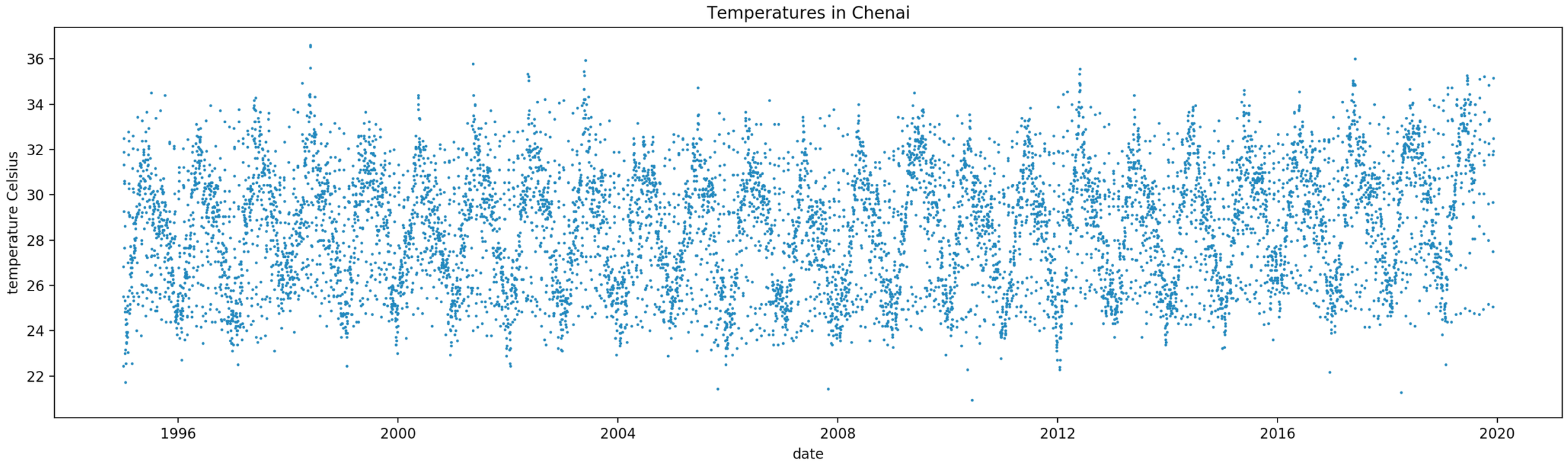
Forecasting and Time Series

Thomas Schwarz, SJ

Time Series

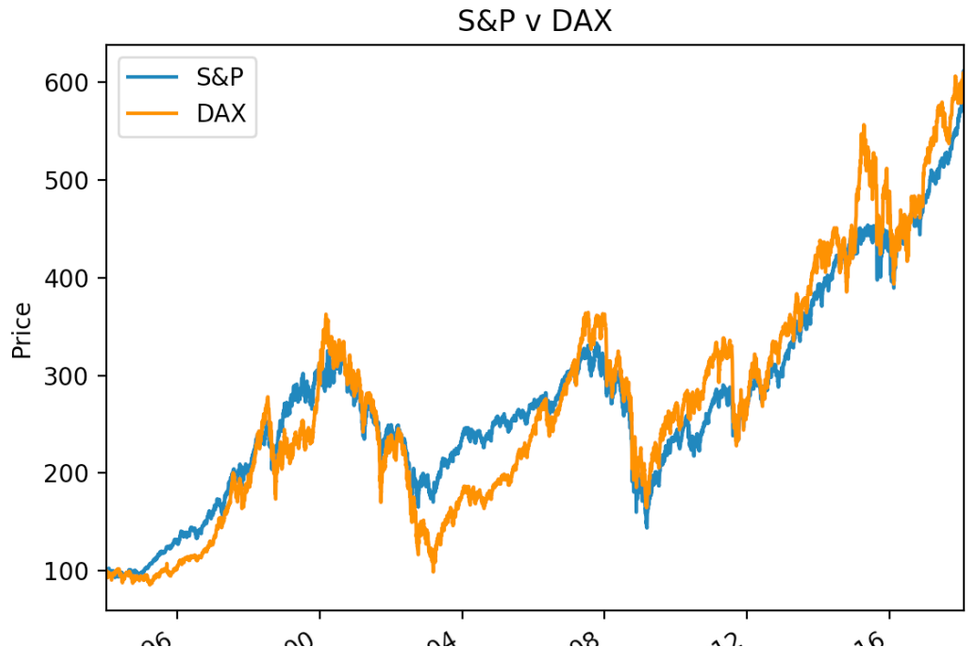
- Study of statistical data that depends on the time
  - Examples:
    - births in the US on a given day
    - names given to children in a certain year
    - exchange rates
    - ...

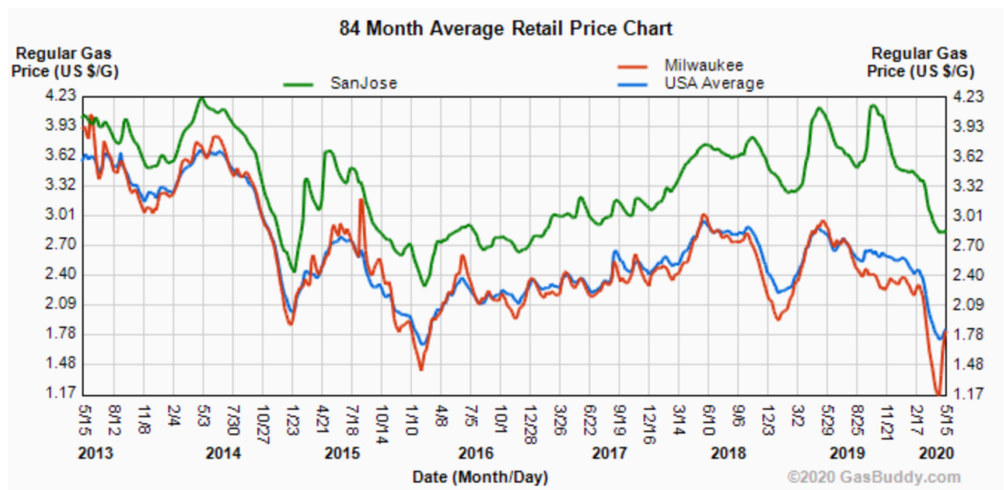
Introduction to Time Series



Introduction to Time Series

- Time Series Data is highly correlated with itself
    - Normal statistical descriptions such as mean are not very useful
- Temperature, stock market, gas prices have long-term trends
- Temperature and gas prices have seasonal trends





Introduction to Time Series

- Dealing with time data:
  - Generate time plot to see what is happening
    - Usually import from csv and transform data
  - Determine optically trends, cycles, outliers, undefined or obviously wrong values
  - Determine whether there is a need for transformation
    - e.g. our stock exchange data is normalized to make DOW and DAX comparable

Introduction to Time Series

- Typical transformations:
  - Linear normalizations
  - Logarithmic, exponential
    - E.g. variance grows with mean —> Logarithmic transform
    - Make a multiplicative dependence additive
      - timevalue = trend * seasonal * random —> Logarithmic: timevalue = trend+seasonal+random

Introduction to Time Series

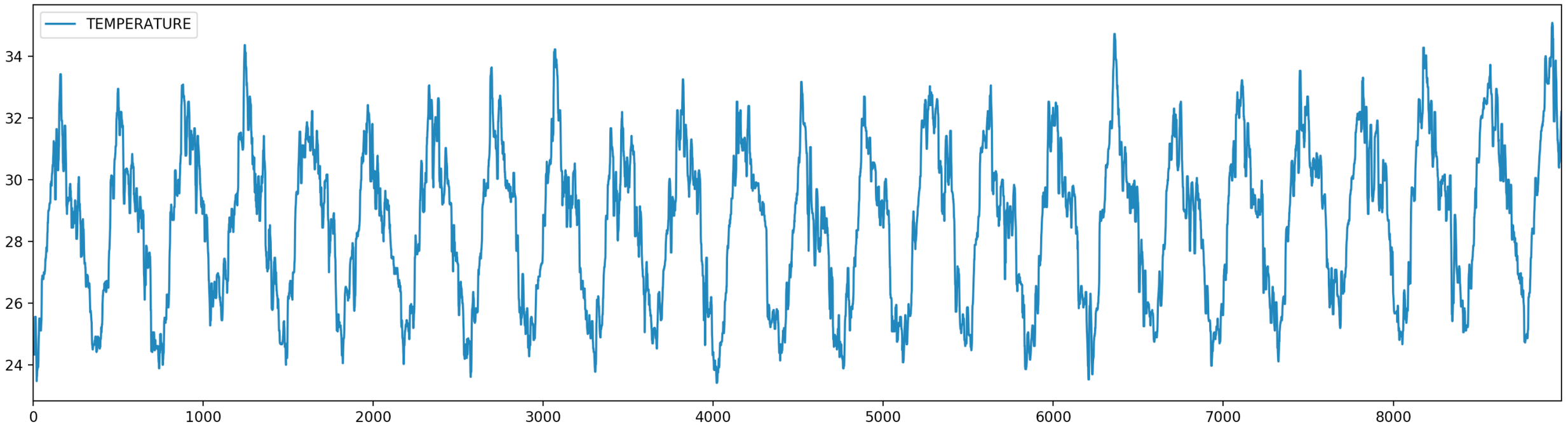
- Filtering: Transform time series (x_t) into other time series (y_t)
  - Smoothing out local variations
    - E.g. Linear smoothing with weights

$$a_{-s}, a_{-s+1}, \ldots, a_{-1}, a_0, a_1, \ldots, a_{r-1}, a_r$$

- $$y_t = \sum_{\nu=-s}^{r} a_\nu x_{t+\nu}$$

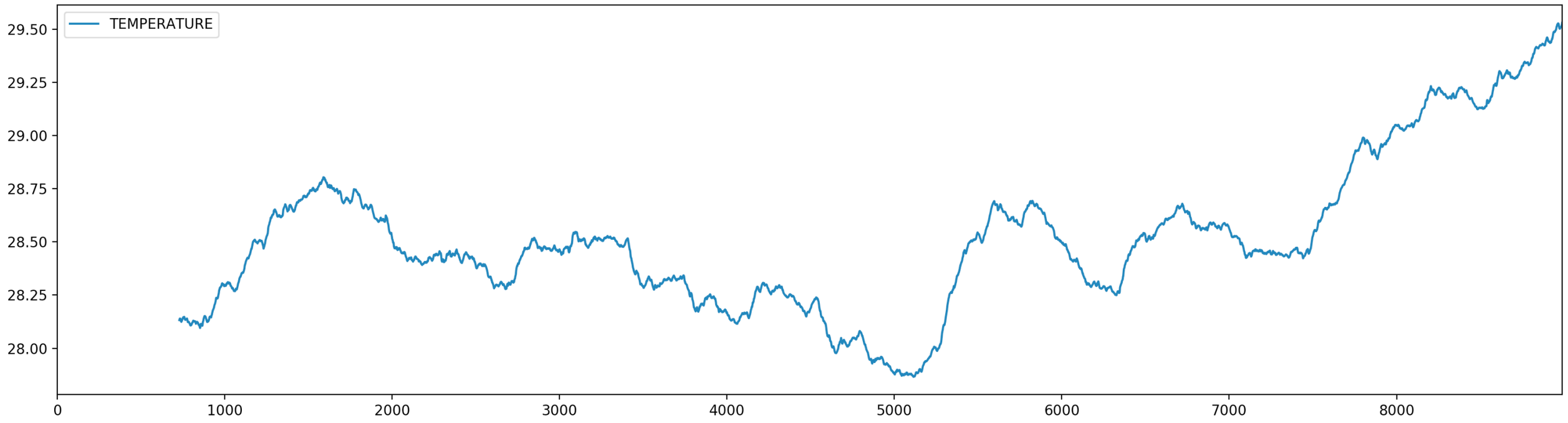
- Eg. Moving average: $$y_t = \frac{1}{2n+1} \sum_{\nu=-n}^{n} x_{t+\nu}$$

Introduction to Time Series



Median Smoothing with a Window of 10 of the Chenai Temperature Data

Introduction to Time Series



Smoothing over 2 years with mean

Introduction to Time Series

- Seasonal variations:
  - Three main models
    - $x_t = m_t + s_t + \epsilon_t$
    - $x_t = m_t \cdot s_t + \epsilon_t$
    - $x_t = m_t \cdot s_t \cdot \epsilon_t$
- Where
  - x_t time-series value
  - m_t long-time trend
  - s_t seasonal component
  - ϵ_t random noise

Auto-Correlation

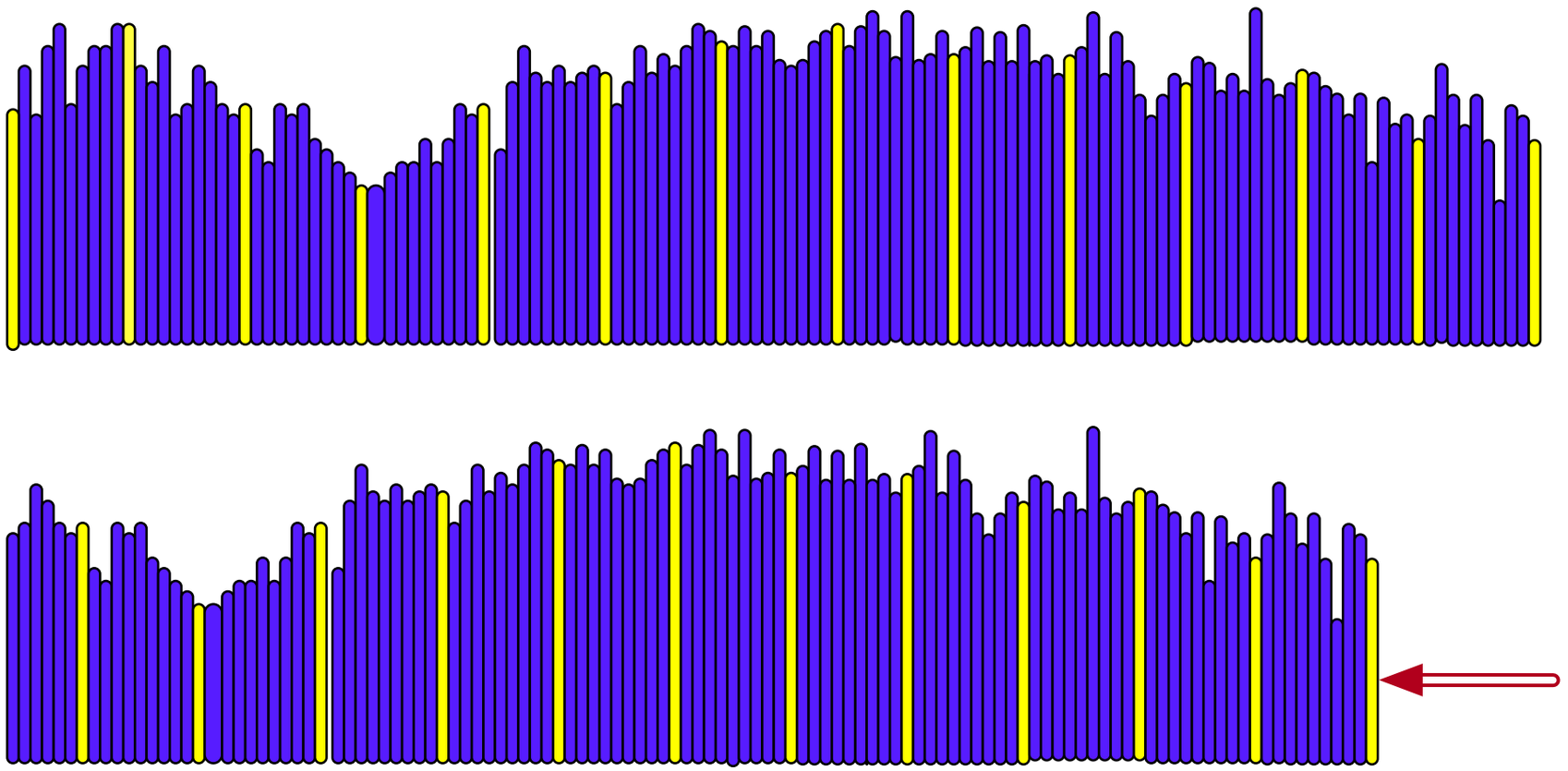
- Covariance:
  - Two random variables X, Y with means $E(X), E(Y)$
  - Covariance $\text{Cov}_{X,Y} = E\left((X - E(X)) \cdot (Y - E(Y))\right)$
  - If X and Y are independent:
    - $\text{Cov}_{X,Y} = E(XY - E(X)Y - E(Y)X + E(X)E(Y)) = 0$
  - If $\text{Cov}_{X,Y} > 0$:
    - High values for X go with high values for Y
  - If $\text{Cov}_{X,Y} < 0$:
    - High values for X go with low values for Y

Auto-Correlation

- Covariance depends on the units in which X and Y are given
- Therefore, we look at the correlation coefficient
- If X and Y have standard deviations σ_X and σ_Y respectively:
  - $\rho_{X,Y} = \dfrac{\mathrm{Cov}_{X,Y}}{\sigma_X \sigma_Y} \in [-1,1]$

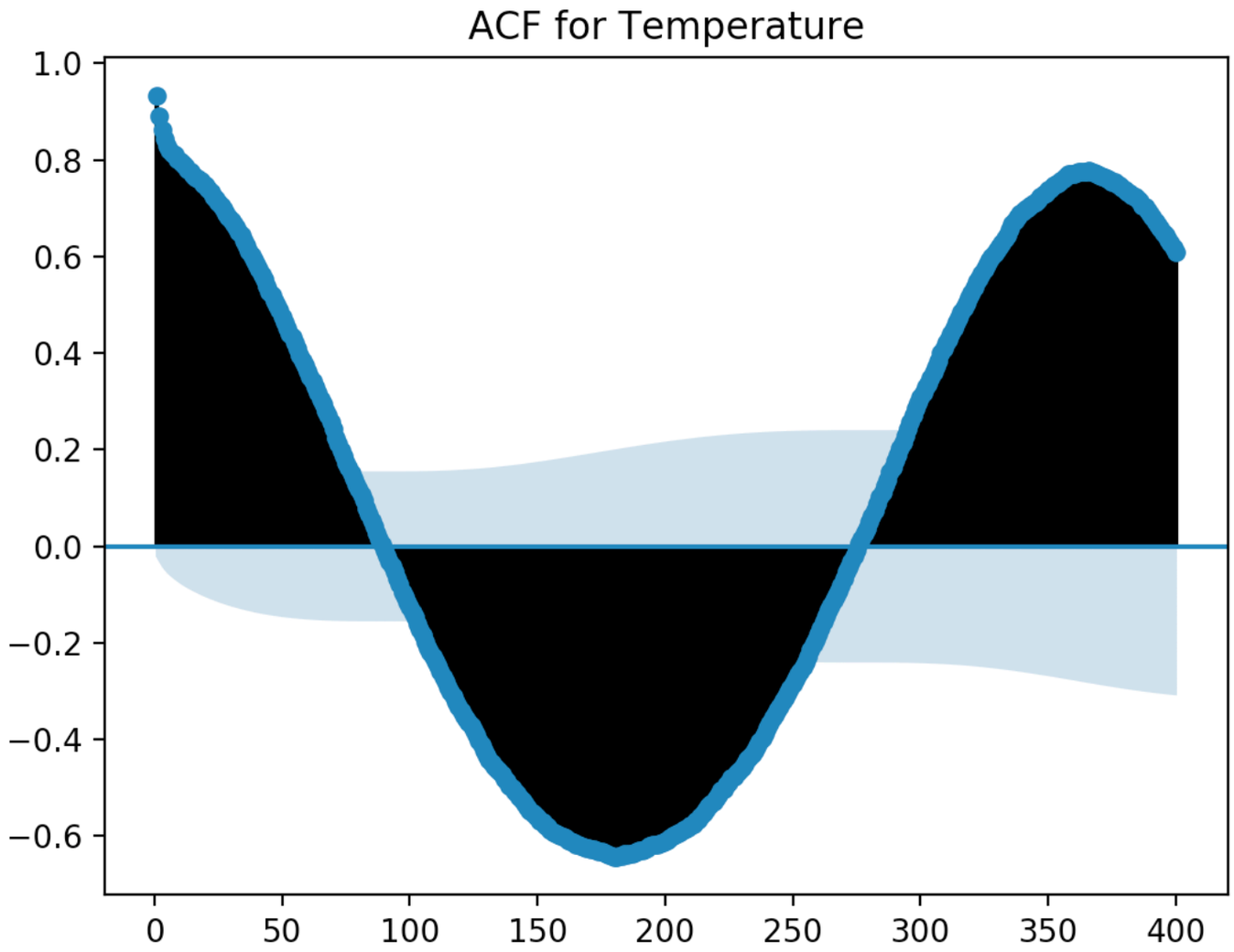
Introduction to Time Series

- Auto-correlation: Compare the correlation of a time series with itself moved by k positions



Introduction to Time Series

- Autocorrelation for Chenai temperatures
  - High auto-correlation for the next day and for a year afterwards
- The light blue denotes significance



Introduction to Time Series

- Goal:
  - Prediction of future value
- Testing:
  - Use latest data and predict based on previous data

Time Series in Pandas Overview

- Time series can be
  - *fixed frequency*
    - data points occur at regular intervals
  - *irregular*
    - no fixed unit of time / offset between data points

Time Series in Pandas Overview

- Time is given in
  - Timestamps
  - Fixed periods (January 2007)
  - Intervals of time such as periods
  - Experiment or elapsed time
    - Each timestamp measures the difference relative to a particular start time
      - For this: use `timedelta` indices in Pandas

Times in Python

- Python time in datetime

```
from datetime import datetime
my_date(year=2020, month = 2, day = 14)
```

  - Can take hours, minutes, seconds, microseconds as well
  - Also timezone information
- Can be combined with timedelta in order to calculate with dates

Times in Python

- datetime has a µsec resolution
- timedelta for the difference between two datetime objects

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> now
datetime.datetime(2021, 8, 23, 20, 30, 8, 285238)
>>> now.year
2021
>>> now.month
8
>>> now.day
23
>>> delta = now - datetime(1955,12,15)
>>> delta
datetime.timedelta(days=23993, seconds=73808, microseconds=285238)
```

Times in Python

- We can calculate with datetime and timedelta objects

```
>>> from datetime import datetime, timedelta
>>> timedelta(14)
datetime.timedelta(days=14)
>>> now = datetime.now()
>>> now + 2*timedelta(7)
datetime.datetime(2021, 9, 6, 20, 32, 57, 444108)
```

Times in Python

- We can convert between string and datetime (or date or time)
  - `strftime` takes a format argument

```
>>> from datetime import date, time, datetime, timedelta
>>> stamp = datetime(1776, 7, 4)
>>> str(stamp)
'1776-07-04 00:00:00'
>>> stamp.strftime('%d-%m-%Y')
'04-07-1776'
```

Times in Python

- Format codes:
  - Can also be used to convert from string to date

Type	Description
%Y	4-digit year
%y	2-digit year
%m	2-digit month
%d	2-digit day
%H	24 hour clock
%I	12 hour clock
%M	2-digit minute
%S	2-digit seconds
%w	weekday as integer
%U	week number in year
%W	week number (monday)
%z	UTC time zone offset
%F	Shortcut %Y-%m-%d
%D	Shortcut %m/%d/%y

Times in Python

```
>>> datetime.strptime('7/4/1776', '%m/%d/%Y')
datetime.datetime(1776, 7, 4, 0, 0)
```

Times in Pandas

- Pandas makes this easier
  - It comes with dateutil.parser
  - Which can guess most formats

```
>>> from dateutil.parser import parse
>>> parse('2020-06-30')
datetime.datetime(2020, 6, 30, 0, 0)
>>> parse('July 4, 2020')
datetime.datetime(2020, 7, 4, 0, 0)
>>> parse('4th of July, 2020')
datetime.datetime(2020, 7, 4, 0, 0)
>>> parse(1/2/2021, dayfirst=True)
```

Times in Pandas

- For the non-US world, need to use `dayfirst=True`

```
>>> parse('1/2/2021', dayfirst=True)
datetime.datetime(2021, 2, 1, 0, 0)
```

Times in Pandas

- A better solution than Python's for us is numpy's datetime
  - Called **datetime64**
    - Example:

```
np.array(['2020-01-01', '2020-01-02'], dtype='datetime64')
```

      - For instance, we can create equidistant arrays of timestamps
        - Notice the array notation " [Y] "

```
np.arange('2000', '2020', dtype = 'datetime64[Y]')
```

Times in Pandas

- In Pandas, there is DatetimeIndex

```
pd.date_range('2020-01-01', periods=12, freq='M')

DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31',
'2020-04-30', '2020-05-31', '2020-06-30', '2020-07-31',
'2020-08-31','2020-09-30', '2020-10-31', '2020-11-30',
'2020-12-31'], dtype='datetime64[ns]', freq='M')
```

  - Notice the 'ns' as the default for the precision
  - Pandas is good at inferring the format of the string
  - For reading in data, we can use pd.to_datetime with the format parameter

Times in Pandas

- Example:
  - Let's generate some random data and then add a time index to it.
  - Use this function to generate some values

```
import numpy as np
import pandas as pd
import random
import math

def v(i,j):
    return 10+math.floor(
          0.3*i+0.1*i**0.4 + random.normalvariate(0,5)
          )
```

Times in Pandas

- We create a value array from numpy with the confusing fromfunction function
- Then we create a time index, the dates being May 1, 2020 to May 31, 2020
- And finally make it into a data-frame

