

Time Series Analysis

Yair Mau

Table of contents

| | |
|---|-----------|
| about | 7 |
| disclaimer | 7 |
| what, who, when and where? | 7 |
| syllabus | 7 |
| course description | 7 |
| course aims | 8 |
| learning outcomes | 8 |
| course content | 8 |
| books and other sources | 8 |
| course evaluation | 9 |
| weekly program | 9 |
| who cares? | 13 |
| why “Time Series Analysis?” | 13 |
| why “Environmental Sciences” | 13 |
| what is it good for? | 13 |
| do I need it? | 14 |
| what will I actually gain from it? | 14 |
| I start here | 15 |
| 1 the boring stuff you absolutely need to do | 16 |
| 1.1 Anaconda | 16 |
| 1.2 VSCode | 16 |
| 1.3 jupyter notebooks | 16 |
| 1.4 folder structure | 17 |
| 2 numpy, pandas, matplotlib | 18 |
| 2.1 pandas | 18 |
| 2.2 pyplot | 19 |
| 3 learn by example | 23 |
| 3.1 import packages | 23 |

| | | |
|------------|---|-----------|
| 3.2 | load data | 24 |
| 3.3 | dealing with dates | 25 |
| 3.4 | first plot | 26 |
| 3.5 | first plot, v2.0 | 27 |
| 3.6 | modifications | 29 |
| 3.7 | playing with the code | 31 |
| 4 | AI policy | 32 |
| II | resampling | 34 |
| 5 | motivation | 35 |
| 5.1 | Challenges | 36 |
| 6 | resampling | 40 |
| 7 | upsampling | 49 |
| 7.1 | Forward/Backward fill | 52 |
| 7.2 | Interpolation | 53 |
| 8 | interpolation | 55 |
| 9 | FAQ | 56 |
| 9.1 | When upsampling, how to fill missing values with zero? | 59 |
| III | smoothing | 60 |
| 10 | motivation | 61 |
| 10.1 | Tumbling vs Sliding | 65 |
| 11 | sliding window | 66 |
| 11.1 | kernels | 67 |
| 11.2 | math | 67 |
| 11.3 | numerics | 68 |
| 11.3.1 | gaussian | 68 |
| 11.3.2 | triangular | 70 |
| 11.4 | which window shape and width to choose? | 70 |
| 12 | LOESS | 72 |

| | |
|--|-----------|
| 13 a perfect smoother | 73 |
| IV outliers and gaps | 74 |
| 14 motivation | 75 |
| 15 Z-score | 77 |
| 16 IQR (Inter Quartile Range) | 78 |
| V best practices | 79 |
| 17 motivation | 80 |
| 18 date formatting | 81 |
| VI stationarity | 89 |
| 19 motivation | 90 |
| 20 stochastic processes | 91 |
| 21 autocorrelation | 92 |
| 21.1 question | 92 |
| 21.2 mean and standard deviation | 93 |
| 21.3 autocorrelation | 94 |
| VII time lags | 95 |
| 22 motivation | 96 |
| 23 cross-correlation | 97 |
| 24 dynamic time warp | 98 |
| 25 LDTW | 99 |

| | |
|---|------------|
| VIII frequency | 100 |
| 26 motivation | 101 |
| 27 Fourier transform | 102 |
| 27.1 basic wave concepts | 102 |
| 27.2 Fourier's theorem | 104 |
| 27.3 Fourier series | 104 |
| 28 filtering | 106 |
| 29 Nyquist-Shannon sampling theorem | 107 |
| IX seasonality | 108 |
| 30 motivation | 109 |
| 31 seasonal decomposition | 110 |
| 31.1 trends in atmospheric carbon dioxide | 110 |
| 31.2 decompose data | 113 |
| 32 Hilbert transform | 116 |
| X rates of change | 117 |
| 33 motivation | 118 |
| 34 derivatives | 124 |
| 35 finite differences | 125 |
| 36 Fourier-based derivatives | 127 |
| 37 LOESS-based derivatives | 128 |
| XI forecasting | 129 |
| 38 motivation | 130 |
| 39 ARIMA | 131 |

| | |
|--------------------------------|------------|
| XII assignments | 132 |
| 40 assignment 1 | 133 |
| 40.1 Task | 133 |
| 40.2 Guidelines | 134 |
| 40.3 Evaluation | 134 |
| 41 assignment 2 | 136 |
| 42 final assignment | 137 |
| 43 technical stuff | 138 |
| 43.1 software | 138 |
| 43.2 python packages | 138 |
| sources | 139 |
| books | 139 |
| videos | 139 |
| references | 140 |
| stuff | 141 |

about

Welcome to **Time Series Analysis for Environmental Sciences** (71606) at the Hebrew University of Jerusalem. This is Yair Mau, your host for today. I am a senior lecturer at the Institute of Environmental Sciences, at the Faculty of Agriculture, Food and Environment, in Rehovot, Israel.

This website contains (almost) all the material you'll need for the course. If you find any mistakes, or have any comments, please email me.

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.

what, who, when and where?

Course number 71606, 3 academic points
Yair Mau (lecturer), Erez Feuer (TA)
Tuesdays, from 14:15 to 17:00
Computer [classroom #18](#)
Office hours: upon request

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

[Click here.](#)

course evaluation

There will be small projects during the semester (totaling 50% of the final grade), and one final project (50%).

weekly program

This year's course will be a bit different than planned due to the shortening of the academic semester. The information below is NOT up to date. Ask Yair what is relevant this year.

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 CO₂ 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.

who cares?

why “Time Series Analysis?”

Time has two aspects. There is the arrow, the running river, without which there is no change, no progress, or direction, or creation. And there is the circle or the cycle, without which there is chaos, meaningless succession of instants, a world without clocks or seasons or promises.

URSULA K. LE GUIN

You are here because you are interested in how things change, evolve. In this course I want to discuss with you how to make sense of data whose temporal nature is in its very essence. We will talk about randomness, cycles, frequencies, correlations, and more.

why “Environmental Sciences”

This same time series analysis (TSA) course could be called instead “TSA for finance”, “TSA for Biology”, or any other application. The emphasis in this course is **not** Environmental Sciences, but the concepts and tools of TSA. Because my research is in Environmental Science, and many of the graduate students at HUJI-Rehovot research this, I chose to use examples “close to home”. The same toolset should be useful for students of other disciplines.

what is it good for?

In many fields of science we are flooded by data, and it’s hard to see the forest for the trees. I hope that the topics we’ll

discuss in this course can help you find meaningful patterns in your data, formulate interesting hypotheses, and design better experiments.

do I need it?

Maybe. If you are a grad student and you have temporal data to analyze, then probably yes. However, I have very fond memories of courses that I took as a grad student that were completely unrelated to my research. Sometimes “because it’s fun” is a perfectly good answer.

what will I actually gain from it?

By the end of this course you will have gained:

- a **hands-on** experience of fundamental time-series analysis tools
- an **intuition** regarding the basic concepts
- **technical** abilities
- a **springboard** for learning more about the subject by yourself

Part I
start here

1 the boring stuff you absolutely need to do

I assume everyone registered has taken a basic Python course. On your computer, do the following:

1.1 Anaconda

Install [Anaconda's Python distribution](#). The Anaconda installation brings with it all the main python packages we will need to use. In order to install extra packages, refer to these two tutorials: [tutorial 1](#), [tutorial 2](#).

1.2 VSCode

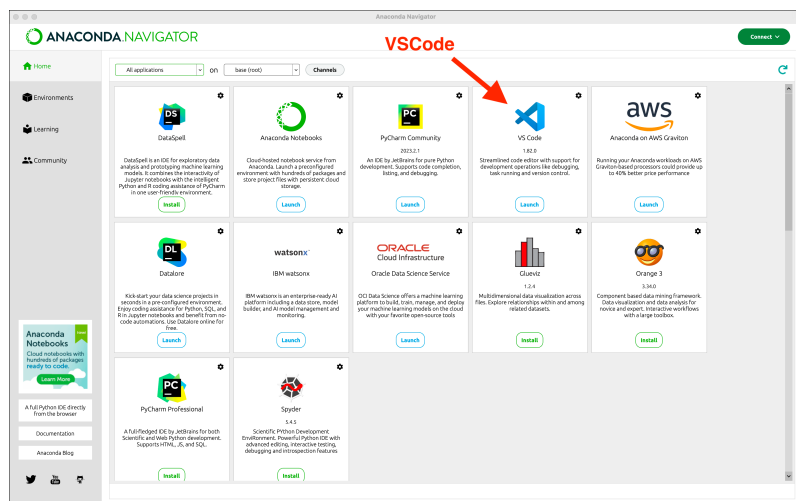
Install [VSCode](#). Visual Studio Code is a very nice IDE (Integrated Development Environment) made by Microsoft, available to all operating systems. Contrary to the title of this page, it is not absolutely necessary to use it, but I like VSCode, and as my student, so do you .

1.3 jupyter notebooks

We will code exclusively in Jupyter Notebooks. [Get acquainted with them](#). Make sure you can [point VSCode](#) to the Anaconda environment of your choice (“base” by default). Don't worry, this is easier than it sounds.

One failproof way of making sure VSCode uses the Anaconda installation is the following:

- Open Anaconda Navigator
- If you are using HUJI's computers, in “Environments”, choose “asgard”. If you are using your own computer, ignore this step.
- open VSCode from inside Anaconda Navigator (see image below).



Sometimes you will need to manually install the Jupyter extension on VSCode. In this case follow [this tutorial](#).

1.4 folder structure

You **NEED** to be comfortable with your computer's folder (or directory) structure. Where are files located? How to navigate through different folders? How is my stuff organized? If you don't feel **absolutely** comfortable with this, then read this, [Windows](#), [MacOS](#). If you use Linux then you surely know this stuff. **Make yourself a “time-series” folder** wherever you want, and have it backed up regularly (use Google Drive, Dropbox, do it manually, etc). “My dog deleted my files” is not an excuse.

2 numpy, pandas, matplotlib

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

The three lines above are the most common way you will start every project in this course.

- **numpy** = numerical python. This library has a ton useful mathematical functions, and most importantly, it has an object called **numpy array**, which is one of the most useful data structures we have for time series analysis.
- **pandas** is built upon numpy, and allows us to easily manipulate data stored in **dataframes**, a fancy name for a table.
- **pyplot** is a submodule of **matplotlib**, and allows us to beautifully plot data.

The best resource I know to get acquainted with all three packages is [Python Data Science Handbook](#), by [Jake VanderPlas](#). This is a free online book, with excellent step by step examples.

2.1 pandas

We will primarily use the Pandas package to deal with data. Pandas has become the standard Python tool to manipulate time series, and you should get acquainted with its basic usage. This course will provide you the opportunity to learn by example, but I'm sure we will only scratch the surface, and you'll be left with lots of questions.

I provide below a (non-comprehensive) list of useful tutorials, they are a good reference for the beginner and for the experienced user.

- [Python Data Science Handbook](#), by Jake VanderPlas
- [Data Wrangling with pandas Cheat Sheet](#)
- [Working with Dates and Times in Python](#)
- [Cheat Sheet: The pandas DataFrame Object](#)
- [YouTube tutorials](#) by Corey Schafer

2.2 pyplot

Matplotlib, and its submodule pyplot, are probably the most common Python plotting tool. Pyplot is both great and horrible:

- Great: you'll have absolutely full control of everything you want to plot. The sky is the limit.
- Horrible: you'll cry as you do it, because there is so much to know, and it is not the most friendly plotting package.

Pyplot is *object oriented*, so you will usually manipulate the **axes** object like this.

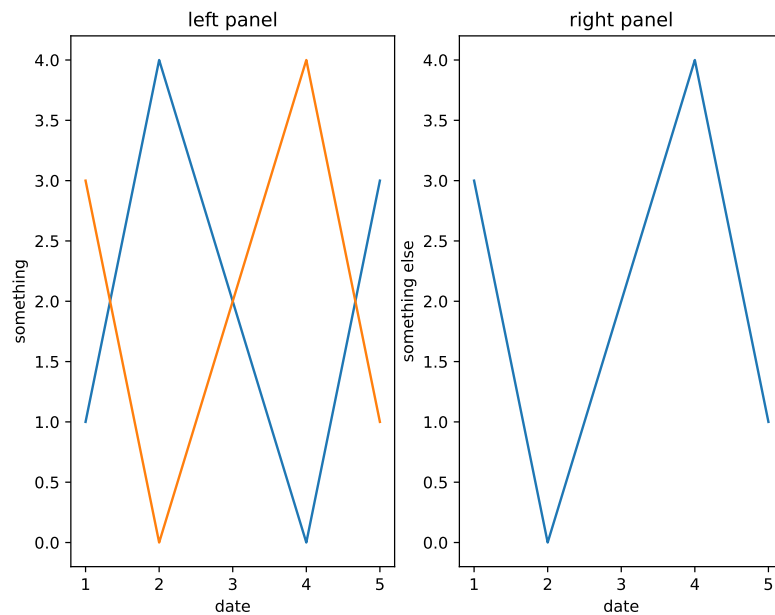
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [1, 4, 2, 0, 3]

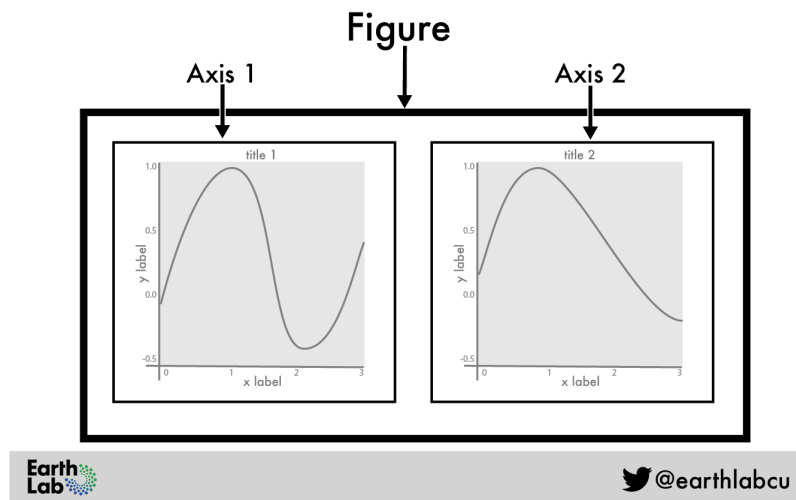
# Figure with two plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (8, 6))
# plot on the left
ax1.plot(x, y, color="tab:blue")
ax1.plot(x, y[:-1], color="tab:orange")
ax1.set(xlabel="date",
        ylabel="something",
        title="left panel")
# plot on the right
ax2.plot(x, y[:-1])
ax2.set(xlabel="date",
```

