Time Series Analysis

Yair Mau

Table of contents

About

Disclaimer

The material here is not comprehensive and `does not` constitute a stand alone course in Time Series Analysis. This is only the support material for the actual presential course I give.

When and Where?

Day of the week, from 00:00 to 24:00
Computer classroom #16

Syllabus

Course description

Data analysis of time series, with practical examples from environmental sciences.

Course aims

This course aims at giving the students a broad overview of the main steps involved in the analysis of time series: data management, data wrangling, visualization, analysis, and forecast. The course will provide a hands-on approach, where students will actively engage with real-life datasets from the field of environmental science.

Learning outcomes

On successful completion of this module,students should be able to:

- Explore a time-series dataset, while formulating interesting questions.
- Choose the appropriate tools to attack the problem and answer the questions.
- Communicate their findings and the methods they used to achieve them, using graphs, statistics, text, and a well-documented code.

Course content

- **Data wrangling:** organization, cleaning, merging, filling gaps, excluding outliers, smoothing, resampling.
- **Visualization:** best practices for graph making using leading python libraries.
- **Analysis:** stationarity, seasonality, (auto)correlations, lags, derivatives, spectral analysis.
- **Forecast:** ARIMA
- **Data management:** how to plan ahead and best organize large quantities of data. If there is enough time, we will build a simple time-series database.

Books and other sources

Course evaluation

There will be 2 projects during the semester (each worth 25% of the final grade), and one final project (50%).

Weekly program

Week 1

- **Lecture:** Course overview, setting of expectations. Introduction, basic concepts, continuous vs discrete time series, sampling, aliasing
- **Exercise:** Loading csv file into python, basic time series manipulation with pandas and plotting

Week 2

- **Lecture:** Filling gaps, removing outliers
- **Exercise:** Practice the same topics learned during the lecture. Data: air temperature and relative humidity

Week 3

- **Lecture:** Interpolation, resampling, binning statistics
- **Exercise:** Practice the same topics learned during the lecture. Data: air temperature and relative humidity, precipitation

Week 4

- **Lecture:** Time series plotting: best practices. Dos and don’ts and maybes
- **Exercise:** Practice with Seaborn, Plotly, Pandas, Matplotlib

Project 1
Basic data wrangling, using real data (temperature, relative humidity, precipitation) downloaded from USGS. 25% of the final grade

Week 5

- **Lecture:** Smoothing, running averages, convolution
- **Exercise:** Practice the same topics learned during the lecture. Data: sap flow, evapotranspiration

Week 6

- **Lecture:** Strong and weak stationarity, stochastic processes, auto-correlation
- **Exercise:** Practice the same topics learned during the lecture. Data: temperature and wind speed

Week 7

- **Lecture:** Correlation between signals. Pearson correlation, time-lagged cross-correlations, dynamic time warping
- **Exercise:** Practice the same topics learned during the lecture. Data: temperature, solar radiation, relative humidity, soil moisture, evapotranspiration

Week 8

Same as lecture 7 above

Week 9

- **Lecture:** Download data from repositories, using API, merging, documentation
- **Exercise:** Download data from USGS, NOAA, Fluxnet, Israel Meteorological Service

Project 2
Students will study a Fluxnet site of their choosing. How do gas fluxes (CO2, H2O) depend on environmental conditions? 25% of the final grade

Week 10

- **Lecture:** Fourier decomposition, filtering, Nyquist–Shannon sampling theorem
- **Exercise:** Practice the same topics learned during the lecture. Data: dendrometer data

Week 11

- **Lecture:** Seasonality, seasonal decomposition (trend, seasonal, residue), Hilbert transform
- **Exercise:** Practice the same topics learned during the lecture. Data: monthly atmospheric CO_2 concentration, hourly air temperature

Week 12

- **Lecture:** Derivatives, differencing
- **Exercise:** Practice the same topics learned during the lecture. Data: dendrometer data

Week 13

- **Lecture:** Forecasting. ARIMA
- **Exercise:** Practice the same topics learned during the lecture. Data: vegetation variables (sap flow, ET, DBH, etc)

Final Project
In consultation with the lecturer, students will ask a specific scientific question about a site of their choosing (from NOAA, USGS, Fluxnet), and answer it using the tools learned during the semester. The report will be written in Jupyter Notebook, combining in one document all the calculations, documentation, figures, analysis, and discussion. 50% of the final grade.

Part I

Outliers and Gaps

Z-score

$$z = \frac{x - \mu}{\sigma},$$

where x is a data point, μ is the time series mean, and σ is the time series standard deviation.

```
def zscore(df, degree=3):
    data = df.copy()
    data['zscore'] = (data - data.mean())/data.std()
    outliers = data[(data['zscore'] <= -degree) | (data['zscore'] >= degree)]
    return outliers['value'], data
```

Now we can simply use this function:

```
threshold = 2.5
outliers, transformed = zscore(tx, threshold)
```

Source: Atwan (2022)

1 Z-score

$$z = \frac{x - \mu}{\sigma},$$

where

- x is a data point,
- μ is the time series mean, and
- σ is the time series standard deviation.

```
def zscore(df, degree=3):
    data = df.copy()
    data['zscore'] = (data - data.mean())/data.std()
    outliers = data[(data['zscore'] <= -degree) | (data['zscore'] >= degree)]
    return outliers['value'], data
```

Now we can simply use this function:

```
threshold = 2.5
outliers, transformed = zscore(tx, threshold)
```

Source: Atwan (2022)

Part II

Introduction

2 First Steps — basic time series analysis

Import packages. If you don't have a certain package, e.g. 'newpackage', just type `pip install newpackage`

```
import urllib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import os.path
import matplotlib.dates as mdates
import datetime as dt
import matplotlib as mpl
from pandas.tseries.frequencies import to_offset
from scipy.signal import savgol_filter
```

This is how you download data from Thingspeak

```
filename1 = "test_elad.csv"
# if file is not there, go fetch it from thingspeak
if not os.path.isfile(filename1):
    # define what to download
    channels = "1690490"
    fields = "1,2,3,4,6,7"
    minutes = "30"

    # https://www.mathworks.com/help/thingspeak/readdata.html
    # format YYYY-MM-DD%20HH:NN:SS
    start = "2022-05-01%2000:00:00"
    end = "2022-05-08%2000:00:00"

    # download using Thingspeak's API
    # url = f"https://api.thingspeak.com/channels/{channels}/fields/{fields}.csv?minutes={
    url = f"https://api.thingspeak.com/channels/{channels}/fields/{fields}.csv?start={star
    data = urllib.request.urlopen(url)
    d = data.read()
```

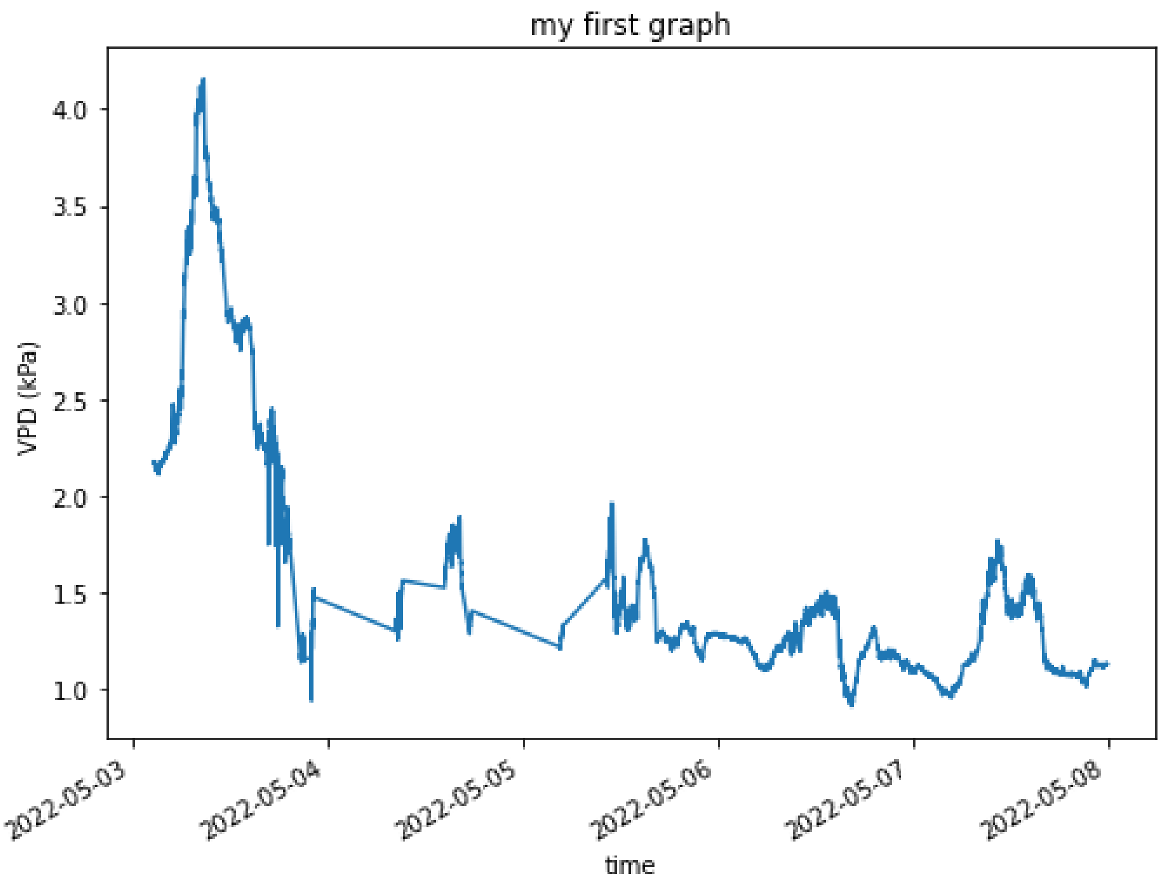
```
    # save data to csv
    file = open(filename1, "w")
    file.write(d.decode('UTF-8'))
    file.close()
```