```
values = np.fromfunction(
                    np.vectorize(v),
                   (31,1)
                )
idx = pd.date_range('2020-05-01', periods = 31, freq='D')
df = pd.DataFrame(values, index=idx, columns=['values'])

print(df)
```

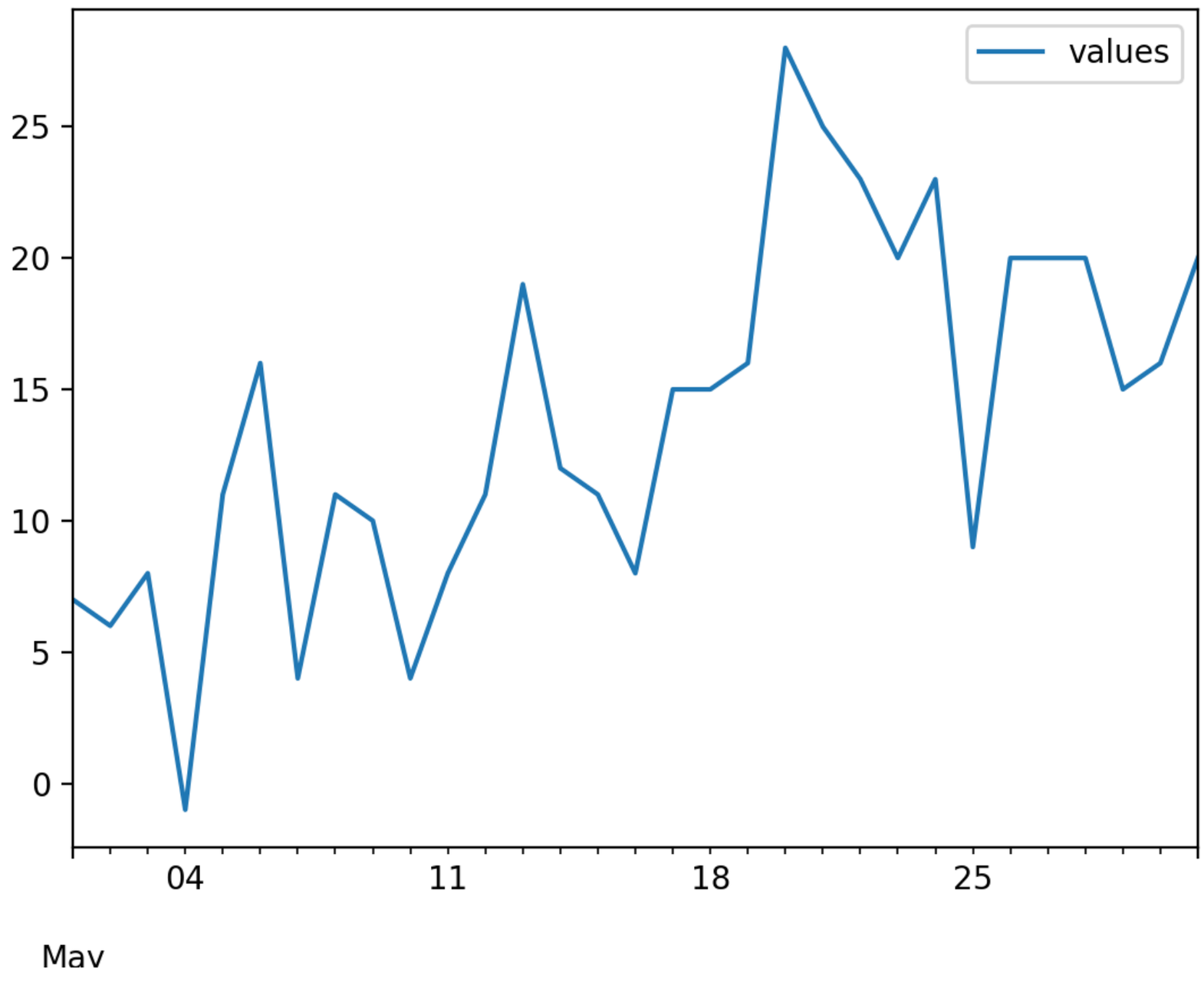
Times in Pandas

- Result:
  - (your values will differ)

	values
2020-05-01	13
2020-05-02	8
2020-05-03	13
2020-05-04	14
2020-05-05	14
2020-05-06	11
2020-05-07	19
2020-05-08	11
2020-05-09	12
2020-05-10	24
2020-05-11	22
2020-05-12	12
2020-05-13	3
2020-05-14	13
2020-05-15	16
2020-05-16	12
2020-05-17	13
2020-05-18	16
2020-05-19	19
2020-05-20	16
2020-05-21	8
2020-05-22	16
2020-05-23	23
2020-05-24	20
2020-05-25	17
2020-05-26	18
2020-05-27	13
2020-05-28	19
2020-05-29	15
2020-05-30	16
2020-05-31	11

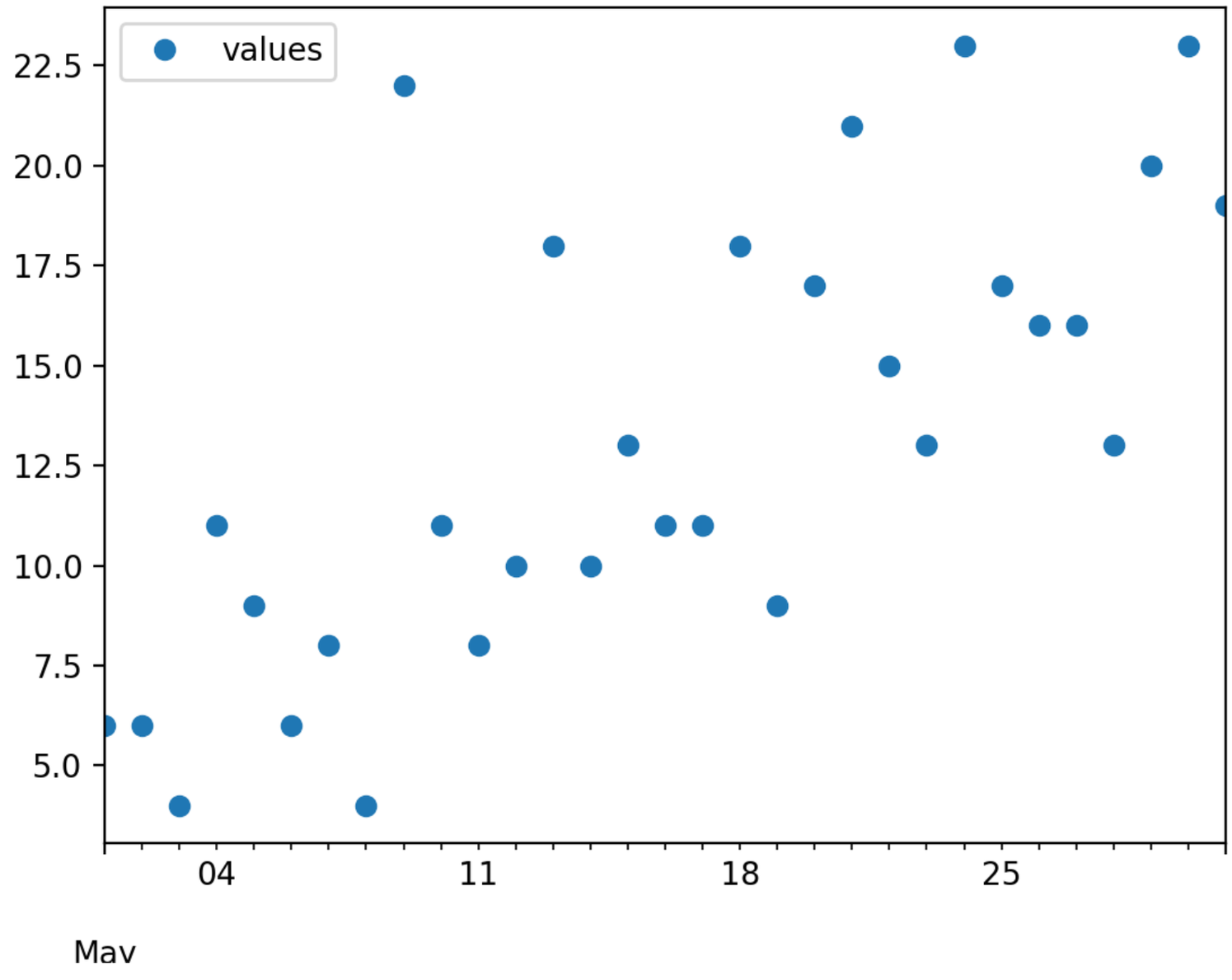
Time Series in Pandas Overview

```
df.plot()
plt.show()
```



Time Series in Pandas

```
df.plot(style ='o')
plt.show()
print(df)
```



Time Series Basics

- Usually a series or data frame object indexed by time-stamps
    - Often, the time stamps are hidden in the raw data as strings
- Arithmetic operations between differently indexed time-series automatically align on the dates

Time Series Basics

- Example:
  - Take previous times series and do average over three points
  - For this, need to create a dataframe with the indices shifted by one day
    - For this we use the timedelta

```
from datetime import timedelta

dfpre = pd.DataFrame(values,
            index=(idx-timedelta(1)),
            columns=['values']
)
```

Time Series Basics

- Example:

```
dfpre = pd.DataFrame(values,
                     index=(idx-timedelta(1)),
                     columns=['values'])
dfpost = pd.DataFrame(values,
                      index=(idx+timedelta(1)),
                      columns=['values'])
dfr = 0.5*df+0.25*dfpre+0.25*dfpost

ax = dfr.plot(style ='r-')
df.plot(style = 'b:', ax = ax)
```

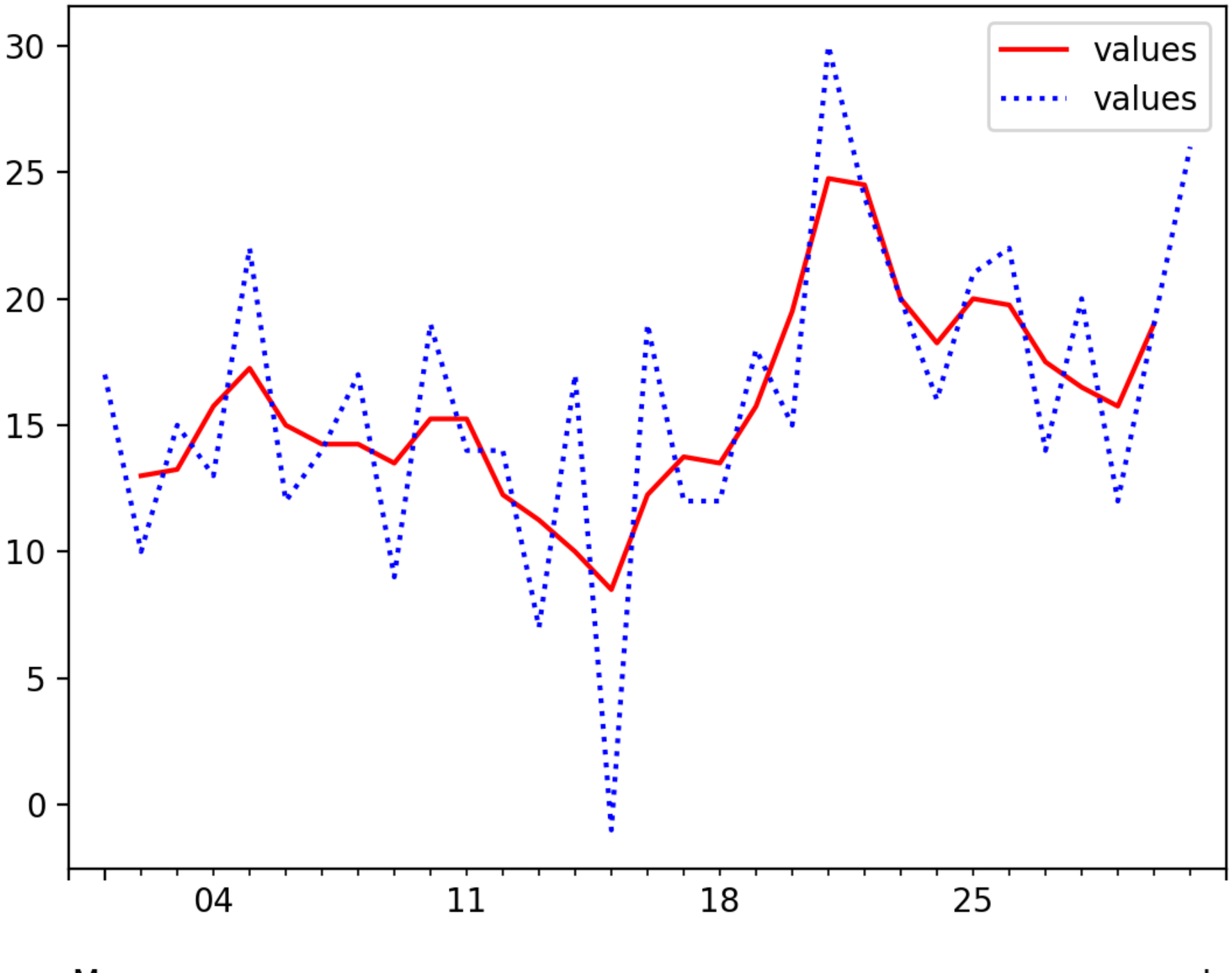
Time Series Basics

- Also: to display to graphs in the same figure:
  - df.plot returns an axes object
  - Which we can specify in the second plot:

```
dfpre = pd.DataFrame(values,
                     index=(idx-timedelta(1)),
                     columns=['values'])
dfpost = pd.DataFrame(values,
                     index=(idx+timedelta(1)),
                     columns=['values'])
dfr = 0.5*df+0.25*dfpre+0.25*dfpost

ax = dfr.plot(style ='r-')
df.plot(style = 'b:', ax = ax)
```

Time Series Basics



Time Series Indexing

- Notice that the "smoothed" value has no values for the first and last day of the month

Time Series Indexing

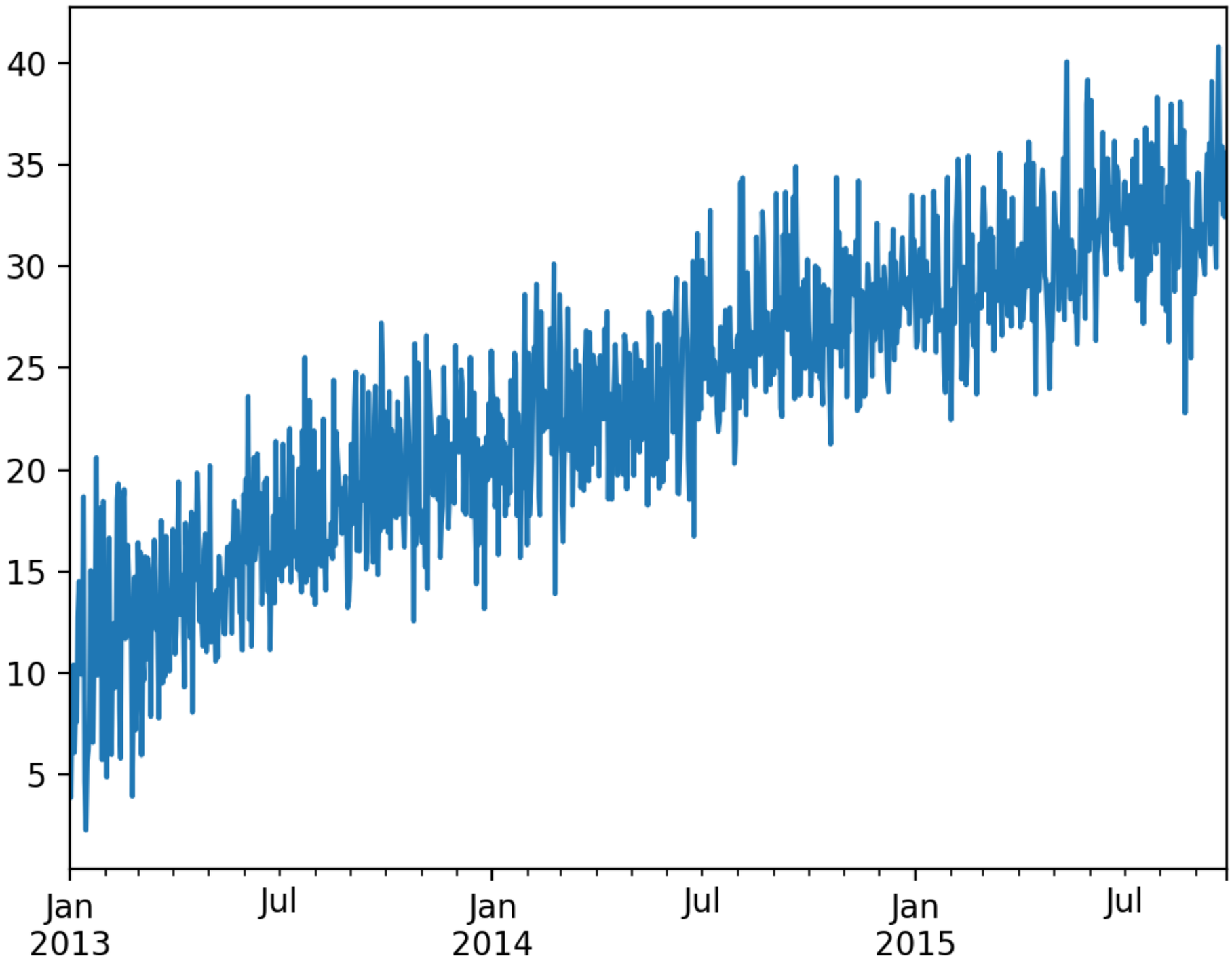
- Indexing and selection works as before
  - Create a **time series** with random component and a trend as sqrt.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

ts = pd.Series(
    3*np.random.randn(1000)
    +np.sqrt(np.linspace(100, 1100, 1000)),
 index = pd.date_range('1/1/2013', periods=1000))

ts.plot()
plt.show()
```

Time Series Indexing



Time Series Indexing

- We can access the value of the timeseries using a string

```
>>> ts['07-01-2014']
30.29958531626109

>>> ts['2014/07/01']
30.29958531626109
```

Time Series Indexing

- We can even slice with a partial date:

```
>>> ts['2015/01']
2015-01-01    29.018824
2015-01-02    26.018584
2015-01-03    26.322633
2015-01-04    29.884471
2015-01-05    30.861241
2015-01-06    27.578311
                …
2015-01-28    31.386897
2015-01-29    34.393147
2015-01-30    24.519918
2015-01-31    28.588378
Freq: D, dtype: float64
```

Time Series Indexing

- We can even slice with a partial date

```
>>> ts['2013']
2013-01-01     6.579922
2013-01-02     3.886306
2013-01-03     8.624116
2013-01-04    10.406576
2013-01-05     6.078220
                  ...
2013-12-27    21.622836
2013-12-28    19.501775
2013-12-29    23.271005
2013-12-30    19.694463
2013-12-31    25.848146
Freq: D, Length: 365, dtype: float64
```

- Or can do range queries

```
>>> ts['2013/12/12' : '2014/1/3']
2013-12-12    22.549265
2013-12-13    25.545776
2013-12-14    17.717606
2013-12-15    17.886024
2013-12-16    23.793512
2013-12-17    17.883687
2013-12-18    14.392229
2013-12-19    21.520931
2013-12-20    16.309788
2013-12-21    20.269989
2013-12-22    20.047367
2013-12-23    20.333124
2013-12-24    21.111016
2013-12-25    13.164632
2013-12-26    19.161525
2013-12-27    21.622836
2013-12-28    19.501775
2013-12-29    23.271005
2013-12-30    19.694463
2013-12-31    25.848146
2014-01-01    23.794902
2014-01-02    23.508325
2014-01-03    18.175757
```

Time Series Indexing

- Works for data frames as well, but remember to use loc or iloc

```
>>> dfr.loc['05-05-2020' : '05-08-2020']
            values
2020-05-05   12.75
2020-05-06   12.25
2020-05-07   13.00
2020-05-08   10.50
```

Time Series Indexing

- Duplicate indices
  - Time series can have multiple observations per time-stamp
  - Use `index.is_unique` method on data frame to find out
  - We would get slices for non-unique indices
  - We can aggregate them using groupby

Time Series Date Ranges

- Can easily generate data ranges with `pd.date_range`
  - Can specify the frequency and number

```
>>> pd.date_range('2020-06-01', periods = 30, freq = 'W')
DatetimeIndex(['2020-06-07', '2020-06-14', '2020-06-21', '2020-06-28',
               '2020-07-05', '2020-07-12', '2020-07-19', '2020-07-26',
               '2020-08-02', '2020-08-09', '2020-08-16', '2020-08-23',
               '2020-08-30', '2020-09-06', '2020-09-13', '2020-09-20',
               '2020-09-27', '2020-10-04', '2020-10-11', '2020-10-18',
               '2020-10-25', '2020-11-01', '2020-11-08', '2020-11-15',
               '2020-11-22', '2020-11-29', '2020-12-06', '2020-12-13',
               '2020-12-20', '2020-12-27'],
              dtype='datetime64[ns]', freq='W-SUN')
```

Time Series Ranges

- Frequencies:
  - D — day
  - B — business day
  - H — hourly
  - min T — minute
  - M — month end
  - MS — month beginning
  - ...

Time Series Ranges

- Can even get third Friday of each month

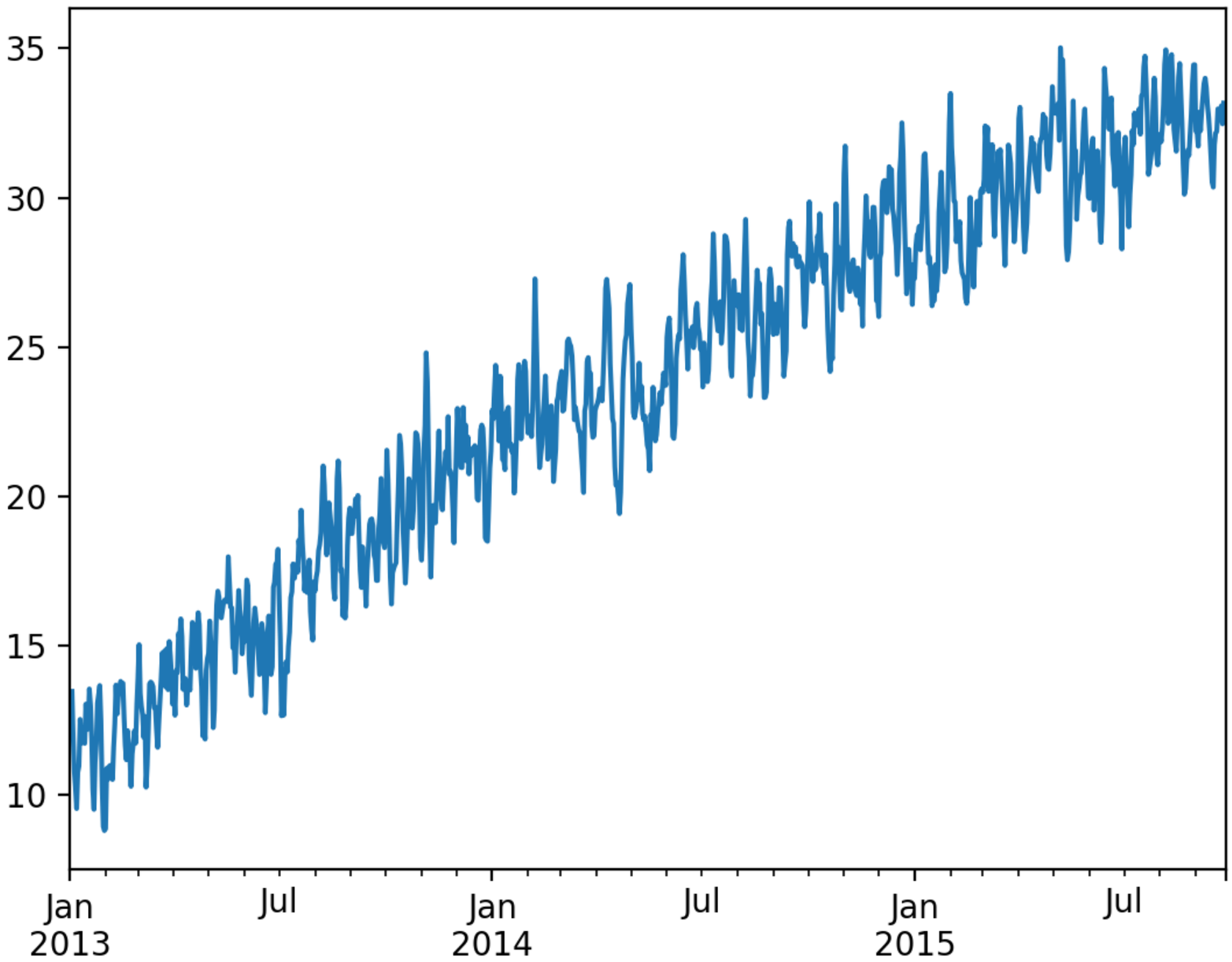
```
>>> pd.date_range('2020-06-28', freq = 'WOM-3FRI', periods = 20)
DatetimeIndex(['2020-07-17', '2020-08-21', '2020-09-18', '2020-10-16',
               '2020-11-20', '2020-12-18', '2021-01-15', '2021-02-19',
               '2021-03-19', '2021-04-16', '2021-05-21', '2021-06-18',
               '2021-07-16', '2021-08-20', '2021-09-17', '2021-10-15',
               '2021-11-19', '2021-12-17', '2022-01-21', '2022-02-18'],
              dtype='datetime64[ns]', freq='WOM-3FRI')
```

Time Series Shifting

- Shifting data — move data backward or forward
  - Can do so be ourselves
  - Or use shift

```
>>> smoother = 0.1*ts.shift(-2) + 0.2*ts.shift(-1) + 0.4*ts
0.2*ts.shift(1)+0.1*ts.shift(2)
>>> smoother.plot()
<matplotlib.axes._subplots.AxesSubplot object at
0x7f9c72f47ee0>
>>> plt.show()
```

Time Series Shifting

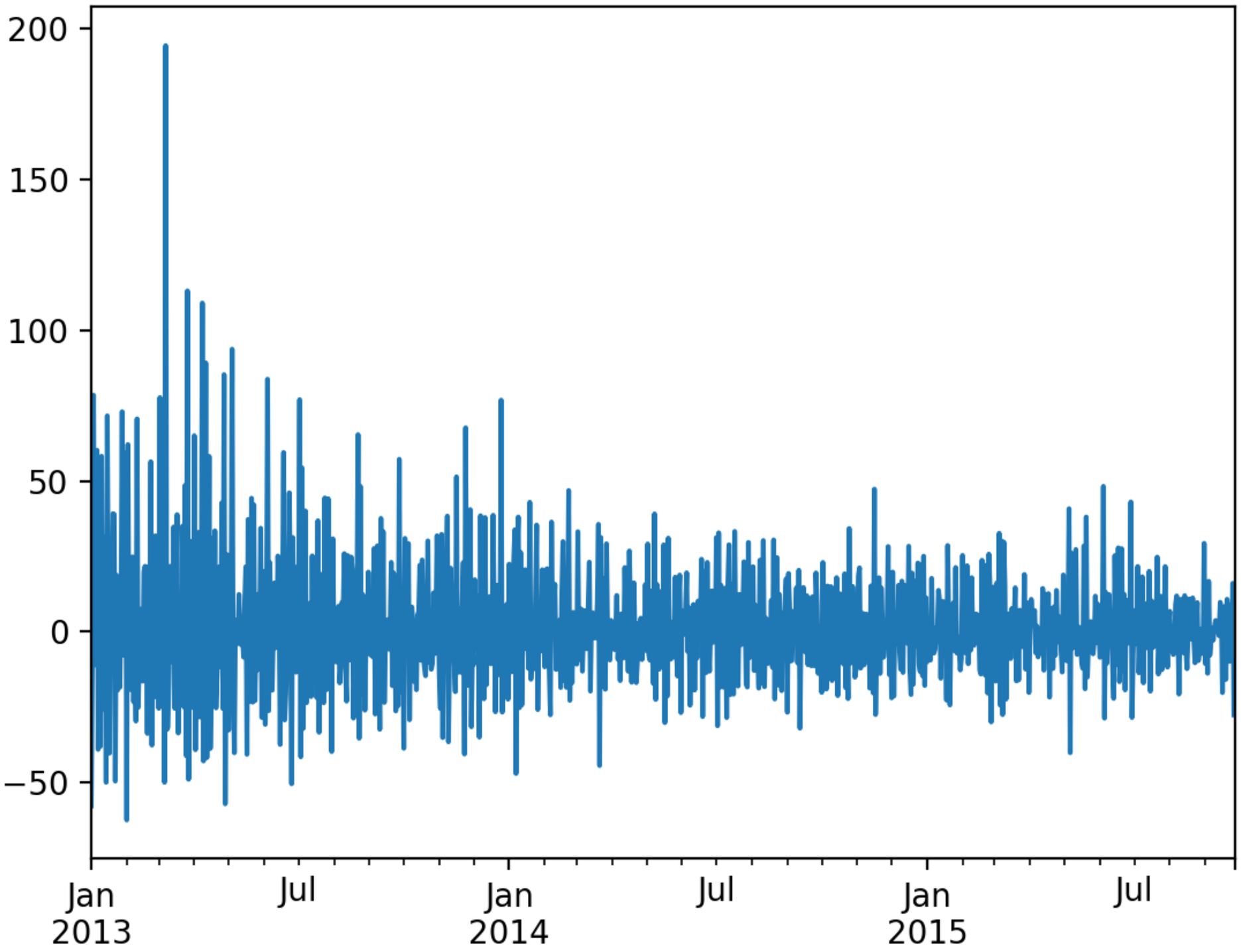


Time Series Shifting

- Using percentage changes

```
>>> plt.show()
>>> pct = (ts/ts.shift(-1)-1)*100
>>> pct.plot()
<matplotlib.axes._subplots.AxesSubplot object at
0x7f9c72f28760>
>>> plt.show()
```

