Time Series Analysis with Pandas

How to handle and manipulate time series data

Soner Yildirim
Feb 27 · 6 min read ★

Figure source

There are many definitions of time series data, all of which indicate the same meaning in a different way. A straightforward definition is that time series data includes data points attached to sequential time stamps. The sources of time series data are periodic measurements or observations. We observe time series data in many industries. Just to give a few examples:

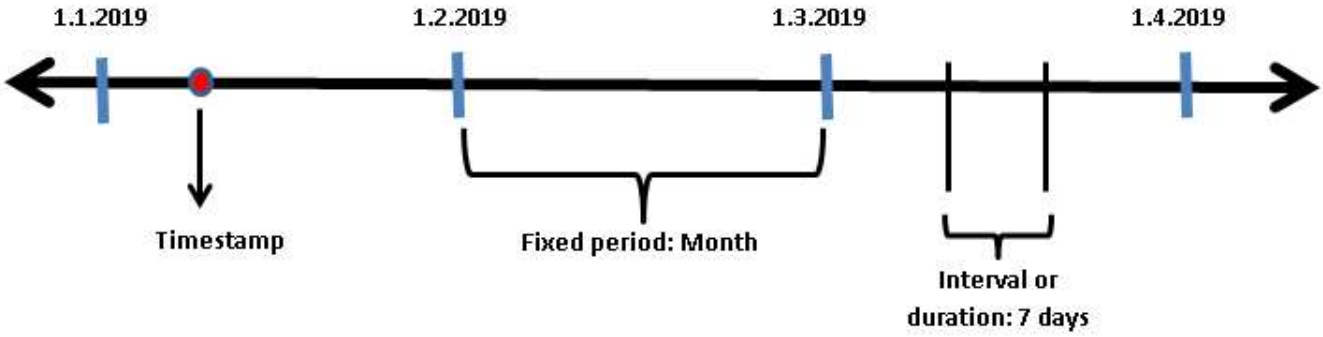
- Stock prices over time
- Daily, weekly, monthly sales
- Periodic measurements in a process
- Power or gas consumption rates over time

Advancements in machine learning have increased the value of time series data. Companies apply machine learning to time series data to make informed business

decisions, do forecasting, compare seasonal or cyclic trends. Large Hadron Collider (LHC) at CERN produces a great amount of time series data with measurements on sub-atomic particles. So, it is everywhere. Handling time series data well is crucial for data analysis process in such fields. Pandas was created by Wes Mckinney to provide an efficient and flexible tool to work with financial data. Therefore, it is a very good choice to work on time series data.

Pandas for time series data

Time series data can be in the form of a specific date, time duration, or fixed defined interval.



Timestamp can be the date of a day or a nanosecond in a given day depending on the precision. For example, '2020–01–01 14:59:30' is a second-based timestamp.

Pandas provides flexible and efficient data structures to work with all kinds of time series data. Following is a table to show basic time series data structures and their corresponding index representations:

Pandas Time Series Data Structures				
Time series data	**Single value**	**Index**	**Data Type**	**Example**
Time stamp	Timestamp	DatetimeIndex	datetime64[ns] datetime64[ns, tz]	"2020-01-10" "2020-01-10 "14:59:00"
Time duration	Timedelta	TimedeltaIndex	timedelta64[ns]	"10 hours" "10 seconds"
Fixed frequency interval	Period	PeriodIndex	period[freq]	"2 business days" "5 months"

For any topic, it is fundamental to learn the basics. Rest can be built-up with practice. Let's explore time series data functionalities of Pandas. As usual, we import the

libraries first:

```
import pandas as pd
import numpy as np
```

The most basic time series data structure is timestamp which can be created using **to_datetime** or **Timestamp** functions:

```
pd.to_datetime('2020-02-10')
```

```
Timestamp('2020-02-10 00:00:00')
```

```
pd.Timestamp('2020-02-10')
```

```
Timestamp('2020-02-10 00:00:00')
```

```
date = pd.to_datetime('2020-02-10')
type(date)
```

```
pandas._libs.tslibs.timestamps.Timestamp
```

In real life cases, we almost always work sequential time series data rather than individual dates. Pandas makes it very simple to work with sequential time series data as well. For example, if we pass multiple dates to to_datetime function, it creates a **DatetimeIndex** which is basically an array of dates.

```
dates = pd.to_datetime(['2020-02-10', '20200211', '2019-Feb-13',
                        '14-February-2020'])
dates
```

```
DatetimeIndex(['2020-02-10', '2020-02-11', '2019-02-13', '2020-02-14'], dtype='datetime64[ns]', freq=None)
```

We don't have to follow a specific format to input a date. to_datetime converts dates in different formats to a standard format. It may not seem convenient to create a time index by passing a list of individual dates. There are, of course, other ways to create an index of time.

