it groups different entries together, and requires some kind of aggregation to produce a new data set. The first parameter to `resample()` is an offset string from Table 10.3: `"D"` for daily, `"H"` for hourly, and so on.

```
>>> import numpy as np

# Get random data for every day from 2000 to 2010.
>>> dates = pd.date_range(start="2000-1-1", end='2009-12-31', freq='D')
>>> df = pd.Series(np.random.random(len(dates)), index=dates)
>>> df
2000-01-01    0.559
2000-01-02    0.874
2000-01-03    0.774
...
2009-12-29    0.837
2009-12-30    0.472
2009-12-31    0.211
Freq: D, Length: 3653, dtype: float64

# Group the data by year.
>>> years = df.resample("A")          # 'A' for 'annual'.
>>> years.agg(len)                  # Number of entries per year.
2000-12-31    366.0
2001-12-31    365.0
2002-12-31    365.0
...
2007-12-31    365.0
2008-12-31    366.0
2009-12-31    365.0
Freq: A-DEC, dtype: float64

>>> years.mean()                  # Average entry by year.
2000-12-31    0.491
2001-12-31    0.514
2002-12-31    0.484
...
```

```

2007-12-31    0.508
2008-12-31    0.521
2009-12-31    0.523
Freq: A-DEC, dtype: float64

# Group the data by month.
>>> months = df.resample("M")
>>> len(months.mean())           # 12 months x 10 years = 120 months.
120

```

Problem 4. The file `website_traffic.csv` contains records for different visits to a fictitious website. Read in the data, calculate the duration of each visit (in seconds), and convert the index to a `DatetimeIndex`. Use downsampling to calculate the average visit duration by minute, and the average visit duration by hour. Plot both results on the same graph.

Elementary Time Series Analysis

Shifting

`DataFrame` and `Series` objects have a `shift()` method that allows you to move data up or down relative to the index. When dealing with time series data, we can also shift the `DatetimeIndex` relative to a time offset.

```

>>> df = pd.DataFrame(dict(VALUE=np.random.rand(5)),
                        index=pd.date_range("2016-10-7", periods=5, freq='D'))
>>> df
          VALUE
2016-10-07  0.127895
2016-10-08  0.811226
2016-10-09  0.656711
2016-10-10  0.351431
2016-10-11  0.608767

>>> df.shift(1)
          VALUE
2016-10-07    NaN
2016-10-08  0.127895
2016-10-09  0.811226
2016-10-10  0.656711
2016-10-11  0.351431

>>> df.shift(-2)
          VALUE
2016-10-07  0.656711
2016-10-08  0.351431
2016-10-09  0.608767

```



```

2016-10-10      NaN
2016-10-11      NaN

>>> df.shift(14, freq="D")
      VALUE
2016-10-21  0.127895
2016-10-22  0.811226
2016-10-23  0.656711
2016-10-24  0.351431
2016-10-25  0.608767

```

Shifting data makes it easy to gather statistics about changes from one timestamp or period to the next.

```

# Find the changes from one period/timestamp to the next
>>> df - df.shift(1)      # Equivalent to df.diff().
      VALUE
2016-10-07      NaN
2016-10-08  0.683331
2016-10-09 -0.154516
2016-10-10 -0.305279
2016-10-11  0.257336

```

Problem 5. Compute the following information about the DJIA dataset from Problem 1.

- The single day with the largest gain.
- The single day with the largest loss.
- The month with the largest gain.
- The month with the largest loss.

For the monthly statistics, define the gain (or loss) to be the difference between the DJIA on the last and first days of the month.

Rolling Functions and Exponentially-Weighted Moving Functions

Many time series are inherently noisy. To analyze general trends in data, we use *rolling functions* and *exponentially-weighted moving (EWM) functions*.

Rolling functions, or *moving window functions*, perform some kind of calculation on just a window of data. There are a few rolling functions that come standard with pandas.