You can load the data using Pandas. Here we create a “dataframe”, which is a fancy name for a table.

```
# load data
df = pd.read_csv(filename1)
# rename columns
df = df.rename(columns={"created_at": "timestamp",
                        "field1": "T1",
                        "field2": "RH",
                        "field3": "T2",
                        "field4": "motion_sensor",
                        "field6": "VWC",
                        "field7": "VPD",})
# set timestamp as index
df['timestamp'] = pd.to_datetime(df['timestamp'])
df = df.set_index('timestamp')
```

2.0.1 First graph

```
# %matplotlib widget

fig, ax = plt.subplots(1, figsize=(8,6))

ax.plot(df['VPD'])
# add labels and title
ax.set(xlabel = "time",
       ylabel = "VPD (kPa)",
       title = "my first graph")
# makes slanted dates
plt.gcf().autofmt_xdate()
```

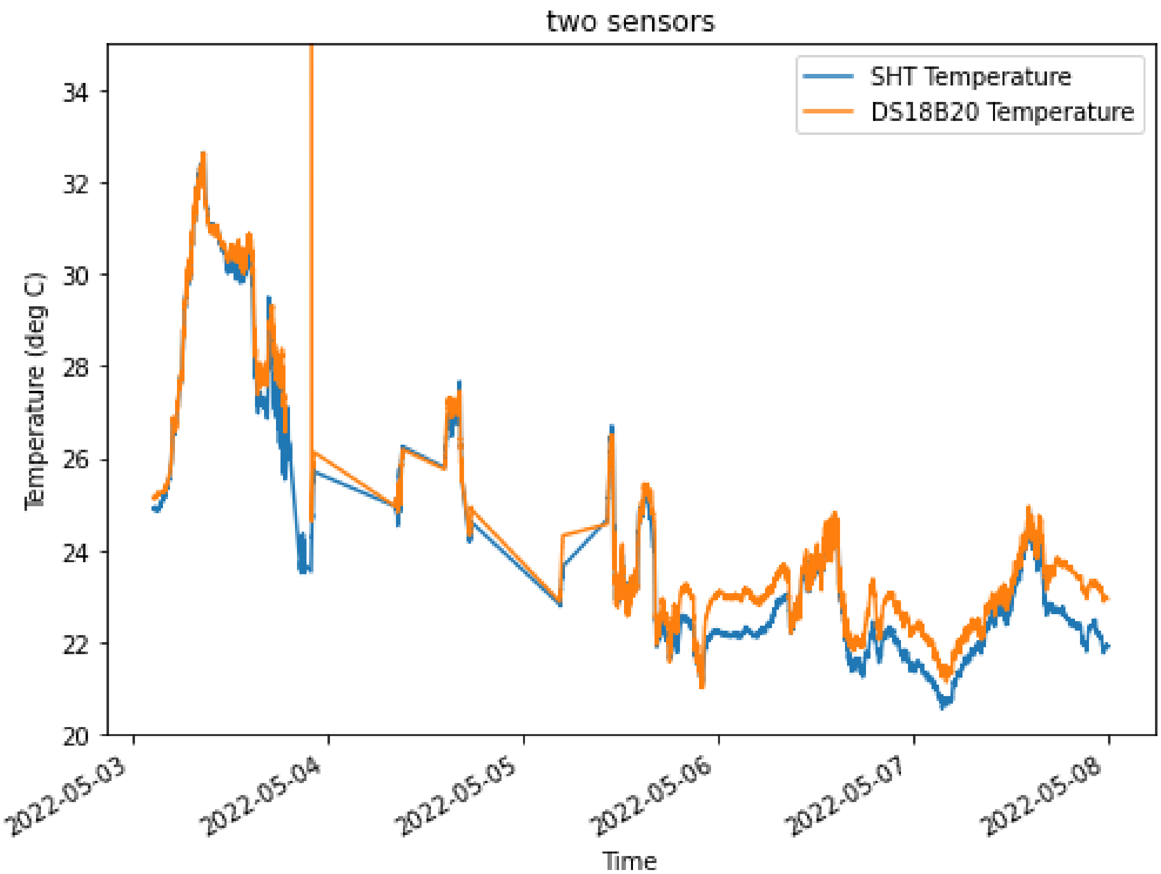


2.0.2 Two columns in the same graph

```
# %matplotlib widget

fig, ax = plt.subplots(1, figsize=(8,6))

ax.plot(df['T1'], color="tab:blue", label="SHT Temperature")
ax.plot(df['T2'], color="tab:orange", label="DS18B20 Temperature")
# add labels and title
ax.set(xlabel = "Time",
       ylabel = "Temperature (deg C)",
       title = "two sensors",
       ylim=[20,35],
       )
# makes slanted dates
```

```
plt.gcf().autofmt_xdate()
ax.legend(loc="upper right")
```

```
<matplotlib.legend.Legend at 0x7fe6c9730610>
```



2.0.3 Calculate stuff

You can calculate new things and save them as new columns of your dataframe.

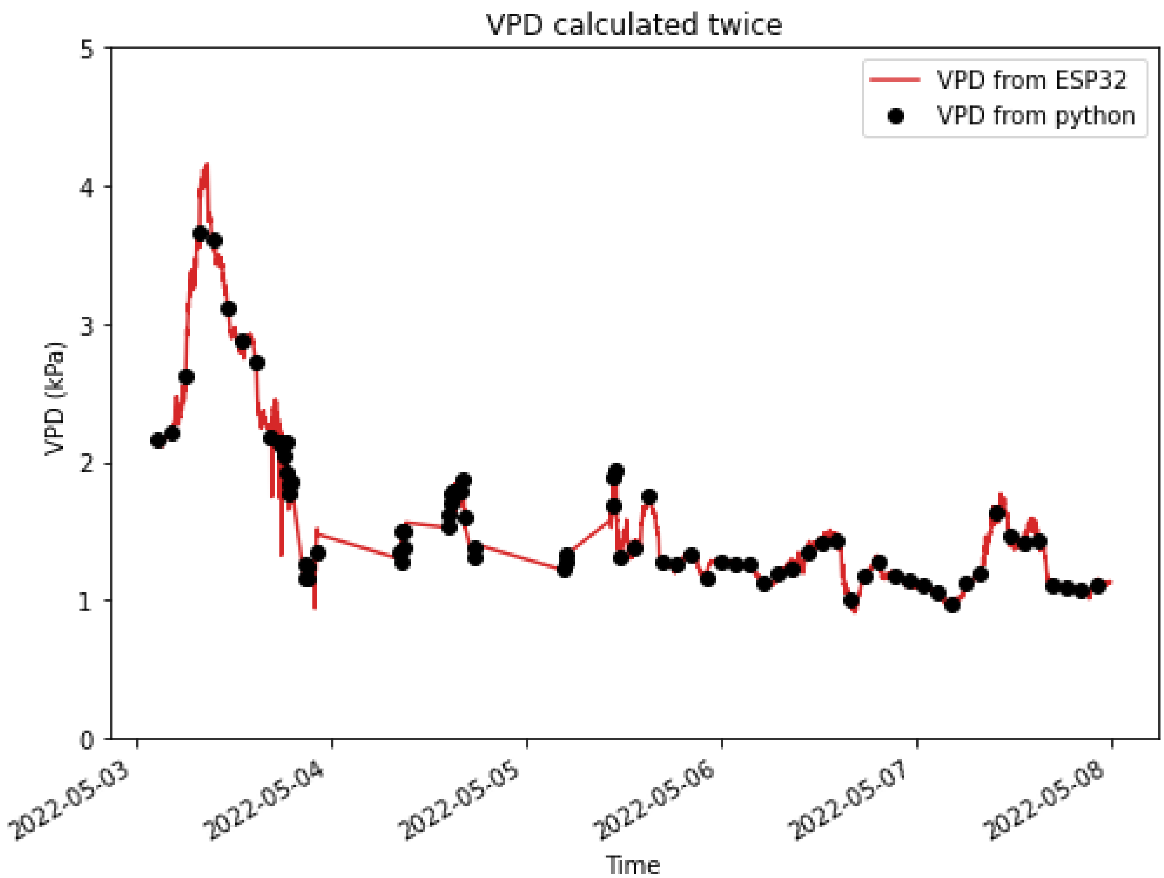
```
def calculate_es(T):
    es = np.exp((16.78 * T - 116.9) / (T + 237.3))
    return es

def calculate_ed(es, rh):
```

```
    return es * rh / 100.0

es = calculate_es(df['T1'])
ed = calculate_ed(es, df['RH'])
df['VPD2'] = es - ed
```

See if what you calculated makes sense.

```
# %matplotlib widget

fig, ax = plt.subplots(1, figsize=(8,6))

ax.plot(df['VPD'], color="tab:red", label="VPD from ESP32")
ax.plot(df['VPD2'][::100], "o", color="black", label="VPD from python")
# add labels and title
ax.set(xlabel = "Time",
       ylabel = "VPD (kPa)",
       title = "VPD calculated twice",
       ylim=[0,5],
       )
# makes slanted dates
plt.gcf().autofmt_xdate()
ax.legend(loc="upper right")
```

```
<matplotlib.legend.Legend at 0x7fe6989ca700>
```