We can create an index of dates using a date and **to_timedelta** function:

```
pd.to_datetime('2020-02-01') + pd.to_timedelta(np.arange(10), 'D')
```

```
DatetimeIndex(['2020-02-01', '2020-02-02', '2020-02-03', '2020-02-04',
               '2020-02-05', '2020-02-06', '2020-02-07', '2020-02-08',
               '2020-02-09', '2020-02-10'],
              dtype='datetime64[ns]', freq=None)
```

'2020–02–01' serves as starting point and to_timedelta creates a sequence with specified time delta. In the above example, 'D' is used for 'day' but there are many other options available. You can check the whole list here.

We can also use **date_range** function to create time index from scratch:

- Using start and end dates

```
pd.date_range('2020-01-01','2020-01-07')
```

```
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-04',
               '2020-01-05', '2020-01-06', '2020-01-07'],
              dtype='datetime64[ns]', freq='D')
```

- Using start or end date and number of periods (default is 'start')

```
pd.date_range('2020-01-01',  periods=7)
```

```
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-04',
               '2020-01-05', '2020-01-06', '2020-01-07'],
              dtype='datetime64[ns]', freq='D')
```

```
pd.date_range(end='2020-01-01',  periods=7)
```

```
DatetimeIndex(['2019-12-26', '2019-12-27', '2019-12-28', '2019-12-29',
               '2019-12-30', '2019-12-31', '2020-01-01'],
              dtype='datetime64[ns]', freq='D')
```

Default frequency is 'day' but there are many options available.

```
pd.date_range('2020-01-01',  periods=7,  freq='M')
```

```
DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31', '2020-04-30',
```

```
               '2020-05-31', '2020-06-30', '2020-07-31'],
              dtype='datetime64[ns]', freq='M')
```

```
pd.date_range('2020-01-01', periods=7, freq='MS')
```

```
DatetimeIndex(['2020-01-01', '2020-02-01', '2020-03-01', '2020-04-01',
               '2020-05-01', '2020-06-01', '2020-07-01'],
              dtype='datetime64[ns]', freq='MS')
```

Note: 'M' indicates the last day of month while 'MS' stands for 'month start'.

We can even derive frequencies from default ones:

```
pd.date_range('2020-01-01', periods=7, freq='3D') #3 days
```

```
DatetimeIndex(['2020-01-01', '2020-01-04', '2020-01-07', '2020-01-10',
               '2020-01-13', '2020-01-16', '2020-01-19'],
              dtype='datetime64[ns]', freq='3D')
```

Pandas also provides **period_range** and **timedelta_range** functions to create PeriodIndex and TimedeltaIndex, respectively:

```
pd.period_range('2001', periods=10, freq='Y')
```

```
PeriodIndex(['2001', '2002', '2003', '2004', '2005', '2006', '2007', '2008',
             '2009', '2010'],
            dtype='period[A-DEC]', freq='A-DEC')
```

```
pd.timedelta_range('0', periods=10, freq='H')
```

```
TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',
                '05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],
               dtype='timedelta64[ns]', freq='H')
```

. . .

We've learned how to create time series data but there are many other operations that Pandas can do with time series data. I will also cover **shifting**, **resampling** and **rolling**

time series data.

Shifting Time Series Data

Time series data analysis may require to shift data points to make a comparison. The **shift** and **tshift** functions shift data in time.