Time Series Shifting



Time Series Shifting

- We can use offsets with shifting
  - Need to import the timestamp objects:
    - `from pandas.tseries.offsets import Day`
  - Can use arithmetic:

```
>>> pd.Timestamp.now()
Timestamp('2020-06-30 16:30:49.660811')
>>> pd.Timestamp.now()+3*Day()
Timestamp('2020-07-03 16:30:51.640833')
```

Time Zones

- Very unpleasant to deal with different time zones
  - Easier to use Coordinated Universal Time (UTC)
    - Successor to Greenwich Mean Time
  - Python: Use Olson database for time zones
    - Need to install pytz with pip

Time Zones

```
>>> kol = pd.date_range('3/9/2020 9:30', periods = 10, freq = 'H',
tz=pytz.timezone('Asia/Kolkata'))
```

```
>>> kol.tz_convert('America/Chicago')
DatetimeIndex(['2020-03-08 23:00:00-05:00', '2020-03-09 00:00:00-05:00',
               '2020-03-09 01:00:00-05:00', '2020-03-09 02:00:00-05:00',
               '2020-03-09 03:00:00-05:00', '2020-03-09 04:00:00-05:00',
               '2020-03-09 05:00:00-05:00', '2020-03-09 06:00:00-05:00',
               '2020-03-09 07:00:00-05:00', '2020-03-09 08:00:00-05:00'],
              dtype='datetime64[ns, America/Chicago]', freq='H')
```

Time Zones

- Many countries outside the tropics have daylight savings or summer times
  - This year: change on November 7, 2021

```
>>> pd.date_range('11/06/2021 20:00', periods = 10, freq = 'H',
tz=pytz.timezone('America/Chicago'))
DatetimeIndex(['2021-11-06 20:00:00-05:00', '2021-11-06 21:00:00-05:00',
               '2021-11-06 22:00:00-05:00', '2021-11-06 23:00:00-05:00',
               '2021-11-07 00:00:00-05:00', '2021-11-07 01:00:00-05:00',
               '2021-11-07 01:00:00-06:00', '2021-11-07 02:00:00-06:00',
               '2021-11-07 03:00:00-06:00', '2021-11-07 04:00:00-06:00'],
              dtype='datetime64[ns, America/Chicago]', freq='H')
```

  - There is a 1:00 am twice on November 7, 2021

Time Zones

- Many countries outside the tropics have daylight savings or summer times
  - This year: change on November 7, 2021

```
>>> from pandas.tseries.offsets import Hour
>>> pd.Timestamp('2021-11-06 20:00') + 5*Hour()
Timestamp('2021-11-07 01:00:00')
```

Periods

- Periods represents time spans
  - Days
  - Months
  - Quarters (with variously defined business quarters)
    - Q-Jan, Q-Feb, Q-MAR, … —> last calendar day of each month
    - BQ-Jan, BQ-Feb, …
    - QS-Jan, QS-Feb, … —> beginning quarter
    - BQS-Jan, BQS-Feb, …
- years: A - calendar year end, BA business year end,

Periods

- Examples:

```
>>> pd.date_range('2021', periods = 5, freq = 'QS')
DatetimeIndex(['2021-01-01', '2021-04-01', '2021-07-01',
               '2021-10-01',
               '2022-01-01'],
              dtype='datetime64[ns]', freq='QS-JAN')

>>> pd.date_range('2021', periods = 5,
                  freq = 'BQS-MAR')
DatetimeIndex(['2021-03-01', '2021-06-01',
               '2021-09-01', '2021-12-01',
               '2022-03-01'],
              dtype='datetime64[ns]', freq='BQS-MAR')
```

Resampling

- Resampling: convert a time series from one frequency to another
    - Down-sampling: Aggregating higher frequency data to lower frequency
    - Up-sampling: Converting lower frequency to higher frequency

Resampling

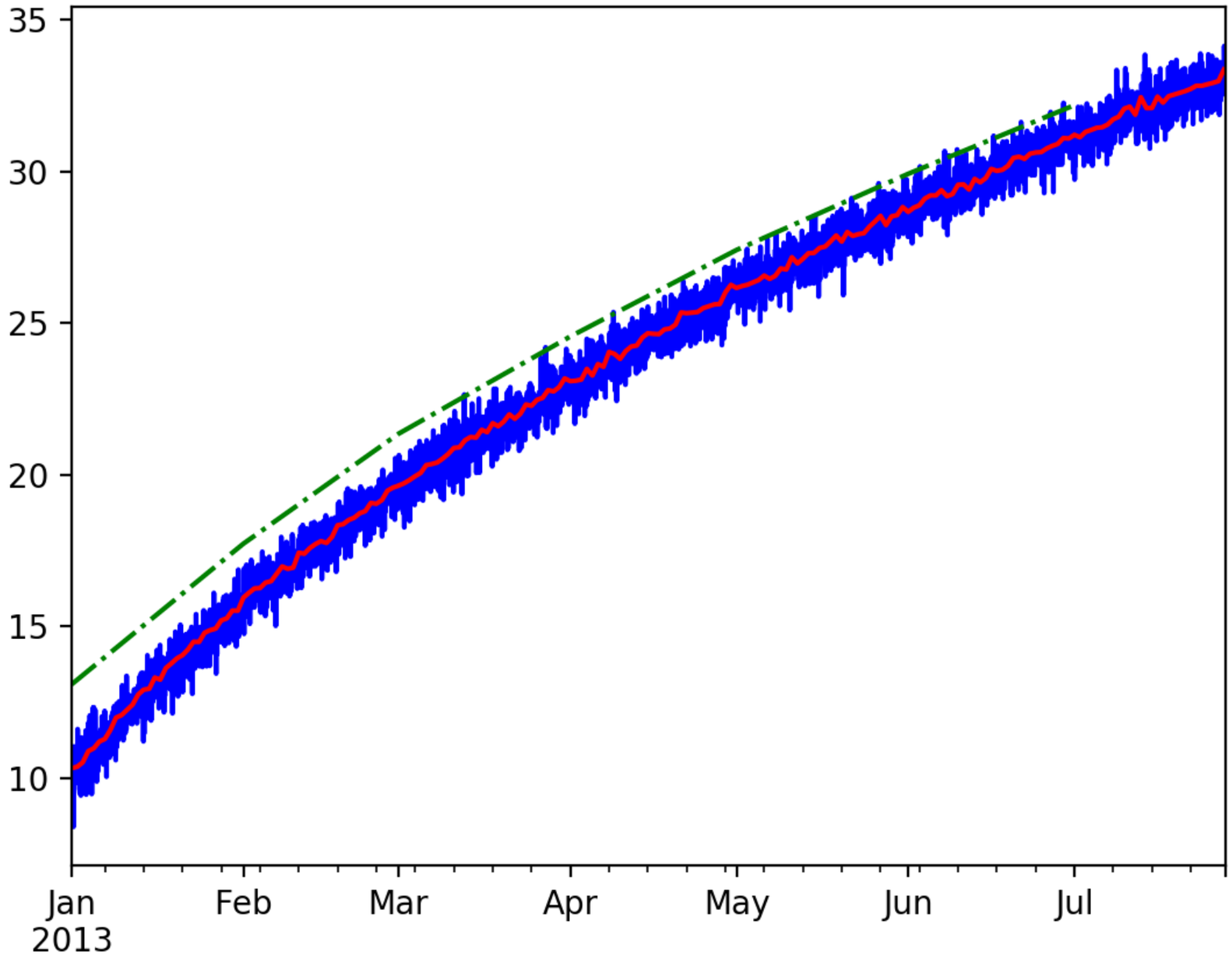
- Pandas has `resample`
  - Need an aggregation function
  - Example:
    - Create time series with hourly data
    - Down-sample to hours and days
      - Down-samples go to the beginning of the period
      - Unless we use lshift

Resampling

```
ts = pd.Series(0.5*np.random.randn(5000)+
          np.sqrt(np.linspace(100, 1100, 5000)),
                index = pd.date_range('1/1/2013',
                periods=5000, freq='H'))
ax = ts.plot(style = 'b-')
ts = ts.resample('D').mean()
ts.plot( style='r-', ax=ax)
ts = ts.resample('M').mean()
ts.plot(style = 'g-.', ax= ax)
plt.show()
```

Resampling

- The monthly sampling starts January 1

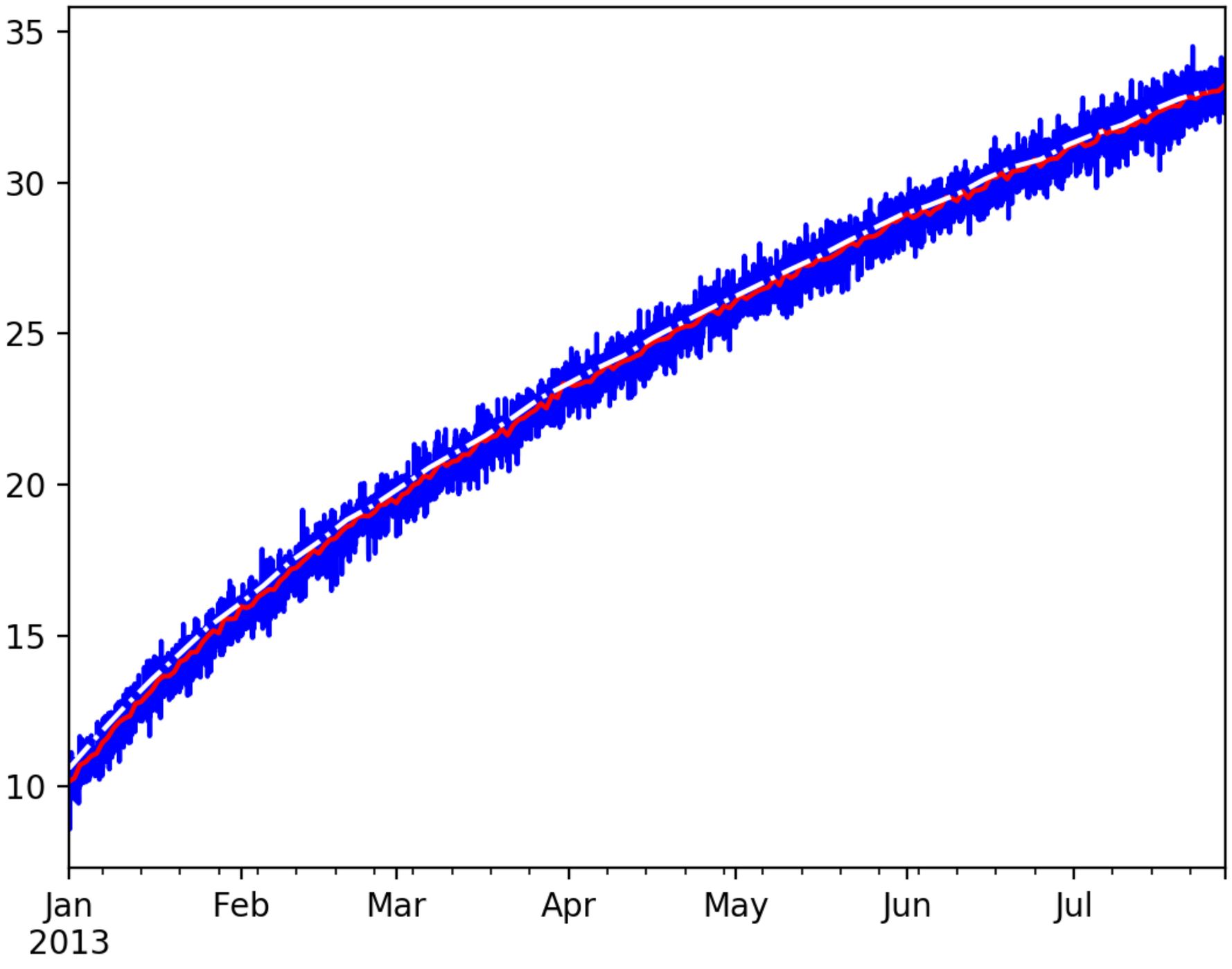


Resampling

- We can use scalars for the resample arguments

```
ts = ts.resample('5D').mean()
ts.plot(style = 'w-.', ax= ax)
```

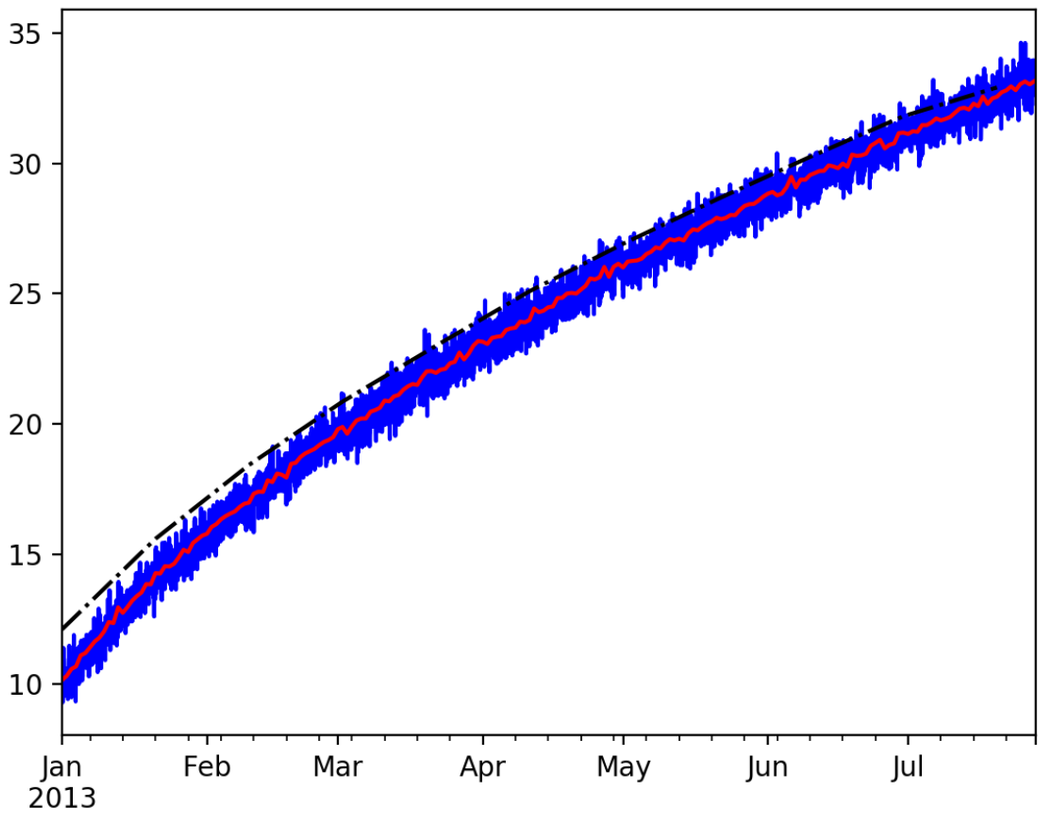
Resampling

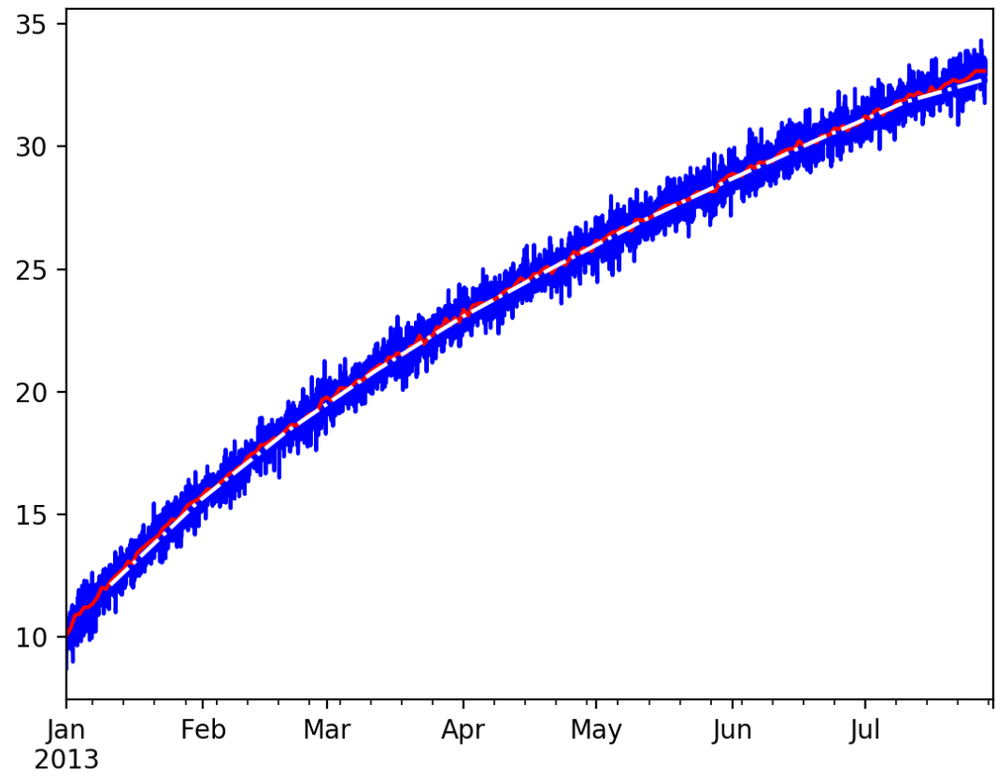


Resampling

- Can use loffset to shift the sample

```
ts = ts.resample('20D', loffset='10D').mean()
ts.plot(style = 'w-.', ax= ax)
```





Resampling

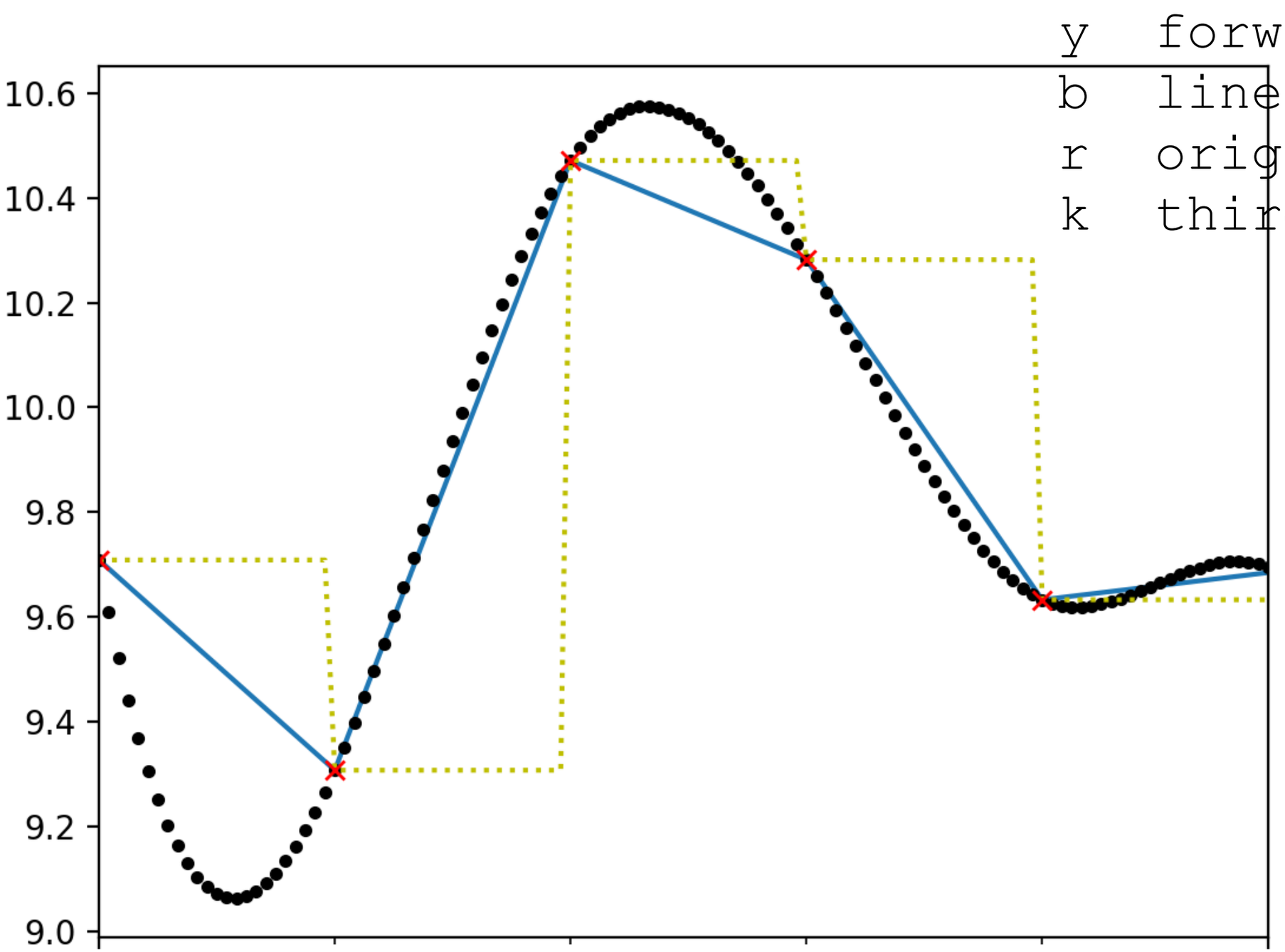
- When up sampling, we do not aggregate but need to interpolate
  - ffill - forward fill
  - bfill -backward fill
  - interpolate
    - needs method and sometimes degree

Resampling

- Example:

```
tshourly = ts.resample('H').interpolate(method='linear')
tshourly2 = ts.resample('H').interpolate(method='polynomial',
                                         order=3)
tshourly3 = ts.resample('H').ffill()
ax = tshourly[:5*24].plot()
tshourly2[:5*24].plot(style='k.', ax = ax)
tshourly3[:5*24].plot(style='y:', ax = ax)
ts[:5].plot(style = 'rx', ax = ax)
```

Resampling


y forward fill
b linear
r original data
k third degree polyn
10.6
10.4
10.2
10.0
9.8
9.6
9.4
9.2
9.0
01
Jan
2013
02
03
04
05

Moving Windows

- An important type of resampling is using moving windows
- `rolling` uses a sliding window
- Together with mean, this provides a way to see a trend

Moving Windows

- Example:
  - min_periods=10: start calculating whenever you have 10 values
  - center=True: place value in the center of sliding window

```
rol = ts.rolling(250, min_periods=10).mean()
rol2 = ts.rolling(250, min_periods=10,
center=True).mean()
myax = ts.plot(style = 'b-')
rol.plot(style = 'r-', ax = myax)
rol2.plot(style = 'k-.', ax = myax)
```

Moving Windows

- rolling window can also accept a fixed-size time offset
  - `rolling('20D')`

Moving Windows

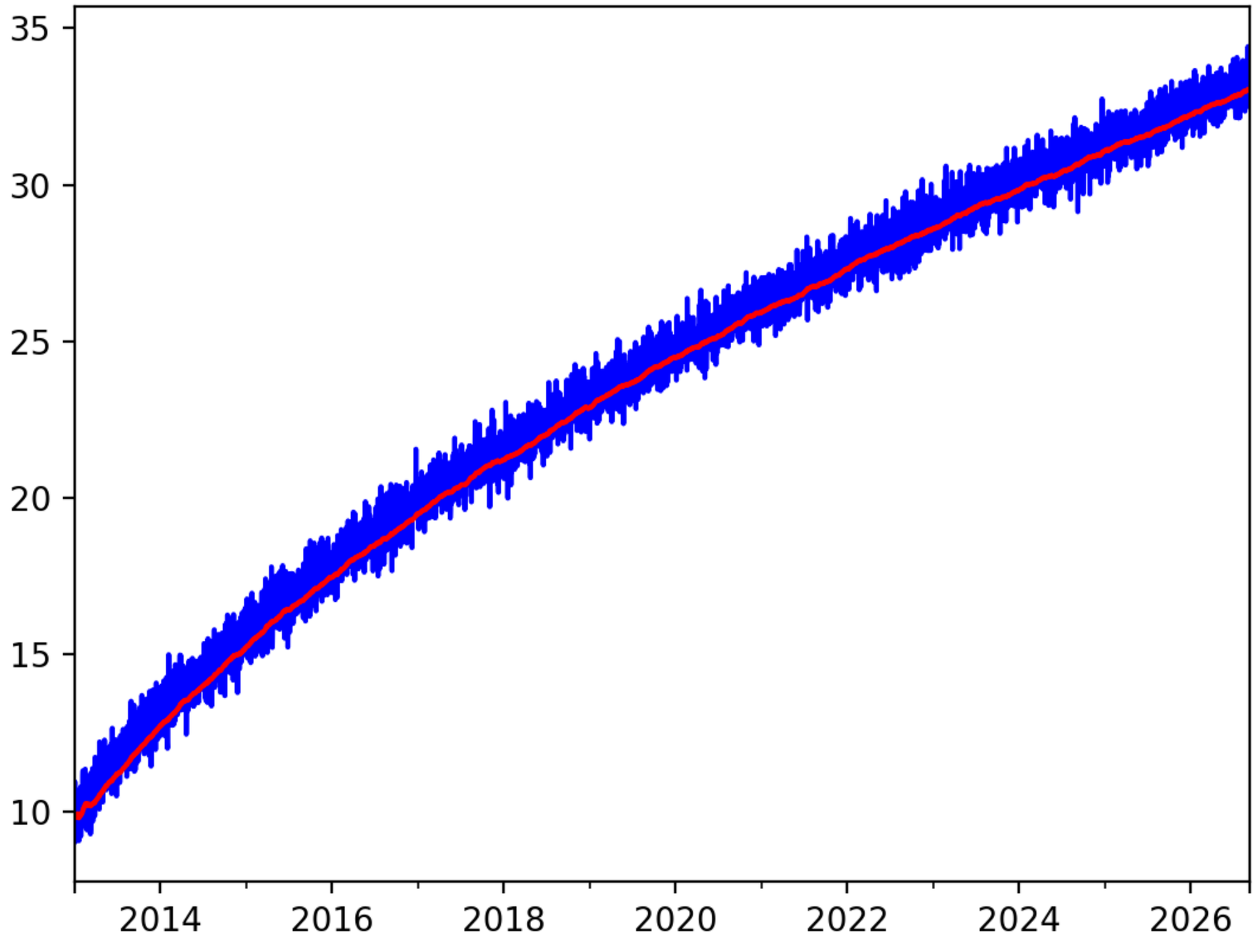
- All data points in a sliding window contribute equally to the aggregator
- Can also use exponential decay
  - Pandas has ewm
    - With span parameter for half the size