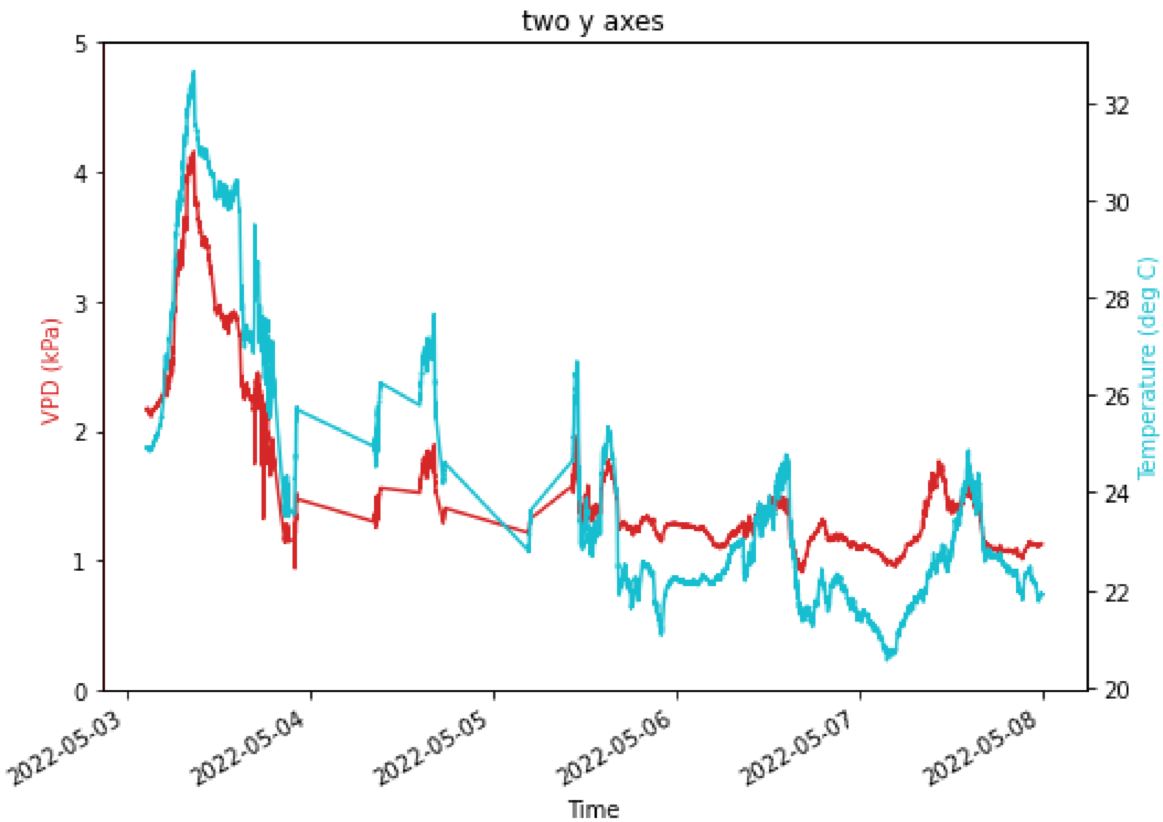


2.0.4 Two y axes

```
# %matplotlib widget

fig, ax = plt.subplots(1, figsize=(8,6))

ax.plot(df['VPD'], color="tab:red", label="VPD")
plt.gcf().autofmt_xdate()
ax2 = ax.twinx()
ax2.plot(df['T1'], color="tab:cyan", label="Temperature")
ax.set(xlabel = "Time",
       title = "two y axes",
       ylim=[0,5],
       )
ax.set_ylabel('VPD (kPa)', color='tab:red')
```

```
ax.spines['left'].set_color('red')

ax2.set_ylabel('Temperature (deg C)', color='tab:cyan')
```

```
Text(0, 0.5, 'Temperature (deg C)')
```

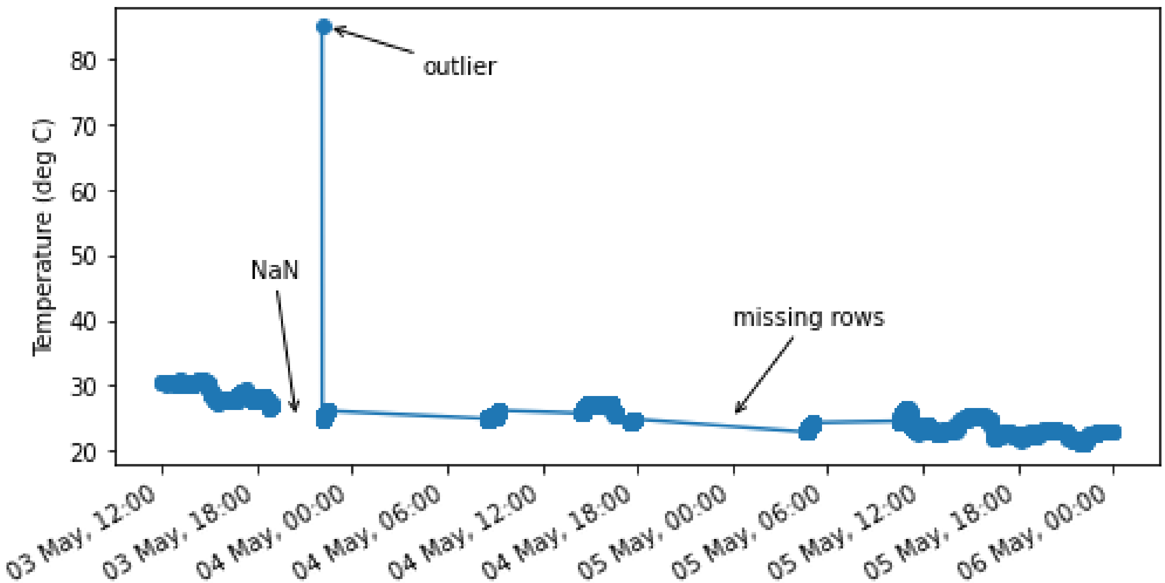


2.1 NaN, Missing data, Outliers

```
# %matplotlib widget

start = "2022-05-03 12:00:00"
end = "2022-05-06 00:00:00"

fig, ax = plt.subplots(1, figsize=(8,4))
```

```python
# plot using pandas' plot method
df.loc[start:end, 'T2'].plot(ax=ax,
                             linestyle='-',
                             marker='o',
                             color="tab:blue",
                             label="data")

# annotate examples here:
# https://jakevdp.github.io/PythonDataScienceHandbook/04.09-text-and-annotation.html
ax.annotate("NaN",                                     # text to write, if nothing, then ""
            xy=('2022-05-03 20:30:00', 25),    # (x,y coordinates for the tip of the arrow
            xycoords='data',                           # xy as 'data' coordinates
            xytext=(-20, 60),                          # xy coordinates for the text
            textcoords='offset points',        # xytext relative to xy
            arrowprops=dict(arrowstyle="->")   # pretty arrow
           )
ax.annotate("outlier",
            xy=('2022-05-03 22:30:00', 85),
            xycoords='data',
            xytext=(40, -20),
            textcoords='offset points',
            arrowprops=dict(arrowstyle="->")
           )
ax.annotate("missing rows",
            xy=('2022-05-05 00:00:00', 25),
            xycoords='data',
            xytext=(0, 40),
            textcoords='offset points',
            arrowprops=dict(arrowstyle="->")
           )

ax.xaxis.set_major_formatter(mdates.DateFormatter('%d %b, %H:00'))
plt.gcf().autofmt_xdate()
ax.set(xlabel="",
       ylabel="Temperature (deg C)")
```

```
[Text(0.5, 0, ''), Text(0, 0.5, 'Temperature (deg C)')]
```



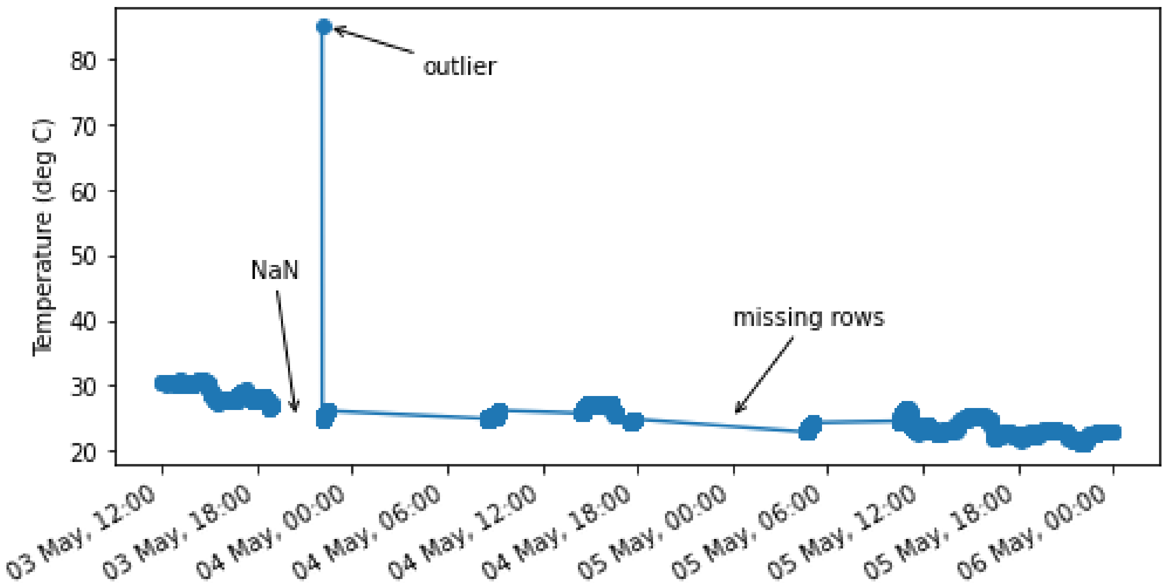
The arrows (annotate) work because the plot was
`df['column'].plot()`

If you use the usual
`ax.plot(df['column'])`
then you matplotlib will not understand timestamps as x-positions. In this case follow the instructions below.

```
# %matplotlib widget

start = "2022-05-03 12:00:00"
end = "2022-05-06 00:00:00"

fig, ax = plt.subplots(1, figsize=(8,4))

ax.plot(df.loc[start:end, 'T2'], linestyle='-', marker='o', color="tab:blue", label="data"

t_nan = '2022-05-03 20:30:00'
x_nan = mdates.date2num(dt.datetime.strptime(t_nan, "%Y-%m-%d %H:%M:%S"))
ax.annotate("NaN",
            xy=(x_nan, 25),
            xycoords='data',
            xytext=(-20, 60),
            textcoords='offset points',
```