- shift: shifts the data
- tshift: shifts the time index

The difference between shift and tshift is better explained with visualizations. Let's first create a sample DataFrame with time series data:

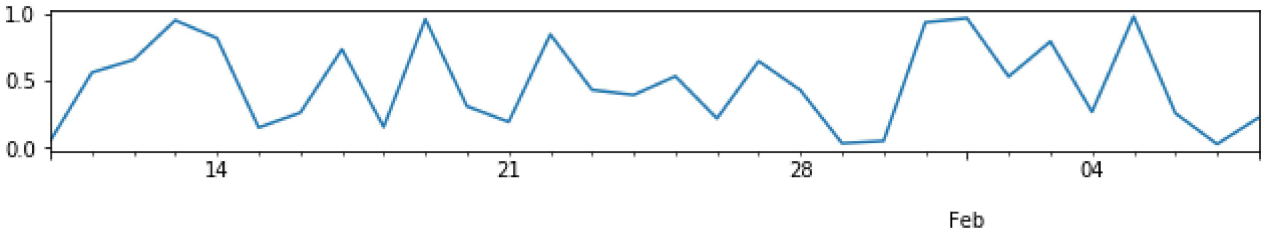
```
dates = pd.date_range('2019-01-10', periods=30)
values = np.random.random(30)
df = pd.DataFrame({'values':values}, index=dates)
df.head()
```

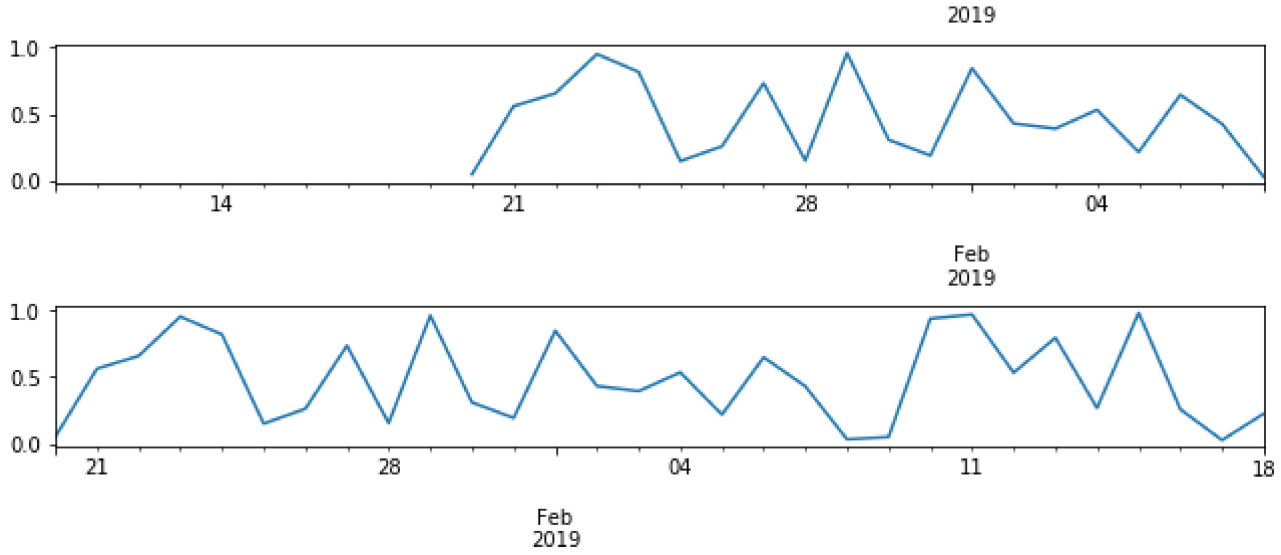
	values
2019-01-10	0.050005
2019-01-11	0.560746
2019-01-12	0.656711
2019-01-13	0.951660
2019-01-14	0.817852

Then we can plot original data and shifted data on the same figure to see the difference:

```
import matplotlib.pyplot as plt
```

```
fig, axs = plt.subplots(nrows=3, figsize=(10,6), sharey=True)
fig.tight_layout(pad=4)
df.plot(ax=axs[0], legend=None)
df.shift(10).plot(ax=axs[1], legend=None)
df.tshift(10).plot(ax=axs[2], legend=None)
```





order: original data, shift, tshift

Resampling

Another common operation with time series data is resampling. Depending on the task, we may need to resample data at a higher or lower frequency. Pandas handles both operations very well. Resampling can be done by **resample** or **asfreq** methods.