Moving Windows

- Example

```
rol = ts.ewm(span=250//2, min_periods=10).mean()
myax = ts.plot(style = 'b-')
rol.plot(style = 'r-', ax = myax)
```

Moving Windows



Pandas Time Series

- Time series need and have their own transformation tools
- Next steps
  - theory of time series
  - recognize static time series
  - learn how to use auto-correlations

Loading Time Series

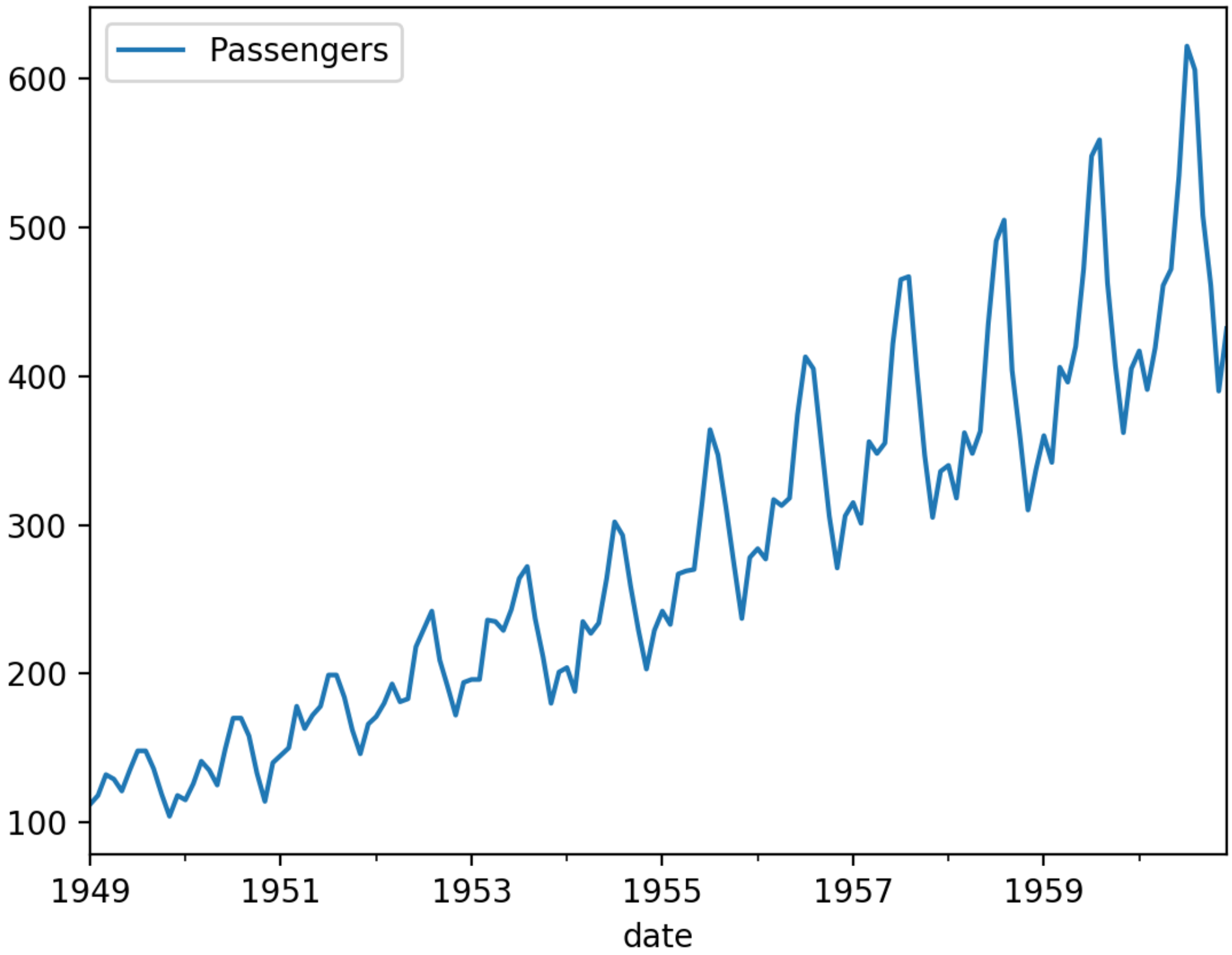
- Pandas can use `parse_dates` in order to extract date information
  - Example: Airline data
  - Has a column called 'date' with month and year

```
            Passengers
date
1949-01-01         112
1949-02-01         118
1949-03-01         132
1949-04-01         129
1949-05-01         121
```

Loading Time Series

```
def get_data_1():
    df = pd.read_csv('airline-passengers.csv',
                     parse_dates= {'date' : ["Month"]})
    df.set_index('date', inplace=True)
    return df
```

Loading Time Series



Loading Time Series

- The date information can also be split over several columns

```
YEAR,LOCATION,STATE ANSI,ASD CODE,COUNTY
ANSI,REFERENCE PERIOD,COMMODITY,"DRY, NONFAT,
HUMAN in LB","SKIM, UNSWEETENED, CONDENSED in
LB"
2016,CENTRAL,  ,  ,  ,JAN,MILK,"25,815,000",
2016,CENTRAL,  ,  ,  ,FEB,MILK,"23,976,000",
2016,CENTRAL,  ,  ,  ,MAR,MILK,"30,956,000",
2016,CENTRAL,  ,  ,  ,APR,MILK,"30,393,000",
2016,CENTRAL,  ,  ,  ,MAY,MILK,"37,183,000",
2016,CENTRAL,  ,  ,  ,JUN,MILK,"29,760,000",
2016,CENTRAL,  ,  ,  ,JUL,MILK,"26,698,000",
```

Loading Time Series

```
def get_data_2():
    df = pd.read_csv('MILK.csv',
        parse_dates= {'date' : ["YEAR", "REFERENCE PERIOD"]})
    df.set_index('date', inplace=True)
    return df
```

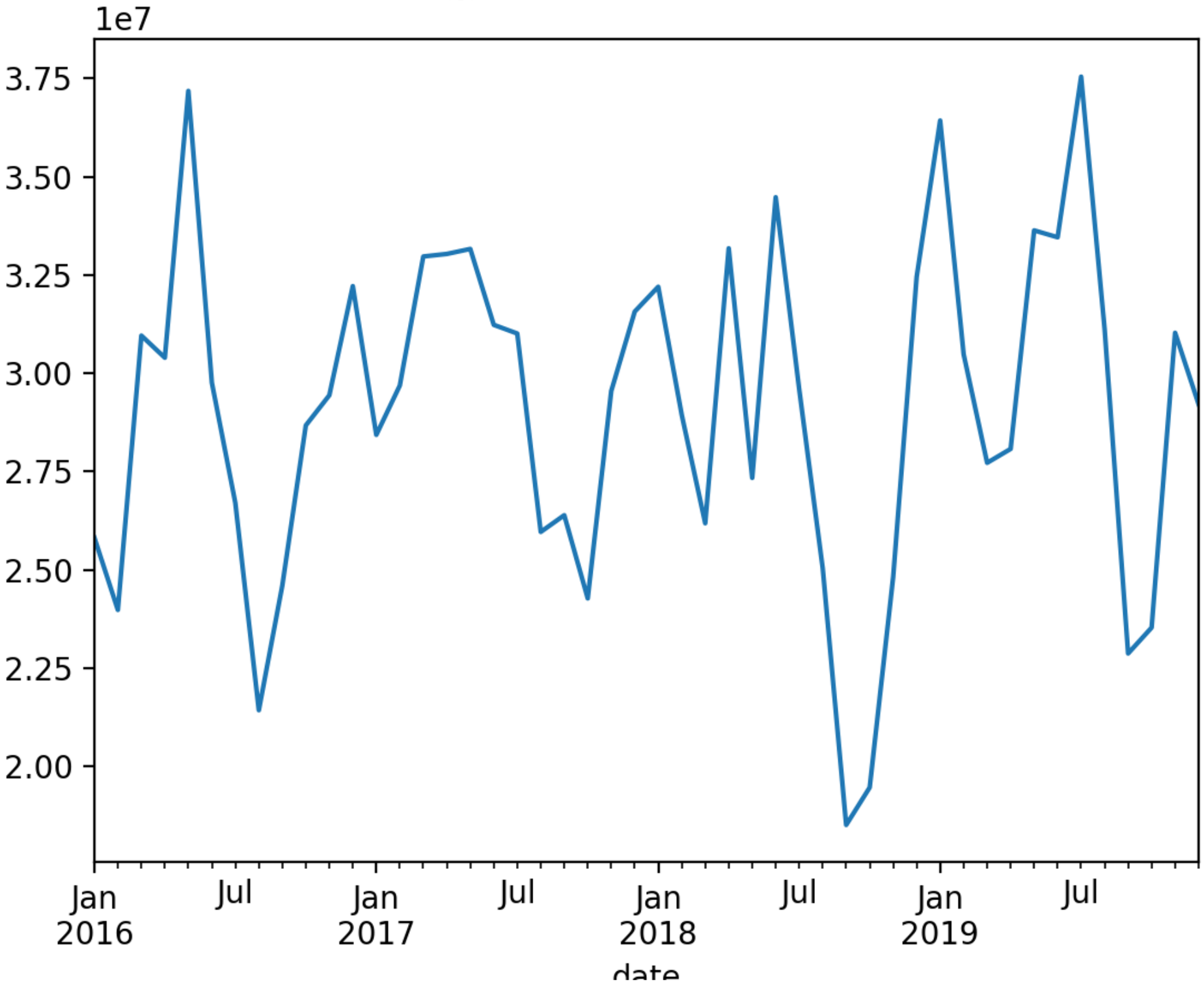
Loading Time Series

```
df = get_data_2()

df['DRY, NONFAT, HUMAN in LB'] = df['DRY, NONFAT, HUMAN
in LB'].str.replace(',','').astype(float)

df['DRY, NONFAT, HUMAN in LB'].plot()
```

Loading Time Series



Analysis of Time Series

Time Series

- General Model of a time series:
  - Trend + seasonal(s) + remainder
  - Trend × seasonal(s) × remainders
- Remainder can be modeled as a random walk

Trends

- Example: Disposable Personal Income Massachusetts and Missouri
  - from Federal Reserve Bank of St. Louis
  - Need to check the raw data: Use separator

```
df_ma = pd.read_csv('MAPCPI.csv',
                            sep = ',',
                           )
df_ma.set_index('DATE', inplace = True)
```

Trends

- Example continued:
  - We have two different files that we want to combine
    - Pandas has a merge function
      - Which needs to have a common column
        - Actually, merge implements an SQL-like join

```
df = pd.merge(df_ma, df_mo, on='DATE')
```

Trends

- We can now use linear recursion

```
from statsmodels.formula.api import ols
…

model = ols("df.MOPCPI ~ df.MAPCPI", df).fit()
inter, coef = model.params
print(inter, coef)
print(model.summary())
```

```
900.173966439816 0.6894112036215065
```

Trends

OLS Regression Results

Dep. Variable:	df.MOPCPI	R-squared:	0.995
Model:	OLS	Adj. R-squared:	0.995
Method:	Least Squares	F-statistic:	1.710e+04
Date:	Fri, 03 Jul 2020	Prob (F-statistic):	1.60e-103
Time:	16:40:01	Log-Likelihood:	-762.99
No. Observations:	91	AIC:	1530.
Df Residuals:	89	BIC:	1535.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	900.1740	147.748	6.093	0.000	606.601	1193.747
df.MAPCPI	0.6894	0.005	130.771	0.000	0.679	0.700

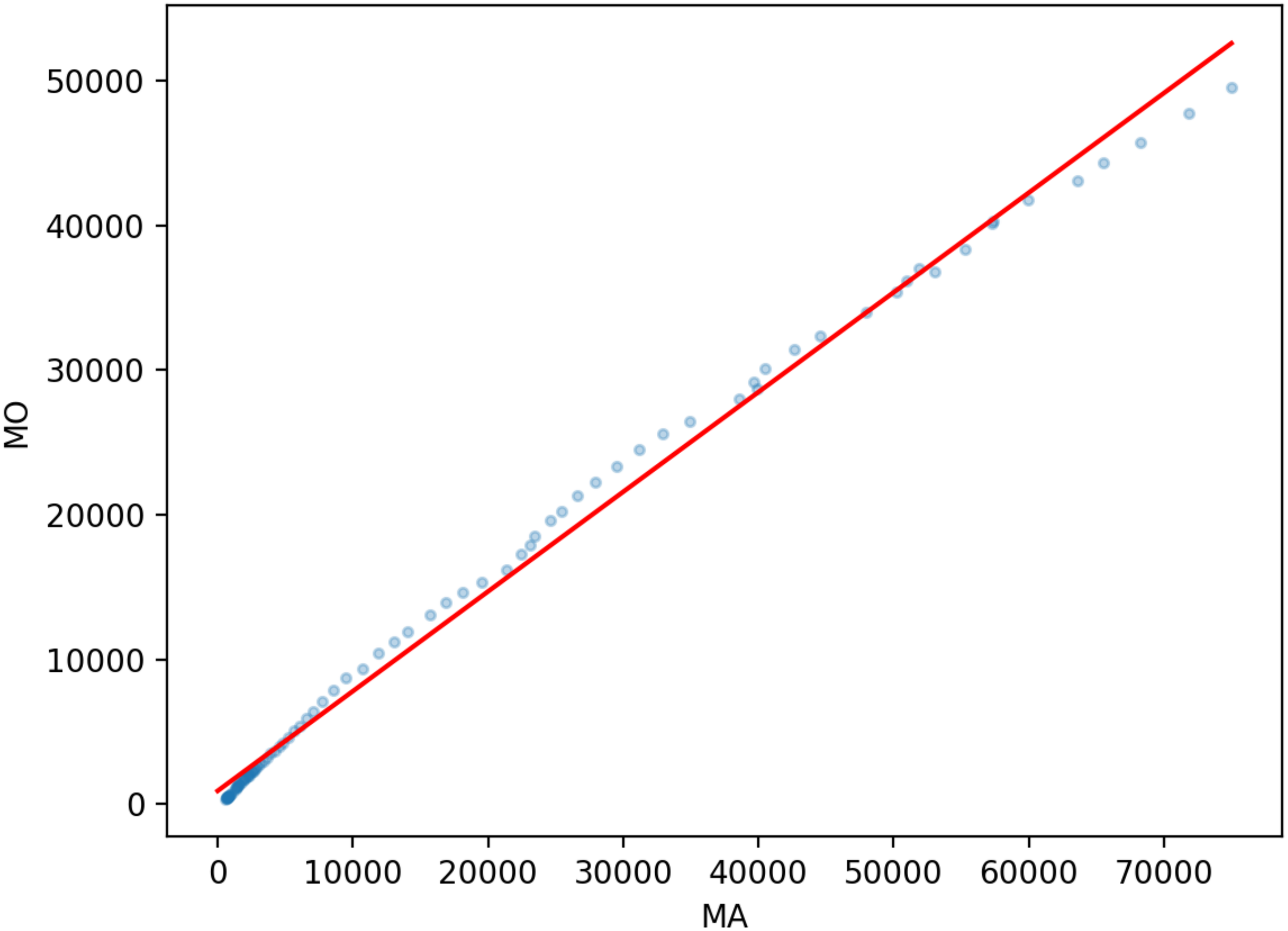
Omnibus:	0.064	Durbin-Watson:	0.066
Prob(Omnibus):	0.969	Jarque-Bera (JB):	0.069
Skew:	0.051	Prob(JB):	0.966
Kurtosis:	2.911	Cond. No.	3.69e+04

Trends

- We then plot the result:

```
plt.plot(df.MAPCPI, df.MOPCPI, '.', alpha=0.3)
plt.plot(np.linspace(0,75000),
         inter + coef*np.linspace(0,75000),
         'r-')

plt.xlabel('MA')
plt.ylabel('MO')
plt.show()
```


50000
40000
30000
20000
10000
0
MO
0
10000
20000
30000
40000
50000
60000
70000
MA

Trends

- We now look at the residual
  - We define it as an additional column

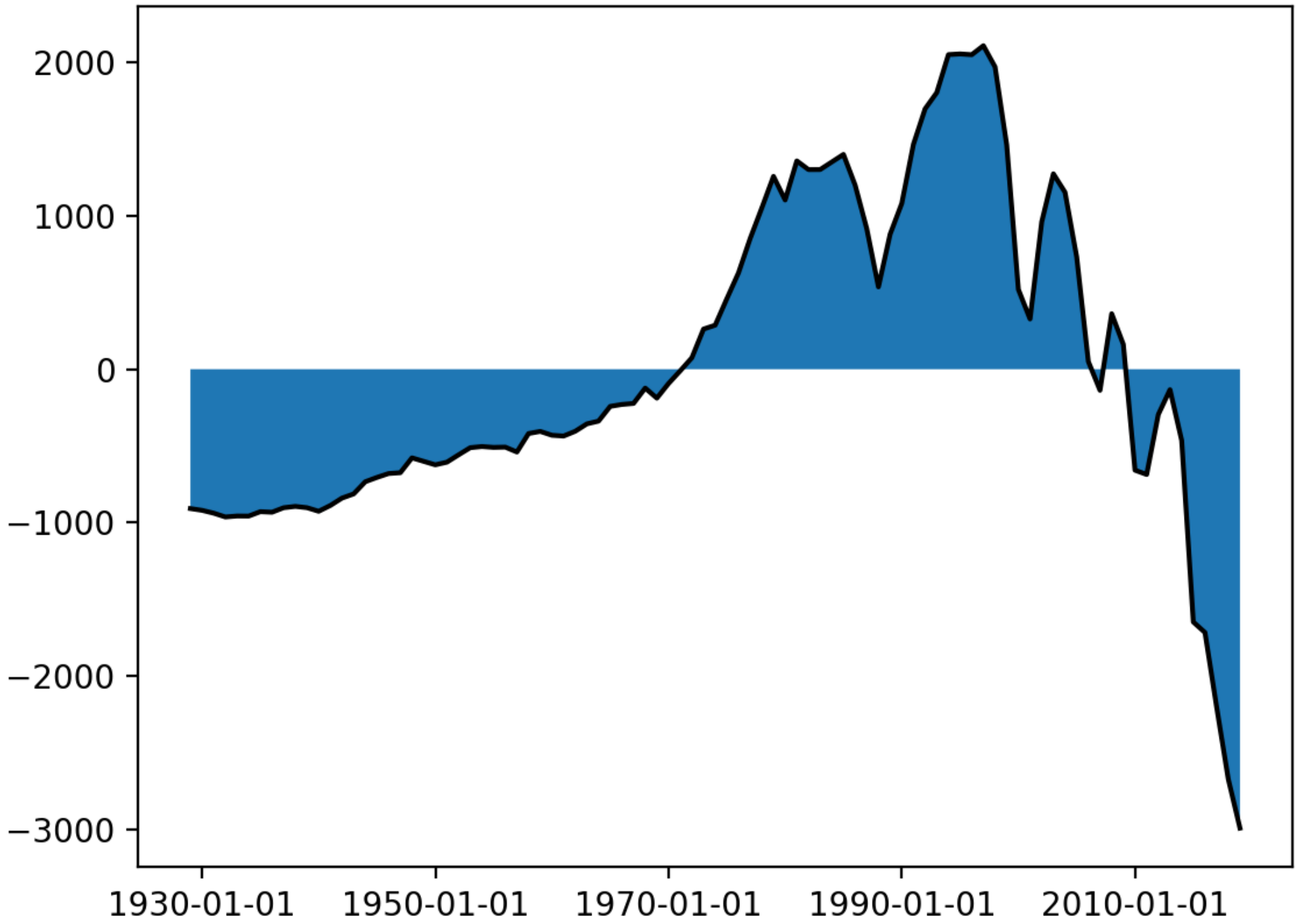
```
df['residual'] = df.MOPCPI - inter-coef*df.MAPCPI
```

Trends

- Showing the residual:
  - Use xticks to make the x-axis readable
  - And use fill to make the result clearer

```
plt.plot(df.residual, 'k-')
plt.xticks(['1930-01-01', '1950-01-01', '1970-01-01',
'1990-01-01', '2010-01-01'])
plt.fill_between(df.index, 0, df.residual)
plt.show()
```

Trends


2000
1000
0
−1000
−2000
−3000
1930-01-01
1950-01-01
1970-01-01
1990-01-01
2010-01-01

Trends

- This would be a bad regression for prediction!
  - We are much lower at the beginning and the end of the time period
  - And much higher in the middle
- Do not be fooled by spurious prediction

Decomposition: Dealing with Seasons and Trends

Seasonal Dummy Variables

- Suppose you want to use linear regression
- Need to account for the effect of week-days
  - Introduce dummy variables

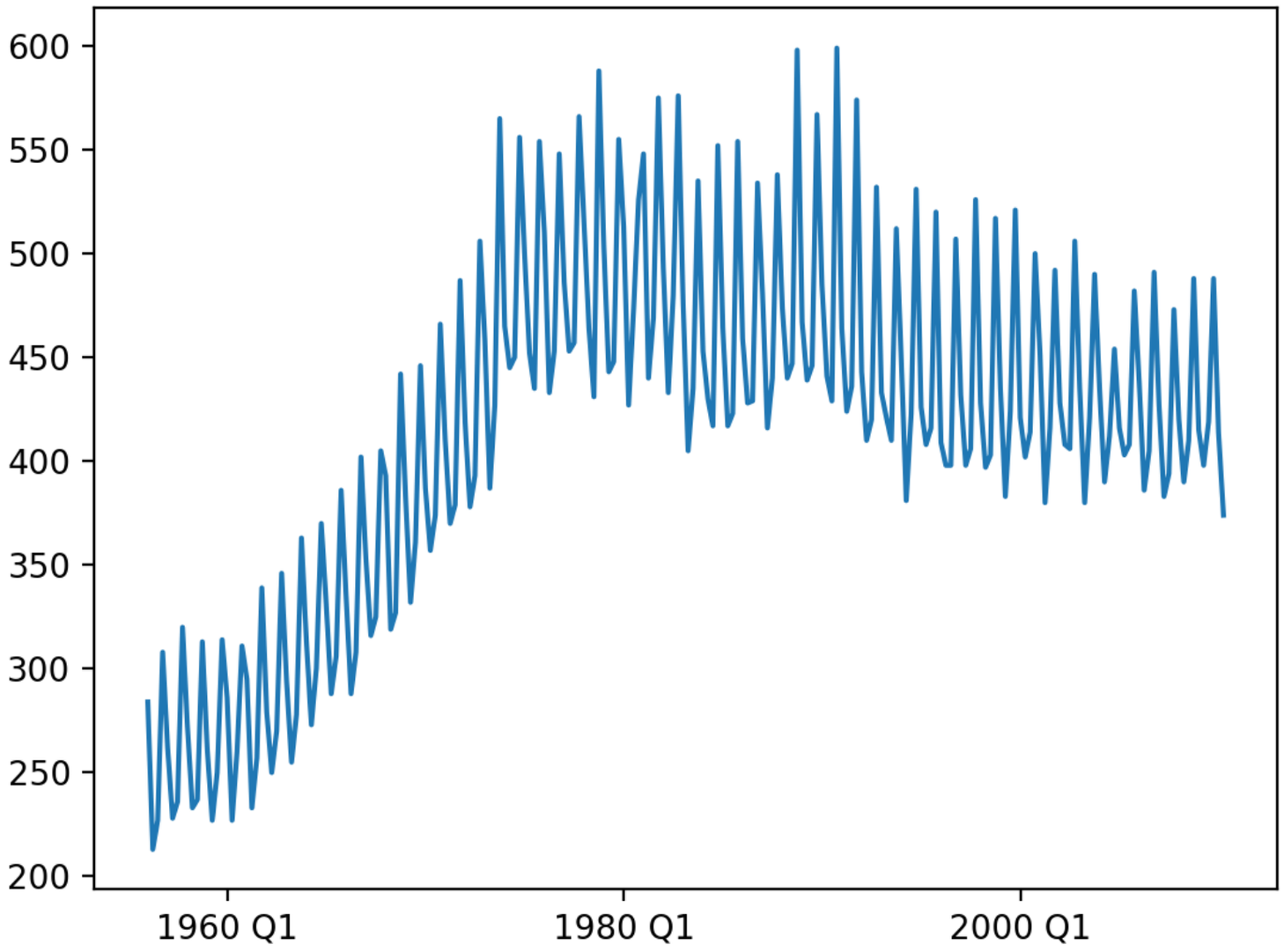
	t1	t2	t3	t4	t5	t6	t7
Mon	1	0	0	0	0	0	0
Tue	0	1	0	0	0	0	0
Wed	0	0	1	0	0	0	0
Thu	0	0	0	1	0	0	0
Fri	0	0	0	0	1	0	0
Sat	0	0	0	0	0	1	0
Sun	0	0	0	0	0	0	1

Seasonal Dummy Variables

- Use linear regression including the dummy variables
  - $y = \beta_0 + \beta_1 x_1 + \ldots \beta_n x_n + \gamma_1 t_1 + \gamma_2 t_2 + \gamma_3 t_3 + \ldots + \gamma_7 t_7$