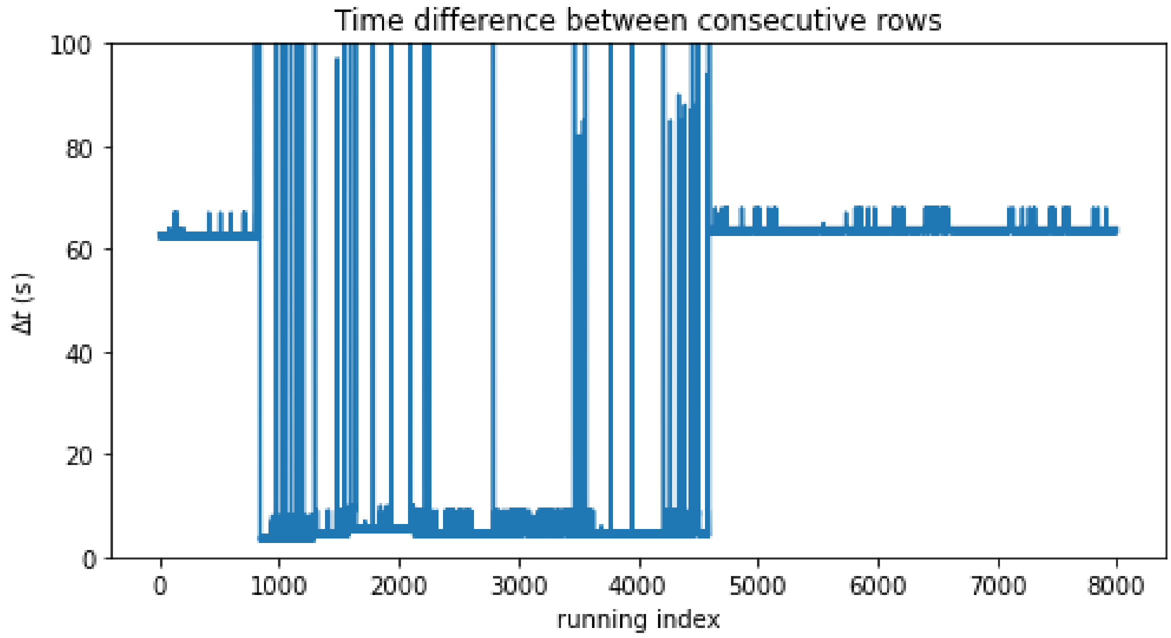
```python
                arrowprops=dict(arrowstyle="->")
               )
t_outlier = '2022-05-03 22:30:00'
x_outlier = mdates.date2num(dt.datetime.strptime(t_outlier, "%Y-%m-%d %H:%M:%S"))
ax.annotate("outlier",
            xy=(x_outlier, 85),
            xycoords='data',
            xytext=(40, -20),
            textcoords='offset points',
            arrowprops=dict(arrowstyle="->")
           )
t_missing = '2022-05-05 00:00:00'
x_missing = mdates.date2num(dt.datetime.strptime(t_missing, "%Y-%m-%d %H:%M:%S"))
ax.annotate("missing rows",
            xy=(x_missing, 25),
            xycoords='data',
            xytext=(0, 40),
            textcoords='offset points',
            arrowprops=dict(arrowstyle="->")
           )
# code for hours, days, etc
# https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes
ax.xaxis.set_major_formatter(mdates.DateFormatter('%d %b, %H:00'))
plt.gcf().autofmt_xdate()
ax.set(xlabel="",
       ylabel="Temperature (deg C)")
```

```
[Text(0.5, 0, ''), Text(0, 0.5, 'Temperature (deg C)')]
```




```
# %matplotlib widget

fig, ax = plt.subplots(1, figsize=(8,4))

delta_index = (df.index.to_series().diff() / pd.Timedelta('1 sec') ).values
ax.plot(delta_index)
ax.set(ylim=[0, 100],
       xlabel="running index",
       ylabel=r"$\Delta t$ (s)",
       title="Time difference between consecutive rows")
```

```
[(0.0, 100.0),
 Text(0.5, 0, 'running index'),
 Text(0, 0.5, '$\\Delta t$ (s)'),
 Text(0.5, 1.0, 'Time difference between consecutive rows')]
```



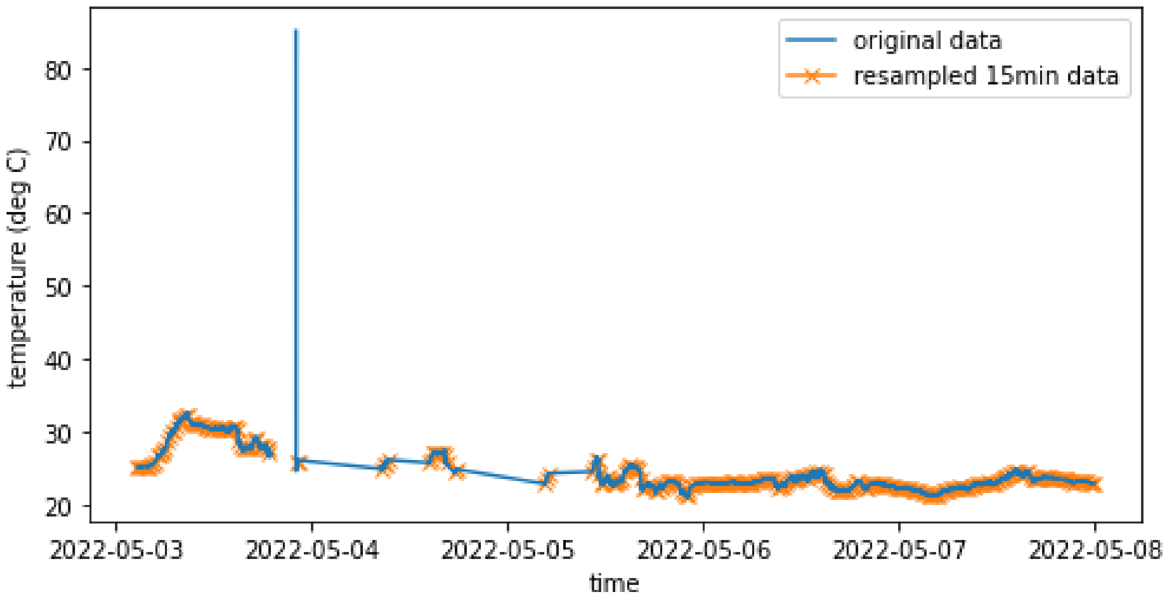
2.2 Resample

2.2.1 Downsampling

```
# %matplotlib widget

fig, ax = plt.subplots(1, figsize=(8,4))

# Downsample to spaced out data points. Change the number below, see what happens.
window_size = '15min'
df_resampled = (df['T2'].resample(window_size)  # resample doesn't do anything yet, just d
                        .mean()                 # this is where stuff happens. you can als
               )
# optional, add half a window size to timestamp
df_resampled.index = df_resampled.index + to_offset(window_size) / 2

ax.plot(df['T2'], color="tab:blue", label="original data")
ax.plot(df_resampled, marker='x', color="tab:orange", zorder=-1,
        label=f"resampled {window_size} data")
ax.legend()
```

```
ax.set(xlabel="time",
       ylabel="temperature (deg C)")
```

```
[Text(0.5, 0, 'time'), Text(0, 0.5, 'temperature (deg C)')]
```

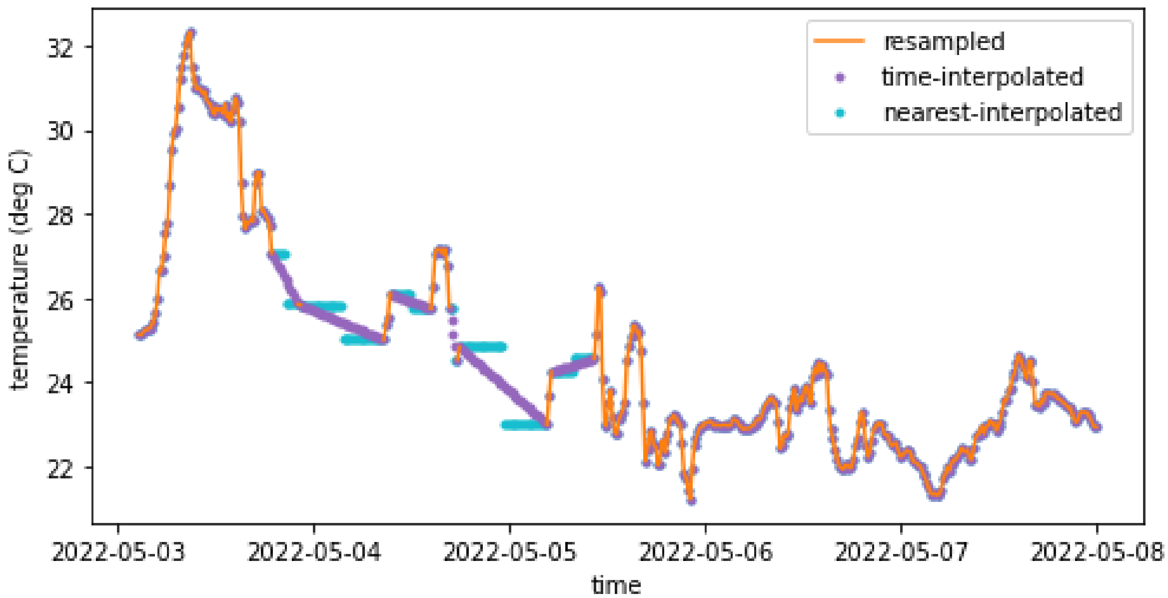


2.2.2 Filling missing data

```
# %matplotlib widget

fig, ax = plt.subplots(1, figsize=(8,4))

# see options for interpolation methods here:
# https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html
df_interpolated1 = df_resampled.interpolate(method='time')
df_interpolated2 = df_resampled.interpolate(method='nearest')

ax.plot(df_resampled, color="tab:orange", label="resampled")
ax.plot(df_interpolated1, '.', color="tab:purple", zorder=-1,
        label=f"time-interpolated")
ax.plot(df_interpolated2, '.', color="tab:cyan", zorder=-2,
        label=f"nearest-interpolated")
```

```
ax.legend()

ax.set(xlabel="time",
       ylabel="temperature (deg C)")
```

```
[Text(0.5, 0, 'time'), Text(0, 0.5, 'temperature (deg C)')]
```