```
ylabel="something else",  
title="right panel")
```

```
[Text(0.5, 0, 'date'),  
Text(0, 0.5, 'something else'),  
Text(0.5, 1.0, 'right panel')]
```

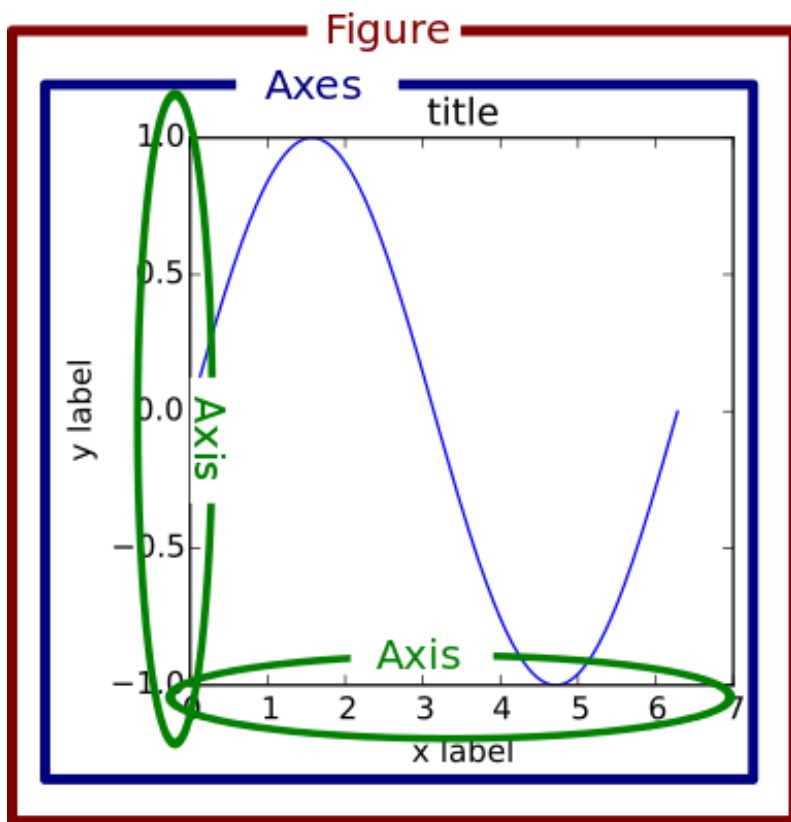


For the very beginners, you need to know that **figure** refers to the whole white canvas, and **axes** means the rectangle inside which something will be plotted:



The image above is good because it has 2 panels, and it's easy to understand what going on. Sadly, they mixed the two terms, axis and axes.

- **axes** is where the whole plot will be drawn. In the figure above it is the same as each panel.
- **axis** is each of the vertical and horizontal lines, where you have ticks and numbers.



If you are new to all this, I recommend that you go to:

- [Earth Lab's Introduction to Plotting in Python Using Matplotlib](#)
- [Jake VanderPlas's Python Data Science Handbook](#)

3 learn by example

Now that everything is installed, try to run the code below *before* the first lecture. Don't worry if you don't understand everything.

- If you manage to run everything without errors, this means that your computer is good to go!
- You might encounter a few problems. That's ok. Make a note and we will solve everything in the first lecture.

Let's make a first plot of real data. We will use NOAA's Global Monitoring Laboratory data on [Trends in Atmospheric Carbon Dioxide](#).

1. On your computer, open the program **Anaconda Navigator** (it may take a while to load).
2. Find the white box called **VS Code** and click **Launch**.
3. Now go to **File > Open Folder**, and open the folder you created for this course. VS Code may ask you if you trust the authors, and the answer is "yes" (it's your computer).
4. **File > New File**, and call it **example.ipynb**
5. You can start copying and pasting code from this website to your Jupyter Notebook. To run a cell, press Shift+Enter.
6. You may be asked to choose to Select Kernel. This is VS Code wanting to know which python installation to use. Click on "Python Environments", and then choose the option with the word **anaconda** in it.
7. That's all! Congratulations!

3.1 import packages

First, import packages to be used. They should all be already included in the Anaconda distribution you installed.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
```

3.2 load data

Load CO2 data into a Pandas dataframe. You can load it directly from the URL (option 1), or first download the CSV to your computer and then load it (option 2). The link to download the data directly from NOAA is [this](https://gml.noaa.gov/webdata/ccgg/trends/co2/co2_weekly_mlo.csv). If for some reason this doesn't work, download [here](#).

```
# option 1: load data directly from URL
# url = "https://gml.noaa.gov/webdata/ccgg/trends/co2/co2_weekly_mlo.csv"
# df = pd.read_csv(url,
#                   header=34,
#                   na_values=[-999.99]
#                   )

# option 2: download first (use the URL above and save it to your computer), then load csv
filename = "co2_weekly_mlo.csv"
df = pd.read_csv(filename,
                  comment='#', # will ignore rows starting with #
                  na_values=[-999.99] # substitute -999.99 for NaN (Not a Number), data not
                  )

# check how the dataframe (table) looks like
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 | 50.39 |
| 1 | 1974 | 5 | 26 | 1974.3986 | 332.95 | 6 | NaN | NaN | 50.05 |
| 2 | 1974 | 6 | 2 | 1974.4178 | 332.35 | 5 | NaN | NaN | 49.59 |
| 3 | 1974 | 6 | 9 | 1974.4370 | 332.20 | 7 | NaN | NaN | 49.64 |
| 4 | 1974 | 6 | 16 | 1974.4562 | 332.37 | 7 | NaN | NaN | 50.06 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 2566 | 2023 | 7 | 23 | 2023.5575 | 421.28 | 4 | 418.03 | 397.30 | 141.60 |

| | year | month | day | decimal | average | ndays | 1 year ago | 10 years ago | increase since 1800 |
|------|------|-------|-----|-----------|---------|-------|------------|--------------|---------------------|
| 2567 | 2023 | 7 | 30 | 2023.5767 | 420.83 | 6 | 418.10 | 396.80 | 141.69 |
| 2568 | 2023 | 8 | 6 | 2023.5959 | 420.02 | 6 | 417.36 | 395.65 | 141.41 |
| 2569 | 2023 | 8 | 13 | 2023.6151 | 418.98 | 4 | 417.25 | 395.24 | 140.89 |
| 2570 | 2023 | 8 | 20 | 2023.6342 | 419.31 | 2 | 416.64 | 395.22 | 141.71 |

3.3 dealing with dates

Create a new column called `date`, that combines the information from three separate columns: `year`, `month`, `day`.

```
# function to_datetime translates the full date into a pandas datetime object,
# that is, pandas knows this is a date, it's not just a string
df['date'] = pd.to_datetime(df[['year', 'month', 'day']])
# make 'date' column the dataframe index
df = df.set_index('date')
# now see if everything is ok
df
```

| | year | month | day | decimal | average | ndays | 1 year ago | 10 years ago | increase since 1800 |
|------------|------|-------|-----|-----------|---------|-------|------------|--------------|---------------------|
| date | | | | | | | | | |
| 1974-05-19 | 1974 | 5 | 19 | 1974.3795 | 333.37 | 5 | NaN | NaN | 50.39 |
| 1974-05-26 | 1974 | 5 | 26 | 1974.3986 | 332.95 | 6 | NaN | NaN | 50.05 |
| 1974-06-02 | 1974 | 6 | 2 | 1974.4178 | 332.35 | 5 | NaN | NaN | 49.59 |
| 1974-06-09 | 1974 | 6 | 9 | 1974.4370 | 332.20 | 7 | NaN | NaN | 49.64 |
| 1974-06-16 | 1974 | 6 | 16 | 1974.4562 | 332.37 | 7 | NaN | NaN | 50.06 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 2023-07-23 | 2023 | 7 | 23 | 2023.5575 | 421.28 | 4 | 418.03 | 397.30 | 141.60 |
| 2023-07-30 | 2023 | 7 | 30 | 2023.5767 | 420.83 | 6 | 418.10 | 396.80 | 141.69 |
| 2023-08-06 | 2023 | 8 | 6 | 2023.5959 | 420.02 | 6 | 417.36 | 395.65 | 141.41 |
| 2023-08-13 | 2023 | 8 | 13 | 2023.6151 | 418.98 | 4 | 417.25 | 395.24 | 140.89 |
| 2023-08-20 | 2023 | 8 | 20 | 2023.6342 | 419.31 | 2 | 416.64 | 395.22 | 141.71 |

3.4 first plot

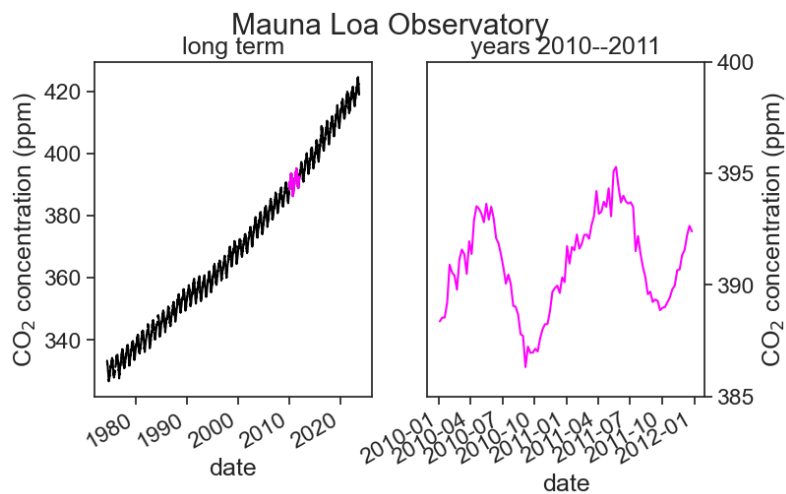
We are now ready for our first plot! Let's see the weekly CO2 average.

```
# %matplotlib widget
# uncomment the above line if you want dynamic control of the figure when using VSCode
fig, (ax1, ax2) = plt.subplots(1, 2, # 1 row, 2 columns
                               figsize=(8,5) # width, height, in inches
                              )

# left panel
ax1.plot(df['average'], color="black")
ax1.plot(df.loc['2010-01-01':'2011-12-31','average'], color="magenta")
ax1.set(xlabel="date",
        ylabel=r"CO$_2$ concentration (ppm)",
        title="long term");

# right panel
ax2.plot(df.loc['2010-01-01':'2011-12-31','average'], color="magenta")
ax2.set(xlabel="date",
        ylabel=r"CO$_2$ concentration (ppm)",
        ylim=[385, 400], # choose y limits
        yticks=np.arange(385, 401, 5), # choose ticks
        title="years 2010--2011");

# put ticks and label on the right for ax2
ax2.yaxis.tick_right()
ax2.yaxis.set_label_position("right")
# title above both panels
fig.suptitle("Mauna Loa Observatory")
# makes slanted dates
plt.gcf().autofmt_xdate()
```



3.5 first plot, v2.0

The dates in the x-label are not great. Let's try to make them prettier.

We need to import a few more packages first.

```
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
```

Now let's replot.

```
# %matplotlib widget
# uncomment the above line if you want dynamic control of the figure when using VSCode
fig, (ax1, ax2) = plt.subplots(1, 2, # 1 row, 2 columns
                               figsize=(8,5) # width, height, in inches
                               )

# left panel
ax1.plot(df['average'], color="black")
ax1.plot(df.loc['2010-01-01':'2011-12-31', 'average'], color="magenta")
ax1.set(xlabel="date",
        ylabel=r"CO$_2$ concentration (ppm)",
```

```

        title="long term");
# right panel
ax2.plot(df.loc['2010-01-01':'2011-12-31','average'], color="magenta")
ax2.set(xlabel="date",
        ylabel=r"CO$_2$ concentration (ppm)",
        ylim=[385, 400], # choose y limits
        yticks=np.arange(385, 401, 5), # choose ticks
        title="years 2010--2011");
# put ticks and label on the right for ax2
ax2.yaxis.tick_right()
ax2.yaxis.set_label_position("right")
# title above both panels
fig.suptitle("Mauna Loa Observatory", y=1.00)

locator = mdates.AutoDateLocator(minticks=3, maxticks=5)
formatter = mdates.ConciseDateFormatter(locator)
ax1.xaxis.set_major_locator(locator)
ax1.xaxis.set_major_formatter(formatter)

locator = mdates.AutoDateLocator(minticks=4, maxticks=5)
formatter = mdates.ConciseDateFormatter(locator)
ax2.xaxis.set_major_locator(locator)
ax2.xaxis.set_major_formatter(formatter)

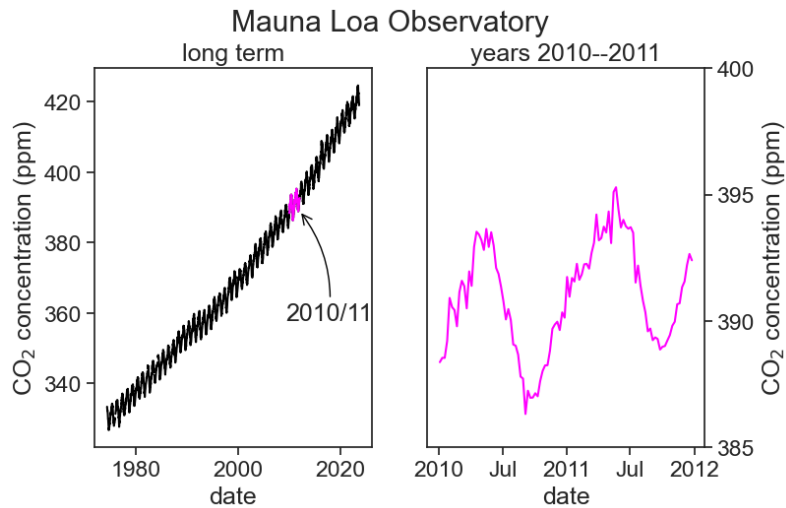
ax1.annotate(
    "2010/11",
    xy=('2011-12-25', 389), xycoords='data',
    xytext=(-10, -80), textcoords='offset points',
    arrowprops=dict(arrowstyle="->",
                    color="black",
                    connectionstyle="arc3,rad=0.2"))
fig.savefig("CO2-graph.png", dpi=300)

```

```

/var/folders/hc/jhnmlst937d27zzq9fhfks780000gn/T/ipykernel_10652/850389963.py:42: UserWarning:
  fig.savefig("CO2-graph.png", dpi=300)
/opt/anaconda3/lib/python3.9/site-packages/IPython/core/pylabtools.py:151: UserWarning: AutoDa
  fig.canvas.print_figure(bytes_io, **kw)

```



The dates on the horizontal axis are determined thus:

1. `locator = mdates.AutoDateLocator(minticks=3, maxticks=5)`
This determines the location of the ticks (between 3 and 5 ticks, whatever “works best”)
2. `ax1.xaxis.set_major_locator(locator)`
This actually puts the ticks in the positions determined above
3. `formatter = mdates.ConciseDateFormatter(locator)`
This says that the labels will be placed at the locations determined in 1.
4. `ax1.xaxis.set_major_formatter(formatter)`
Finally, labels are written down

The arrow is placed in the graph using `annotate`. It has a tricky syntax and a million options. Read [Jake VanderPlas's](#) excellent examples to learn more.

3.6 modifications

Let's change a lot of plotting options to see how things could be different.

