Lab 1

Pandas IV: Time Series

Lab Objective: *Learn how to manipulate and prepare time series in pandas in preparation for analysis*

Introduction: What is time series data?

Time series data is ubiquitous in the real world. Time series data is any form of data that comes attached to a timestamp (i.e. Sept 28, 2016 20:32:24) or a period of time (i.e. Q3 2012). Some examples of time series data include:

- stock market data
- ocean tide levels
- number of sales over a period of time
- website traffic
- concentrations of a certain compound in a solution over time
- audio signals
- seismograph data
- and more...

Notice that a common feature of all these types of data is that the values can be tied to a specific time or period of time.

In this lab, we will not go into depth on the analysis of such data. Rather, we will discuss some of the tools provided in pandas for cleaning and preparing time series data for further analysis.

Initializing Time Series in `pandas`

To take advantage of all the time series tools available to us in pandas, we need to make a few adjustments to our normal DataFrame.

The `datetime` Module and Initializing a `DatetimeIndex`

For pandas to know to treat a DataFrame or Series object as time series data, the index must be a `DatetimeIndex`. pandas utilizes the `datetime.datetime` object from the `datetime` module to standardize the format in which dates or timestamps are represented.

```
>>> from datetime import datetime

>>> datetime(2016, 9, 28) # 9/28/2016
datetime.datetime(2016,9,28,0,0)

>>> datetime(2016, 9, 28, 21, 12, 48) # 9/28/2016 9:12:48 PM
datetime.datetime(2016, 9, 28, 21, 12, 48)
```

Unsurprisingly, the format for dates varies greatly from dataset to dataset. The `datetime` module comes with a string parser (`datetime.strptime()`) flexible enough to translate nearly any format into a `datetime.datetime` object. This method accepts the string representation of the date, and a string representing the format of the string. See Table **??** for all the options.

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

Table 1.1: Formats recognized by `datetime.strptime()`

Here are some examples of using `datetime.strptime()` to parse the same date from different formats.

```
>>> datetime.strptime("2016-9-28", "%Y-%m-%d")
datetime.datetime(2016, 9, 2, 0, 0)

>>> datetime.strptime("9/28/16", "%m/%d/%y")
datetime.datetime(2016, 9, 2, 0, 0)

>>> datetime.strptime("2016-9-28 9:12:48", "%Y-%m-%d %I:%M:%S")
datetime.datetime(2016, 9, 28, 9, 12, 48)
```

If the dates are in an easily parsible format, pandas has a method `pd.to_datetime()` that can turn a whole pandas `Series` into `datetime.datetime` objects. In the case of the index, the index is automatically converted to a `DatetimeIndex`. This index type is what distinguishes a regular `Series` or `DataFrame` from a time series.

```
>>> dates = ["2010-1-1", "2010-2-1", "2010-3-1", "2011-1-1",
... "2012-1-1", "2012-1-2", "2012-1-3"]
>>> values = np.random.randn(7,2)
```

```
>>> df = pd.DataFrame(values, index=dates)
>>> df
                 0         1
2010-1-1  0.566694  1.093125
2010-2-1 -0.219856  0.852917
2010-3-1  1.511347 -1.324036
2011-1-1  0.300766  0.934895
2012-1-1  0.212141  0.859555
2012-1-2  1.483123 -0.520873
2012-1-3  1.436843  0.596143

>>> df.index = pd.to_datetime(df.index)
```

Note

In earlier versions of pandas, there was a dedicated TimeSeries data type. This has since been depricated, however the functionality remains. Therefore, if you happen to read any materials that reference the TimeSeries data type, know that the cooresponding functionality is likely still in place as long as you have a `DatetimeIndex` associated with your Series or DataFrame.

Problem 1. The provided dataset, "DJIA.csv" is the daily closing value of the Dow Jones Industrial Average for every day over the past 10 years. Read this dataset into a Series with a DatetimeIndex. Replace any missing values with `np.nan`. Lastly, cast all the values in the Series to floats. We will use this dataset for many problems in this lab.

Handling Data Without Marked Timestamps

There will be times you will need to analyze time series data that does not come marked with an index. For example, you may have the a list of bank account balances at the beginning of every month for the last 5 years. You may have heart rate readings every 10 minutes for the past week. pandas provides efficient tools for generating indices for these kinds of siutations.