2.3 Smoothing noisy data

Let's first download data from a different project.

```
filename2 = "test_peleg.csv"
# if file is not there, go fetch it from thingspeak
if not os.path.isfile(filename2):
    # define what to download
    channels = "1708067"
    fields = "1,2,3,4,5"
    minutes = "30"

    # https://www.mathworks.com/help/thingspeak/readdata.html
    # format YYYY-MM-DD%20HH:NN:SS
```

```python
    start = "2022-05-15%2000:00:00"
    end = "2022-05-25%2000:00:00"

    # download using Thingspeak's API
    # url = f"https://api.thingspeak.com/channels/{channels}/fields/{fields}.csv?minutes={
    url = f"https://api.thingspeak.com/channels/{channels}/fields/{fields}.csv?start={star
    data = urllib.request.urlopen(url)
    d = data.read()

    # save data to csv
    file = open(filename2, "w")
    file.write(d.decode('UTF-8'))
    file.close()
```

```python
# load data
df = pd.read_csv(filename2)
# rename columns
df = df.rename(columns={"created_at": "timestamp",
                        "field1": "T",
                        "field2": "Tw",
                        "field3": "RH",
                        "field4": "VPD",
                        "field5": "dist",
                        })
# set timestamp as index
df['timestamp'] = pd.to_datetime(df['timestamp'])
df = df.set_index('timestamp')
```

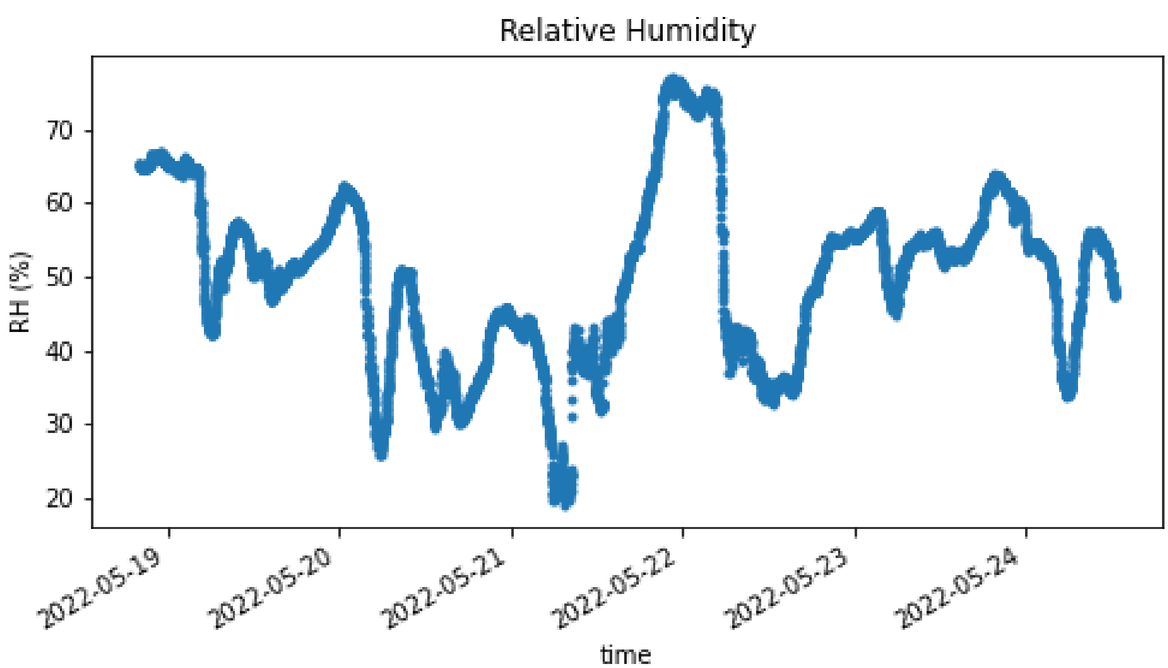
```python
df
```

	entry_id	T	Tw	RH	VPD	dist
timestamp						
2022-05-18 20:09:31+00:00	24716	23.85	23.3125	65.32	1.02532	7.208
2022-05-18 20:10:32+00:00	24717	23.88	23.2500	65.32	1.02717	7.208
2022-05-18 20:11:33+00:00	24718	23.90	23.2500	65.23	1.03107	7.276
2022-05-18 20:12:33+00:00	24719	23.90	23.2500	65.19	1.03226	7.208
2022-05-18 20:13:34+00:00	24720	23.89	23.2500	65.15	1.03282	7.633
...	...	...	...	...	...	...
2022-05-24 12:18:35+00:00	32711	27.47	26.1250	47.49	1.92397	8.925
2022-05-24 12:19:36+00:00	32712	27.47	26.1250	47.62	1.91921	8.925
2022-05-24 12:20:39+00:00	32713	27.47	26.1250	47.96	1.90675	8.925

	entry_id	T	Tw	RH	VPD	dist
timestamp						
2022-05-24 12:21:40+00:00	32714	27.47	26.1875	47.75	1.91444	8.925
2022-05-24 12:22:41+00:00	32715	27.49	26.1875	47.94	1.90971	8.925

2.4 Smoothing noisy data

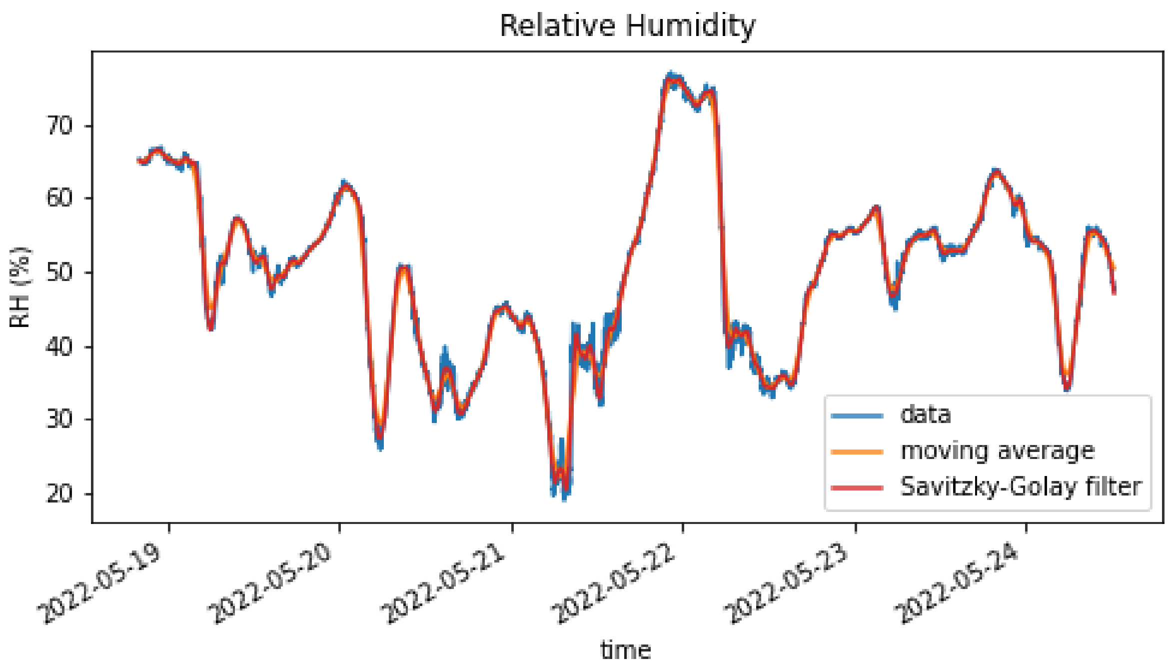
```
# %matplotlib widget

fig, ax = plt.subplots(1, figsize=(8,4))

ax.plot(df['RH'], '.')
# add labels and title
ax.set(xlabel = "time",
       ylabel = "RH (%)",
       title = "Relative Humidity")
# makes slanted dates
plt.gcf().autofmt_xdate()
```



2.4.1 Moving average and SavGol

```python
# %matplotlib widget

fig, ax = plt.subplots(1, figsize=(8,4))

# apply a rolling average of size "window_size",
# it can be either by number of points, or by window time
# window_size = 30  # number of measurements
window_size = '120min'  # minutes
RH_smooth = df['RH'].rolling(window_size, center=True).mean().to_frame()
RH_smooth.rename(columns={'RH': 'rolling_avg'}, inplace=True)

RH_smooth['SG'] = savgol_filter(df['RH'], window_length=121, polyorder=2)

ax.plot(df['RH'], color="tab:blue", label="data")
ax.plot(RH_smooth['rolling_avg'], color="tab:orange", label="moving average")
ax.plot(RH_smooth['SG'], color="tab:red", label="Savitzky-Golay filter")
# add labels and title
ax.set(xlabel = "time",
       ylabel = "RH (%)",
       title = "Relative Humidity")
# makes slanted dates
plt.gcf().autofmt_xdate()
ax.legend()
```

```
<matplotlib.legend.Legend at 0x7fe6a0525730>
```