```

sns.set(style="darkgrid")
sns.set_context("notebook")

# %matplotlib widget
# uncomment the above line if you want dynamic control of the figure when using VSCode
fig, (ax1, ax2) = plt.subplots(1, 2, # 1 row, 2 columns
                               figsize=(8,4) # width, height, in inches
                               )

# left panel
ax1.plot(df['average'], color="tab:blue")
ax1.plot(df.loc['2010-01-01':'2011-12-31','average'], color="tab:orange")
ax1.set(xlabel="date",
        ylabel=r"CO$_2$ concentration (ppm)",
        title="long term");

# right panel
ax2.plot(df.loc['2010-01-01':'2011-12-31','average'], color="tab:orange")
ax2.set(xlabel="date",
        ylim=[385, 400], # choose y limits
        yticks=np.arange(385, 401, 5), # choose ticks
        title="years 2010--2011");

# title above both panels
fig.suptitle("Mauna Loa Observatory", y=1.00)

locator = mdates.AutoDateLocator(minticks=3, maxticks=5)
formatter = mdates.ConciseDateFormatter(locator)
ax1.xaxis.set_major_locator(locator)
ax1.xaxis.set_major_formatter(formatter)

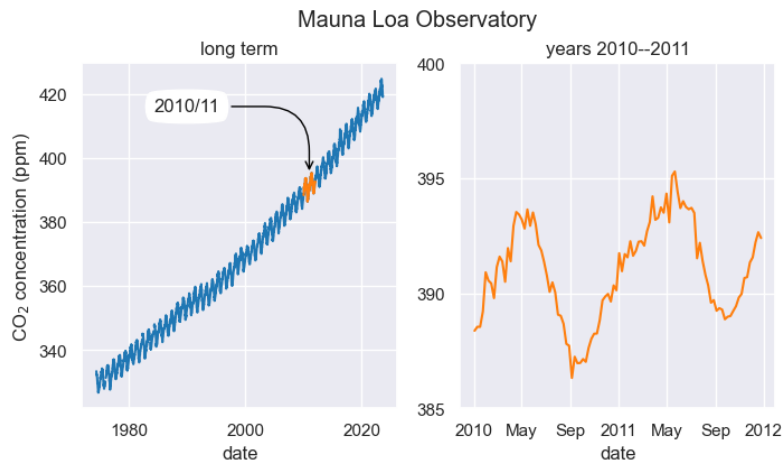
locator = mdates.AutoDateLocator(minticks=5, maxticks=8)
formatter = mdates.ConciseDateFormatter(locator)
ax2.xaxis.set_major_locator(locator)
ax2.xaxis.set_major_formatter(formatter)

ax1.annotate(
    "2010/11",
    xy=('2010-12-25', 395), xycoords='data',
    xytext=(-100, 40), textcoords='offset points',
    bbox=dict(boxstyle="round4,pad=.5", fc="white"),
    arrowprops=dict(arrowstyle="->"),

```

```
color="black",
connectionstyle="angle,angleA=0,angleB=-90,rad=40"))
```

```
Text(-100, 40, '2010/11')
```



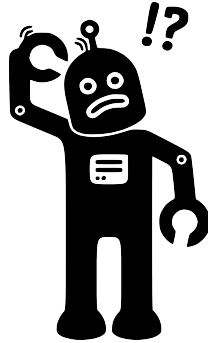
The main changes were:

1. Using the Seaborn package, we changed the fontsize and the overall plot style. [Read more.](#)
`sns.set(style="darkgrid")`
`sns.set_context("notebook")`
2. We changed the colors of the lineplots. To know what colors exist, [click here.](#)
3. The arrow annotation has a different style. [Read more.](#)

3.7 playing with the code

I encourage you to play with the code you just ran. An easy way of learning what each line does is to comment something out and see what changes in the output you see. If you feel brave, try to modify the code a little bit.

4 AI policy



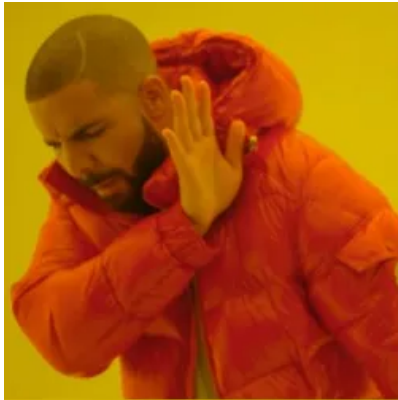
The guidelines below are an adaptation of [Ethan Mollick's extremely useful ideas on AI](#) as an assistant tool for teaching.

I EXPECT YOU to use LLMs (large language models) such as ChatGPT, Bing AI, Google Bard, or whatever else springs up since the time of this writing. You should familiarize yourself with the AI's capabilities and limitations.

Use LLMs to help you learn, chat with them about what you want to accomplish and learn from them how to do it. **Ask** your LLM what each part of the code means, copy and pasting blindly is unacceptable. You are here to learn.

Consider the following important points:

- Ultimately, you, the student, are responsible for the assignment.
- Acknowledge the use of AI in your assignment. Be transparent about your use of the tool and the extent of assistance it provided.



A.I
taking my job



Learning
to use A.I
and make
my job easier

Part II

resampling

5 motivation

Data from the [Israel Meteorological Service](#), IMS.

See the temperature at a weather station in Jerusalem, for the whole 2019 year. This is an interactive graph: to zoom in, play with the bottom panel.

```
alt.VConcatChart(...)
```

5.0.0.0.1 * discussion

The temperature fluctuates on various time scales, from daily to yearly. Let's think together a few questions we'd like to ask about the data above.

Now let's see precipitation data:

```
alt.VConcatChart(...)
```

5.0.0.0.2 * discussion

What would be interesting to know about precipitation?

We have not talked about what kind of data we have in our hands here. The csv file provided by the IMS looks like this:

| | Station | Date & Time (Winter) | Diffused radiation (W/m ²) | Global radiation (W/m ²) |
|-------|---------------------|----------------------|--|--------------------------------------|
| 0 | Jerusalem Givat Ram | 01/01/2019 00:00 | 0.0 | 0.0 |
| 1 | Jerusalem Givat Ram | 01/01/2019 00:10 | 0.0 | 0.0 |
| 2 | Jerusalem Givat Ram | 01/01/2019 00:20 | 0.0 | 0.0 |
| 3 | Jerusalem Givat Ram | 01/01/2019 00:30 | 0.0 | 0.0 |
| 4 | Jerusalem Givat Ram | 01/01/2019 00:40 | 0.0 | 0.0 |
| ... | ... | ... | ... | ... |
| 52549 | Jerusalem Givat Ram | 31/12/2019 22:20 | 0.0 | 0.0 |

| | Station | Date & Time (Winter) | Diffused radiation (W/m ²) | Global radiation (W/m ²) |
|-------|---------------------|----------------------|--|--------------------------------------|
| 52550 | Jerusalem Givat Ram | 31/12/2019 22:30 | 0.0 | 0.0 |
| 52551 | Jerusalem Givat Ram | 31/12/2019 22:40 | 0.0 | 0.0 |
| 52552 | Jerusalem Givat Ram | 31/12/2019 22:50 | 0.0 | 0.0 |
| 52553 | Jerusalem Givat Ram | 31/12/2019 23:00 | 0.0 | 0.0 |

We see that we have data points spaced out evenly every 10 minutes.

5.1 Challenges

Let's try to answer the following questions:

What is the mean temperature for each month?

First we have to divide temperature data by month, and then take the average for each month.
a possible solution

```
df_month = df['temperature'].resample('M').mean()
```

For each month, what is the mean of the daily maximum temperature? What about the minimum?

This is a bit trickier.

1. We need to find the maximum/minimum temperature for each day.
2. Only then we split the daily data by month and take the average.

a possible solution

```
df_day['max temp'] = df['temperature'].resample('D').max()
df_month['max temp'] = df_day['max temp'].resample('MS').mean()
```

What is the average night temperature for every season?
What about the day temperature?

1. We need to filter our data to contain only night times.
2. We need to divide rain data by seasons (3 months), and then take the mean for each season.

a possible solution

```
# filter only night data
df_night = df.loc[((df.index.hour < 6) | (df.index.hour >= 18))]
season_average_night_temp = df_night['temperature'].resample('Q').mean()
```

another possible solution

```
# filter using between_time
df_night = df.between_time('18:00', '06:00', inclusive='left')
season_average_night_temp = df_night['temperature'].resample('Q').mean()
```

What is the daily precipitation?

First we have to divide rain data by day, and then take the sum for each day.

a possible solution

```
daily_precipitation = df['rain'].resample('D').sum()
```

How much rain was there every month?

We have to divide rain data by month, and then sum the totals of each month.

a possible solution

```
monthly_precipitation = df['rain'].resample('M').sum()
```

How many rainy days were there each month?

1. We need to sum rain by day.
2. We need to count how many days are there each month where `rain > 0`.

a possible solution

```
daily_precipitation = df['rain'].resample('D').sum()
only_rainy_days = daily_precipitation.loc[daily_precipitation > 0]
rain_days_per_month = only_rainy_days.resample('M').count()
```

How many days, hours, and minutes were between the last rain of the season (Malkosh) to the first (Yoreh)?

1. We need to divide our data into two: `rainy_season_1` and `rainy_season_2`.
2. We need to find the time of the last rain in `rainy_season_1`.
3. We need to find the time of the first rain in `rainy_season_2`.
4. We need to compute the time difference between the two dates.

a possible solution

```
split_date = '2019-08-01'
rainy_season_1 = df[:split_date] # everything before split date
rainy_season_2 = df[split_date:] # everything after split date
malkosh = rainy_season_1['rain'].loc[rainy_season_1['rain'] > 0].last_valid_index()
yoreh = rainy_season_2['rain'].loc[rainy_season_2['rain'] > 0].first_valid_index()
dry_period = yoreh - malkosh
# extracting days, hours, and minutes
days = dry_period.days
hours = dry_period.components.hours
minutes = dry_period.components.minutes
print(f'The dry period of 2019 was {days} days, {hours} hours and {minutes} minutes.')
```

i What was the rainiest morning (6am-12pm) of the year?
Bonus, what about the rainiest night (6pm-6am)?

1. We need to filter our data to contain only morning times.
2. We need to sum rain by day.
3. We need to find the day with the maximum value.

a possible solution

```
# filter to only day data
morning_df = df.loc[((df.index.hour >= 6) & (df.index.hour < 18))]
morning_rain = morning_df['rain'].resample('D').sum()
rainiest_morning = morning_rain.idxmax()
# plot
morning_rain.plot()
plt.axvline(rainiest_morning, c='r', alpha=0.5, linestyle='--')
```

bonus solution

```
# filter to only night data
df_night = df.loc[((df.index.hour < 6) | (df.index.hour >= 18))]
# resampling night for each day is tricky because the date changes at 12:00. We can do this
# we shift the time back by 6 hours so all the data for the same night will have the same date
df_shifted = df_night.tshift(-6, freq='H')
night_rain = df_shifted['rain'].resample('D').sum()
rainiest_night = night_rain.idxmax()
# plot
night_rain.plot()
plt.axvline(rainiest_night, c='r', alpha=0.5, linestyle='--')
```

Note: this whole webpage is actually a Jupyter Notebook rendered as html. If you want to know how to make interactive graphs, go to the top of the page and click on “ Code”

Useful functions compatible with `pandas.resample()` can be found [here](#). The full list of resampling frequencies can be found [here](#).

6 resampling

We can only really understand how to calculate monthly means if we do it ourselves.

First, let's import a bunch of packages we need to use.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
```

Now we load the csv file for Jerusalem (2019), provided by the [IMS](#).

6.0.0.0.1 * discussion

We will go to the IMS website together and see what are the options available and how to download. If you just need the csv right away, download it [here](#).

- We substitute every occurrence of - for NaN (not a number, that is, the data is missing).
- We call the columns **Temperature** (°C) and **Rainfall** (mm) with more convenient names, since we will be using them a lot.
- We interpret the column **Date & Time (Winter)** as a date, saying to python that day comes first.
- We make **date** the index of the dataframe.


```

filename = "../archive/data/jerusalem2019.csv"
df = pd.read_csv(filename, na_values=['-'])
df.rename(columns={'Temperature (°C)': 'temperature',
                  'Rainfall (mm)': 'rain'}, inplace=True)
df['date'] = pd.to_datetime(df['Date & Time (Winter)'], dayfirst=True)
df = df.set_index('date')
df

```

| date | Station | Date & Time (Winter) | Diffused radiation (W/m ²) | Global rad |
|---------------------|---------------------|----------------------|--|------------|
| 2019-01-01 00:00:00 | Jerusalem Givat Ram | 01/01/2019 00:00 | 0.0 | 0.0 |
| 2019-01-01 00:10:00 | Jerusalem Givat Ram | 01/01/2019 00:10 | 0.0 | 0.0 |
| 2019-01-01 00:20:00 | Jerusalem Givat Ram | 01/01/2019 00:20 | 0.0 | 0.0 |
| 2019-01-01 00:30:00 | Jerusalem Givat Ram | 01/01/2019 00:30 | 0.0 | 0.0 |
| 2019-01-01 00:40:00 | Jerusalem Givat Ram | 01/01/2019 00:40 | 0.0 | 0.0 |
| ... | ... | ... | ... | ... |
| 2019-12-31 22:20:00 | Jerusalem Givat Ram | 31/12/2019 22:20 | 0.0 | 0.0 |
| 2019-12-31 22:30:00 | Jerusalem Givat Ram | 31/12/2019 22:30 | 0.0 | 0.0 |
| 2019-12-31 22:40:00 | Jerusalem Givat Ram | 31/12/2019 22:40 | 0.0 | 0.0 |
| 2019-12-31 22:50:00 | Jerusalem Givat Ram | 31/12/2019 22:50 | 0.0 | 0.0 |
| 2019-12-31 23:00:00 | Jerusalem Givat Ram | 31/12/2019 23:00 | 0.0 | 0.0 |

With `resample` it's easy to compute monthly averages. Resample by itself only divides the data into buckets (in this case monthly buckets), and waits for a further instruction. Here, the next instruction is `mean`.

```

df_month = df['temperature'].resample('M').mean()
df_month

```

| date | |
|------------|-----------|
| 2019-01-31 | 9.119937 |
| 2019-02-28 | 9.629812 |
| 2019-03-31 | 10.731571 |
| 2019-04-30 | 14.514329 |
| 2019-05-31 | 22.916894 |
| 2019-06-30 | 23.587361 |
| 2019-07-31 | 24.019403 |

```

2019-08-31    24.050822
2019-09-30    22.313287
2019-10-31    20.641868
2019-11-30    17.257153
2019-12-31    11.224131
Freq: M, Name: temperature, dtype: float64

```

Instead of M for month, which other options do I have? The full list can be [found here](#), but the most commonly used are:

```

M          month end frequency
MS         month start frequency
A          year end frequency
AS, YS     year start frequency
D          calendar day frequency
H          hourly frequency
T, min     minutely frequency
S          secondly frequency

```

The results we got for the monthly means were given as a pandas series, not dataframe. Let's correct this:

```

df_month = (df['temperature'].resample('M')          # resample by month
            .mean()                                # take the mean
            .to_frame('mean temp')                  # make output a dataframe
            )
df_month

```

| | mean temp |
|------------|-----------|
| 2019-01-31 | 9.119937 |
| 2019-02-28 | 9.629812 |
| 2019-03-31 | 10.731571 |
| 2019-04-30 | 14.514329 |
| 2019-05-31 | 22.916894 |
| 2019-06-30 | 23.587361 |
| 2019-07-31 | 24.019403 |
| 2019-08-31 | 24.050822 |

| | mean temp |
|------------|-----------|
| date | |
| 2019-09-30 | 22.313287 |
| 2019-10-31 | 20.641868 |
| 2019-11-30 | 17.257153 |
| 2019-12-31 | 11.224131 |

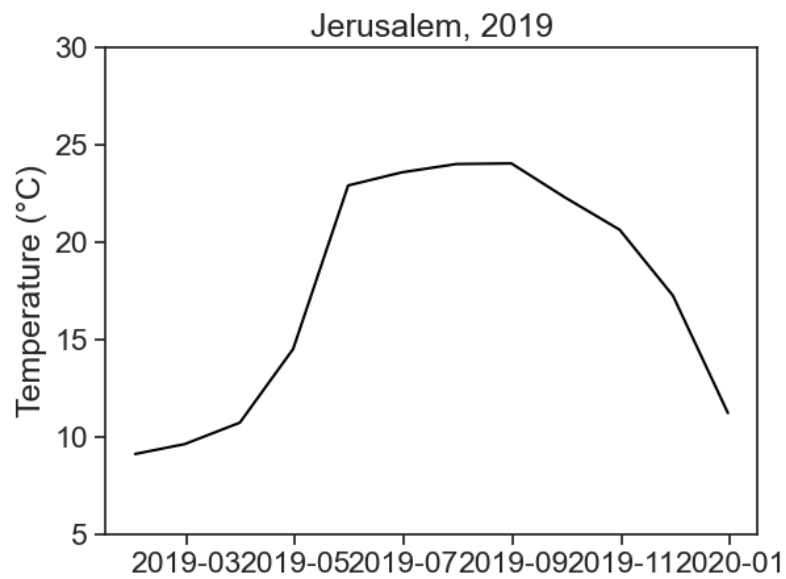
6.0.0.0.2 * hot tip

Sometimes, a line of code can get too long and messy. In the code above, we broke line for every step, which makes the process so much cleaner. We **highly** advise you to do the same. **Attention:** This trick works as long as all the elements are inside the same parenthesis.