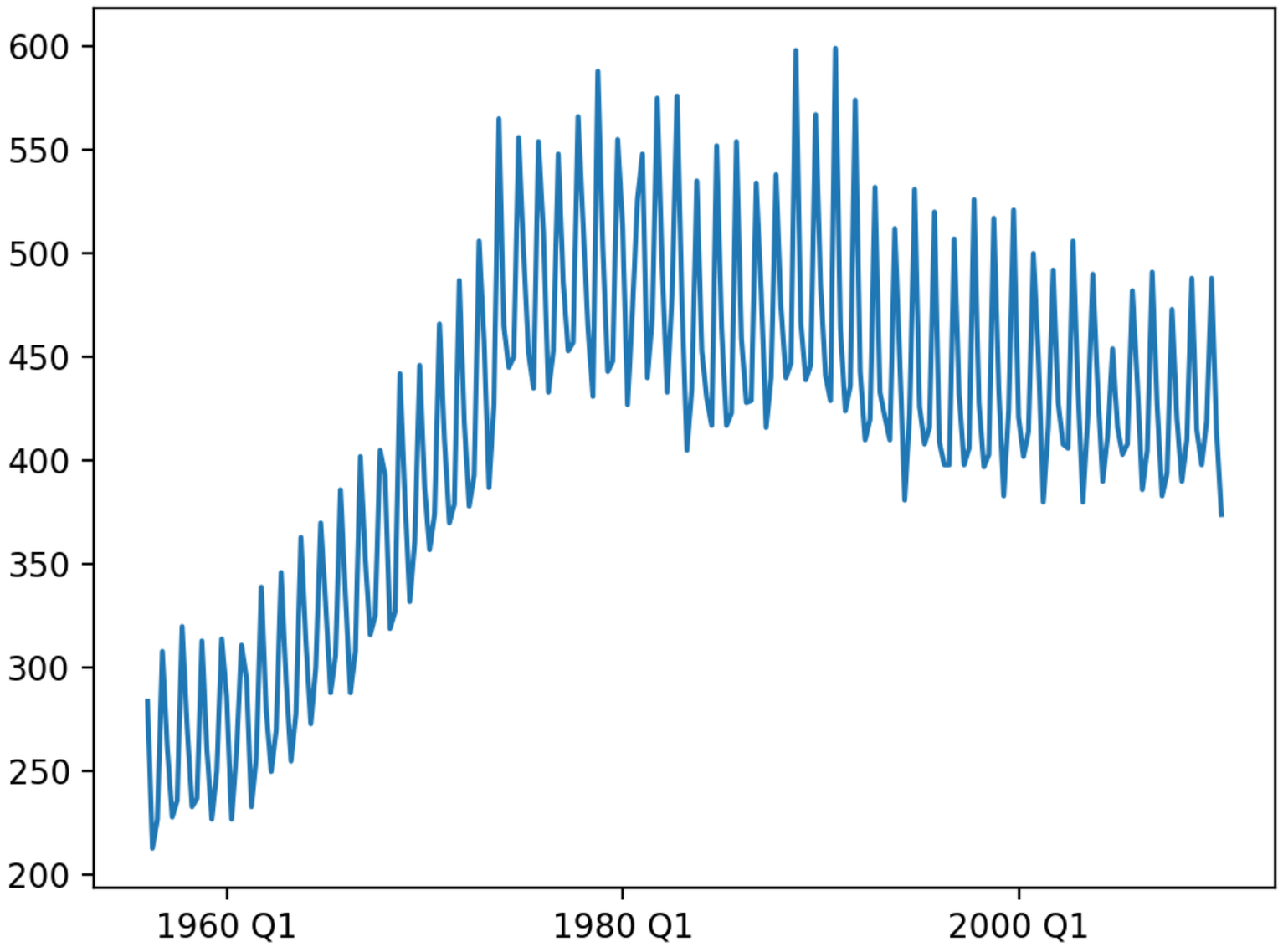
Seasonal Dummy Variables

- Example:
  - Australian beer production (quarterly)



Seasonal Dummy Variables

- Obviously, two different trends, 1955 - 1975 and 1975 -



Seasonal Dummy Variables

- Look at the original data:
  - Need to parse time stamps from two different columns
  - Luckily, parse data is up to it

```
Time,Year,Quarter,Beer.Production
1,1956,Q1,284
2,1956,Q2,213
3,1956,Q3,227
4,1956,Q4,308
5,1957,Q1,262
6,1957,Q2,228
7,1957,Q3,236
8,1957,Q4,320
9,1958,Q1,272
10,1958,Q2,233
```

```
def get_data():
    df_ab = pd.read_csv('AusBeer.csv',
                        sep = ',',
                        parse_dates={'period':['Year', 'Quarter']}
                        )
    df_ab = df_ab.set_index('period')
    return df_ab
```

Seasonal Dummy Variables

- Aside:
  - When we draw the graph, need to select x-ticks

```
def show(df_ab):
    plt.plot(df_ab['Beer.Production'])
    plt.plot(df_ab['pred'])
    plt.xticks(['1960 Q1', '1980 Q1', '2000 Q1'])
    plt.show()
```

Seasonal Dummy Variables

- First, linear regression just on beer production
  - without accounting for the influence of the quarters

```
df_ab = get_data()
df = df_ab.loc['1979 Q1':]
y = df['Beer.Production']
x = df['Time']
model = ols("y~ x",df).fit()
print(model.summary())
```

Seasonal Dummy Variables

- This gives so-so values (as should be expected)

OLS Regression Results

Dep. Variable:	y	R-squared:	0.273
Model:	OLS	Adj. R-squared:	0.270
Method:	Least Squares	F-statistic:	81.30
Date:	Tue, 24 Aug 2021	Prob (F-statistic):	1.06e-16
Time:	17:22:20	Log-Likelihood:	-1244.8
No. Observations:	218	AIC:	2494.
Df Residuals:	216	BIC:	2500.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	337.4062	9.973	33.832	0.000	317.749	357.063
x	0.7120	0.079	9.017	0.000	0.556	0.868

Omnibus:	11.115	Durbin-Watson:	0.855
Prob(Omnibus):	0.004	Jarque-Bera (JB):	11.202
Skew:	0.516	Prob(JB):	0.00369
Kurtosis:	2.589	Cond. No.	253.

Seasonal Dummy Variables

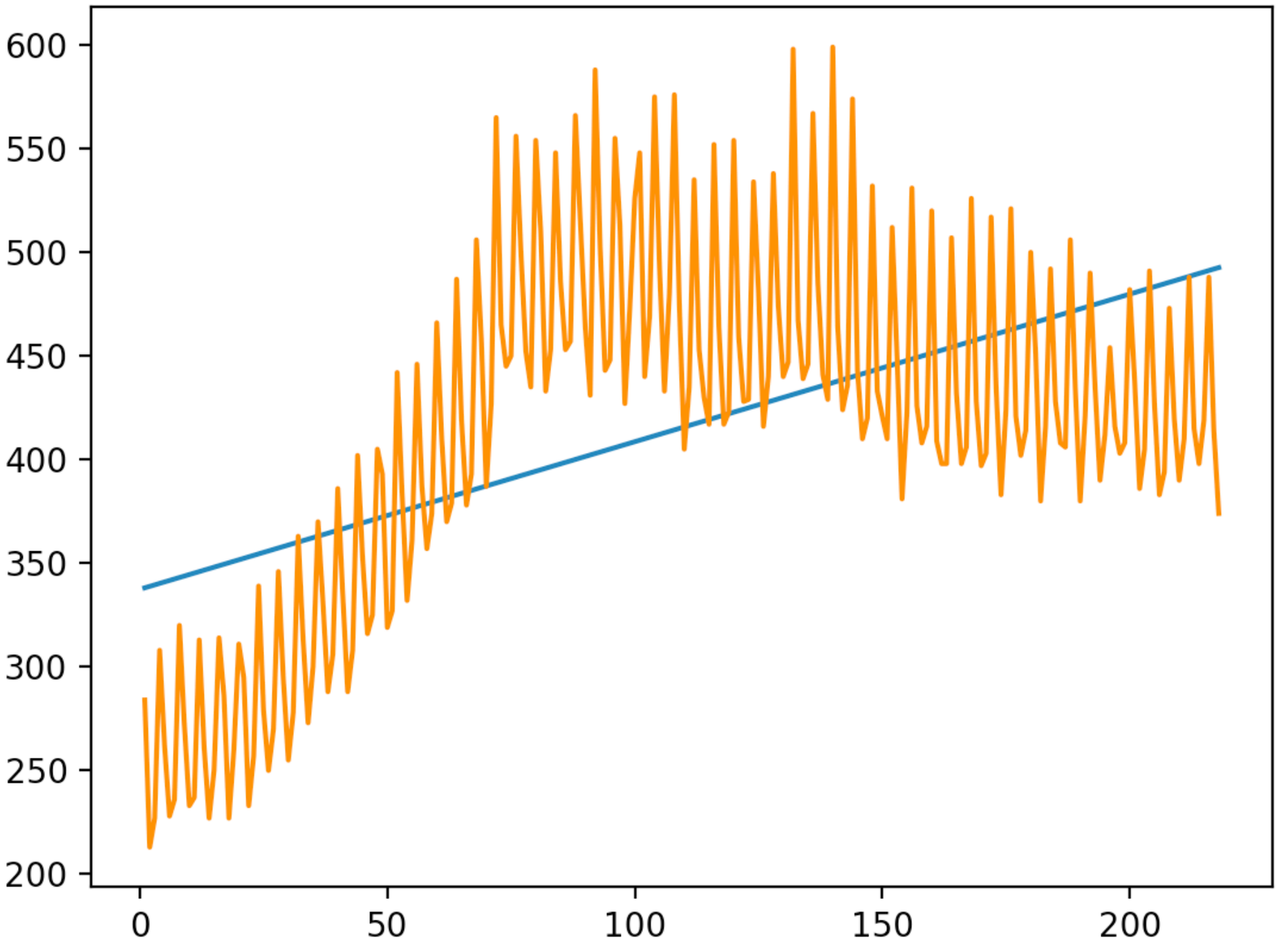
- But already some nice trend
  - Get the model parameters and add a new column with predicted values

```
x = beer['Time']
y = beer['Beer.Production']
model = ols("y~x", beer).fit()
print(model.summary())
intercept, slope = model.params
```

  - Then display

```
beer['without'] = intercept+slope*beer.Time
beer.without.plot()
beer['Beer.Production'].plot()
plt.show()
```

Seasonal Dummy Variables

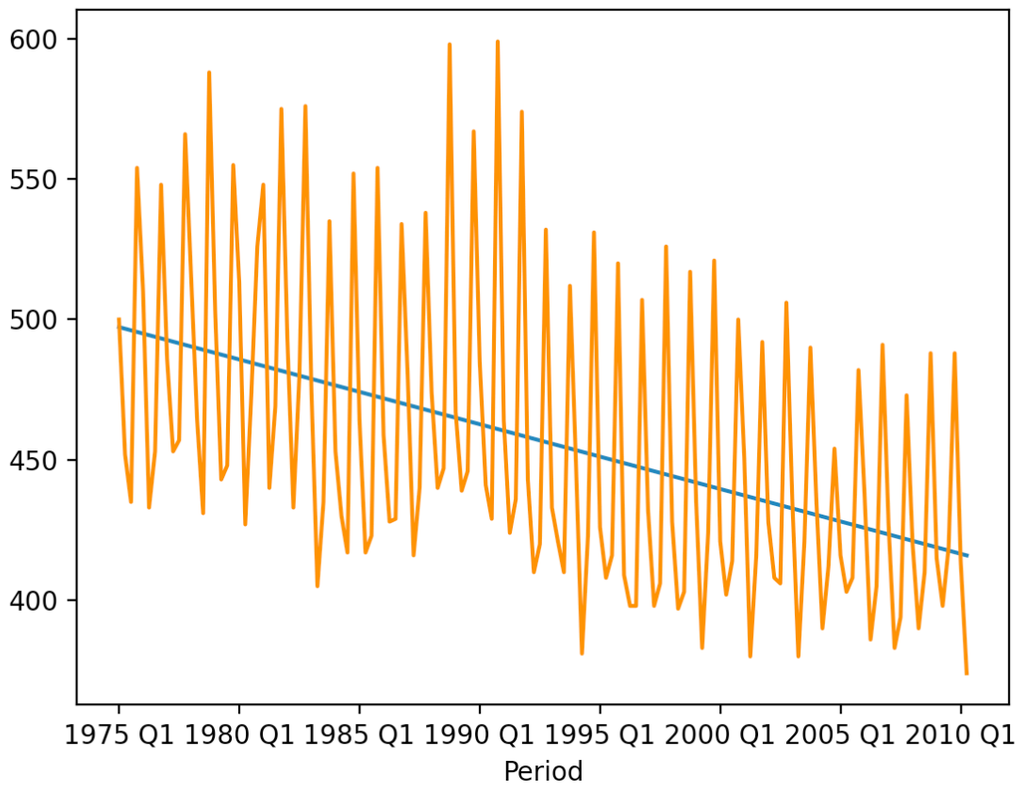


Seasonal Dummy Variable

- Restrict to values after 1975:

```
beer = beer.loc['1975 Q1':]
```

- Lowers R^2 to 0.191



Seasonal Dummy Variables

- Now, let's try using seasonal dummy variables
  - Create four new columns

```
def transform(df_ab):
    df_ab['s1'] = [ 1 if 'Q1' in df_ab.index[i]
           else 0 for i in range(len(df_ab.index))]
    df_ab['s2'] = [ 1 if 'Q2' in df_ab.index[i]
           else 0 for i in range(len(df_ab.index))]
    df_ab['s3'] = [ 1 if 'Q3' in df_ab.index[i]
           else 0 for i in range(len(df_ab.index))]
    df_ab['s4'] = [ 1 if 'Q4' in df_ab.index[i]
           else 0 for i in range(len(df_ab.index))]
```

Seasonal Dummy Variables

- Set up the model including the seasonal parameters

```
y = df['Beer.Production']
x = df['Time']
x1 = df['s1']
x2 = df['s2']
x3 = df['s3']
x4 = df['s4']
model = ols("y~ x",df).fit()
print(model.summary())
a, b, b1, b2, b3, b4 = model.params
print(a, b, b1, b2, b3, b4)
```

Seasonal Dummy Variables

OLS Regression Results

Dep. Variable:	y	R-squared:	0.891
Model:	OLS	Adj. R-squared:	0.888
Method:	Least Squares	F-statistic:	280.2
Date:	Tue, 24 Aug 2021	Prob (F-statistic):	6.77e-65
Time:	17:35:02	Log-Likelihood:	-610.68
No. Observations:	142	AIC:	1231.
Df Residuals:	137	BIC:	1246.
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	434.1185	4.555	95.303	0.000	425.111	443.126
x	-0.5811	0.037	-15.623	0.000	-0.655	-0.508
s1	108.1018	2.855	37.864	0.000	102.456	113.747
s2	65.0717	2.869	22.679	0.000	59.398	70.746
s3	78.5248	2.882	27.243	0.000	72.825	84.225
s4	182.4202	2.897	62.979	0.000	176.692	188.148

Omnibus:	19.732	Durbin-Watson:	1.947
Prob(Omnibus):	0.000	Jarque-Bera (JB):	32.379
Skew:	0.683	Prob(JB):	9.31e-08
Kurtosis:	4.899	Cond. No.	2.63e+18

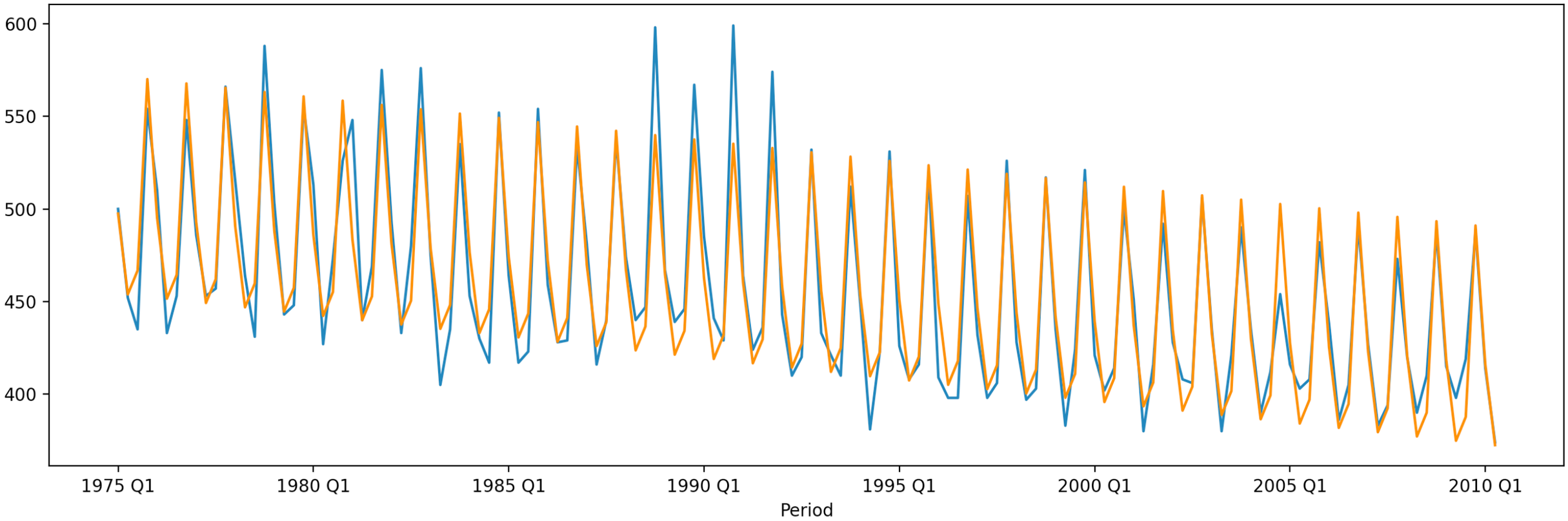
Seasonal Dummy Variables

- Add a new column for the prediction
  - `df_ab['pred'] = a + b*df.Time +b1*df.s1+b2*df.s2+b3*df.s3+b4*df.s4`
    `df_ab.dropna(inplace=True)`
    - And plot raw data and prediction values

```
plt.plot(df_ab['Beer.Production'])
plt.plot(df_ab['pred'], alpha = 0.5)

plt.xticks(['1980 Q1', '1985 Q1','1990 Q1',
          '1995 Q1', '2000 Q1', '2005 Q1', '2010 Q1'],
           ['80', '85', '90', '95', '00', '05', '10'])
plt.show()
```

Seasonal Dummy Variables



Seasonal Dummy Variables

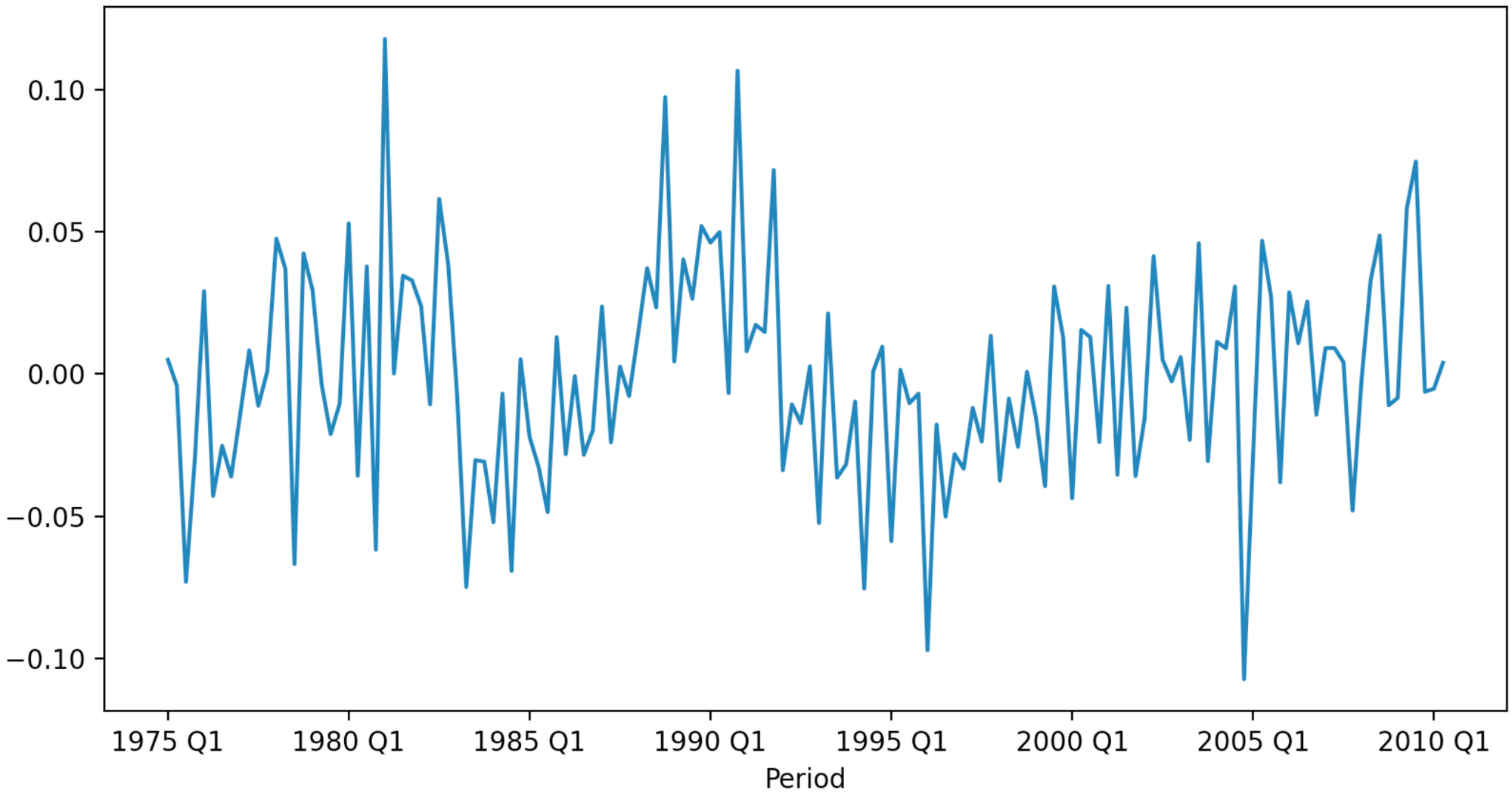
- Why are we not doing better?
  - We only have time and season as explanatory variables
    - Temperature and economy could also explain beer consumption
  - And maybe exports?

Seasonal Dummy Variables

- Let's look at the average error of the prediction
  - Add one more column to the data frame
    - ```
      df_ab['res'] =
      (df_ab['Beer.Production']-
      df_ab['pred'])/df_ab['Beer.Production']
      ```
  - And display
      ```

Seasonal Dummy Variables

- Shows that we are historically with 10% of the linear regression calculated value

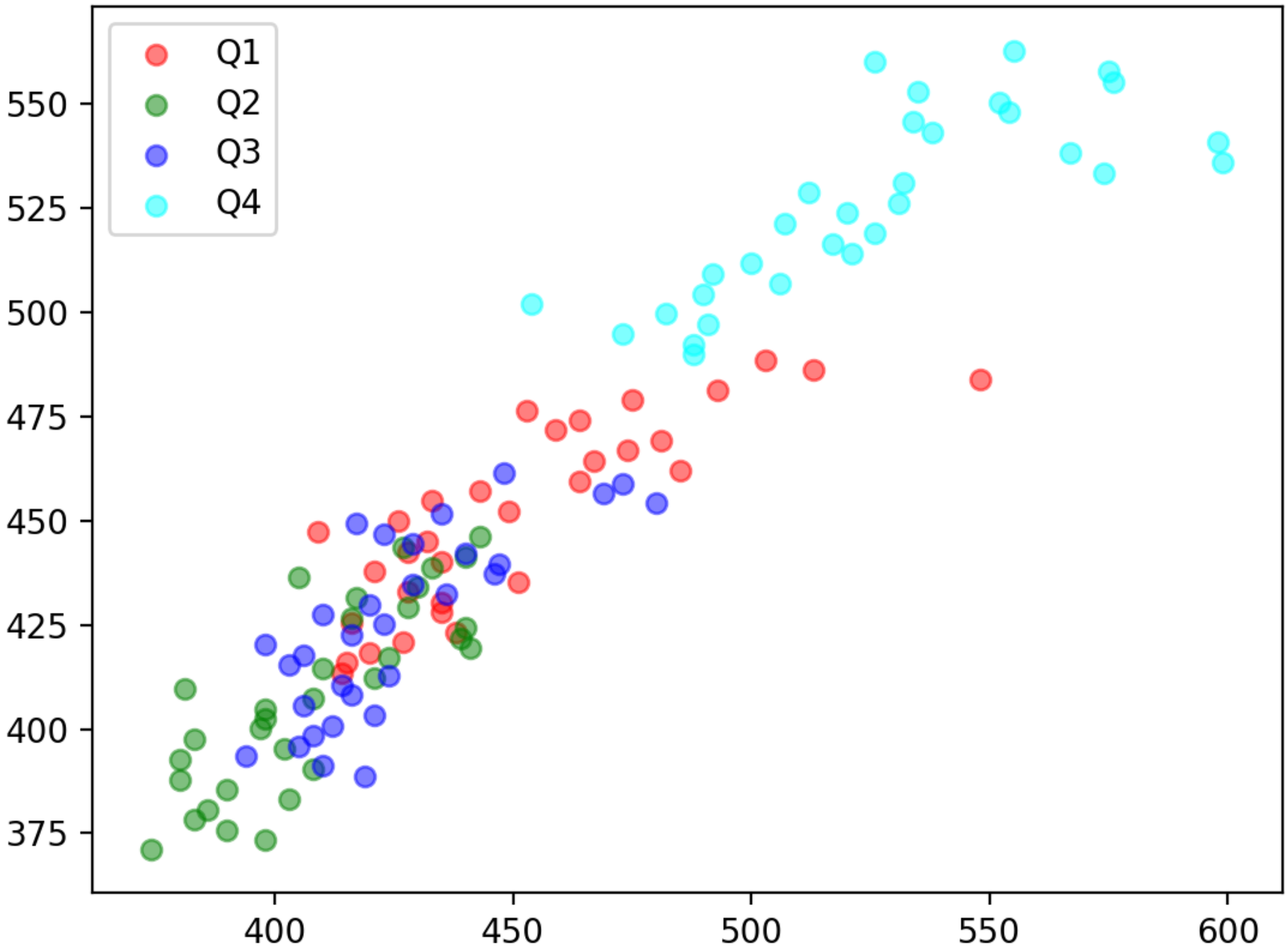


Seasonal Dummy Variables

- We can also look at how well the prediction works

```
plt.scatter(df_ab['Beer.Production'][df_ab['s1']==1],
            df_ab['pred'][df_ab['s1']==1],
            alpha = 0.5, c = 'red', label='Q1')
plt.scatter(df_ab['Beer.Production'][df_ab['s2']==1],
            df_ab['pred'][df_ab['s2']==1],
            alpha = 0.5, c = 'green', label='Q2')
plt.scatter(df_ab['Beer.Production'][df_ab['s3']==1],
            df_ab['pred'][df_ab['s3']==1],
            alpha = 0.5, c = 'blue', label='Q3')
plt.scatter(df_ab['Beer.Production'][df_ab['s4']==1],
            df_ab['pred'][df_ab['s4']==1],
            alpha = 0.5, c = 'cyan', label='Q4')
plt.legend()
plt.show()
```

Seasonal Dummy Variables



Classical Decomposition

Classical Decomposition

- Simple method with periods of *m* on y
  - quarters: *m*=4, business weeks: *m*=5, years: *m* = 365
  - Additive decomposition: $y = T + S + R$
    - Use moving average of size *2m* (or *m*) to estimate T
    - Calculate the "*detrended*" series $d_t = y_t - T_t$
    - Seasonal component is the average over all values for that period
      $S_t = \text{Average}\left(\{\ldots, d_{t-2m}, d_{t-m}, d_t, d_{t+m}, d_{t+2m}, \ldots\}\right)$
    - Residual is what is left