Relative Humidity
RH (%)
70
60
50
40
30
20
data
moving average
Savitzky-Golay filter
2022-05-19
2022-05-20
2022-05-21
2022-05-22
2022-05-23
2022-05-24
time

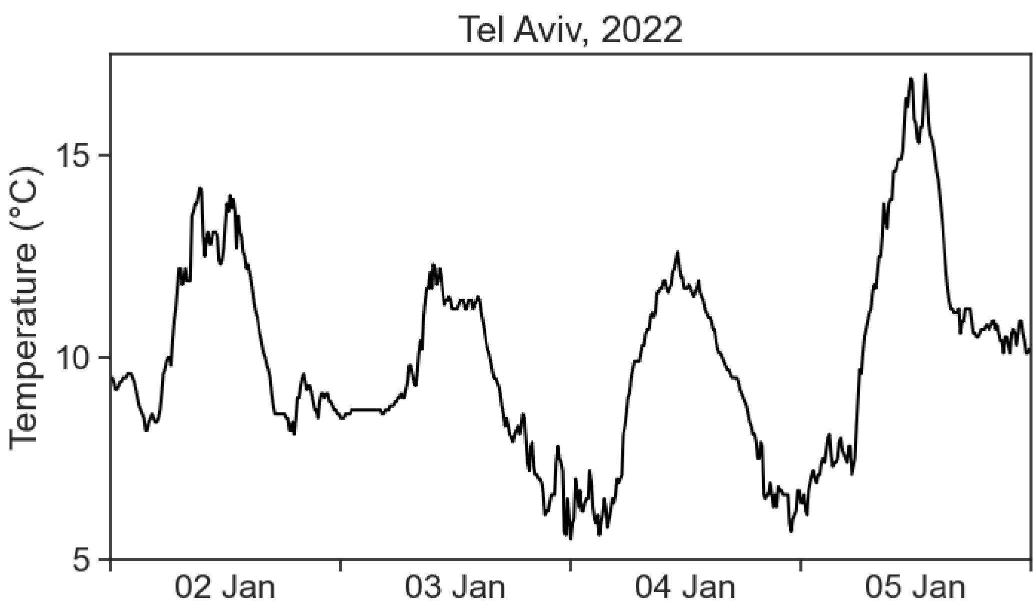
Part III

Smoothing

3 Convolution

Running windows of different shapes (kernels)

This is the temperature for Tel Aviv, between 2 and 5 of January 2022. Data is in intervals of 10 minutes, and was downloaded from the Israel Meteorological Service.



We see that the temperature curve has a rough profile. Can we find ways of getting smoother curves?

3.1 Convolution

Convolution is a fancy word for averaging a time series using a running window. We will use the terms **convolution, running average, and rolling average** interchangeably. See the animation below. We take all temperature values inside a window of width 500 minutes (51 points), and average them with equal weights. The weights profile is called `kernel`.

The pink curve is much smoother than the original! However, the running average cannot describe sharp temperature changes. If we decrease the window width to 200 minutes (21 points), we get the following result.

There is a tradeoff between the smoothness of a curve, and its ability to describe sharp temporal changes.

3.2 Kernels

We can modify our running average, so that values closer to the center of the window have higher weights, and those further away count less. This is achieved by changing the weight profile, or the shape of the kernel. We see below the result of a running average using a triangular window of base 500 minutes (51 points).

Things can get as fancy as we want. Instead of a triangular kernel, which has sharp edges, we can choose a smoother gaussian kernel, see the difference below. We used a gaussian kernel with 60-minute standard deviation (the window in the animation is 4 standard deviations wide).

3.3 Math

The definition of a convolution between signal $f(t)$ and kernel $k(t)$ is

$$(f * k)(t) = \int f(\tau)k(t-\tau)d\tau.$$

The expression $f*k$ denotes the convolution of these two functions. The argument of k is $t-\tau$, meaning that the kernel runs from left to right (as t does), and at every point the two signals (f and k) are multiplied together. It is the product of the signal with the weight function k that gives us an average. Because of $-\tau$, the kernel is flipped backwards, but this has no effect to symmetric kernels, like to ones in the examples above. Finally, the actual running average is not the convolution, but

$$\frac{(f * k)(t)}{\displaystyle\int k(t)dt}.$$

Whenever the integral of the kernel is 1, then the convolution will be identical with the running average.

3.4 Numerics

Running averages are very common tools in time-series analysis. The `pandas` package makes life quite simple. For example, in order to calculate the running average of temperature using a rectangular kernel, one writes

```
df['temperature'].rolling(window='20', center=True).mean()
```

- `window=20` means that the width of the window is 20 points. Pandas lets us define a window width in time units, for example, `window='120min'`.
- `center=True` is needed in order to assign the result of averaging to the center of the window. Make it `False` and see what happens.
- `mean()` is the actual calculation, the average of temperature over the window. The `rolling` part does not compute anything, it just creates a moving window, and we are free to calculate whatever we want. Try to calculate the standard deviation or the maximum, for example.

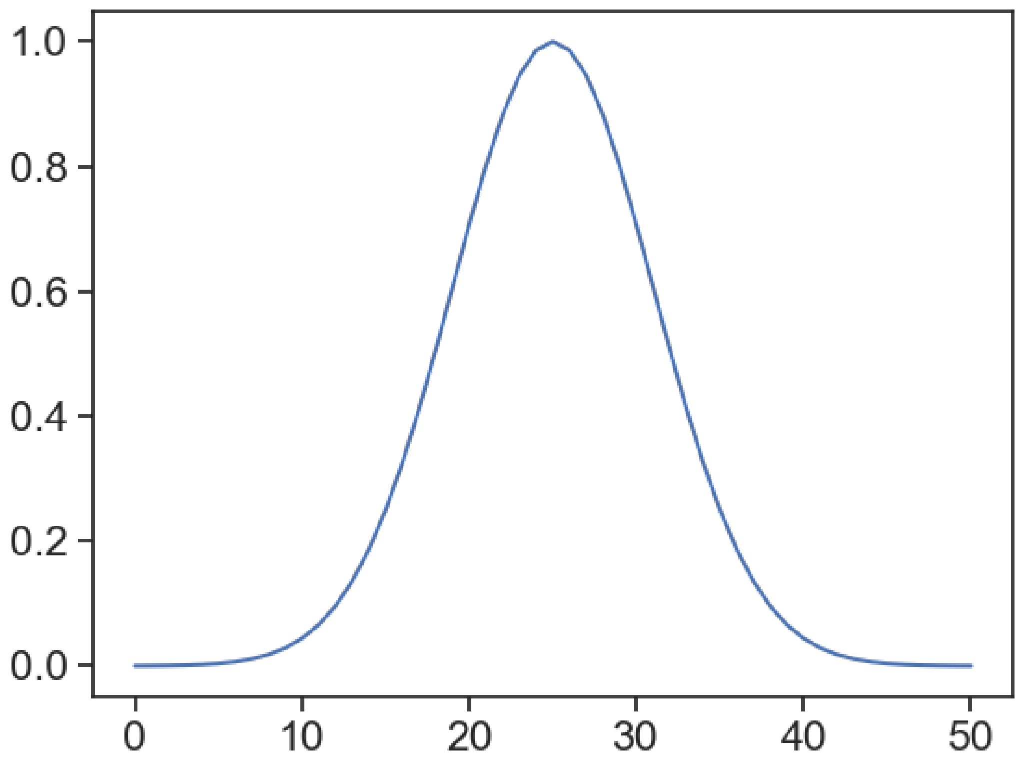
It is implicit in the command above a "rectangular" kernel. What if we want other shapes?

3.4.1 Gaussian

```
(
df['temperature'].rolling(window=window_width,
                          center=True,
                          win_type="gaussian")
                 .mean(std=std_gaussian)
)
```

where * `window_width` is an integer, number of points in your window * `std_gaussian` is the standard deviation of your gaussian, measured in sample points, not time!

For instance, if we have measurements every 10 minutes, and our window width is 500 minutes, then `window_width = 500/10 + 1` (first and last included). If we want a standard deviation of 60 minutes, then `std_gaussian = 6`. The gaussian kernel will look like this:



You can take a look at various options for kernel shapes here, provided by the `scipy` package. The graph above was achieved by running:

```
g = scipy.signal.gaussian(window_width, std)
plt.plot(g)
```

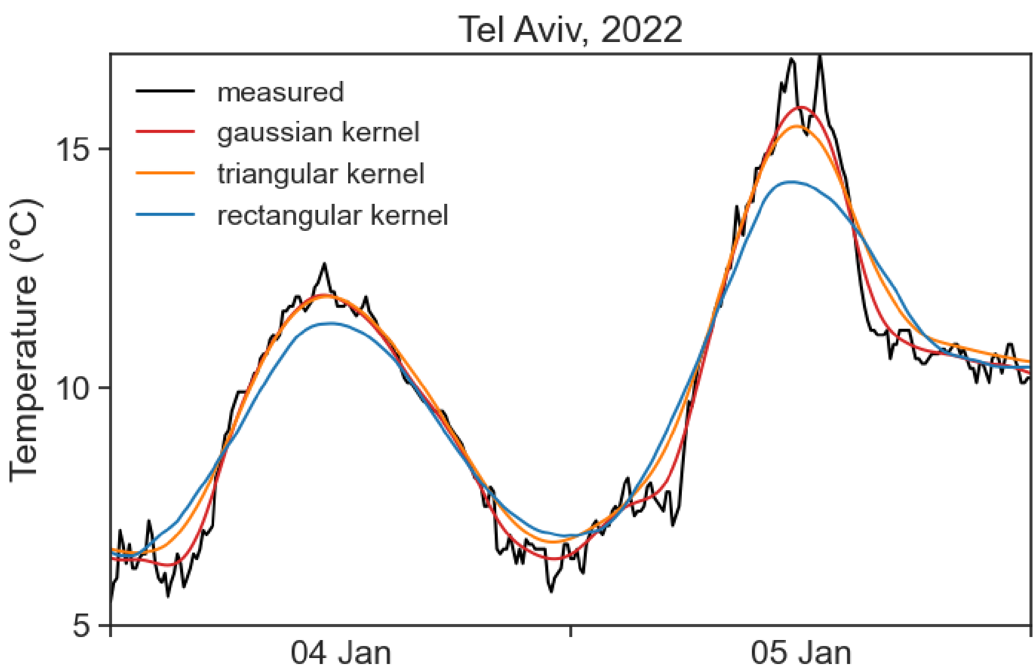
3.4.2 Triangular

Same idea as gaussian, but simpler, because we don't need to think about standard deviation.

```
(
df['temperature'].rolling(window=window_width,
                          center=True,
                          win_type="triang")
                 .mean()
)
```

3.5 Which window shape and width to choose?

Sorry, there is not definite answer here... It really depends on your data and what you need to do with it. See below a comparison of all examples in the videos above.