The `pd.date_range()` Method

The `pd.date_range()` method is analogous to `np.arange()`. The parameters we will use most are described in Table **??**.

Exactly two of the parameters `start`, `end`, and `periods` must be defined to generate a range of dates. The `freqs` parameter accepts a variety of string representations. The accepted strings are referred to as *offset aliases* in the documentation. See Table **??** 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`.

`start`	start of date range
`end`	end of date range
`periods`	the number of dates to include in the date range
`freq`	the amount of time between dates (similar to "step")
`normalize`	trim the time of the date to midnight

Table 1.2: Parameters for `datetime.strptime()`

`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 1.3: Parameters for `datetime.strptime()`

```
>>> 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')

>>> 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')


>>> pd.date_range(start='9/28/2016 16:00',
            end='9/29/2016 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 "paychecks.csv" dataset has 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. However, "paychecks.csv" is not indexed explicitly as such. To be able to manipulate it as a time series in pandas, we will need to add a `DatetimeIndex` to it. Read in the data and use `pd.date_range()` to generate the `DatetimeIndex`.

Hint: to combine two `DatetimeIndex` objects, you can use the `.union()` method of `DatetimeIndex` objects.

Plotting Time Series

The process for plotting a time series is identical to plotting any other Series or DataFrame. For more information and examples, refer back to Lab **??**.

Problem 3. Plot the DJIA dataset that you read in as part of Problem **??**. Label your axes and title the plot.

Dealing with Periods Instead of Timestamps

It is often important to distinguish whether a given figure corresponds to a single point in time or to a whole month, quater, year, decade, etc. A `Period` object is better suited for the summary over a *period* of time rather than the timestamp of a specific event.

Some example of time series that would merit the use of periods would include,

- The number of steps a given person walks in a day.
- The box office results per week for a summer blockbuster.
- The population changes of a given city per year.
- etc.

The `Period` Object

The principle parameters of the `Period` are `"value"` and `"freq"`. The `"value"` paramter 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 **??**.

These nuances are best clarified through examples.

```
# The default value for `freq' is "M" meaning month.
>>> p1 = pd.Period("2016-10")
>>> p1.start_time
Timestamp('2016-10-01 00:00:00')

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

# A `freq' value of 'A-DEC' indicates a annual period
#    with the end of the period occuring in December.
>>> 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')

# Notice that the Period object that is created is the
#   week of the year when the given date occurs. Also
#   note that "W-SAT" indicates we wish to treat
#   Saturday as the last day of the week (and therefore
#   Sunday as the first day of the week.)
>>> p3 = pd.Period("2016-10-03", freq="W-SAT")
>>> p3
Period('2016-10-02/2016-10-08', 'W-SAT')
```

The `pd.period_range()` Method

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.

```
# a 'freq' value of 'Q-DEC' means that Q4 ends in December.
>>> pd.period_range("2008", "2010-12", freq="Q-DEC")
PeriodIndex(['2009Q1', '2009Q2', '2009Q3', '2009Q4', '2010Q1', '2010Q2',
             '2010Q3', '2010Q4'], dtype='int64', freq='Q-DEC')
```

Other Useful Functionality and Methods

After creating a `PeriodIndex`, you have the option to redefine the `"freq"` parameter using the `asfreq()` method.

```
>>> p = pd.period_range("2010-03", "2011", freq="3M")
PeriodIndex(['2010-03', '2010-06', '2010-09', '2010-12'],
            dtype='int64', freq='3M')

# Change frequency to be "Q-DEC" instead of "3M"
>>> p = p.asfreq("Q-DEC")
>>> p
PeriodIndex(['2010Q2', '2010Q3', '2010Q4', '2011Q1'],
            dtype='int64', 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` obejcts by adding or subtracting an integer, n. The resulting `PeriodIndex` will be shifted by $n \times$ `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 4. The "finances.csv" dataset has a list of simulated quarterly earnings and expense totals from a fictional company. Load these values into a DataFrame with a PeriodIndex with a quarterly frequency. Assume the fiscal year starts at the beginning of September and that the dataset goes back to September 1978.

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 a Time Series

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). We will only address downsampling in detail.