Now it's time to plot!

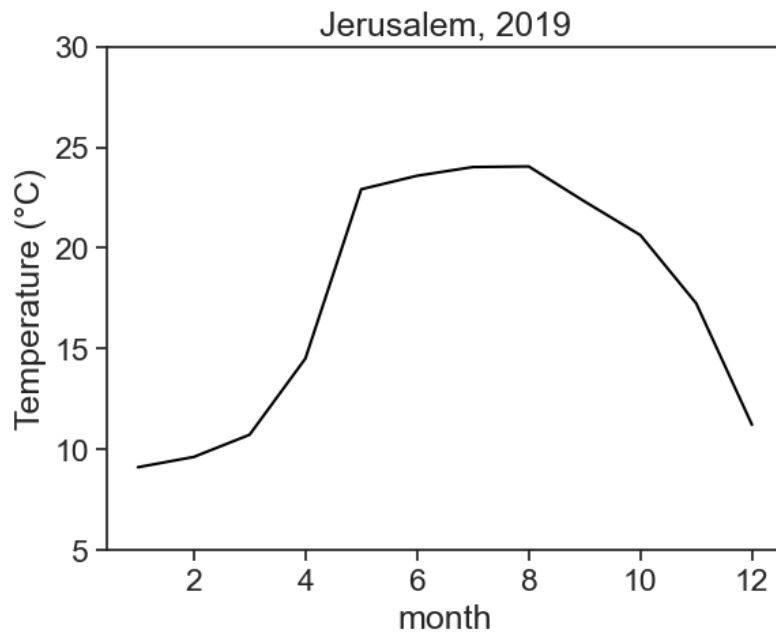
```
fig, ax = plt.subplots()
ax.plot(df_month['mean temp'], color='black')
ax.set(ylabel='Temperature (°C)',
       yticks=np.arange(5,35,5),
       title="Jerusalem, 2019")
```

```
[Text(0, 0.5, 'Temperature (°C)'),
 [<matplotlib.axis.YTick at 0x7faf784c6d60>,
  <matplotlib.axis.YTick at 0x7faf7843a220>,
  <matplotlib.axis.YTick at 0x7faf784c62b0>,
  <matplotlib.axis.YTick at 0x7faf784f3400>,
  <matplotlib.axis.YTick at 0x7faf784f3760>,
  <matplotlib.axis.YTick at 0x7faf784fa5b0>],
 Text(0.5, 1.0, 'Jerusalem, 2019')]
```



The dates in the horizontal axis are not great. An easy fix is to use the month numbers instead of dates.

```
fig, ax = plt.subplots()
ax.plot(df_month.index.month, df_month['mean temp'], color='black')
ax.set(xlabel="month",
      ylabel='Temperature (°C)',
      yticks=np.arange(5,35,5),
      title="Jerusalem, 2019",);
```



6.0.0.0.3 * discussion

When you have datetime as the dataframe index, you don't need to give the function `plot` two arguments, date and values. You can just tell `plot` to use the column you want, the function will take the dates by itself.

What does this line mean?

```
df_month['mean temp'].index.month
```

Print on the screen the following, and see yourself what each thing is:

- `df_month`
- `df_month.index`
- `df_month.index.month`
- `df_month.index.day`

We're done! Congratulations :)

Now we need to calculate the average minimum/maximum daily temperatures. We start by creating an empty dataframe.

```
df_day = pd.DataFrame()
```

Now resample data by day (D), and take the min/max of each day.

```
df_day['min temp'] = df['temperature'].resample('D').min()
df_day['max temp'] = df['temperature'].resample('D').max()
df_day
```

| | min temp | max temp |
|------------|----------|----------|
| date | | |
| 2019-01-01 | 7.5 | 14.1 |
| 2019-01-02 | 6.6 | 11.5 |
| 2019-01-03 | 6.3 | 10.7 |
| 2019-01-04 | 6.6 | 14.6 |
| 2019-01-05 | 7.0 | 11.4 |
| ... | ... | ... |
| 2019-12-27 | 4.4 | 7.4 |
| 2019-12-28 | 6.6 | 10.3 |
| 2019-12-29 | 8.1 | 12.5 |
| 2019-12-30 | 6.9 | 13.0 |
| 2019-12-31 | 5.2 | 13.3 |

The next step is to calculate the average minimum/maximum for each month. This is similar to what we did above.

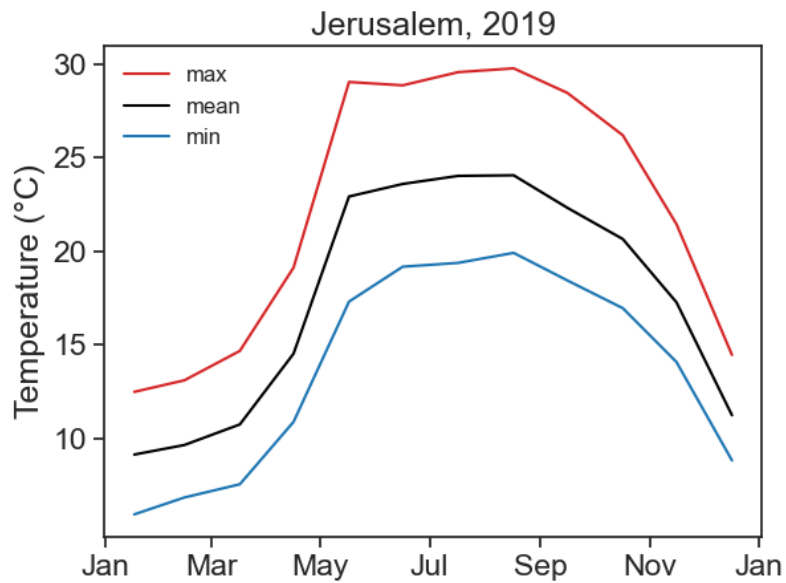
```
df_month['min temp'] = df_day['min temp'].resample('M').mean()
df_month['max temp'] = df_day['max temp'].resample('M').mean()
df_month
```

| | mean temp | min temp | max temp |
|------------|-----------|-----------|-----------|
| date | | | |
| 2019-01-31 | 9.119937 | 5.922581 | 12.470968 |
| 2019-02-28 | 9.629812 | 6.825000 | 13.089286 |
| 2019-03-31 | 10.731571 | 7.532258 | 14.661290 |
| 2019-04-30 | 14.514329 | 10.866667 | 19.113333 |
| 2019-05-31 | 22.916894 | 17.296774 | 29.038710 |

| date | mean temp | min temp | max temp |
|------------|-----------|-----------|-----------|
| 2019-06-30 | 23.587361 | 19.163333 | 28.860000 |
| 2019-07-31 | 24.019403 | 19.367742 | 29.564516 |
| 2019-08-31 | 24.050822 | 19.903226 | 29.767742 |
| 2019-09-30 | 22.313287 | 18.430000 | 28.456667 |
| 2019-10-31 | 20.641868 | 16.945161 | 26.190323 |
| 2019-11-30 | 17.257153 | 14.066667 | 21.436667 |
| 2019-12-31 | 11.224131 | 8.806452 | 14.448387 |

Let's plot...

```
fig, ax = plt.subplots()
ax.plot(df_month['max temp'], color='tab:red', label='max')
ax.plot(df_month['mean temp'], color='black', label='mean')
ax.plot(df_month['min temp'], color='tab:blue', label='min')
ax.set(ylabel='Temperature (°C)',
       yticks=np.arange(10,35,5),
       title="Jerusalem, 2019")
ax.xaxis.set_major_locator(mdates.MonthLocator(range(1, 13, 2), bymonthday=15))
date_form = DateFormatter("%b")
ax.xaxis.set_major_formatter(date_form)
ax.legend(fontsize=12, frameon=False);
```



Voilà! You made a beautiful graph!

6.0.0.0.4 * discussion

This time we did not put month numbers in the horizontal axis, we now have month names. How did we do this black magic, you ask? See lines 8–10 above. Matplotlib gives you absolute power over what to put in the axis, if you can only know how to tell it to... Wanna know more? [Click here](#).

7 upsampling

In the previous chapter, we resampled from fine temporal resolution to a coarser one. This is also called **downsampling**. We will learn the **upsampling** now: how to go from coarse data to a finer scale.

Sadly, there is no free lunch, and we just can't get data that was not measured. What to do then?

It's best to consider a practical example.

We want to calculate the daily potential evapotranspiration using [Penman's equation](#). Part of the calculation involves characterizing the energy budget on soil surface. When direct solar radiation measurements are not available, we can estimate the energy balance by knowing the “cloudless skies mean solar radiation”, R_{so} . This is the amount of energy (MJ/m²/d) that hits the surface, assuming no clouds. This radiation depends on the season and on the latitude you are. For Israel, located at latitude 32° N, we can use the following data for 30°:

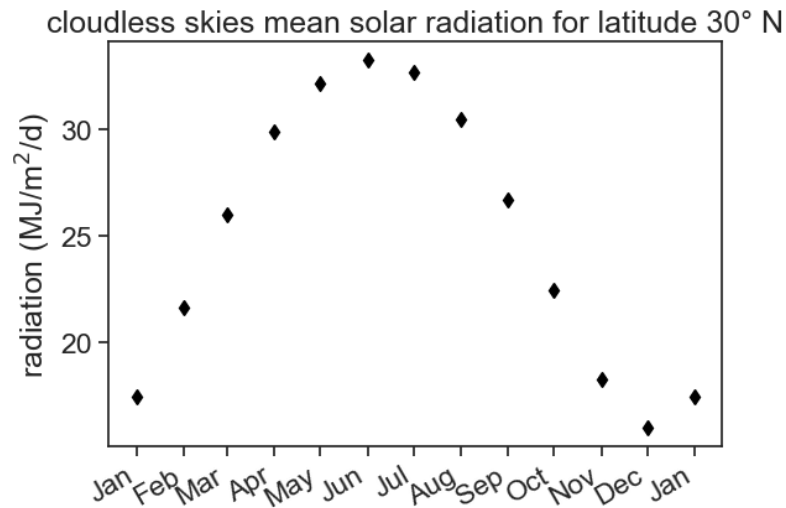
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters

dates = pd.date_range(start='2021-01-01', periods=13, freq='MS')
values = [17.46, 21.65, 25.96, 29.85, 32.11, 33.20, 32.66, 30.44, 26.67, 22.48, 18.30, 16.04]
df = pd.DataFrame({'date': dates, 'radiation': values})
df = df.set_index('date')
```

df

| radiation | |
|------------|-------|
| date | |
| 2021-01-01 | 17.46 |
| 2021-02-01 | 21.65 |
| 2021-03-01 | 25.96 |
| 2021-04-01 | 29.85 |
| 2021-05-01 | 32.11 |
| 2021-06-01 | 33.20 |
| 2021-07-01 | 32.66 |
| 2021-08-01 | 30.44 |
| 2021-09-01 | 26.67 |
| 2021-10-01 | 22.48 |
| 2021-11-01 | 18.30 |
| 2021-12-01 | 16.04 |
| 2022-01-01 | 17.46 |

```
fig, ax = plt.subplots()
ax.plot(df['radiation'], color='black', marker='d', linestyle='None')
ax.set(ylabel=r'radiation (MJ/m$^2$/d)',
       title="cloudless skies mean solar radiation for latitude 30° N")
ax.xaxis.set_major_locator(mdates.MonthLocator())
date_form = DateFormatter("%b")
ax.xaxis.set_major_formatter(date_form)
plt.gcf().autofmt_xdate() # makes slanted dates
```



We only have 12 values for the whole year, and we can't use this dataframe to compute daily ET. We need to upsample!

In the example below, we resample the monthly data into daily data, and do nothing else. Pandas doesn't know what to do with the new points, so it fills them with NaN.

```
df_nan = df['radiation'].resample('D').asfreq().to_frame()
df_nan.head(33)
```

| | radiation |
|------------|-----------|
| date | |
| 2021-01-01 | 17.46 |
| 2021-01-02 | NaN |
| 2021-01-03 | NaN |
| 2021-01-04 | NaN |
| 2021-01-05 | NaN |
| 2021-01-06 | NaN |
| 2021-01-07 | NaN |
| 2021-01-08 | NaN |
| 2021-01-09 | NaN |
| 2021-01-10 | NaN |
| 2021-01-11 | NaN |
| 2021-01-12 | NaN |

| | radiation |
|------------|-----------|
| date | |
| 2021-01-13 | NaN |
| 2021-01-14 | NaN |
| 2021-01-15 | NaN |
| 2021-01-16 | NaN |
| 2021-01-17 | NaN |
| 2021-01-18 | NaN |
| 2021-01-19 | NaN |
| 2021-01-20 | NaN |
| 2021-01-21 | NaN |
| 2021-01-22 | NaN |
| 2021-01-23 | NaN |
| 2021-01-24 | NaN |
| 2021-01-25 | NaN |
| 2021-01-26 | NaN |
| 2021-01-27 | NaN |
| 2021-01-28 | NaN |
| 2021-01-29 | NaN |
| 2021-01-30 | NaN |
| 2021-01-31 | NaN |
| 2021-02-01 | 21.65 |
| 2021-02-02 | NaN |

7.1 Forward/Backward fill

We can forward/backward fill these NaNs:

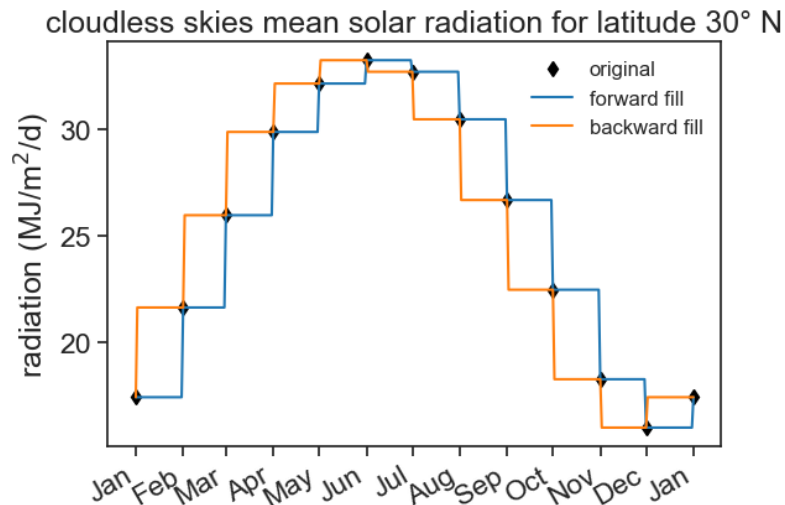
```
df_forw = df['radiation'].resample('D').ffill().to_frame()
df_back = df['radiation'].resample('D').bfill().to_frame()

fig, ax = plt.subplots()
ax.plot(df['radiation'], color='black', marker='d', linestyle='None', label="original")
ax.plot(df_forw['radiation'], color='tab:blue', label="forward fill")
ax.plot(df_back['radiation'], color='tab:orange', label="backward fill")
ax.set(ylabel=r'radiation (MJ/m$^2$/d)',
       title="cloudless skies mean solar radiation for latitude 30° N")
```