Classical Decomposition

- Problems with classical decomposition:
  - Trend cycle not available for the first or last times without extrapolation
  - Rapid rises and falls are smoothed out too much
  - Assumes seasonal components repeat
    - Counter-example:
      - Used to be that electricity consumption peaked in the winter (heating)
      - Now peaks in the summer (air-conditioning)
  - Really difficult to deal with extra-ordinary events (strikes, weather, catastrophes, …)

Classical Decomposition

- Implemented in statsmodels

```
from statsmodels.tsa.seasonal import seasonal_decompose
```

- Needs a time series `x`
- Needs a model: "additive" (default), "multiplicative"
- Can give filtering weights
- Period (if `x` not Pandas with frequency)
- two-sided: method used for filtering
- extrapolate_trend: non-zero value or 'freq' to extrapolate
  - Otherwise, NaN values

Classical Decomposition

- Call decomposition

```
result = seasonal_decompose(
                df['Beer.Production'],
                model = 'additive',
                period = 4,
                extrapolate_trend = 3)
```

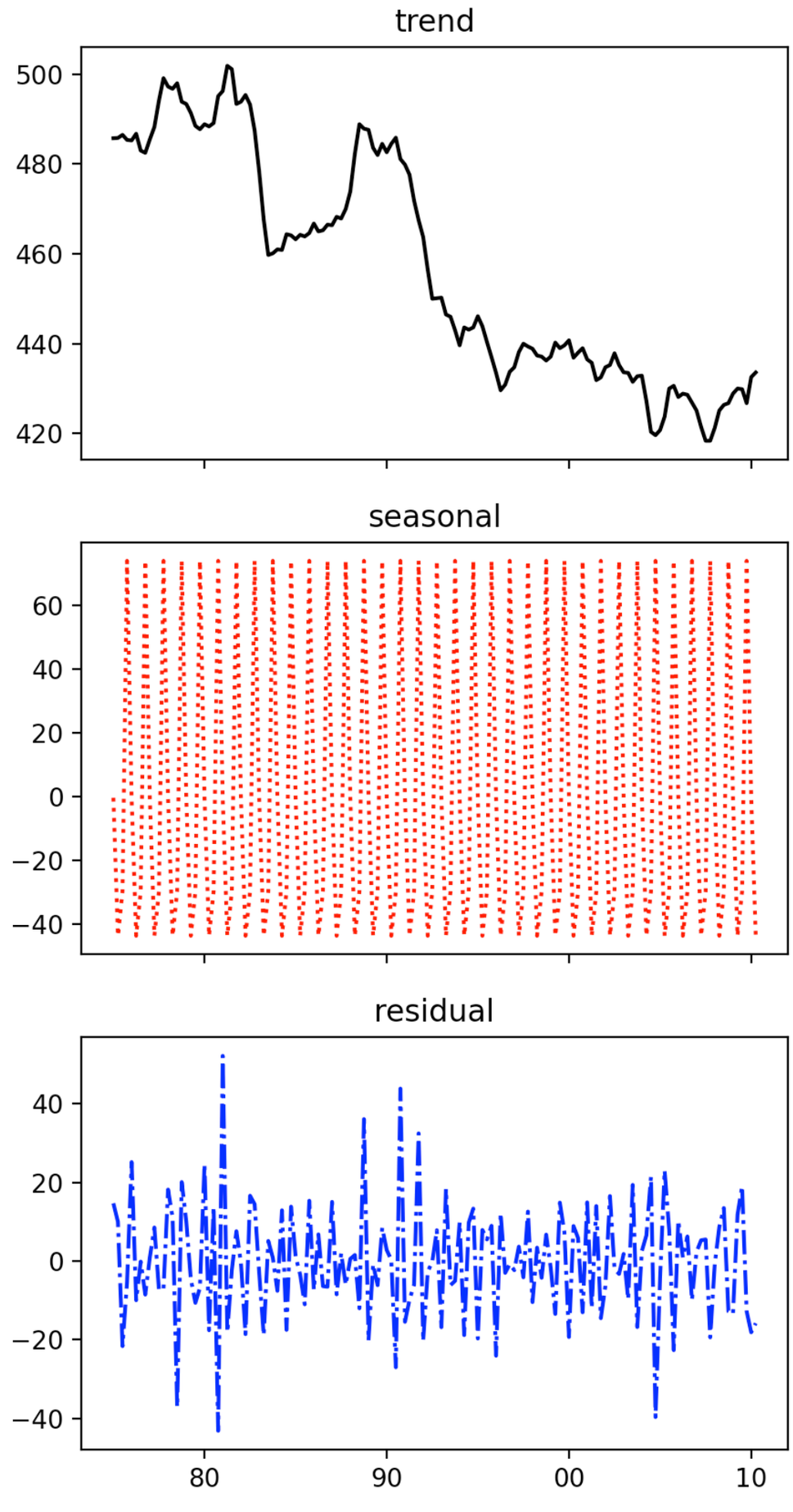
- result has components `trend, seasonal, resid`

Classical Decomposition

- Show results:

```
def show(result):
    fig, axs = plt.subplots(3, sharex=True, figsize=(5,10))
    axs[0].plot(result.trend, 'k-')
    axs[1].plot(result.seasonal, 'r:')
    axs[2].plot(result.resid, 'b-.')
    axs[0].set_title('trend')
    axs[1].set_title('seasonal')
    axs[2].set_title('residual')
    axs[2].set_xticks(['1980 Q1', '1990 Q1', '2000 Q1', '2010 Q1'])
    axs[2].set_xticklabels(['80', '90', '00', '10'])
    plt.show()
```

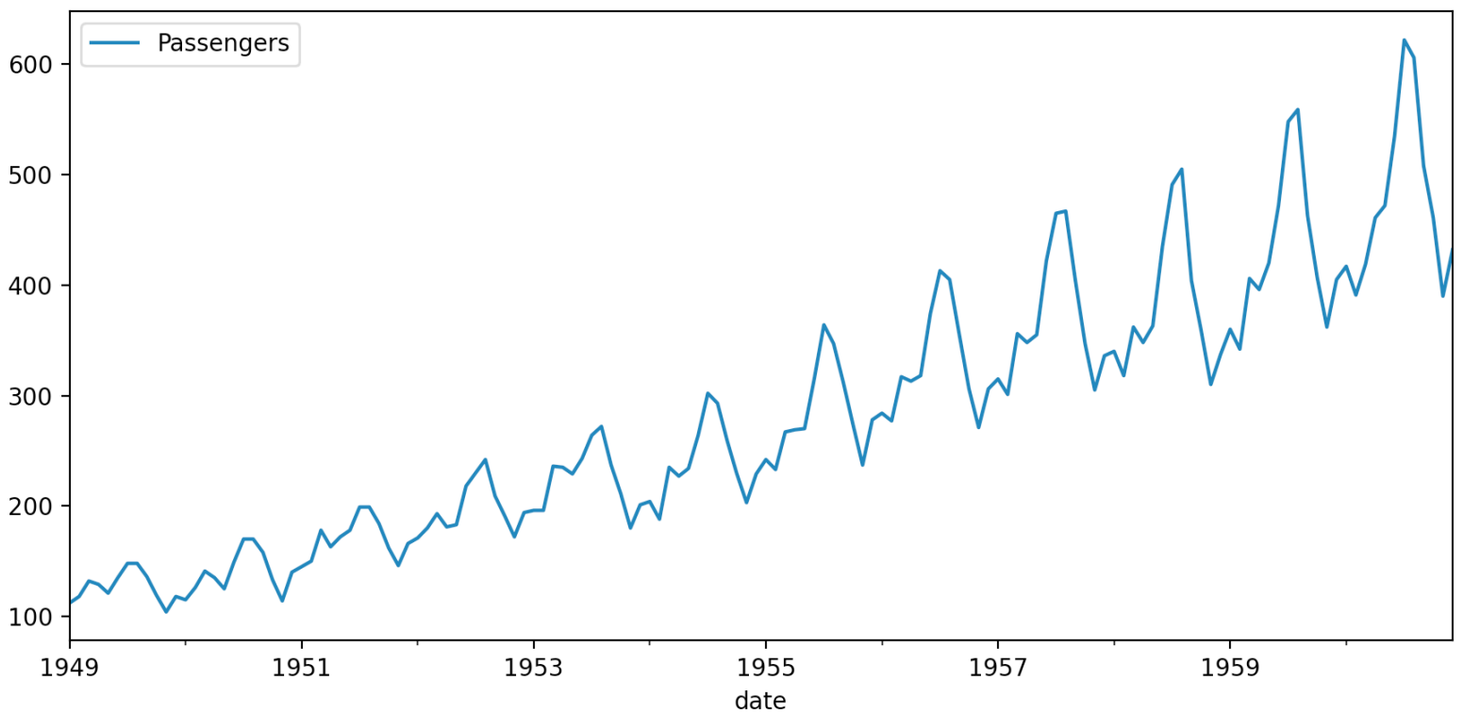
- Can extrapolate the trend
- Add seasonal
- Use residual as a measure of uncertainty

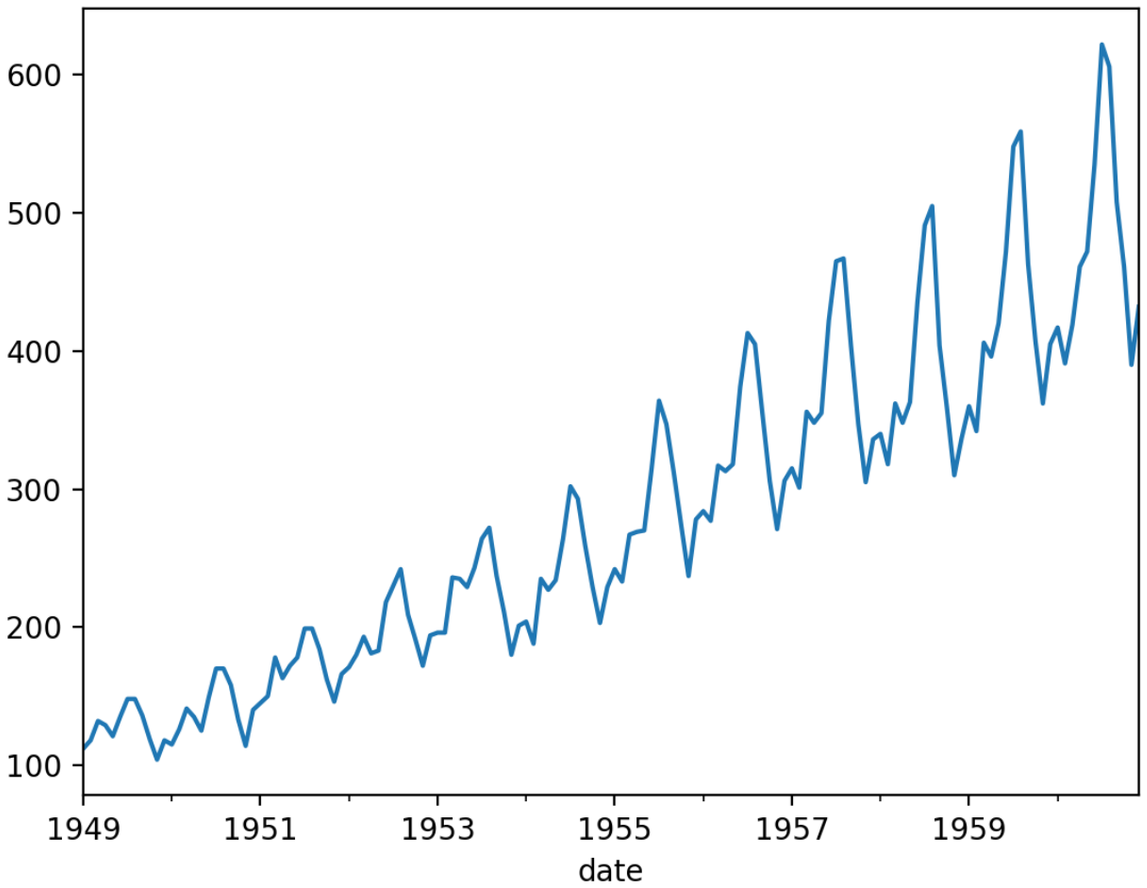


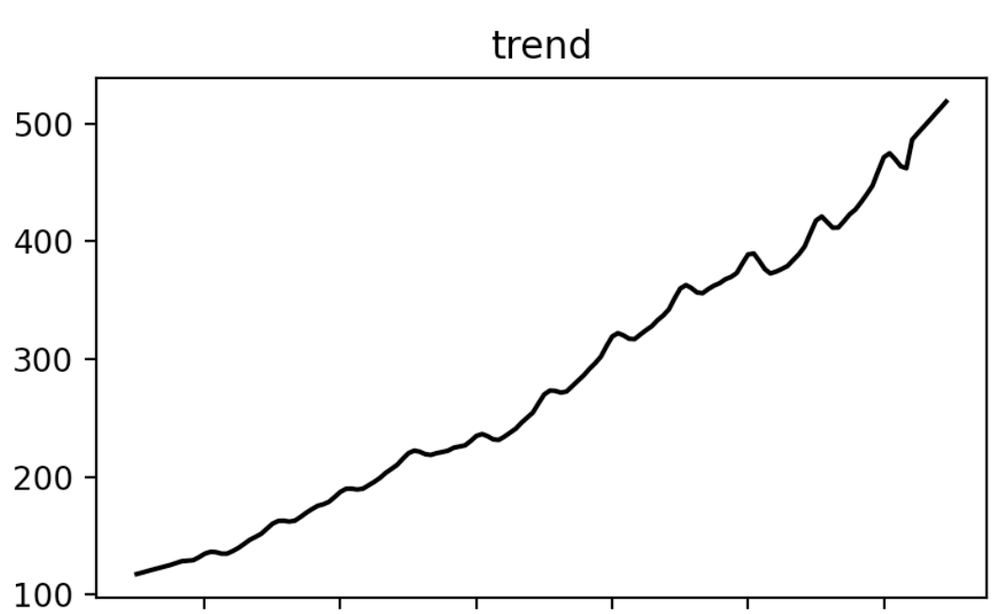
Classical Decomposition

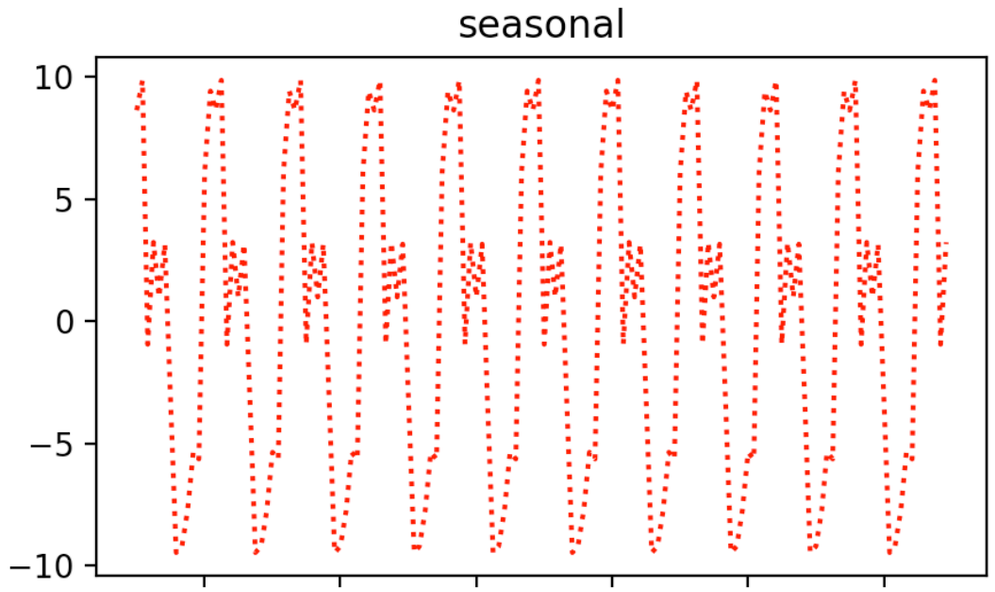
- airline passengers

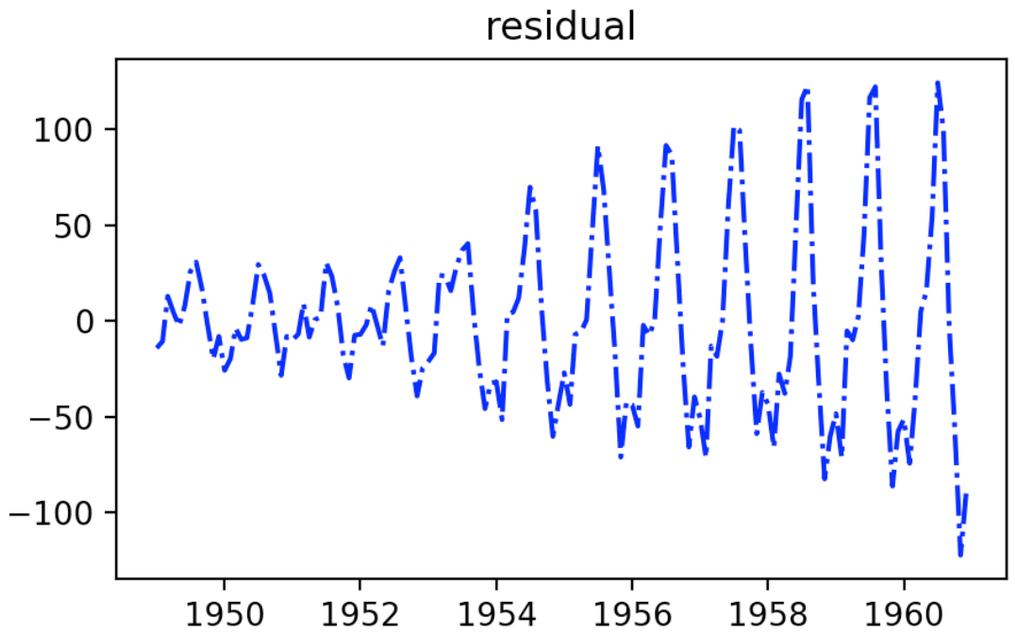
```
result = seasonal_decompose(df.Passengers,
                            model = 'additive',
                            period = 14,
                            extrapolate_trend = 'freq'
                            )
```




600
500
400
300
200
100
1949
1951
1953
1955
1957
1959
date


trend
500
400
300
200
100


seasonal
10
5
0
−5
−10


residual
100
50
0
−50
−100
1950
1952
1954
1956
1958
1960

Better Decompositions

- Decomposition has had a 100 year history
  - Classical decomposition finds seasonal adjustment based on means so that they are constant
  - Better decompositions allow seasonal values to vary
  - **S**easonal **D**ecomposition using **L**OESS (STL)
    - LOESS is based on estimating the trend with a range of functions within a certain window

Better Decompositions

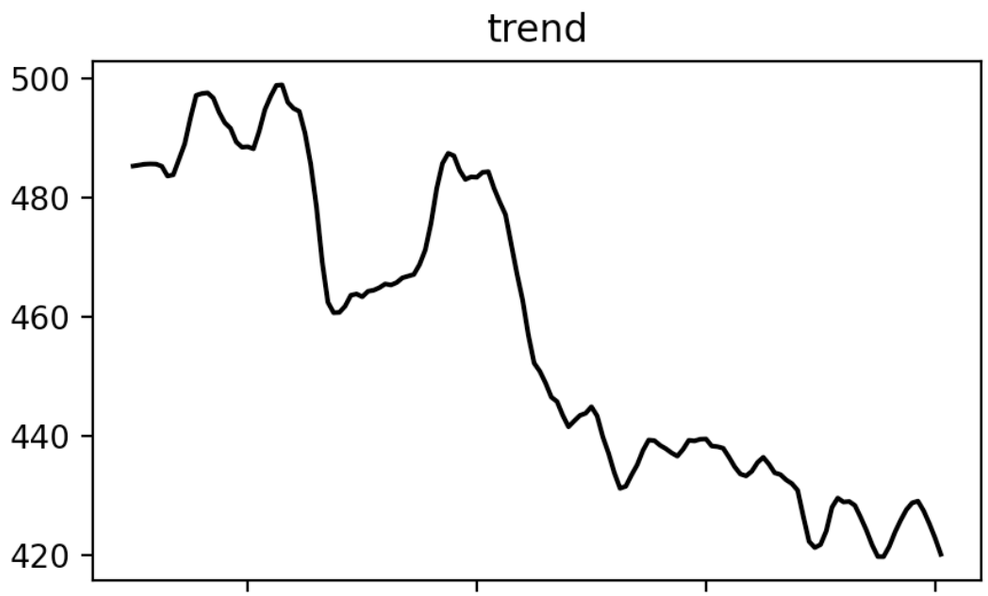
- STL is implemented in statsmodels
  - `from statsmodels.tsa.seasonal import STL`
  - Uses the assumed cyclicity as input (periods)

Better Decompositions

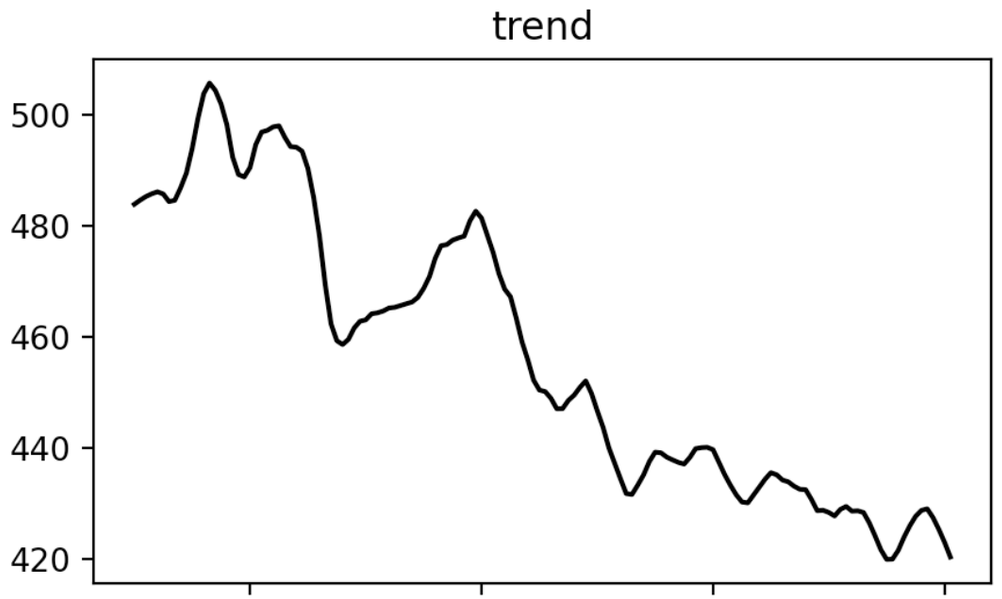
- Example: Australian Beer:

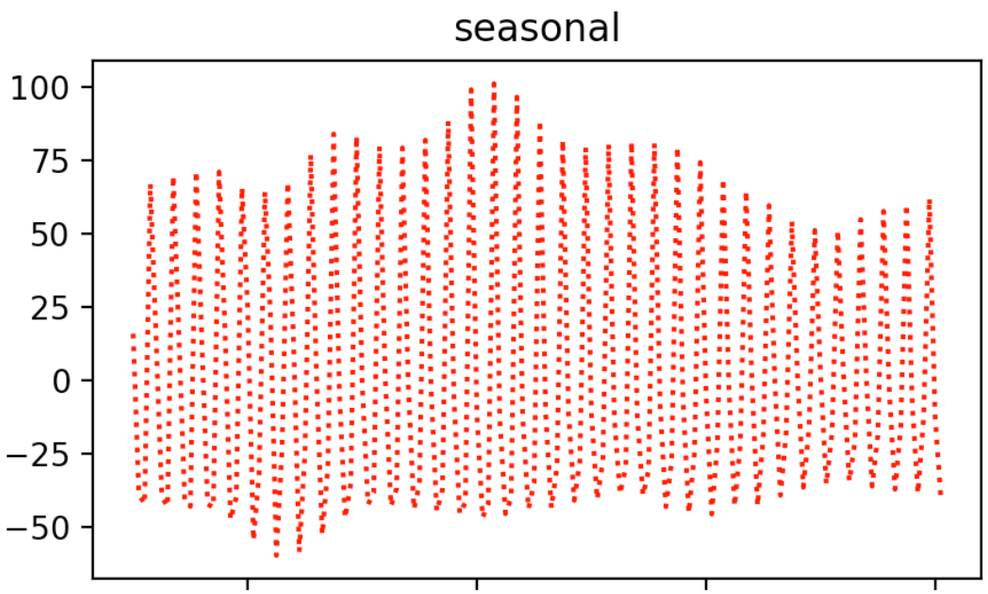
```
stl = STL(df['Beer.Production'],
          period=4,
          robust=False)
result = stl.fit()
```