- asfreq returns the value at the end of the specified interval
- resample creates groups (or bins) of specified internal and lets you do aggregations on groups

It will be more clear with examples. Let's first create time series data for year.

```
dates = pd.date_range('2019-01-01', periods=365)
values = np.random.random(365)
df = pd.DataFrame({'values':values}, index=dates)
```

asfreq('M') returns the value on the last day of each month. We can confirm by checking the value at the end of January:

```
df.asfreq('M')[:5]
```

	values
2019-01-31	0.397073
2019-02-28	0.908748
2019-03-31	0.995094
2019-04-30	0.258743

2019-05-31	0.798235

```
df.iloc[30]
```

```
values     0.397073
Name: 2019-01-31 00:00:00, dtype: float64
```

resample('M') creates bins of months but we need to apply an aggregate function to get values. Let's calculate the average monthly values. We can also confirm the result by comparing the average value of January:

```
df.resample('M')
```

```
DatetimeIndexResampler [freq=<MonthEnd>, axis=0, closed=right, label=right, convention=start, base=0]
```

```
df.resample('M').mean()[:5]
```

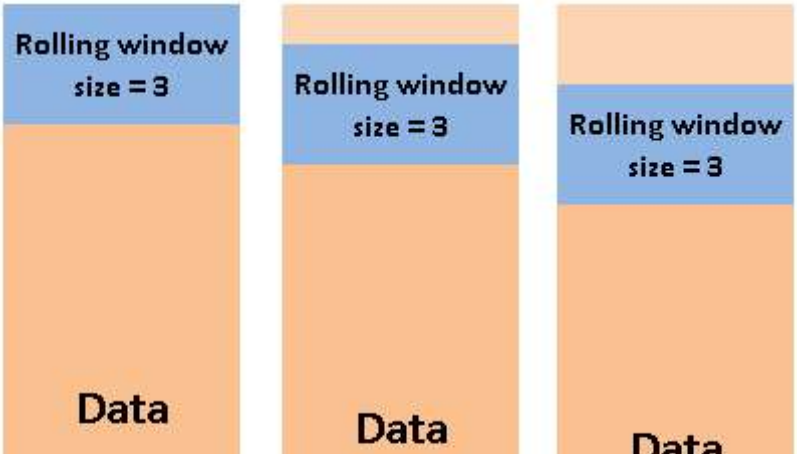
	values
2019-01-31	0.547003
2019-02-28	0.552702
2019-03-31	0.513727
2019-04-30	0.441477
2019-05-31	0.559626

```
df[:31].mean()
```

```
values     0.547003
dtype: float64
```

Rolling

Rolling is a very useful operation for time series data. Rolling means creating a rolling window with a specified size and perform calculations on the data in this window which, of course, rolls through the data. The figure below explains the concept of rolling.



It is worth noting that the calculation starts when the whole window is in the data. In other words, if the size of the window is three, the first aggregation is done at the third row. Let's apply rolling with size 3 to the DataFrame we created:

```
df.head()
```

	values
2019-01-01	0.482918
2019-01-02	0.856506
2019-01-03	0.838757
2019-01-04	0.246114
2019-01-05	0.113285

```
df['values'].rolling(3).mean()[:10]
```

```
2019-01-01         NaN
2019-01-02         NaN
2019-01-03    0.726060
2019-01-04    0.647126
2019-01-05    0.399385
2019-01-06    0.400386
2019-01-07    0.613169
2019-01-08    0.718845
2019-01-09    0.720514
2019-01-10    0.717290
Freq: D, Name: values, dtype: float64
```

. . .

Conclusion

Predictive analytics is highly valuable in the data science field and time series data is at the core of many problems that predictive analytics aims to solve. Hence, if you plan to work in the field of predictive analytics, you should definitely learn how to handle time series data.

• • •

Thank you for reading. Please let me know if you have any feedback.

My other posts on data manipulation and analysis

- The Most Underrated Tool in Data Science: NumPy
- The Most Underrated Tool in Data Science: NumPy (Part 2)
- Combining DataFrames Using Pandas
- Handling Missing Values with Pandas
- 3 Useful Functionalities of Pandas

References

- https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html
- https://jakevdp.github.io/PythonDataScienceHandbook/

Machine Learning | Predictive Analytics | Pandas | Time Series Analysis | Data Analysis