```

ax.legend(frameon=False, fontsize=12)
ax.xaxis.set_major_locator(mdates.MonthLocator())
date_form = DateFormatter("%b")
ax.xaxis.set_major_formatter(date_form)
plt.gcf().autofmt_xdate() # makes slanted dates

```



This does the job, but I want something better, not step functions. The radiation should vary smoothly from day to day. Let's use interpolation.

7.2 Interpolation

```

df_linear = df['radiation'].resample('D').interpolate(method='time').to_frame()
df_cubic = df['radiation'].resample('D').interpolate(method='cubic').to_frame()

```

```

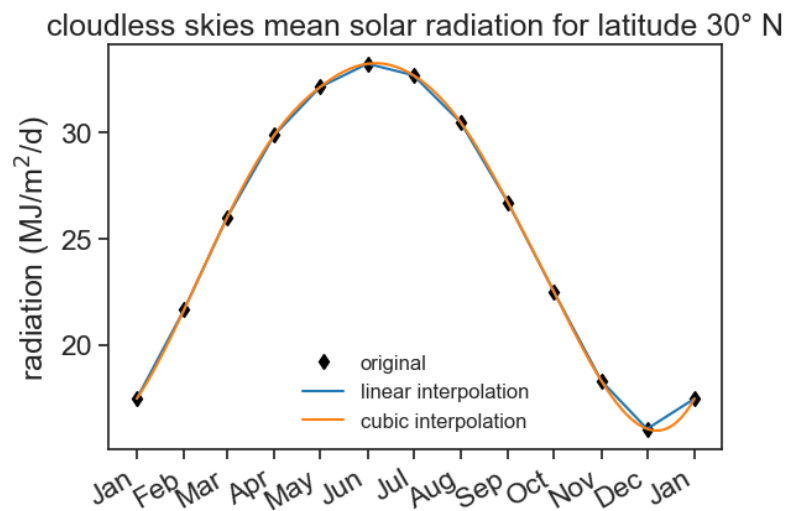
fig, ax = plt.subplots()
ax.plot(df['radiation'], color='black', marker='d', linestyle='None', label="original")
ax.plot(df_linear['radiation'], color='tab:blue', label="linear interpolation")
ax.plot(df_cubic['radiation'], color='tab:orange', label="cubic interpolation")
ax.set(ylabel=r'radiation (MJ/m$^2$/d)',
       title="cloudless skies mean solar radiation for latitude 30° N")

```

```

ax.legend(frameon=False, fontsize=12)
ax.xaxis.set_major_locator(mdates.MonthLocator())
date_form = DateFormatter("%b")
ax.xaxis.set_major_formatter(date_form)
plt.gcf().autofmt_xdate() # makes slanted dates

```



There are many ways to fill NaNs and to interpolate. A nice detailed guide can be [found here](#).

8 interpolation

Interpolation is the act of getting data you don't have from data you already have. We used some interpolation when upsampling, and now it is time to talk about it a little bit more in depth.

There is no one correct way of interpolating, the method you use depends in the end on what you want to accomplish, what are your (hidden or explicit) assumptions, etc. Let's see a few examples.

9 FAQ

This is called [anchored offset](#). One possible use to it is to calculate statistics according to the hydrological year that, for example, ends in September.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters

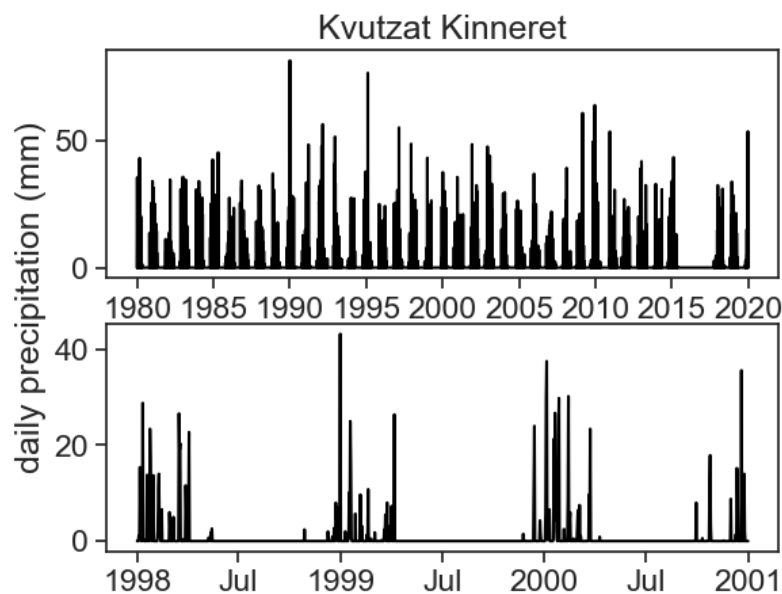
filename = "../archive/data/Kinneret_Kvuza_daily_rainfall.csv"
df = pd.read_csv(filename, na_values=['-'])
df.rename(columns={'Date': 'date',
                  'Daily Rainfall (mm)': 'rain'}, inplace=True)
df['date'] = pd.to_datetime(df['date'], dayfirst=True)
df = df.set_index('date')
df = df.resample('D').asfreq().fillna(0) # asfreq = replace
df
```

| | Station | rain |
|------------|--------------------------------|------|
| date | | |
| 1980-01-02 | Kinneret Kvuza 09/1977-08/2023 | 0.0 |
| 1980-01-03 | 0 | 0.0 |
| 1980-01-04 | 0 | 0.0 |
| 1980-01-05 | Kinneret Kvuza 09/1977-08/2023 | 35.5 |
| 1980-01-06 | Kinneret Kvuza 09/1977-08/2023 | 2.2 |
| ... | ... | ... |
| 2019-12-26 | Kinneret Kvuza 09/1977-08/2023 | 39.4 |

| | Station | rain |
|------------|--------------------------------|------|
| date | | |
| 2019-12-27 | Kinneret Kvuza 09/1977-08/2023 | 5.2 |
| 2019-12-28 | Kinneret Kvuza 09/1977-08/2023 | 1.6 |
| 2019-12-29 | 0 | 0.0 |
| 2019-12-30 | Kinneret Kvuza 09/1977-08/2023 | 0.1 |

```
fig, ax = plt.subplots(2,1)
ax[0].plot(df['rain'], color='black')
ax[1].plot(df.loc['1998':'2000', 'rain'], color='black')
locator = mdates.AutoDateLocator(minticks=4, maxticks=8)
formatter = mdates.ConciseDateFormatter(locator)
ax[1].xaxis.set_major_locator(locator)
ax[1].xaxis.set_major_formatter(formatter)
fig.text(0.02, 0.5, 'daily precipitation (mm)', va='center', rotation='vertical')
ax[0].set_title("Kvutzat Kinneret")
```

Text(0.5, 1.0, 'Kvutzat Kinneret')



We see a marked dry season during the summer, so let's assume the Hydrological Year ends in September.

```
df_year = df.resample('A-SEP').sum()
df_year = df_year.loc['1980':'2003']
df_year
```

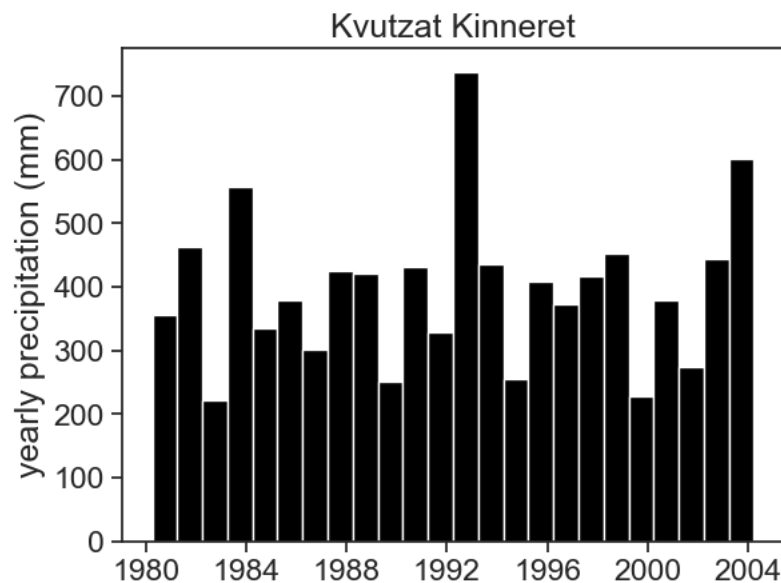
/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_94063/2047090134.py:1: FutureWarning

```
df_year = df.resample('A-SEP').sum()
```

| | rain |
|------------|-------|
| date | |
| 1980-09-30 | 355.5 |
| 1981-09-30 | 463.1 |
| 1982-09-30 | 221.7 |
| 1983-09-30 | 557.1 |
| 1984-09-30 | 335.3 |
| 1985-09-30 | 379.8 |
| 1986-09-30 | 300.7 |
| 1987-09-30 | 424.7 |
| 1988-09-30 | 421.6 |
| 1989-09-30 | 251.6 |
| 1990-09-30 | 432.5 |
| 1991-09-30 | 328.3 |
| 1992-09-30 | 738.4 |
| 1993-09-30 | 434.9 |
| 1994-09-30 | 255.4 |
| 1995-09-30 | 408.6 |
| 1996-09-30 | 373.0 |
| 1997-09-30 | 416.2 |
| 1998-09-30 | 451.9 |
| 1999-09-30 | 227.8 |
| 2000-09-30 | 378.9 |
| 2001-09-30 | 273.9 |
| 2002-09-30 | 445.2 |
| 2003-09-30 | 602.4 |

```
fig, ax = plt.subplots()
ax.bar(df_year.index, df_year['rain'], color='black',
      width=365)
ax.set_ylabel("yearly precipitation (mm)")
ax.set_title("Kvutzat Kinneret")
```

```
Text(0.5, 1.0, 'Kvutzat Kinneret')
```



9.1 When upsampling, how to fill missing values with zero?

We did that in the example above, like this:

```
df = df.resample('D').asfreq().fillna(0) # asfreq = replace
```

Part III

smoothing

10 motivation

This is the temperature for the Yatir Forest (Shani station, see [map](#)), between 2 and 5 of January 2022. Data is in intervals of 10 minutes, and was downloaded from the Israel Meteorological Service.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import datetime as dt
import matplotlib.ticker as ticker
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
import requests
import json
# %matplotlib widget

# useful links
# https://ims.gov.il/he/ObservationDataAPI
# https://ims.gov.il/sites/default/files/2021-09/API%20explanation.pdf
# https://ims.gov.il/sites/default/files/2022-04/Python%20API%20example.pdf

with open('IMS-token.txt', 'r') as file:
    TOKEN = file.readline()
STATION_NUM = 28 # 28 = "SHANI"
DATA = 10 # 10 = TDmax (max temperature)
start = "2022/01/01"
end = "2022/02/01"
url = f"https://api.ims.gov.il/v1/envista/stations/{STATION_NUM}/data/{DATA}/?from={start}&t
```

```

headers = {'Authorization': f'ApiToken {TOKEN}'}
response = requests.request("GET", url, headers=headers)
data= json.loads(response.text.encode('utf8'))

df = pd.json_normalize(data['data'],record_path=['channels'], meta=['datetime'])
df['date'] = pd.to_datetime(df['datetime'])
df = df.set_index('date')
# df

# dirty trick to have dates in the middle of the 24-hour period
# make minor ticks in the middle, put the labels there!
# from https://matplotlib.org/stable/gallery/ticks/centered_ticklabels.html

def centered_dates(ax):
    date_form = DateFormatter("%d %b") # %d 3-letter-Month
    # major ticks at midnight, every day
    ax.xaxis.set_major_locator(mdates.DayLocator(interval=1))
    ax.xaxis.set_major_formatter(date_form)
    # minor ticks at noon, every day
    ax.xaxis.set_minor_locator(mdates.HourLocator(byhour=[12]))
    # erase major tick labels
    ax.xaxis.set_major_formatter(ticker.NullFormatter())
    # set minor tick labels as define above
    ax.xaxis.set_minor_formatter(date_form)
    # completely erase minor ticks, center tick labels
    for tick in ax.xaxis.get_minor_ticks():
        tick.tick1line.set_markersize(0)
        tick.tick2line.set_markersize(0)
        tick.label1.set_horizontalalignment('center')

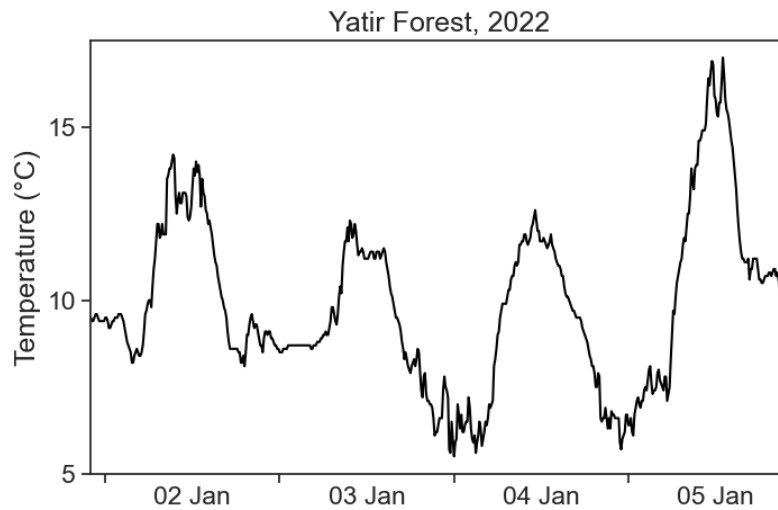
fig, ax = plt.subplots(figsize=(8,5))
start = "2022-01-02"
end = "2022-01-05"
df = df.loc[start:end]
ax.plot(df['value'], color='black')
ax.set(ylim=[5, 17.5],
        xlim=[df.index[0], df.index[-1]],
        ylabel="Temperature (°C)",

```

```

    title="Yatir Forest, 2022",
    yticks=[5,10,15])
centered_dates(ax)

```



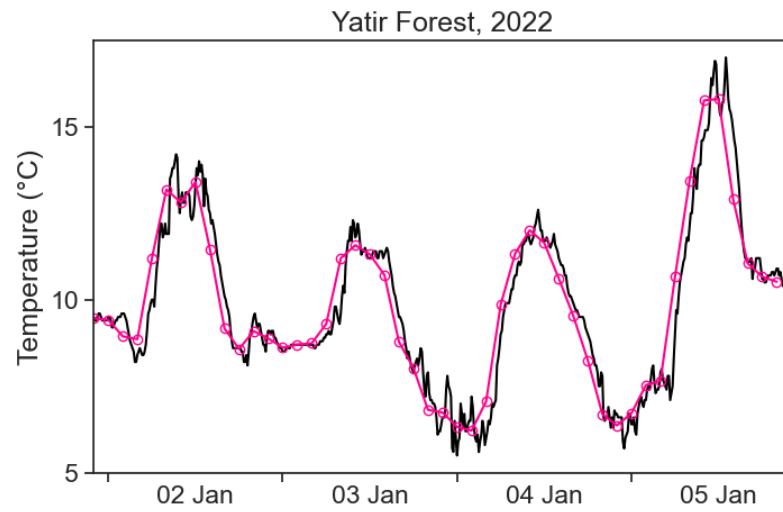
We see that the temperature curve has a rough profile. Can we find ways of getting smoother curves?

We learned how to average over a window with `resample`. Let's try that for a 2-hour window:

```

fig, ax = plt.subplots(figsize=(8,5))
ax.plot(df['value'], color='black')
ax.plot(df['value'].resample('2H').mean(),
        color='xkcd:hot pink', ls='-',
        marker="o", mfc="None")
ax.set(ylim=[5, 17.5],
        xlim=[df.index[0], df.index[-1]],
        ylabel="Temperature (°C)",
        title="Yatir Forest, 2022",
        yticks=[5,10,15])
centered_dates(ax)

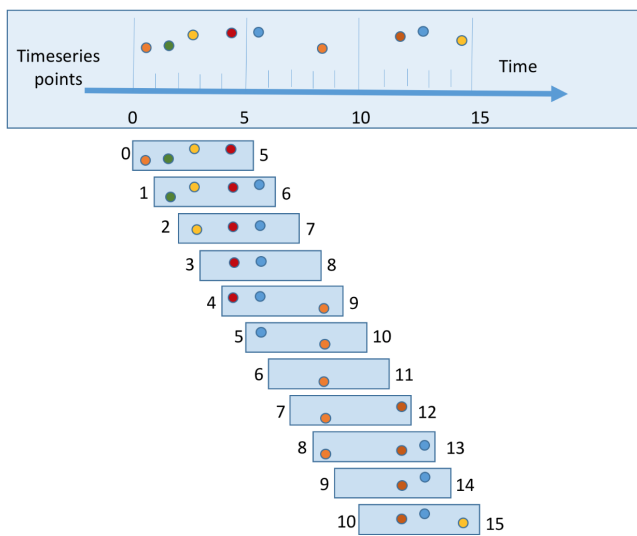
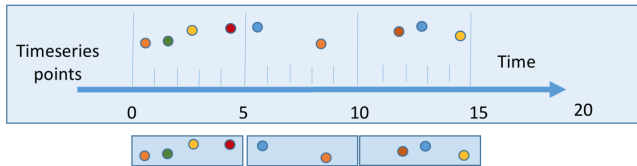
```



The temperature profile now is much smoother, but when using `resample`, we lost temporal resolution. Our original data had 10-minute frequency, and now we have a 2-hour frequency.

How can we get a smoother curve without losing resolution?

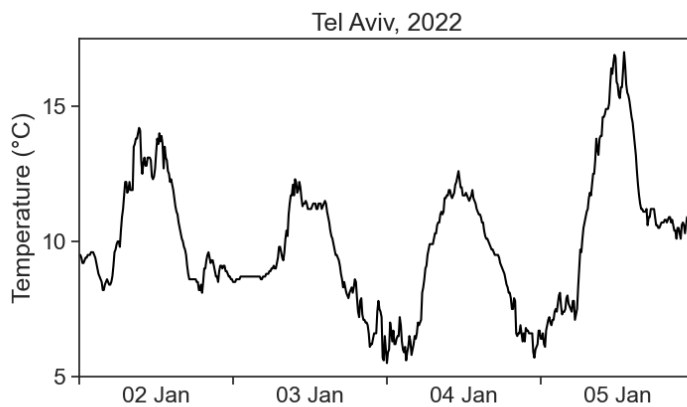
10.1 Tumbling vs Sliding



11 sliding window

Sliding windows of different shapes (kernels)

This is the temperature for the Yatir Forest, 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?

Convolution is a fancy word for averaging a time series using a sliding 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.

11.1 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).

11.2 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

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

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

11.3 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?

11.3.1 gaussian

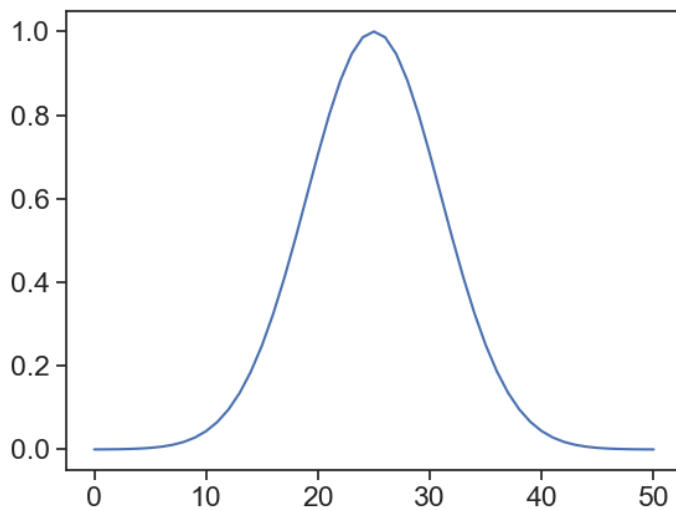
```
(
df['temperature'].rolling(window=window_width,
                           center=True,
                           win_type="gaussian")
```

```
)  
    .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)
```

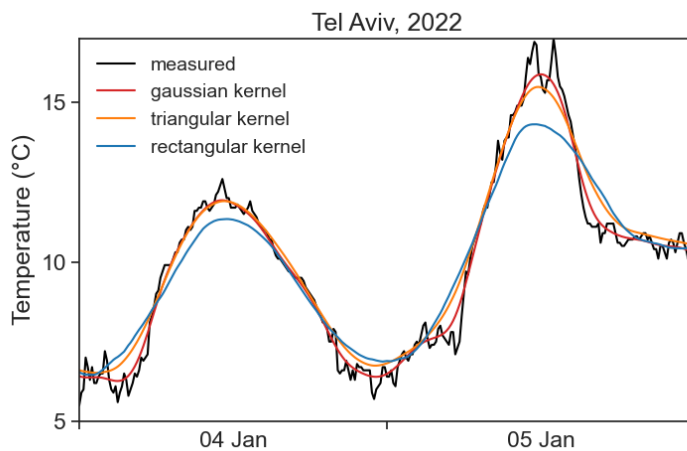
11.3.2 triangular

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()  
)
```

11.4 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.



One important question you need to ask is: what are the time scales associated with the processes I'm interested in? For example, if I'm interested in the daily temperature pattern, getting rid of 1-minute-long fluctuations would probably be ok.

On the other hand, if we were to smooth the signal so much that all that can be seen are the temperature changes between summer and winter, then my smoothing got out of hand, and I threw away the very process I wanted to study.

All this is to say that you need to know in advance a few things about the system you are studying, otherwise you can't know what is "noise" that can be smoothed away.

12 LOESS

13 a perfect smoother

Source: Eilers (2003)

[GitHub repository](#)

Noisy series y of length m .

The smoothed series is called z .

We have conflicting interests:

- we want a z series “as smooth as possible”.
- however, the smoother z is, the farthest from y it will be (low fidelity).

Roughness:

$$R = \sum_i (z_i - z_{i-1})^2$$

Fit to data:

$$S = \sum_i (y_i - z_i)^2$$

Cost functional to be minimized:

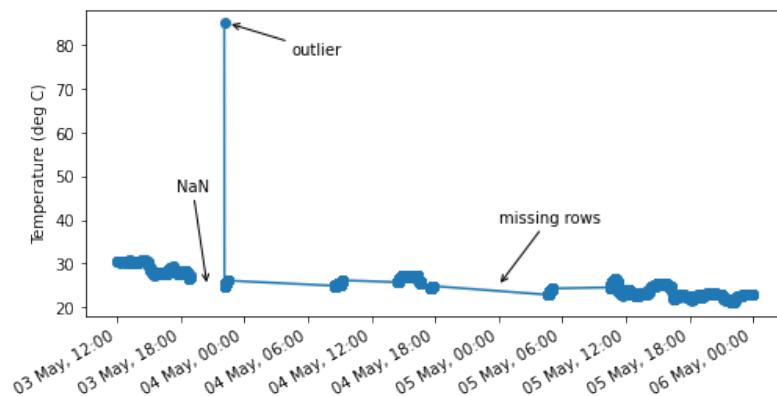
$$Q = S + \lambda R$$

Part IV

outliers and gaps

14 motivation

Outliers are observations significantly different from all other observations. Consider, for example, this temperature graph:



While most measured points are between 20 and 30 °C, there is obviously something very wrong with the one data point above 80 °C.

How could such a thing come about? This could be the result of non-natural causes, such as measurement errors, wrong data collection, or wrong data entry. On the other hand, this point could have natural sources, such as a very hot spark flying next to the temperature sensor.

Identifying outliers is important, because they might greatly impact measures like mean and standard deviation. When left untouched, outliers might make us reach wrong conclusions about our data. See what happens to the slope of this linear regression with and without the outliers.

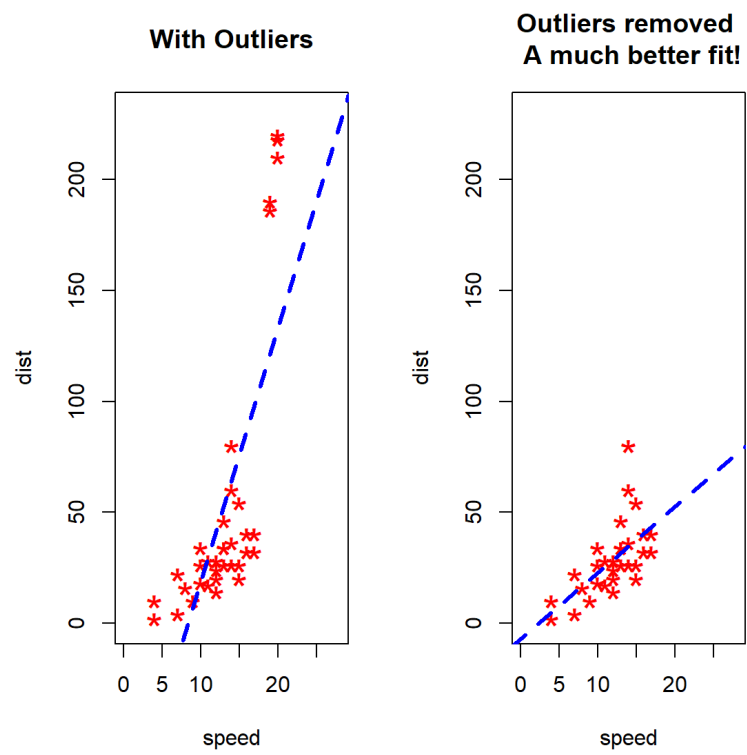


Figure 14.1: Source: Zhang (2020)

15 Z-score

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

Let's write a function that identifies outliers according to the Z-score.

where

```
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
```

- x = data point,
- μ = time series mean
- σ = time series standard deviation.

Now we can simply use this function:

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

Source: Atwan (2022)

16 IQR (Inter Quartile Range)

The IQR (Inter Quartile Range) is the distance between the 25th percentile (Q1) and the 75th percentile (Q3). In a box plot, the whiskers usually extend $1.5 \times \text{IQR}$ beyond Q1 and Q3, see below.

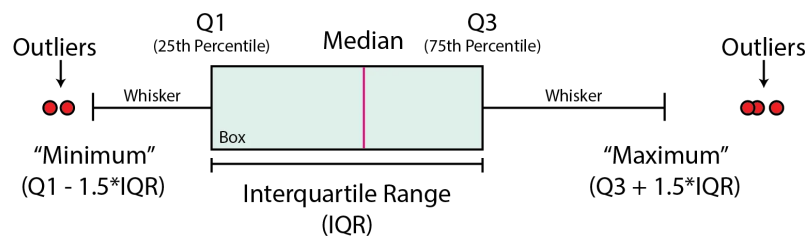


Figure 16.1: Source: McDonald (2022)

A common outlier detection method is to consider whatever points outside the whisker range as outliers.

Part V

best practices

17 motivation

18 date formatting

Here you will find several examples of how to format dates in your plots. Not many explanations are provided.

How to use this page? Find first an example of a plot you like, only then go to the code and see how it's done.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import datetime
from datetime import timedelta
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
import matplotlib.gridspec as gridspec
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker

import pandas as pd

start_date = '2018-01-01'
end_date = '2018-04-30'

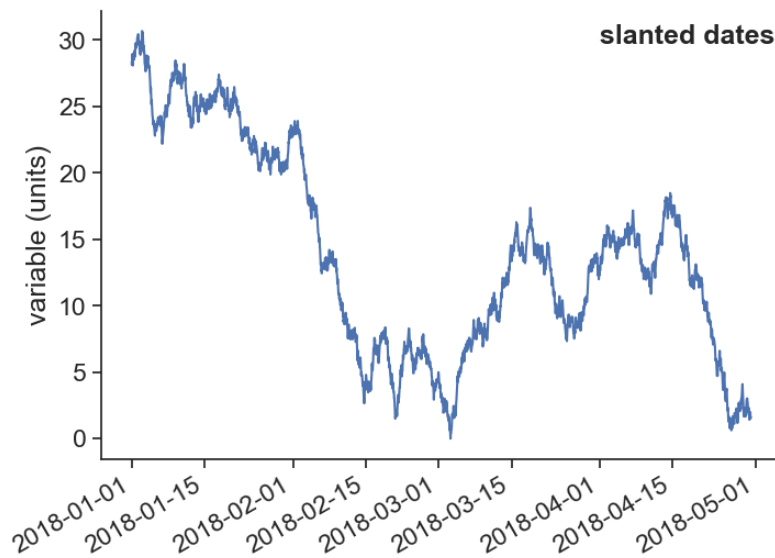
dates = pd.date_range(start_date, end_date, freq='1H')
# create a random variable to plot
var = np.random.rand(len(dates)) - 0.51
var = var.cumsum()
var = var - var.min()
# create dataframe, make "date" the index
df = pd.DataFrame({'date': dates, 'variable': var})
df.set_index(df['date'], inplace=True)
df
```

| | date | variable |
|---------------------|---------------------|-----------|
| date | | |
| 2018-01-01 00:00:00 | 2018-01-01 00:00:00 | 28.317035 |
| 2018-01-01 01:00:00 | 2018-01-01 01:00:00 | 28.120523 |
| 2018-01-01 02:00:00 | 2018-01-01 02:00:00 | 28.596894 |
| 2018-01-01 03:00:00 | 2018-01-01 03:00:00 | 28.931941 |
| 2018-01-01 04:00:00 | 2018-01-01 04:00:00 | 28.561778 |
| ... | ... | ... |
| 2018-04-29 20:00:00 | 2018-04-29 20:00:00 | 1.914343 |
| 2018-04-29 21:00:00 | 2018-04-29 21:00:00 | 1.648757 |
| 2018-04-29 22:00:00 | 2018-04-29 22:00:00 | 1.992956 |
| 2018-04-29 23:00:00 | 2018-04-29 23:00:00 | 1.500860 |
| 2018-04-30 00:00:00 | 2018-04-30 00:00:00 | 1.650439 |

define a useful function to plot the graphs below

```
def explanation(ax, text, letter):
    ax.text(0.99, 0.97, text,
            transform=ax.transAxes,
            horizontalalignment='right', verticalalignment='top',
            fontweight="bold")
    ax.text(0.01, 0.01, letter,
            transform=ax.transAxes,
            horizontalalignment='left', verticalalignment='bottom',
            fontweight="bold")
    ax.set(ylabel="variable (units)")
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)

fig, ax = plt.subplots(1, 1, figsize=(8, 6))
ax.plot(df['variable'])
plt.gcf().autofmt_xdate() # makes slanted dates
explanation(ax, "slanted dates", "")
fig.savefig("dates1.png")
```



```
fig, ax = plt.subplots(4, 1, figsize=(10, 16),
                        gridspec_kw={'hspace': 0.3})

### plot a ###
ax[0].plot(df['variable'])
date_form = DateFormatter("%b")
ax[0].xaxis.set_major_locator(mdates.MonthLocator(interval=2))
ax[0].xaxis.set_major_formatter(date_form)