- Result is a triple of trend, seasonal, and residual

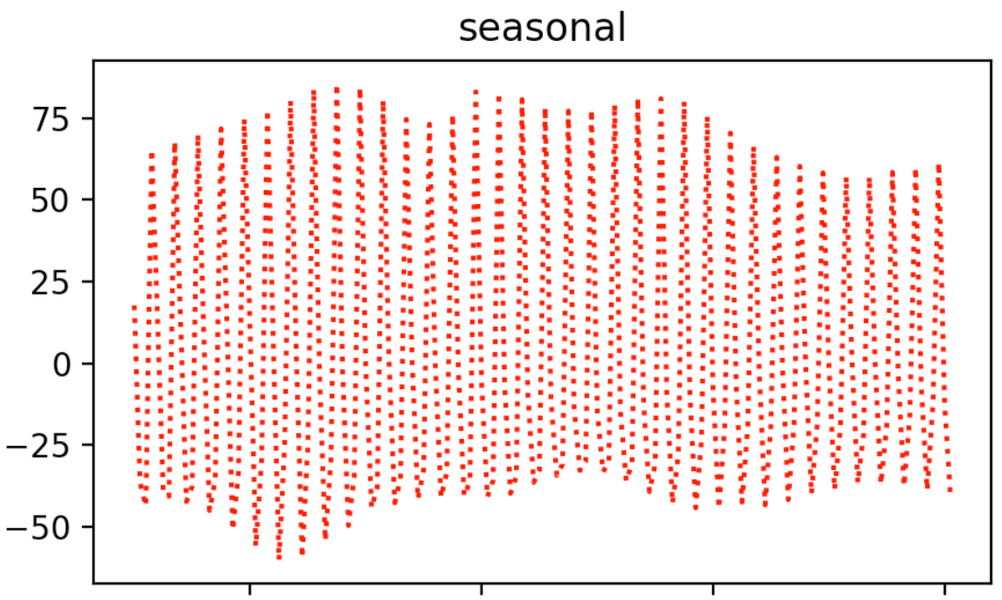


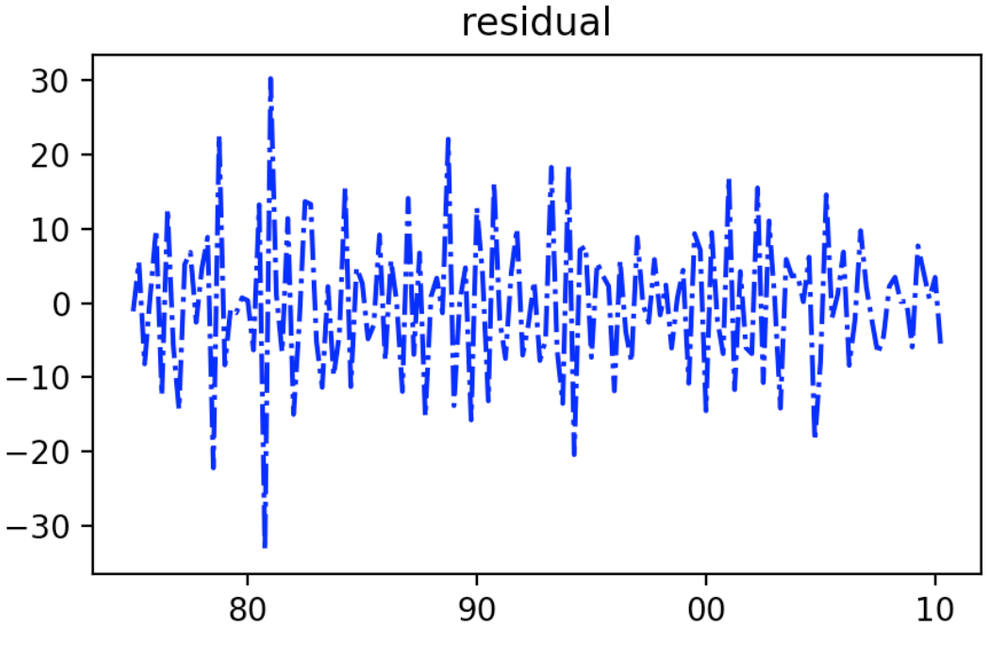
robust

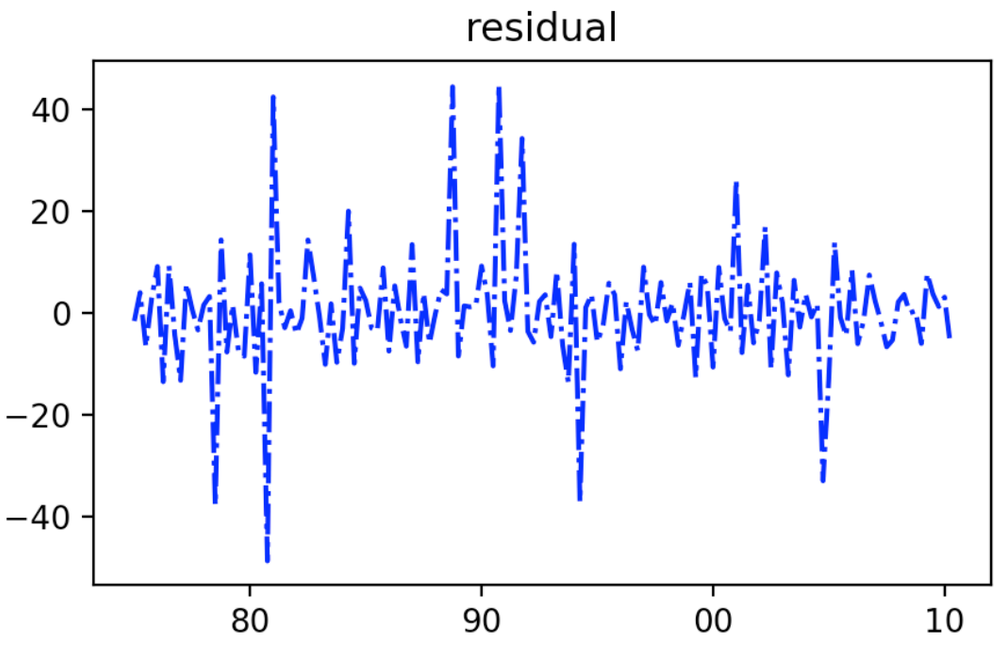


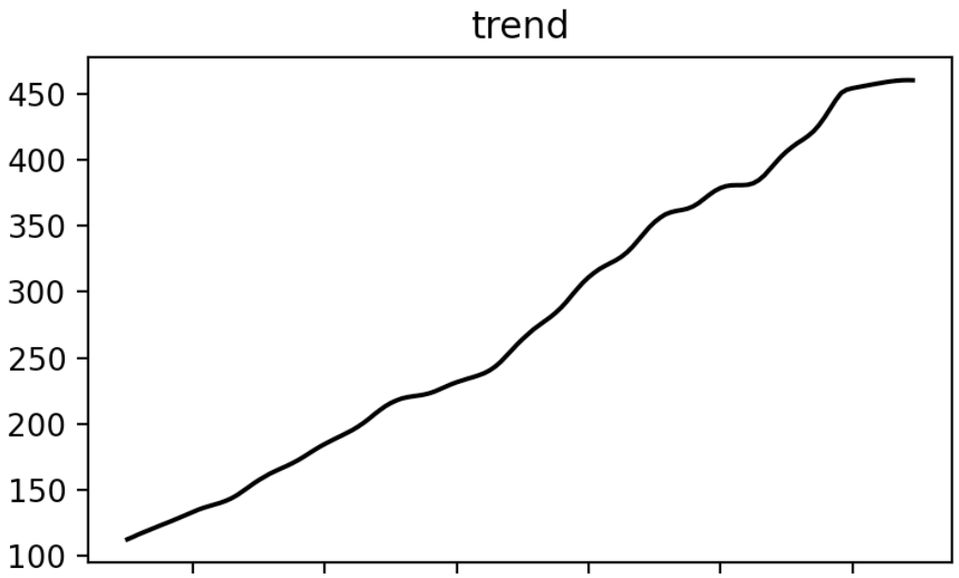


not
robust

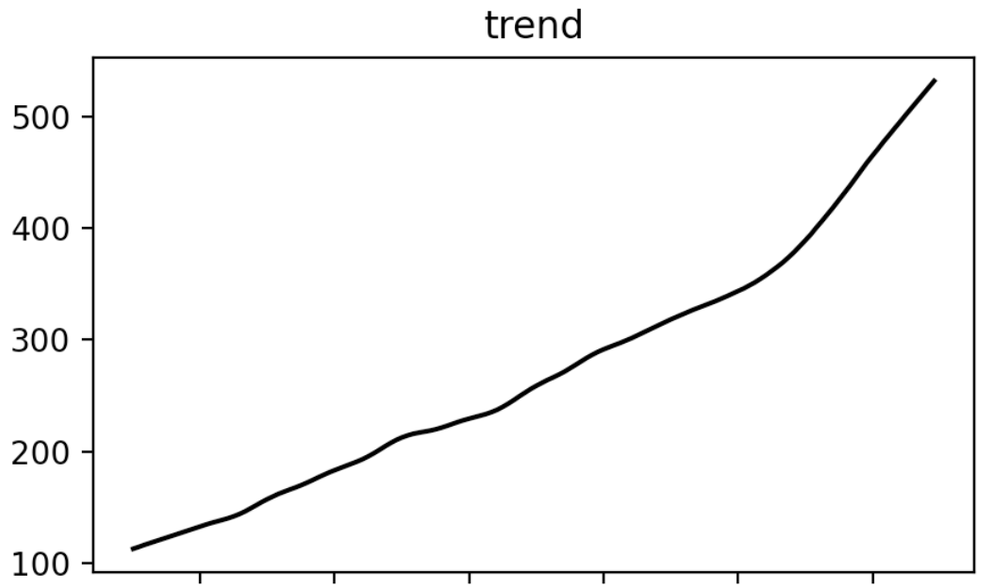


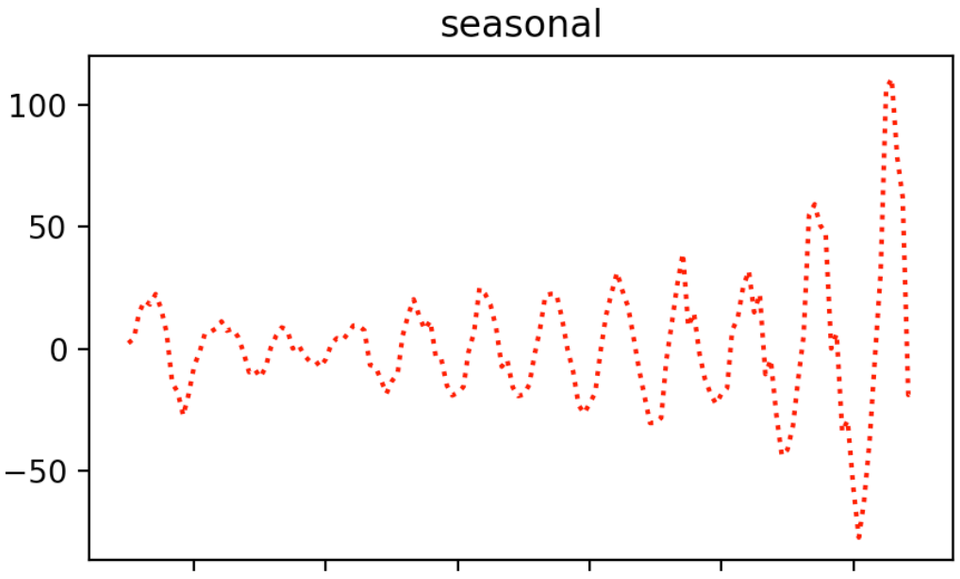




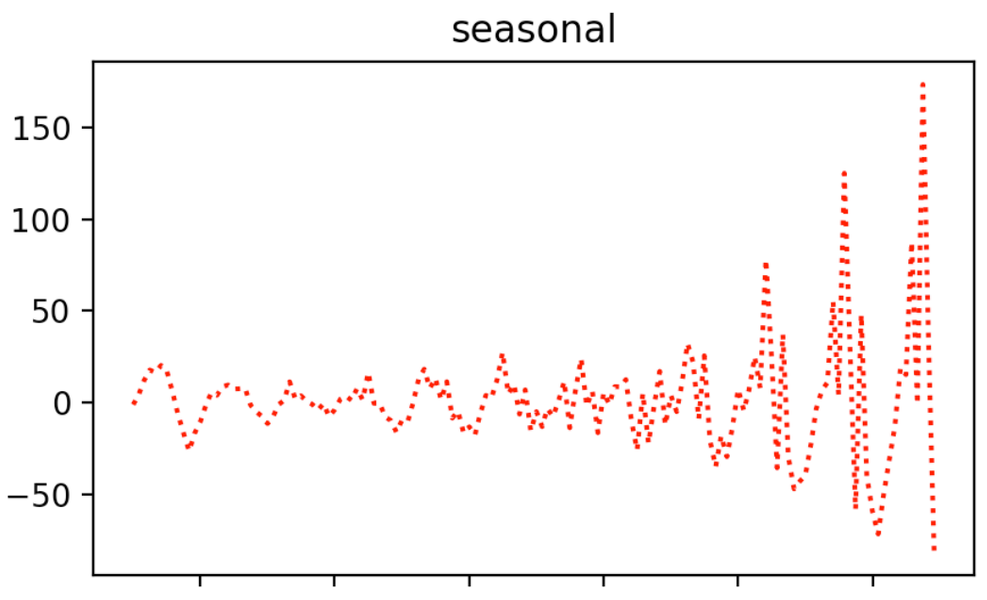


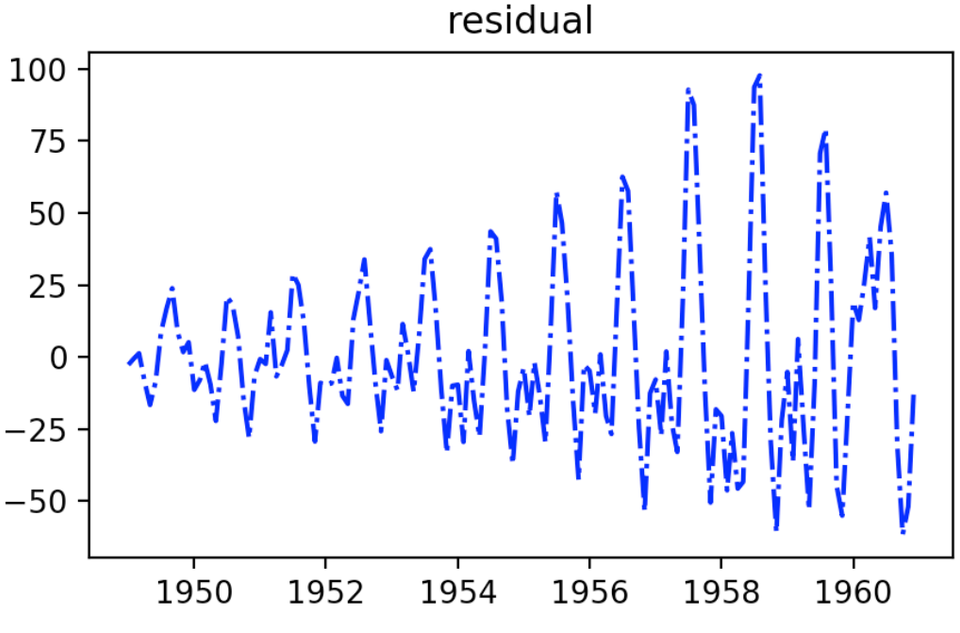
airline passengers



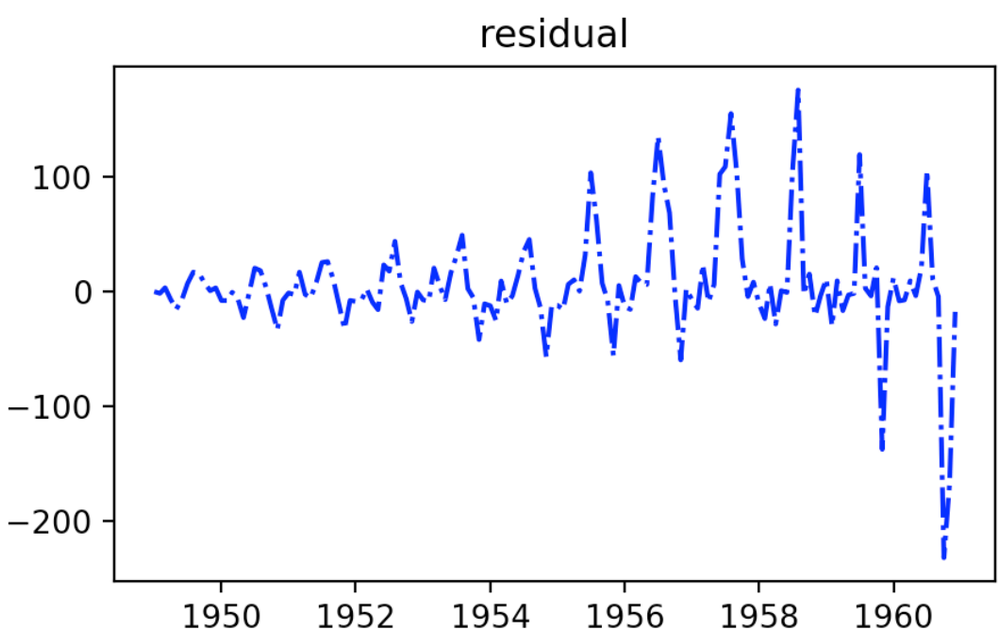


not robust





robust



Better Decompositions

- Holt's linear trend method:
  - Simple exponential smoothing for data with trend
    - Estimates series at time t using estimates of the slope obtained as a weighted average
- Holt-Winter's Seasonal Method
  - Adds a seasonal component

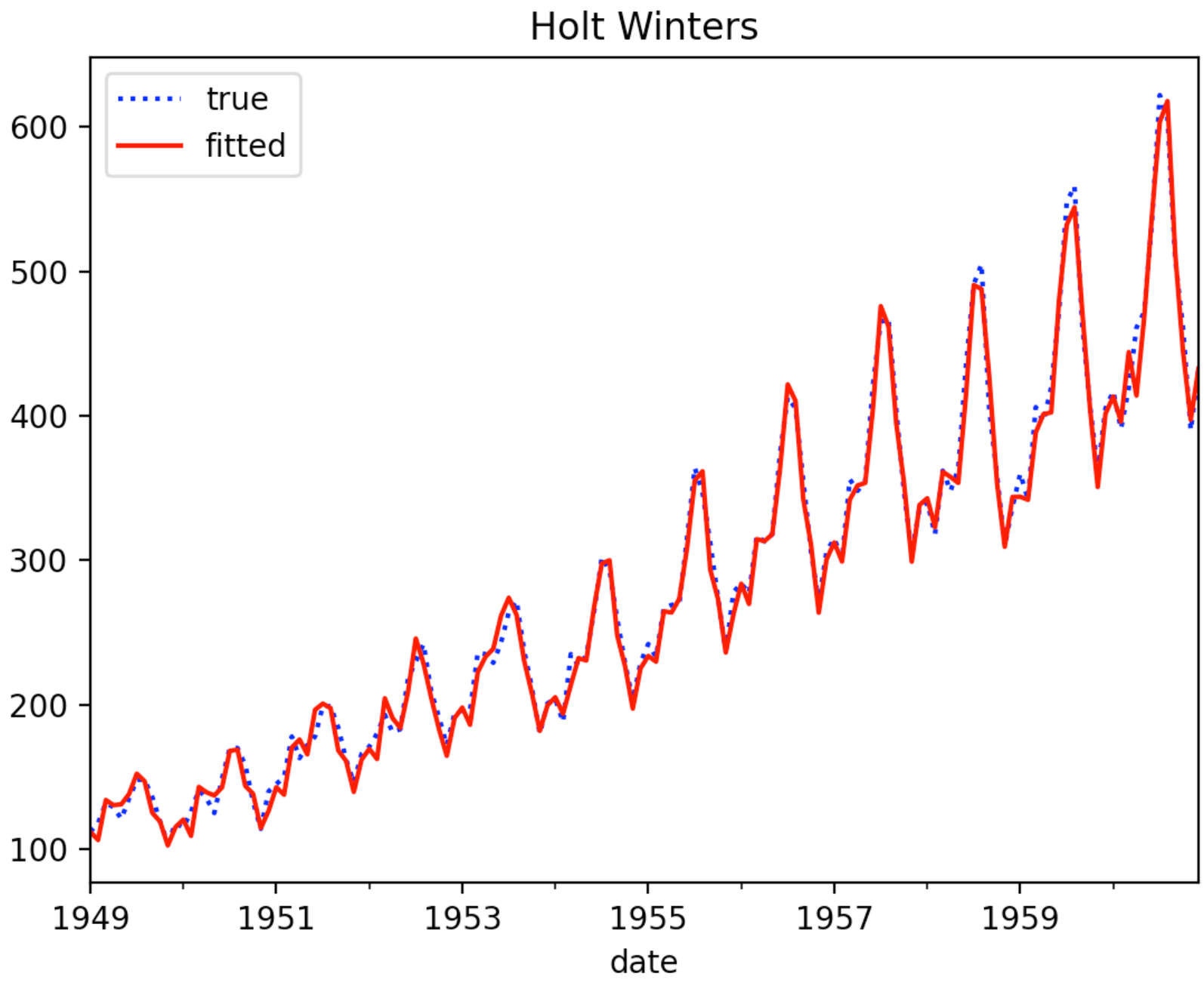
Better Decompositions

- Implemented in statsmodels
  - `from statsmodels.tsa.holtwinters import ExponentialSmoothing as esm`
  - Produces a fitted value (fit) and allows predictions

```
df = get_data()
esm = esm(df['Passengers'],
          seasonal='mul',
          seasonal_periods=12
          )
result = esm.fit()
```

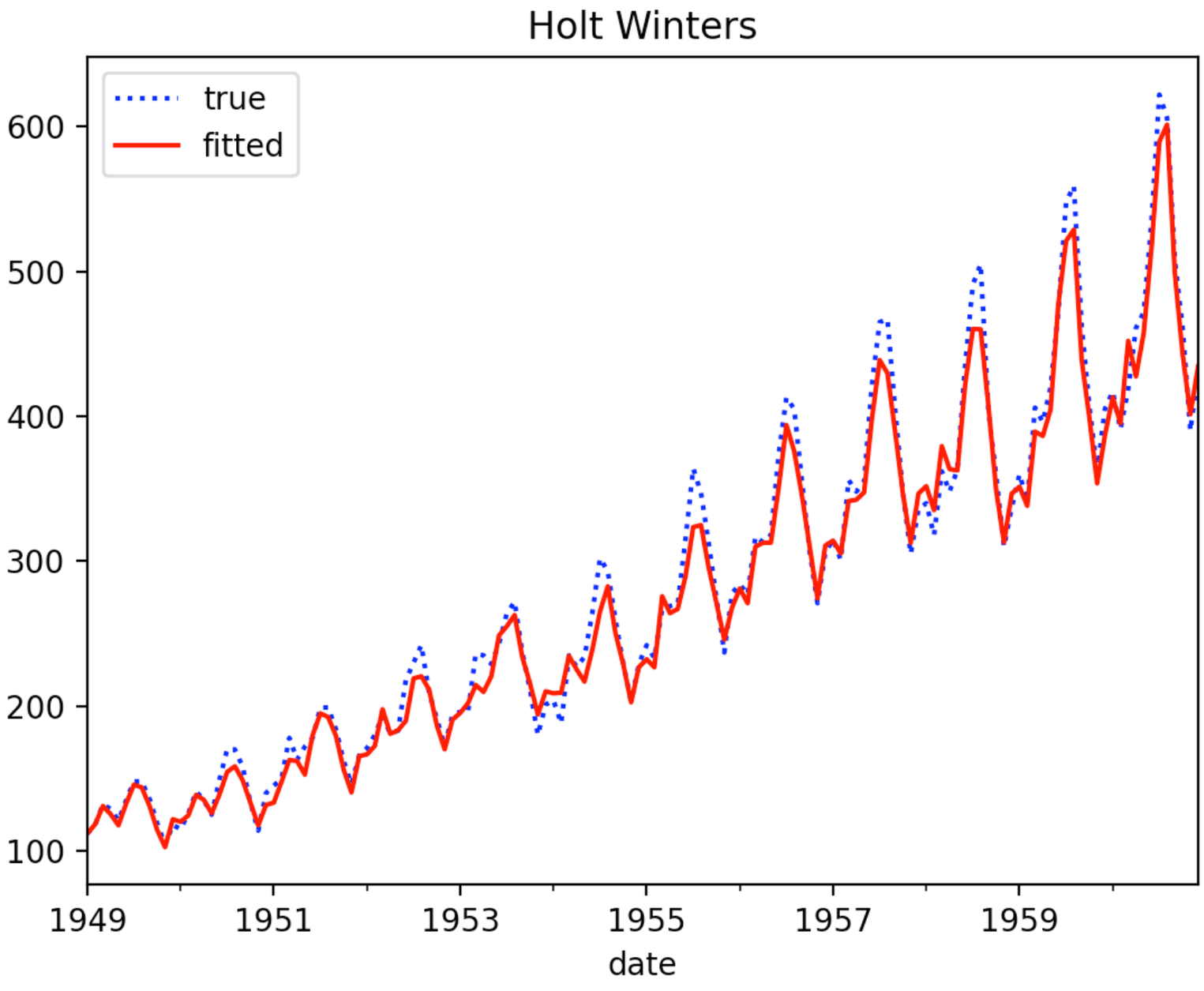
Better Decompositions

- Airline Example (multiplicative fit)



Better Decompositions

- Airline Example additive seasonality



Stationary Time Series

Time Series Analysis

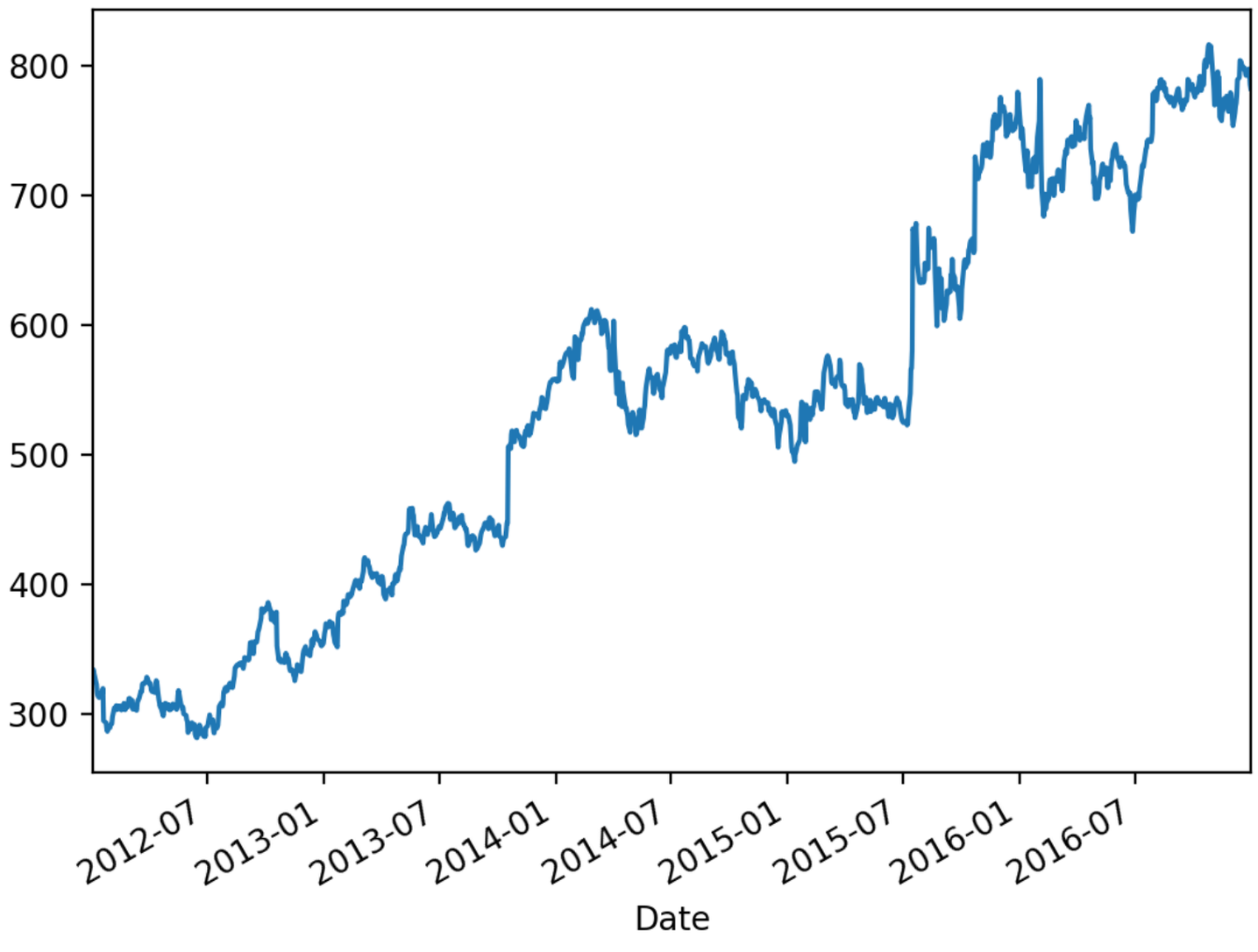
- After defining and removing
  - Seasonal components
  - Trends
- Residual needs to be analyzed