Rolling Functions (Moving Window Functions)

One of the most commonly used rolling functions is the *rolling average*, which takes the average value over a window of data.

```
# Generate a time series using random walk from a uniform distribution.
N = 10000
bias = 0.01
s = np.zeros(N)
s[1:] = np.random.uniform(low=-1, high=1, size=N-1) + bias
s = pd.Series(s.cumsum(),
              index=pd.date_range("2015-10-20", freq='H', periods=N))

# Plot the original data together with a rolling average.
ax1 = plt.subplot(121)
s.plot(color="gray", lw=.3, ax=ax1)
s.rolling(window=200).mean().plot(color='r', lw=1, ax=ax1)
ax1.legend(["Actual", "Rolling"], loc="lower right")
ax1.set_title("Rolling Average")
```

The function call `s.rolling(window=200)` creates a `pd.core.rolling.Window` object that can be aggregated with a function like `mean()`, `std()`, `var()`, `min()`, `max()`, and so on.

Exponentially-Weighted Moving (EWM) Functions

Whereas a moving window function gives equal weight to the whole window, an *exponentially-weighted moving* function gives more weight to the most recent data points.

In the case of a *exponentially-weighted moving average* (EWMA), each data point is calculated as follows.

$$z_i = \alpha \bar{x}_i + (1 - \alpha) z_{i-1},$$

where z_i is the value of the EWMA at time i , \bar{x}_i is the average for the i -th window, and α is the decay factor that controls the importance of previous data points. Notice that $\alpha = 1$ reduces to the rolling average.

More commonly, the decay is expressed as a function of the window size. In fact, the `span` for an EWMA is nearly analogous to `window` size for a rolling average.

Notice the syntax for EWM functions is very similar to that of rolling functions.

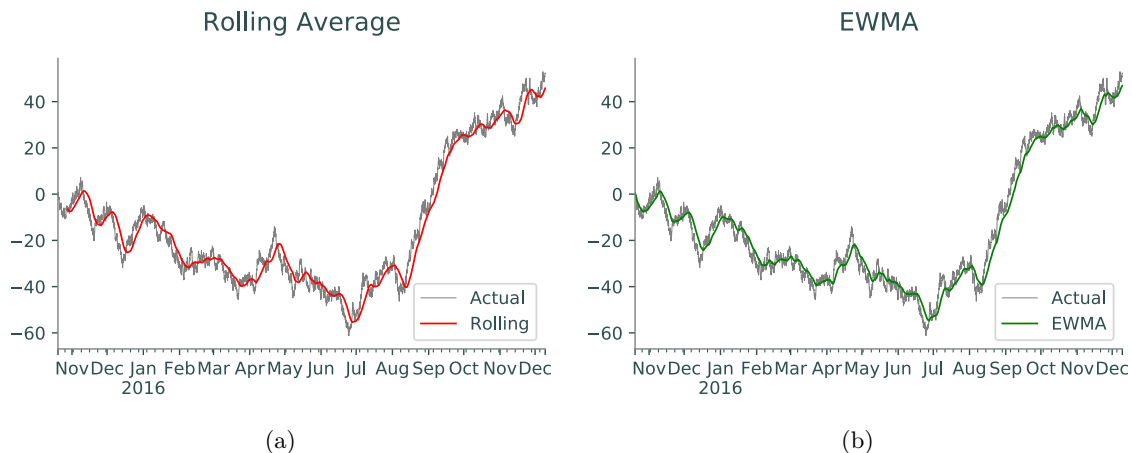


Figure 10.1: Rolling average and EWMA.

```
ax2 = plt.subplot(121)
s.plot(color="gray", lw=.3, ax=ax2)
s.ewm(span=200).mean().plot(color='g', lw=1, ax=ax2)
ax2.legend(["Actual", "EWMA"], loc="lower right")
ax2.set_title("EWMA")
```

Problem 6. Plot the following from the DJIA dataset with a window or span of 30, 120, and 365.

- The original data points.
- Rolling average.
- Exponential average.
- Minimum rolling values.
- Maximum rolling values.

Describe how varying the length of the window changes the approximation to the data.