### plot b ###
ax[1].plot(df['variable'])
date_form = DateFormatter("%B")
ax[1].xaxis.set_major_locator(mdates.MonthLocator(interval=1))
ax[1].xaxis.set_major_formatter(date_form)

### plot c ###
ax[2].plot(df['variable'])
ax[2].xaxis.set_major_locator(mdates.MonthLocator())
# 16 is a slight approximation for the center, since months differ in number of days.
ax[2].xaxis.set_minor_locator(mdates.MonthLocator(bymonthday=16))
ax[2].xaxis.set_major_formatter(ticker.NullFormatter())
ax[2].xaxis.set_minor_formatter(DateFormatter('%B'))
```

```

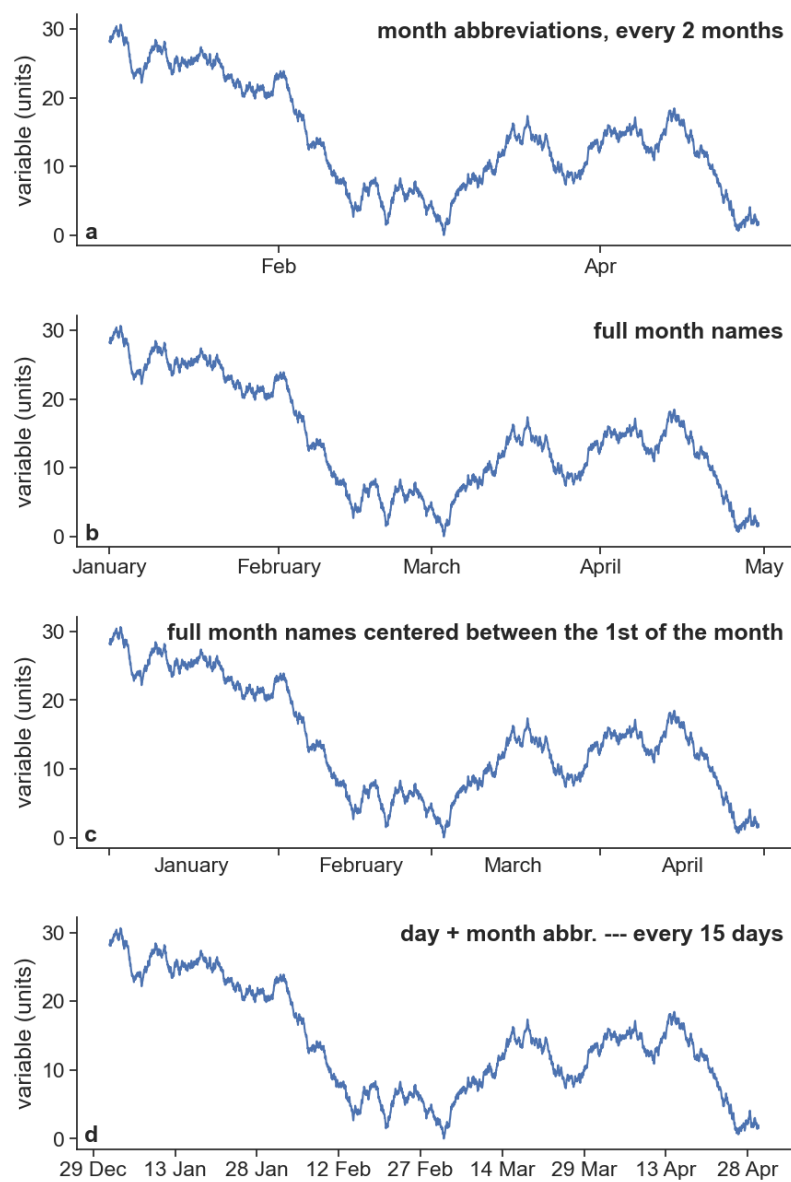
for tick in ax[2].xaxis.get_minor_ticks():
    tick.tick1line.set_markersize(0)
    tick.tick2line.set_markersize(0)
    tick.label1.set_horizontalalignment('center')

### plot d ###
ax[3].plot(df['variable'])
date_form = DateFormatter("%d %b")
ax[3].xaxis.set_major_locator(mdates.DayLocator(interval=15))
ax[3].xaxis.set_major_formatter(date_form)

explanation(ax[0], "month abbreviations, every 2 months", "a")
explanation(ax[1], "full month names", "b")
explanation(ax[2], "full month names centered between the 1st of the month", "c")
explanation(ax[3], "day + month abbr. --- every 15 days", "d")

fig.savefig("dates2.png")

```



```
fig, ax = plt.subplots(4, 1, figsize=(10, 16),
                        gridspec_kw={'hspace': 0.3})

### plot e ###
ax[0].plot(df['variable'])
```

```

date_form = DateFormatter("%d/%m")
ax[0].xaxis.set_major_locator(mdates.DayLocator(bymonthday=[5, 20]))
ax[0].xaxis.set_major_formatter(date_form)

### plot f ###
ax[1].plot(df['variable'])
locator = mdates.AutoDateLocator(minticks=11, maxticks=17)
formatter = mdates.ConciseDateFormatter(locator)
ax[1].xaxis.set_major_locator(locator)
ax[1].xaxis.set_major_formatter(formatter)

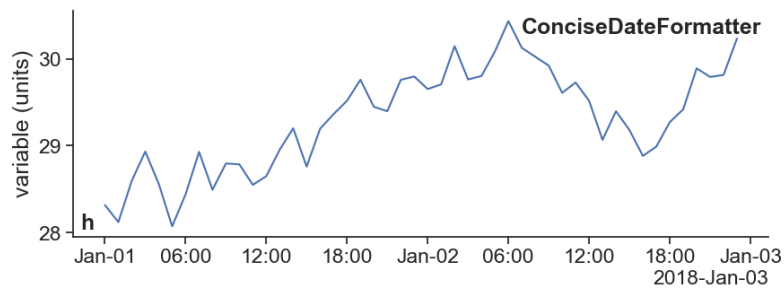
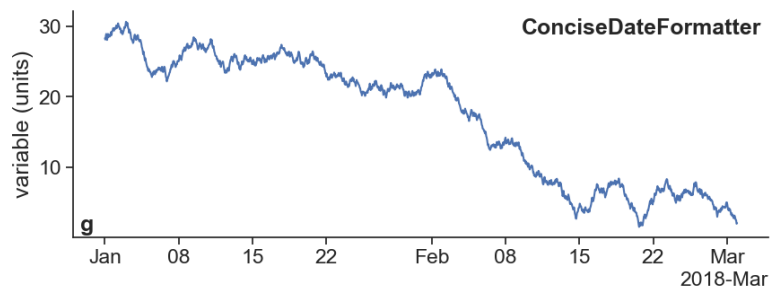
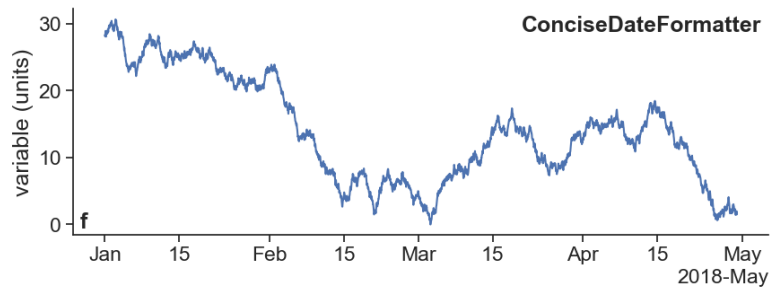
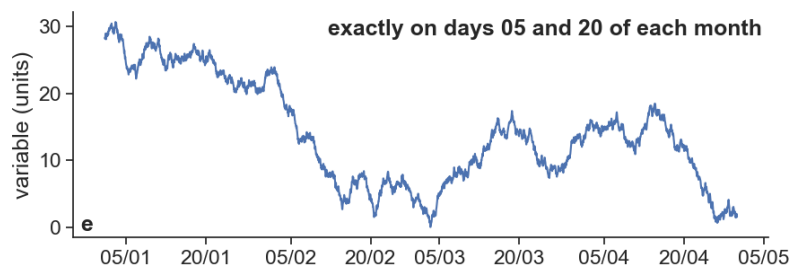
### plot g ###
ax[2].plot(df.loc['2018-01-01':'2018-03-01', 'variable'])
locator = mdates.AutoDateLocator(minticks=6, maxticks=14)
formatter = mdates.ConciseDateFormatter(locator)
ax[2].xaxis.set_major_locator(locator)
ax[2].xaxis.set_major_formatter(formatter)

### plot h ###
ax[3].plot(df.loc['2018-01-01':'2018-01-02', 'variable'])
locator = mdates.AutoDateLocator(minticks=6, maxticks=10)
formatter = mdates.ConciseDateFormatter(locator)
ax[3].xaxis.set_major_locator(locator)
ax[3].xaxis.set_major_formatter(formatter)

explanation(ax[0], "exactly on days 05 and 20 of each month", "e")
explanation(ax[1], "ConciseDateFormatter", "f")
explanation(ax[2], "ConciseDateFormatter", "g")
explanation(ax[3], "ConciseDateFormatter", "h")

fig.savefig("dates3.png")

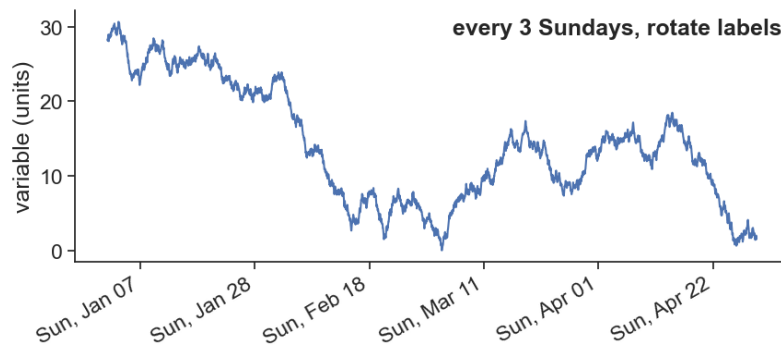
```



```
fig, ax = plt.subplots(1, 1, figsize=(10, 4),
                        gridspec_kw={'hspace': 0.3})

# import constants for the days of the week
from matplotlib.dates import MO, TU, WE, TH, FR, SA, SU
ax.plot(df['variable'])
```

```
# tick on sundays every third week
loc = mdates.WeekdayLocator(byweekday=SU, interval=3)
ax.xaxis.set_major_locator(loc)
date_form = DateFormatter("%a, %b %d")
ax.xaxis.set_major_formatter(date_form)
fig.autofmt_xdate(bottom=0.2, rotation=30, ha='right')
explanation(ax, "every 3 Sundays, rotate labels", "")
```



| Code | Explanation |
|------|---|
| %Y | 4-digit year (e.g., 2022) |
| %y | 2-digit year (e.g., 22) |
| %m | 2-digit month (e.g., 12) |
| %B | Full month name (e.g., December) |
| %b | Abbreviated month name (e.g., Dec) |
| %d | 2-digit day of the month (e.g., 09) |
| %A | Full weekday name (e.g., Tuesday) |
| %a | Abbreviated weekday name (e.g., Tue) |
| %H | 24-hour clock hour (e.g., 23) |
| %I | 12-hour clock hour (e.g., 11) |
| %M | 2-digit minute (e.g., 59) |
| %S | 2-digit second (e.g., 59) |
| %p | “AM” or “PM” |
| %Z | Time zone name |
| %z | Time zone offset from UTC (e.g., -0500) |

Part VI

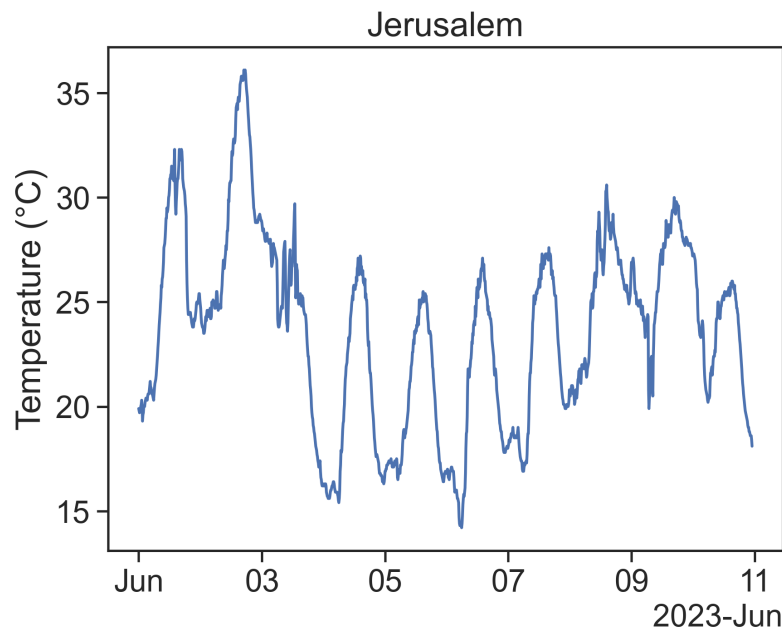
stationarity

19 motivation

20 stochastic processes

21 autocorrelation

See the temperatures for Jerusalem in a 4-day interval:



21.1 question

If I know the temperature right now, what does that tell me about the temperature 10 minutes from now? How about 100 minutes? 1000 minutes?

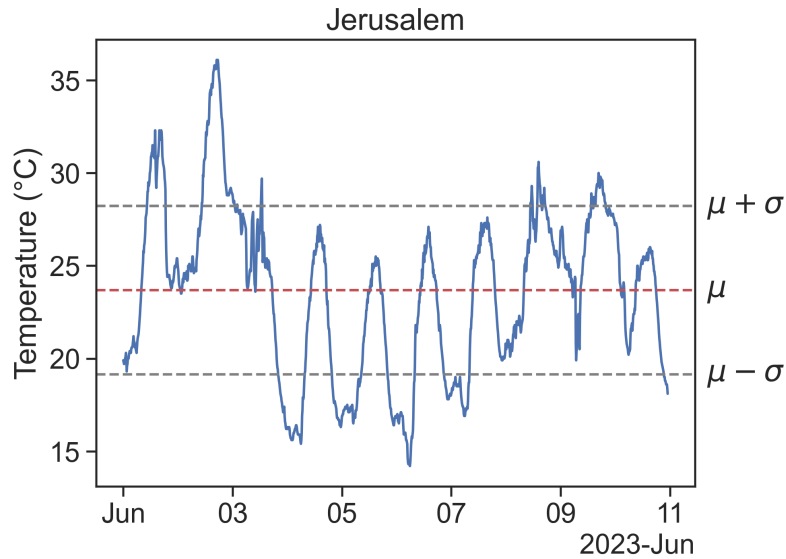
To answer this, we need to talk about **autocorrelation**. Let's start by introducing the necessary concepts.

21.2 mean and standard deviation

Let's call our time series from above X , and its length N .
Then:

$$\begin{aligned}\text{mean } \mu &= \frac{\sum_{i=1}^N X_i}{N} \\ \text{standard deviation } \sigma &= \sqrt{\frac{\sum_{i=1}^N (X_i - \mu)^2}{N}}\end{aligned}$$

The mean and standard deviation can be visualized thus:



One last basic concept we need is the expected value:

$$E[X] = \sum_{i=1}^N X_i p_i$$

For our time series, the probability p_i that a given point X_i is in the dataset is simply $1/N$, therefore the expectation becomes

$$E[X] = \frac{\sum_{i=1}^N X_i}{N}$$

21.3 autocorrelation

The autocorrelation of a time series X is the answer to the following question:

if we shift X by τ units, how similar will this be to the original signal?

In other words:

how correlated are $X(t)$ and $X(t + \tau)$?

Using the Pearson correlation coefficient

we get

$$\rho_{XX}(\tau) = \frac{E[(X_t - \mu)(X_{t+\tau} - \mu)]}{\sigma^2}$$

Pearson correlation coefficient
between X and Y :

$$\rho_{X,Y} = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y}$$

A video is worth a billion words, so let's see the autocorrelation in action:

<https://youtu.be/tpf-tuYHR5w>

A few comments:

- The autocorrelation for $\tau = 0$ (zero shift) is always 1.
[Can you prove this? All the necessary equations are above!]

Part VII

time lags

22 motivation

23 cross-correlation