Part IV

Time Lags

4 Cross-correlation

```python
import numpy as np
```

```python
print('dfvdfv')
```

```
dfvdfv
```

5 Dynamic Time Warp

6 LDTW

according to this paper

Part V

Seasonality

7 Seasonal Decomposition

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()  # datetime converter for a matplotlib
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter
```

8 Trends in Atmospheric Carbon Dioxide

Mauna Loa CO2 concentration.
data from NOAA

```
url = "https://gml.noaa.gov/webdata/ccgg/trends/co2/co2_weekly_mlo.csv"
df = pd.read_csv(url, header=47, na_values=[-999.99])

# you can first download, and then read the csv
# filename = "co2_weekly_mlo.csv"
# df = pd.read_csv(filename, header=47, na_values=[-999.99])

df
```

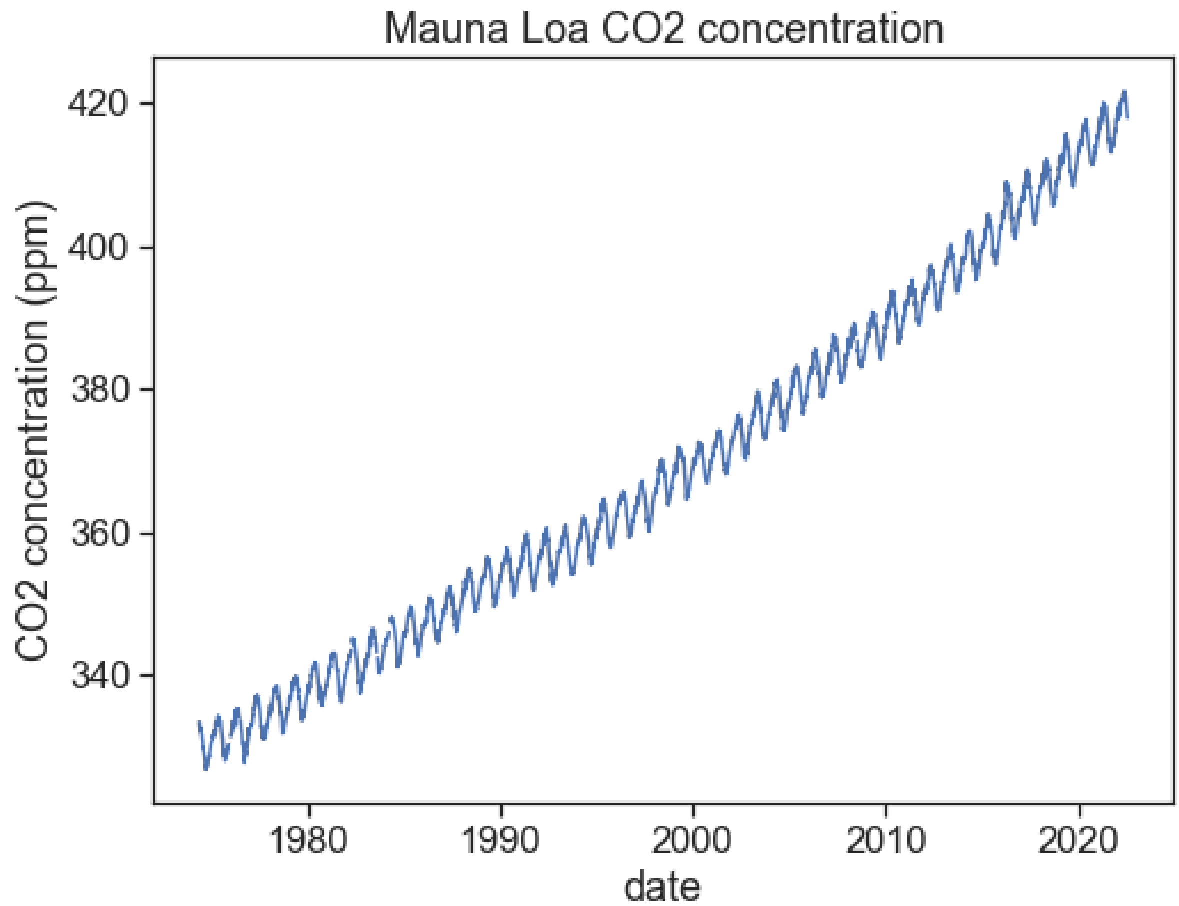
	year	month	day	decimal	average	ndays	1 year ago	10 years ago	increase since 1800
0	1974	5	19	1974.3795	333.37	5	NaN	NaN	50.40
1	1974	5	26	1974.3986	332.95	6	NaN	NaN	50.06
2	1974	6	2	1974.4178	332.35	5	NaN	NaN	49.60
3	1974	6	9	1974.4370	332.20	7	NaN	NaN	49.65
4	1974	6	16	1974.4562	332.37	7	NaN	NaN	50.06
...	...	...	...	...	...	...	...	...	...
2510	2022	6	26	2022.4836	420.31	7	418.14	395.36	138.71
2511	2022	7	3	2022.5027	419.73	6	417.49	395.15	138.64
2512	2022	7	10	2022.5219	419.08	6	417.25	394.59	138.52
2513	2022	7	17	2022.5411	418.43	6	417.14	394.64	138.41
2514	2022	7	24	2022.5603	417.84	6	415.68	394.11	138.36

```
df['date'] = pd.to_datetime(df[['year', 'month', 'day']])
df = df.set_index('date')
df
```

	year	month	day	decimal	average	ndays	1 year ago	10 years ago	increase since 180
date									
1974-05-19	1974	5	19	1974.3795	333.37	5	NaN	NaN	50.40

	year	month	day	decimal	average	ndays	1 year ago	10 years ago	increase since 180
date									
1974-05-26	1974	5	26	1974.3986	332.95	6	NaN	NaN	50.06
1974-06-02	1974	6	2	1974.4178	332.35	5	NaN	NaN	49.60
1974-06-09	1974	6	9	1974.4370	332.20	7	NaN	NaN	49.65
1974-06-16	1974	6	16	1974.4562	332.37	7	NaN	NaN	50.06
...	...	...	...	...	...	...	...	...	...
2022-06-26	2022	6	26	2022.4836	420.31	7	418.14	395.36	138.71
2022-07-03	2022	7	3	2022.5027	419.73	6	417.49	395.15	138.64
2022-07-10	2022	7	10	2022.5219	419.08	6	417.25	394.59	138.52
2022-07-17	2022	7	17	2022.5411	418.43	6	417.14	394.64	138.41
2022-07-24	2022	7	24	2022.5603	417.84	6	415.68	394.11	138.36

```
# %matplotlib widget

fig, ax = plt.subplots(1, figsize=(8,6))
ax.plot(df['average'])
ax.set(xlabel="date",
       ylabel="CO2 concentration (ppm)",
       # ylim=[0, 430],
       title="Mauna Loa CO2 concentration");
```



fill missing data. interpolate method: 'time'

interpolation methods visualized