Downsampling

When downsampling data, we aggregate our time series data into fewer intervals. Let's futher consider the website traffic examples given above.

Say we have a data set of website traffic indexed by timestamp. Each entry in the dataset contains the IP address of the user, the time the user entered the website, and the time the user left the website.

If we were interested in having information regarding the average visit duration for any given day, we could downsample the data to a daily timescale. To aggregate the data, we would take the average across all the datapoints for the day.

Instead, if we were interested in the number of visits to the website per hour, we could downsample to a weekly timescale. To aggregate the data in this case, we would count up the number of visits in a hour.

The task of downsampling is handled using the `resample()` method. The parameters we will use are `rule`, `how`, `closed`, and `label`. For explanations of these parameters, see Table **??**.

`rule`	the offset string (see Table **??**)
`how`	the data aggregation method (i.e. "sum" or "mean")
`closed`	the boundary of the interval that is closed/inclusive (default "right")
`label`	the boundary of the interval that is assigned to the label (default "right")

Table 1.4: Parameters for `DataFrame.resample()`

Problem 5. Using the "website_traffic.csv" dataset, solve the problem described above. Namely, downsample the dataset to show daily average visit duration. In a different DataFrame, also downsample the dataset to show total number of visits per hour. Your resulting time series should use a `PeriodIndex`.

Basic Time Series Analysis

As mentioned in the beginning of this lab, the focus of this lab is not meant to be on Time Series Analysis. However, we would like to address a few very basic methods that are built into pandas.

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. See the following example code for clarification:

```
>>> 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 has a particularly useful application. We can easily 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)
              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 6. From the Dow Jones Industrial Average dataset referenced in Problem 1, use shifting to find the following information:

- 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

Hint: Downsample or use `groupby()` to answer the questions regarding months.

Rolling Functions and Exponentially-Weighted Moving Functions

In many situations, your data will be noisy. You will often be more interested in general trends. We can accomplish this through *rolling functions* and *exponentially-weighted moving (EWM)* functions.

Problem 7. Generate a time series plot with the following information from the DJIA dataset:

- The original data points.
- Rolling average with window 30.
- Rolling average with window 365.
- Exponential average with span 30.
- Exponential average with span 365.

Include a legend, axis labels, and title. Your plot should look like Figure **??**.

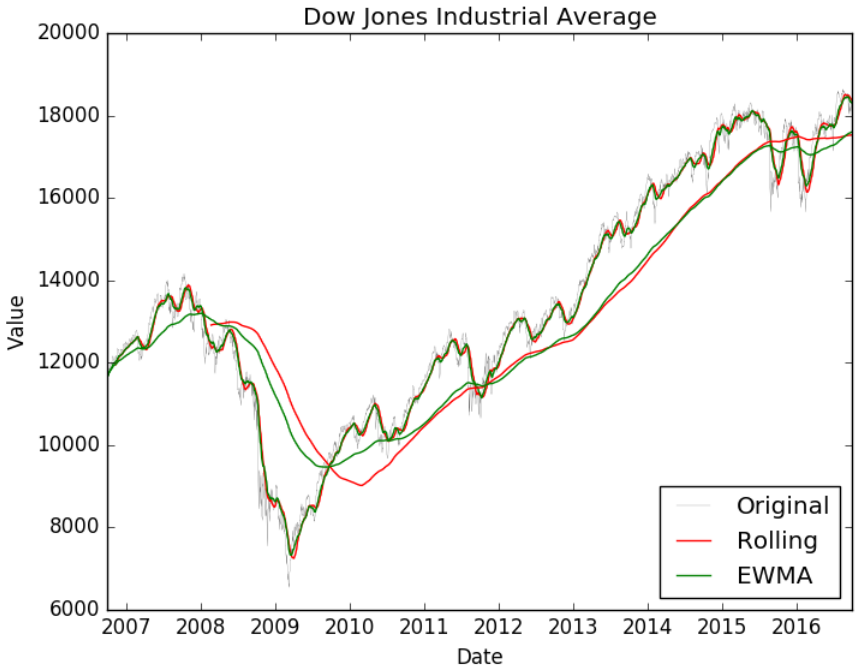

Figure 1.1: Rolling average and EWMA with windows 30 and 365.