Stationary Time Series

- Properties do not depend on the time at which the series is observed
  - No trend, no seasonality
  - But could be cyclic if cycles have no fixed length

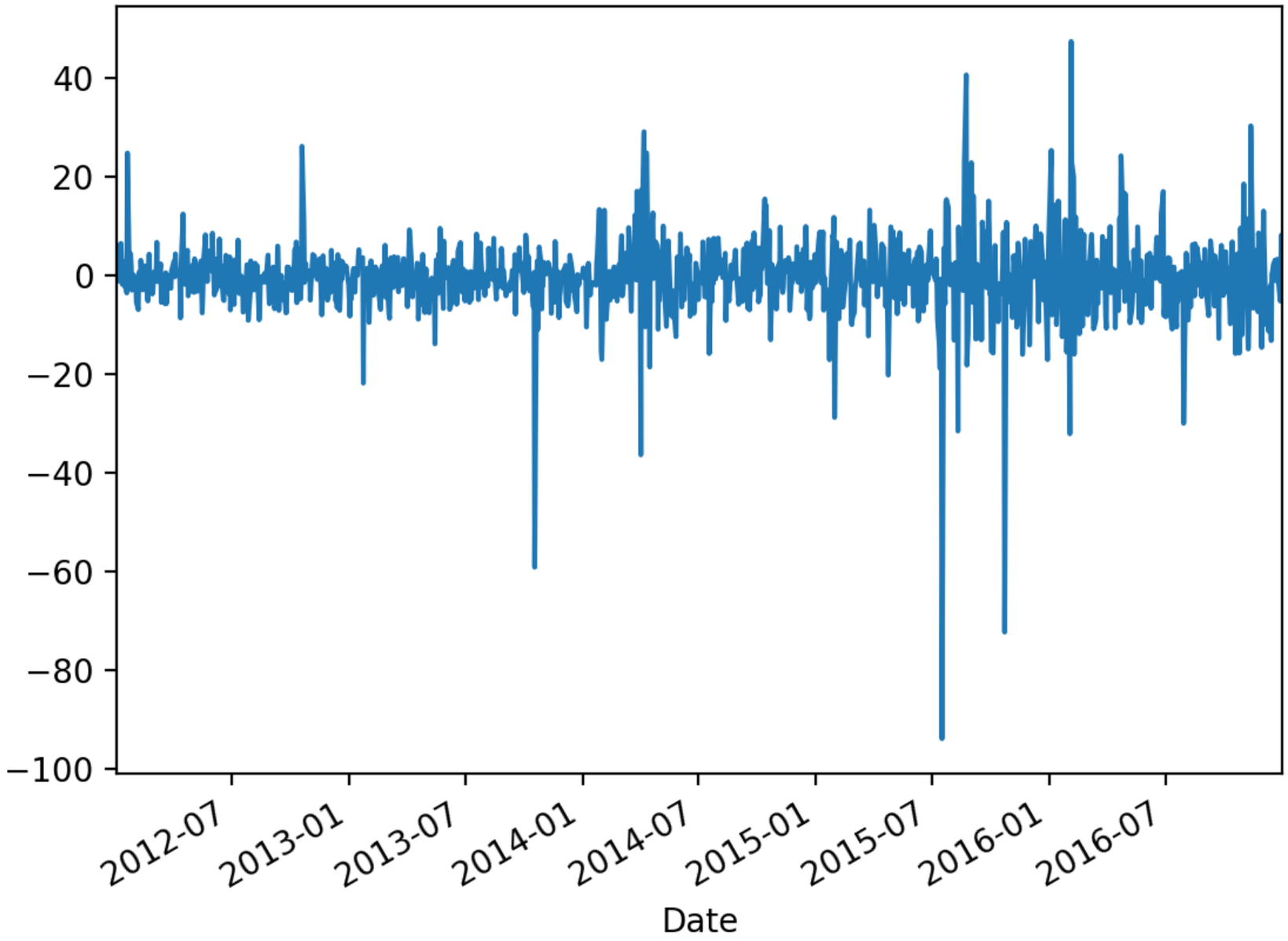
Stationary Time Series

- Example:
  - Google High stock value is not stationary



Stationary Time Series

- But it's daily change is



Stationary Time Series

- Use shift in order to obtain the differences

```
def get_google_data():
    my_df = pd.read_csv('../Pandas/google.csv',
                        parse_dates=[0],
                        index_col=0,
                        converters={
                            'Close': my_converter,
                            'Volume': convert_volume}
                        )
    print(my_df.info())
    #my_df.High.plot()
    (my_df.shift(1).High - my_df.High).plot()
    plt.show()
    return my_df
```

Stationary Time Series

- This is typical
  - **Differencing**
    - Make a non-stationary time series stationary
    - Might have to be repeated several times

Stationary Time Series

- Can look at the Auto-Correlation Function
  - How does the value of a time series relate to the values shifted by *m* periods
  - Implemented in Pandas as `autocorr(lag = m)`

Stationary Time Series

- Example: google change

```
my_df = get_google_data()
for i in range(20):
    print(i,
(my_df.shift(1).High - my_df.High).autocorr(lag=i))
```

```
0 0.9999999999999999
1 0.12080309541084427
2 -0.035334997385017435
3 -0.048524948019570004
4 -0.055606938788878337
5 -0.006479704665710260 5
6 0.011124537759089287
7 -0.039016502314726 03
8 -0.012520771888603816
```

Stationary Time Series

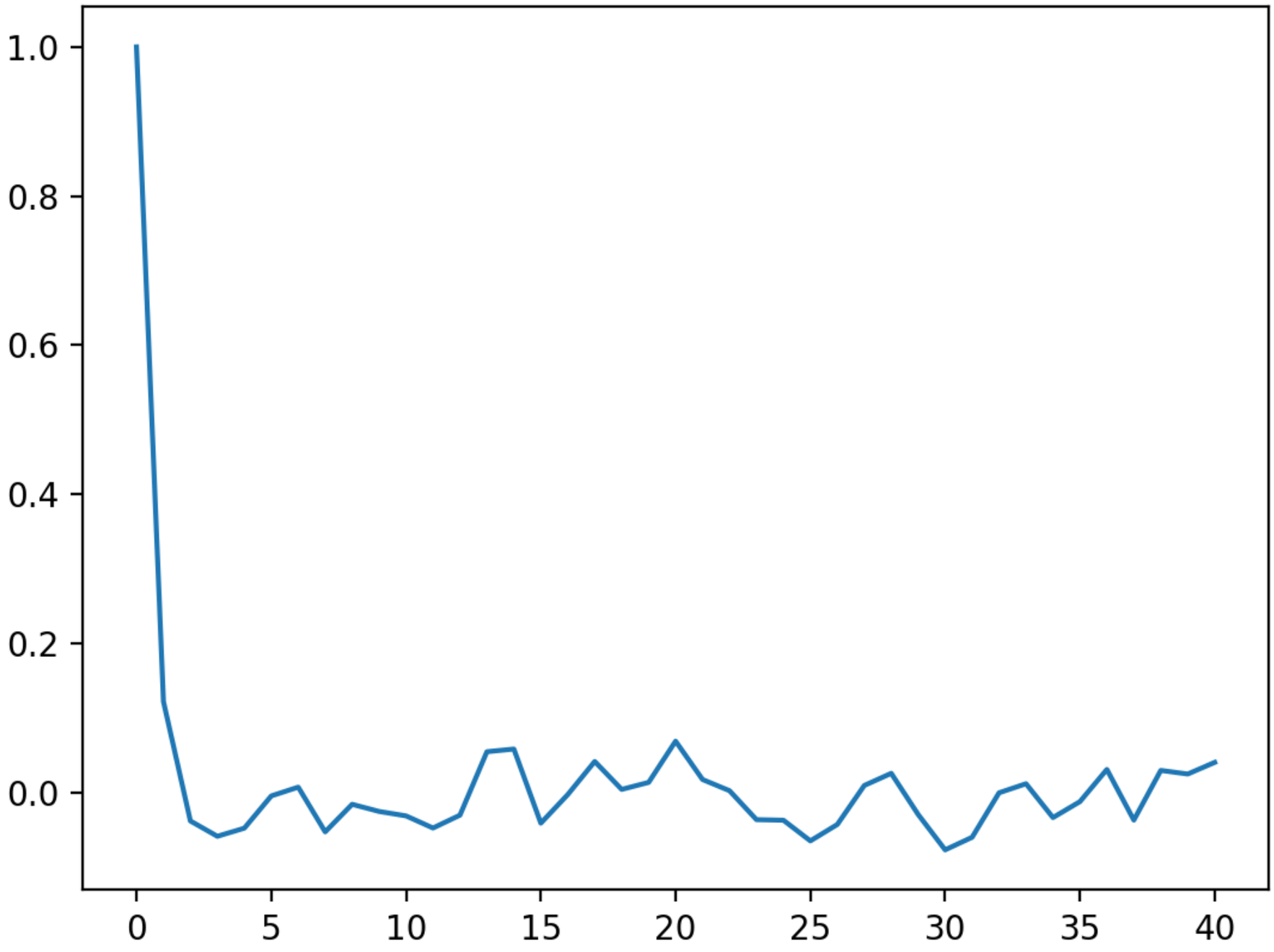
- One version in statsmodels

```
from statsmodels.tsa.stattools import acf, pacf
```

- Calculates ACF as a numpy array

```
my_df = get_google_data()
google = (my_df.shift(1) - my_df).dropna(axis=0)
my_acf = acf(google.High, fft = False)
fig, ax = plt.subplots(1)
ax.plot(my_acf)
```

Stationary Time Series

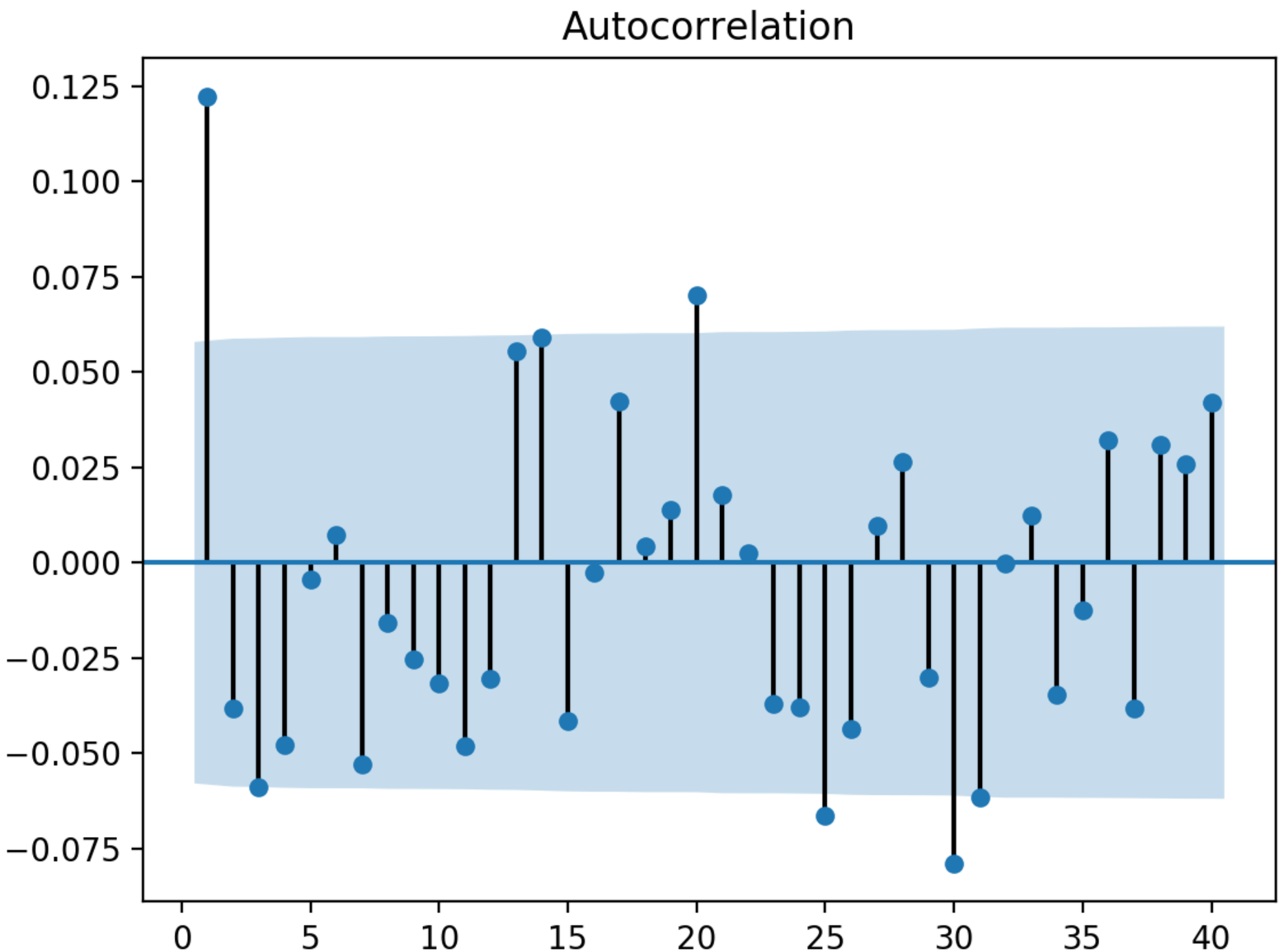


Stationary Time Series

- A graphical version is also available:
  - `import statsmodels.graphics.tsaplots as sgt`

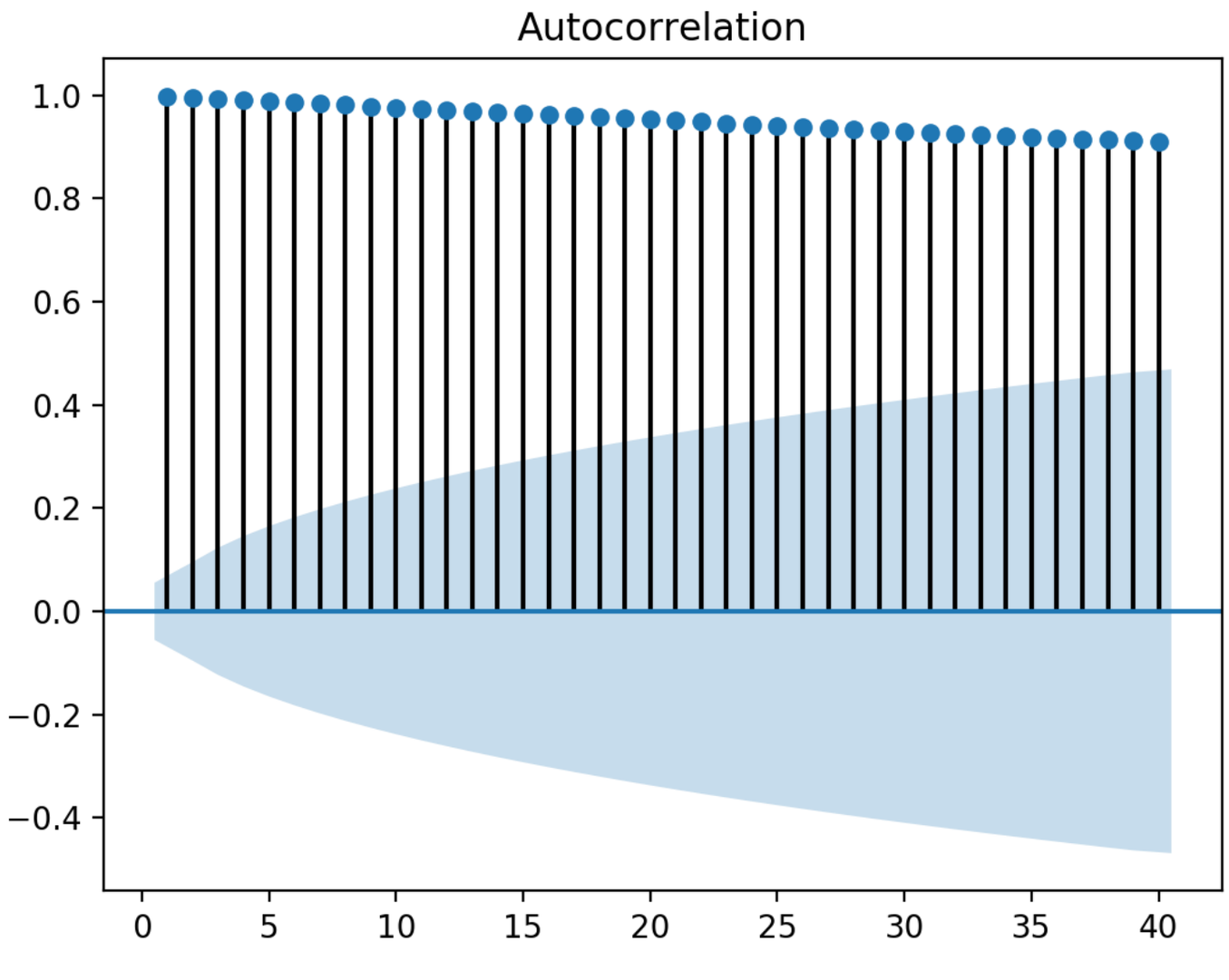
```
sgt.plot_acf(google.High, unbiased = True,
zero=False, lags = 40)
```

Stationary Time Series



Stationary Time Series

- If we apply the same methodology to the original data
  - Much more autocorrelation

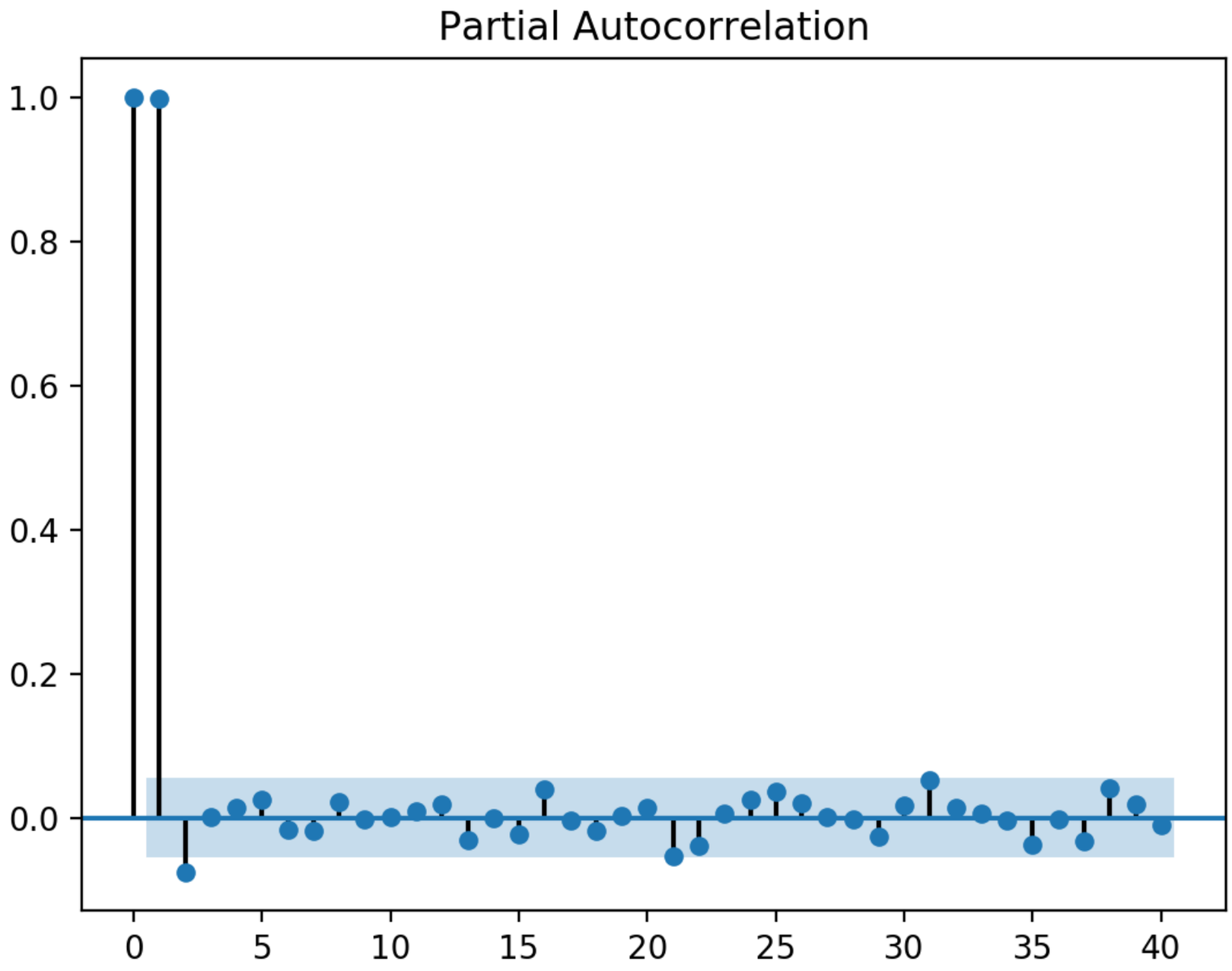


Stationary Time Series

- For lag = 2, the value is mostly because of the correlation between lag = 1
- Use pacm instead
- Calculates only the auto-correlation not explained by auto-correlation for smaller lags

```
sgt.plot_pacf(my_df.High,  lags = 40)
```

Stationary Time Series



Stationary Time Series

- Autoregression Models
  - Forecast variable of interest using ***linear combination of past values of the variable***
  - $y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_m y_{t-m} + \epsilon_t$
  - With "white error" ϵ_t
  - ***Autoregressive Model of order*** m

Stationary Time Series

- Moving average models
  - Uses past forecast errors
    - $y_t = c + \epsilon_t + \theta_1 \epsilon_{t-1} + \ldots + \theta_q \epsilon_{t-q}$
    - with white noise $\epsilon_t,\ \epsilon_{t-1}, \ldots, \epsilon_{t-q}$

Stationary Time Series

- ARIMA models
  - AutoRegressive Integrated Moving Average

$$y'_t = c + \phi_1 y'_{t-1} + \ldots + \phi_p y'_{t-p} + \theta_1 \epsilon_{t-1} + \ldots + \theta_m \epsilon_{t-q} + \epsilon_t$$

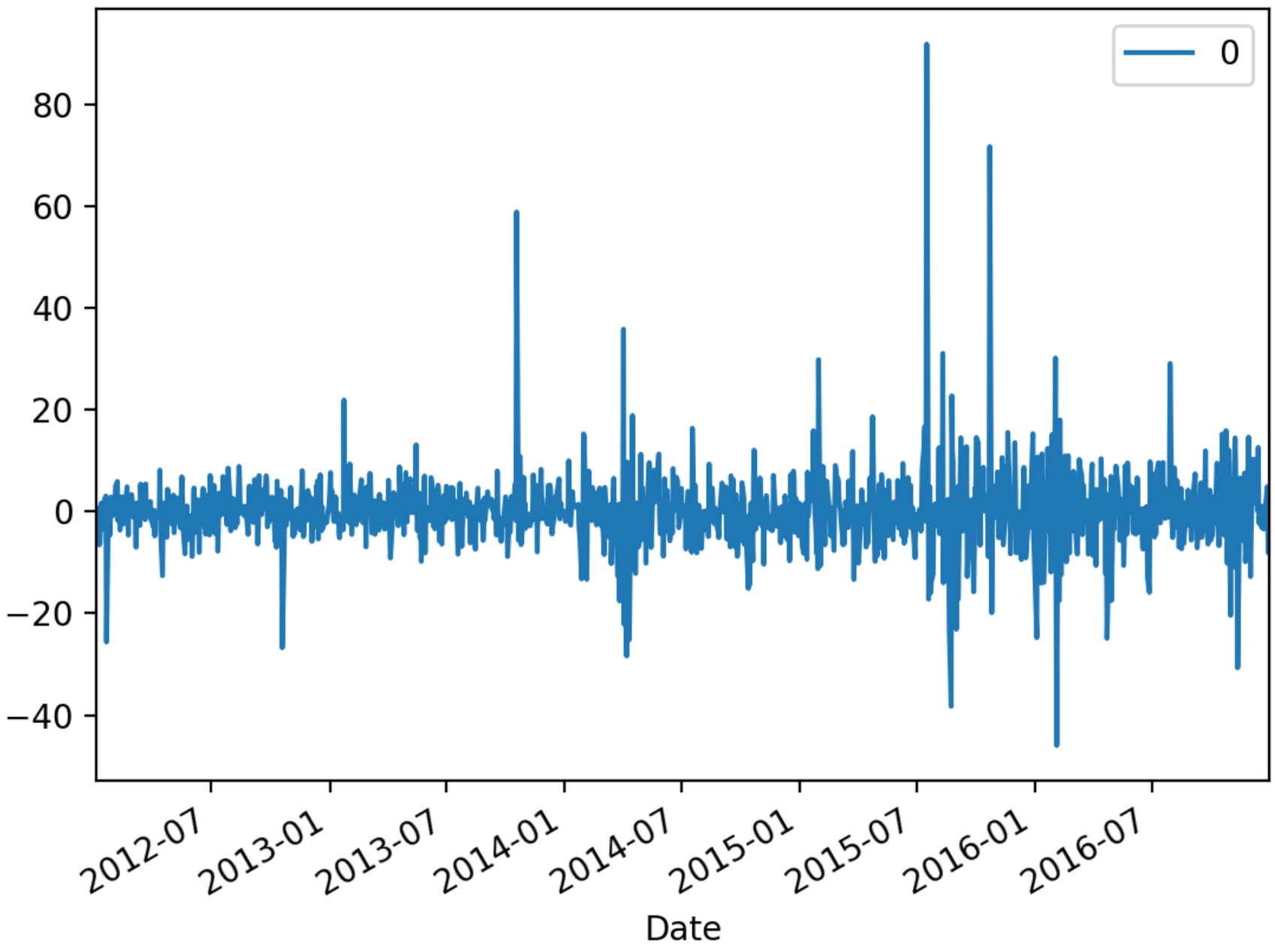
- Differenced value at *t* is
  - autoregressive part of order *p*
  - *d* times differenciated
  - moving average of order *q*
- *ARIMA(p,d,q)*

Stationary Time Series

- Can use statsmodels
  - `from statsmodels.tsa.arima_model import ARIMA`
- Specify degree of ARIMA model and print out its parameters
- Display residual

```
model = ARIMA(my_df.High, order=(1,1,0))
model_fit = model.fit(disp=0)
print(model_fit.summary())
residuals = pd.DataFrame(model_fit.resid)
residuals.plot(label='residual')
plt.show()
```

Stationary Time Series



Seasonal Time Series

- It is possible to extend ARIMA to include a seasonal component
  - In which case we could even put in a trend

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
df = get_data()
my_order = (1,1,1)
my_seasonal_order=(1,1,1,12)
model = SARIMAX(endog = df.Passengers,
                order = my_order,
                seasonal_order = my_seasonal_order)
results = model.fit()
print(results.summary())
results.resid.plot()
plt.show()
```