```
df['co2'] = (df['average'].resample("D") #resample daily
                          .interpolate(method='time') #interpolate by time
             )
df
```

date	year	month	day	decimal	average	ndays	1 year ago	10 years ago	increase since 180
1974-05-19	1974	5	19	1974.3795	333.37	5	NaN	NaN	50.40
1974-05-26	1974	5	26	1974.3986	332.95	6	NaN	NaN	50.06
1974-06-02	1974	6	2	1974.4178	332.35	5	NaN	NaN	49.60
1974-06-09	1974	6	9	1974.4370	332.20	7	NaN	NaN	49.65
1974-06-16	1974	6	16	1974.4562	332.37	7	NaN	NaN	50.06

date	year	month	day	decimal	average	ndays	1 year ago	10 years ago	increase since 180
...	...	...	...	...	...	...	...	...	...
2022-06-26	2022	6	26	2022.4836	420.31	7	418.14	395.36	138.71
2022-07-03	2022	7	3	2022.5027	419.73	6	417.49	395.15	138.64
2022-07-10	2022	7	10	2022.5219	419.08	6	417.25	394.59	138.52
2022-07-17	2022	7	17	2022.5411	418.43	6	417.14	394.64	138.41
2022-07-24	2022	7	24	2022.5603	417.84	6	415.68	394.11	138.36

8.1 decompose data

`seasonal_decompose` returns an object with four components:

- observed: $Y(t)$
- trend: $T(t)$
- seasonal: $S(t)$
- resid: $e(t)$

Additive model:

$$Y(t) = T(t) + S(t) + e(t)$$

Multiplicative model:

$$Y(t) = T(t) \times S(t) \times e(t)$$

8.1.0.1 Interlude

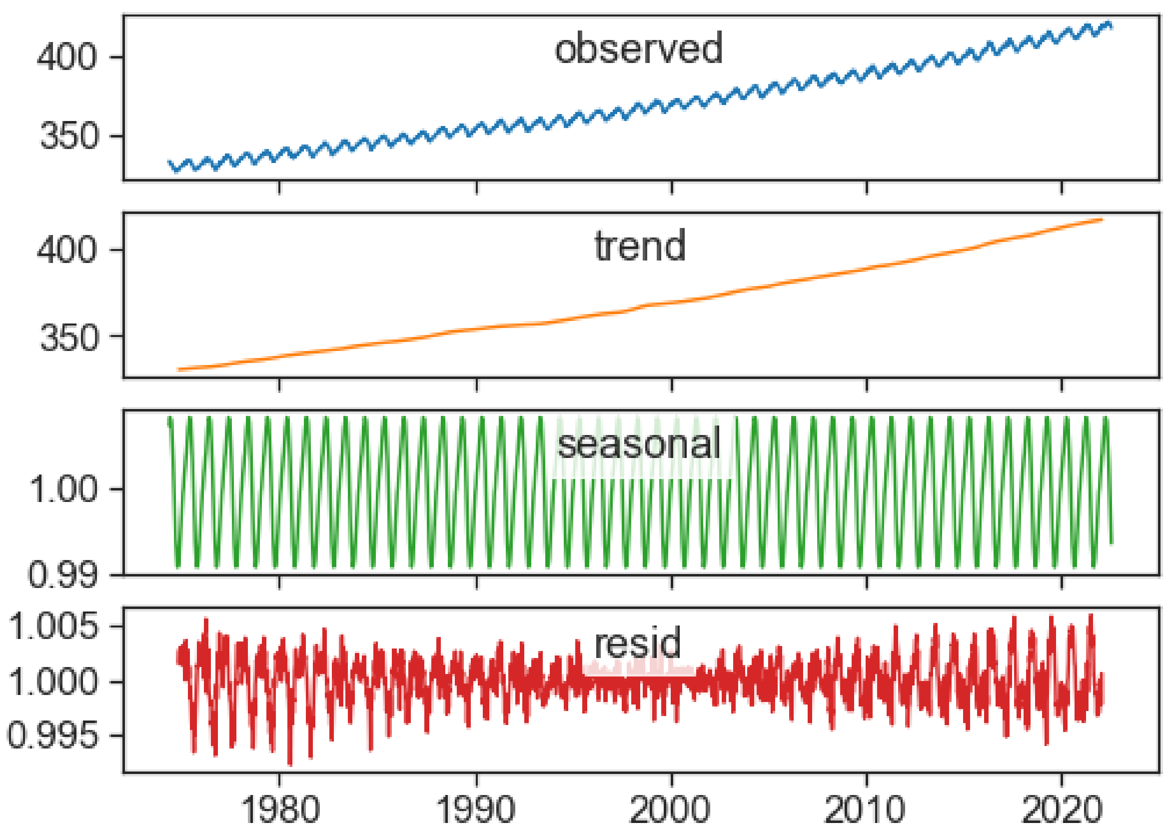
learn how to use `zip` in a loop

```python
letters = ['a', 'b', 'c', 'd', 'e']
numbers = [1, 2, 3, 4, 5]
# zip let's us iterate over to lists at the same time
for l, n in zip(letters, numbers):
    print(f"{l} = {n}")
```

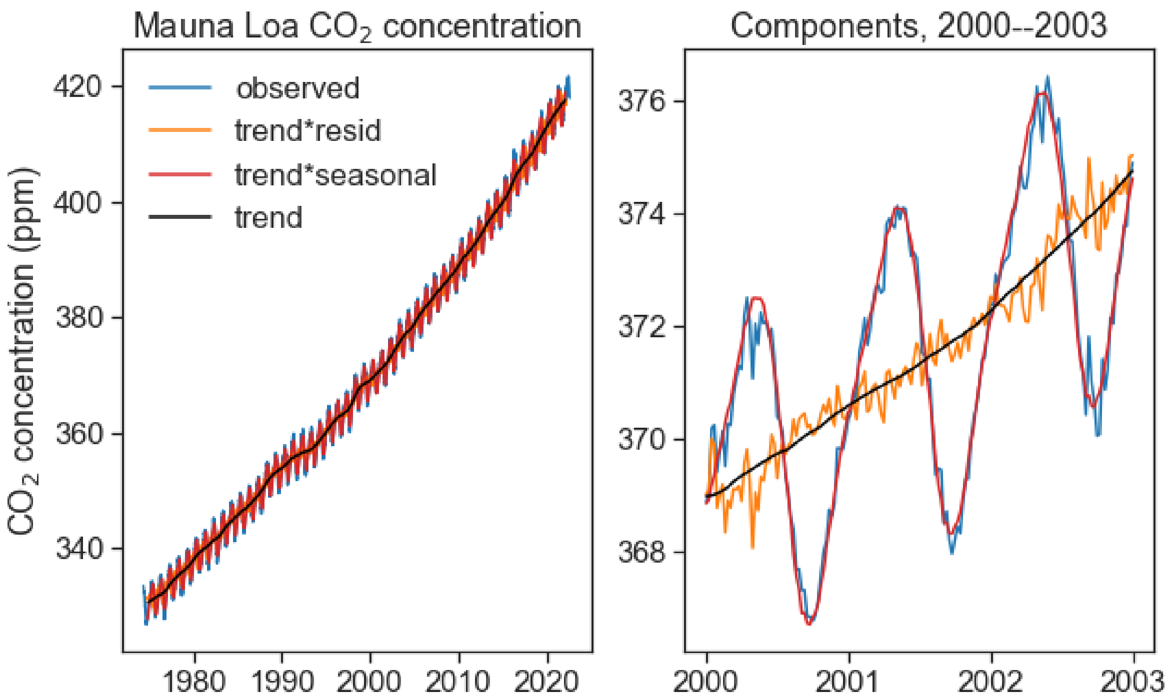
```
a = 1
b = 2
c = 3
d = 4
e = 5
```

Plot each component separately.

```
# %matplotlib widget

fig, ax = plt.subplots(4, 1, figsize=(8,6), sharex=True)
decomposed_m = seasonal_decompose(df['co2'], model='multiplicative')
decomposed_a = seasonal_decompose(df['co2'], model='additive')
decomposed = decomposed_m
pos = (0.5, 0.9)
components =["observed", "trend", "seasonal", "resid"]
colors = ["tab:blue", "tab:orange", "tab:green", "tab:red"]
for axx, component, color in zip(ax, components, colors):
    data = getattr(decomposed, component)
    axx.plot(data, color=color)
    axx.text(*pos, component, bbox=dict(facecolor='white', alpha=0.8),
             transform=axx.transAxes, ha='center', va='top')
```



```
# %matplotlib widget

decomposed = decomposed_m

fig, ax = plt.subplots(1, 2, figsize=(10,6))
ax[0].plot(df['co2'], color="tab:blue", label="observed")
ax[0].plot(decomposed.trend * decomposed.resid, color="tab:orange", label="trend*resid")
ax[0].plot(decomposed.trend * decomposed.seasonal, color="tab:red", label="trend*seasonal"
ax[0].plot(decomposed.trend, color="black", label="trend")
ax[0].set(ylabel="CO$_2$ concentration (ppm)",
          title="Mauna Loa CO$_2$ concentration")
ax[0].legend(frameon=False)

start = "2000-01-01"
end = "2003-01-01"
zoom = slice(start, end)
ax[1].plot(df.loc[zoom, 'co2'], color="tab:blue", label="observed")
ax[1].plot((decomposed.trend * decomposed.resid)[zoom], color="tab:orange", label="trend*r
ax[1].plot((decomposed.trend * decomposed.seasonal)[zoom], color="tab:red", label="trend*s
ax[1].plot(decomposed.trend[zoom], color="black", label="trend")
date_form = DateFormatter("%Y")
ax[1].xaxis.set_major_formatter(date_form)
ax[1].xaxis.set_major_locator(mdates.YearLocator(1))
ax[1].set_title("Components, 2000--2003");
```



Mauna Loa CO_2 concentration
CO_2 concentration (ppm)
observed
trend*resid
trend*seasonal
trend
420
400
380
360
340
1980
1990
2000
2010
2020
Components, 2000--2003
376
374
372
370
368
2000
2001
2002
2003

Technical Stuff

Operating systems

I recommend working with UNIX-based operating systems (MacOS or Linux). Everything is easier.

If you use Windows, consider installing Linux on Windows with WSL.

Software

Anaconda's Python distribution

VSCode

Python packages

Kats — a one-stop shop for time series analysis
Developed by Meta

statsmodels statsmodels is a Python package that provides a complement to scipy for statistical computations including descriptive statistics and estimation and inference for statistical models.

ydata-profiling
Quick Exploratory Data Analysis on time-series data. Read also this.

Sources

Books

from Data to Viz

Fundamentals of Data Visualization, by Claus O. Wilke

PyNotes in Agriscience

Forecasting: Principles and Practice (3rd ed), by Rob J Hyndman and George Athanasopoulos

Python for Finance Cookbook 2nd Edition - Code Repository

Practical time series analysis,: prediction with statistics and machine learning, by Aileen Nielsen
The online edition of this book is available for Hebrew University staff and students.

Time series analysis with Python cookbook : practical recipes for exploratory data analysis, data preparation, forecasting, and model evaluation, by Tarek A. Atwan
The online edition of this book is available for Hebrew University staff and students.

Hands-on Time Series Analysis with Python: From Basics to Bleeding Edge Techniques, by B V Vishwas, Ashish Patel
The online edition of this book is available for Hebrew University staff and students.

Videos

Times Series Analysis for Everyone, by Bruno Goncalves
This series is available for Hebrew University staff and students.

Time Series Analysis with Pandas, by Joshua Malina This video is available for Hebrew University staff and students.

References

Atwan, Tarek A. 2022. *Time Series Analysis with Python Cookbook: Practical Recipes for Exploratory Data Analysis, Data Preparation, Forecasting, and Model Evaluation*. Packt.