```
import numpy as np
```

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

dfvdfv

24 dynamic time warp

25 LDTW

according to this paper

Part VIII

frequency

26 motivation

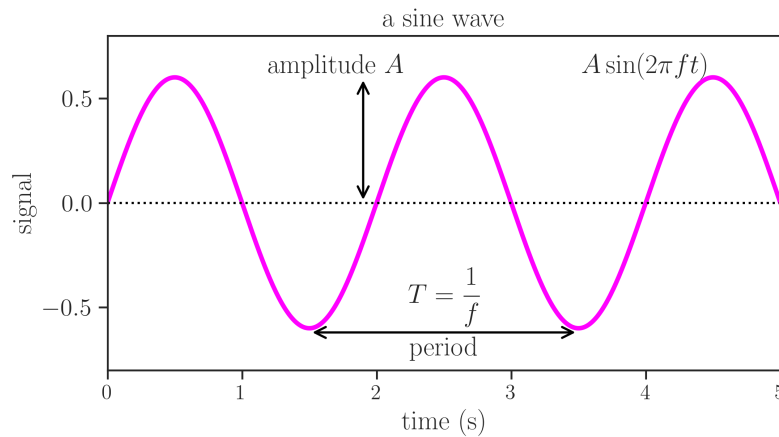
27 Fourier transform

27.1 basic wave concepts

The function

$$f(t) = B \sin(2\pi ft) \quad (27.1)$$

has two basic characteristics, its amplitude B and frequency f .

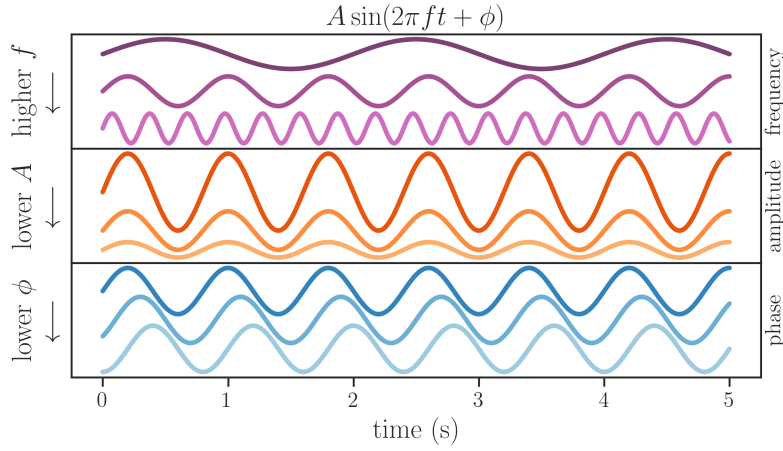


In the figure above, the amplitude $B = 0.6$ and we see that the distance between two peaks is called period, $T = 2$ s. The frequency is defined as the inverse of the period:

$$f = \frac{1}{T}. \quad (27.2)$$

When time is in seconds, then the frequency is measured in Hertz (Hz). For the graph above, therefore, we see a wave whose frequency is $f = 1/(2 \text{ s}) = 0.5$ Hz.

In the figure below, we see what happens when we vary the values of the frequency and amplitude.



The graph above introduces two new characteristics of a wave, its phase ϕ , and its offset B . A more general description of a sine wave is

$$f(t) = B \sin(2\pi ft + \phi) + B_0. \quad (27.3)$$

The offset B_0 moves the wave up and down, while changing the value of ϕ makes the sine wave move left and right. When the phase $\phi = 2\pi$, the sine wave will have shifted a full period, and the resulting wave is identical to the original:

$$B \sin(2\pi ft) = B \sin(2\pi ft + 2\pi). \quad (27.4)$$

All the above can also be said about a cosine, whose general form can be given as

$$A \cos(2\pi ft + \phi) + A_0 \quad (27.5)$$

One final point before we jump into the deep waters is that the sine and cosine functions are related through a simple phase shift:

$$\cos\left(2\pi ft + \frac{\pi}{2}\right) = \sin(2\pi ft)$$

27.2 Fourier's theorem

Fourier's theorem states that

Any periodic signal is composed of a superposition of pure sine waves, with suitably chosen amplitudes and phases, whose frequencies are harmonics of the fundamental frequency of the signal.

See the following animations to visualize the theorem in action.

Source: https://en.wikipedia.org/wiki/File:Fourier_series_and_transform.gif

Source: https://commons.wikimedia.org/wiki/File:Fourier_synthesis_square_wave_animated.gif

Source: https://commons.wikimedia.org/wiki/File:Sawtooth_Fourier_Animation.gif

Source: https://commons.wikimedia.org/wiki/File:Continuous_Fourier_transform_of_rect_and_sinc_functions.gif

27.3 Fourier series

a periodic function can be described as a sum of sines and cosines.

The classic examples are usually the square function and the sawtooth function:

[Source: <https://www.geogebra.org/m/tkajbzmng>]

<https://www.geogebra.org/m/k4eq4fkr>

Not any function, but certainly most functions we will deal with in this course. The function has to fulfill the [Dirichlet conditions](#)

$$F[x(t)] = F(f) = \int_{-\infty}^{\infty} x(t)e^{-2\pi i f t} dt$$

$$f(t) = \int_{-\infty}^{\infty} F(f) e^{2\pi i f t} df$$

<https://dibsmethodsmeetings.github.io/fourier-transforms/>

<https://www.jezzamon.com/fourier/index.html>

28 filtering

29 Nyquist-Shannon sampling theorem

Part IX

seasonality

30 motivation

31 seasonal decomposition

```
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
```

31.1 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=35, na_values=[-999.99])

df
```

| | 1974 | 5 | 19 | 1974.3795 | 333.37 | 5.1 | -999.99 | -999.99.1 | 50.39 |
|---|------|---|----|-----------|--------|-----|---------|-----------|-------|
| 0 | 1974 | 5 | 26 | 1974.3986 | 332.95 | 6 | NaN | NaN | 50.05 |
| 1 | 1974 | 6 | 2 | 1974.4178 | 332.35 | 5 | NaN | NaN | 49.59 |
| 2 | 1974 | 6 | 9 | 1974.4370 | 332.20 | 7 | NaN | NaN | 49.64 |

| | 1974 | 5 | 19 | 1974.3795 | 333.37 | 5.1 | -999.99 | -999.99.1 | 50.39 |
|------|------|-----|-----|-----------|--------|-----|---------|-----------|--------|
| 3 | 1974 | 6 | 16 | 1974.4562 | 332.37 | 7 | NaN | NaN | 50.06 |
| 4 | 1974 | 6 | 23 | 1974.4753 | 331.73 | 5 | NaN | NaN | 49.72 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 2565 | 2023 | 7 | 23 | 2023.5575 | 421.28 | 4 | 418.03 | 397.30 | 141.60 |
| 2566 | 2023 | 7 | 30 | 2023.5767 | 420.83 | 6 | 418.10 | 396.80 | 141.69 |
| 2567 | 2023 | 8 | 6 | 2023.5959 | 420.02 | 6 | 417.36 | 395.65 | 141.41 |
| 2568 | 2023 | 8 | 13 | 2023.6151 | 418.98 | 4 | 417.25 | 395.24 | 140.89 |
| 2569 | 2023 | 8 | 20 | 2023.6342 | 419.31 | 2 | 416.64 | 395.22 | 141.71 |

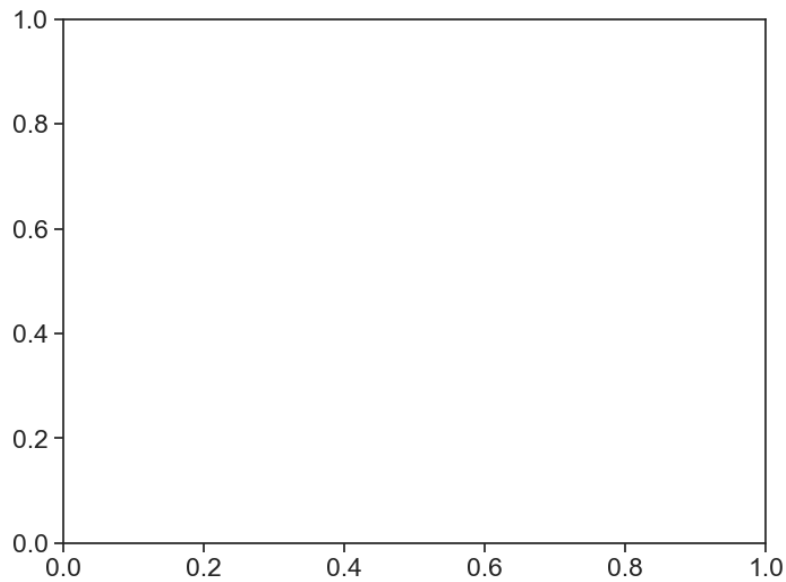
```
df['date'] = pd.to_datetime(df[['year', 'month', 'day']])
df = df.set_index('date')
df
```

| | year | month | day | decimal | average | ndays | 1 year ago | 10 years ago | increase since 1800 |
|------------|------|-------|-----|-----------|---------|-------|------------|--------------|---------------------|
| date | | | | | | | | | |
| 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 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 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");
```

KeyError: 'average'



fill missing data. interpolate method: 'time'

[interpolation methods visualized](#)

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

| | year | month | day | decimal | average | ndays | 1 year ago | 10 years ago | increase since 1800 |
|------------|------|-------|-----|-----------|---------|-------|------------|--------------|---------------------|
| date | | | | | | | | | |
| 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 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 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 |

| date | year | month | day | decimal | average | ndays | 1 year ago | 10 years ago | increase since 1800 |
|------------|------|-------|-----|-----------|---------|-------|------------|--------------|---------------------|
| 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 |

31.2 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:

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

31.2.0.1 Interlude

learn how to use `zip` in a loop

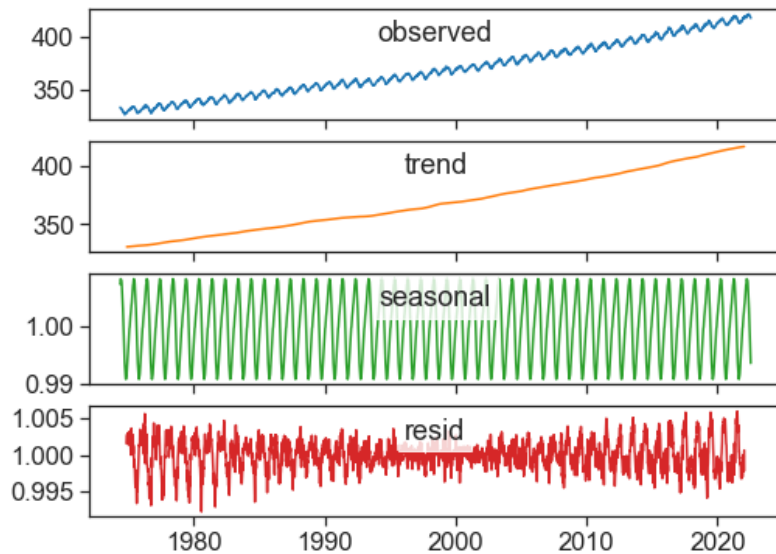
```
letters = ['a', 'b', 'c', 'd', 'e']
numbers = [1, 2, 3, 4, 5]
# zip let's us iterate over to lists at the same time
for l, n in zip(letters, numbers):
    print(f"{l} = {n}")
```

```
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

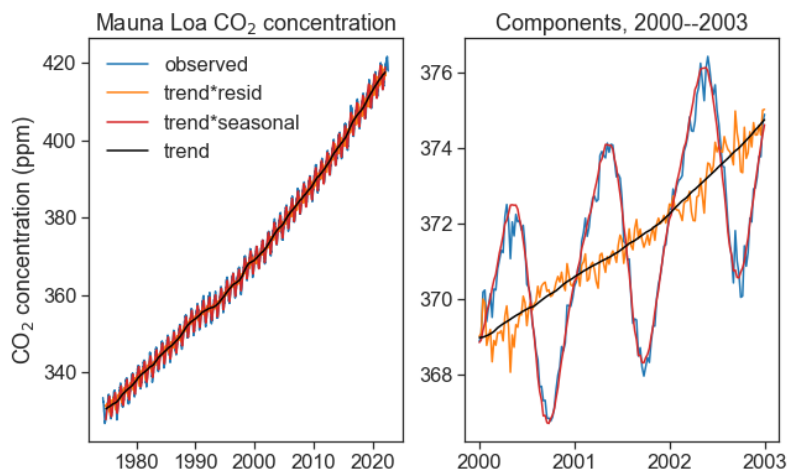
fig, ax = plt.subplots(1, 2, figsize=(10,6))
```

```

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*resid")
ax[1].plot((decomposed.trend * decomposed.seasonal)[zoom], color="tab:red", label="trend*seasonal")
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");

```



32 Hilbert transform

Part X

rates of change

33 motivation

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
%matplotlib widget
```

```
filename = "../archive/data/kinneret_cleaned.csv"
df = pd.read_csv(filename)
df['date'] = pd.to_datetime(df['date'], dayfirst=True)
df = df.set_index('date')
df
```

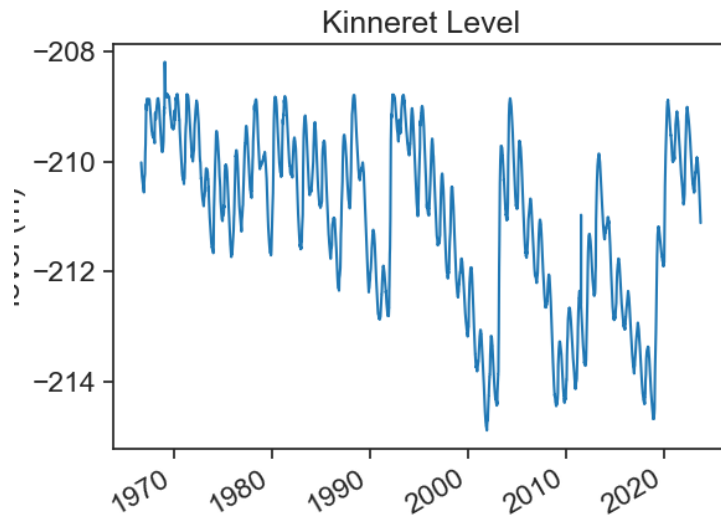
| | level |
|------------|----------|
| date | |
| 2023-09-12 | -211.115 |
| 2023-09-11 | -211.105 |
| 2023-09-10 | -211.095 |
| 2023-09-09 | -211.085 |
| 2023-09-08 | -211.070 |
| ... | ... |
| 1966-11-01 | -210.390 |
| 1966-10-15 | -210.320 |
| 1966-10-01 | -210.270 |
| 1966-09-15 | -210.130 |
| 1966-09-01 | -210.020 |

```
fig, ax = plt.subplots()
ax.plot(df['level'], color="tab:blue")
ax.set(title="Kinneret Level",
```

```

        ylabel="level (m)")
plt.gcf().autofmt_xdate() # makes slanted dates

```



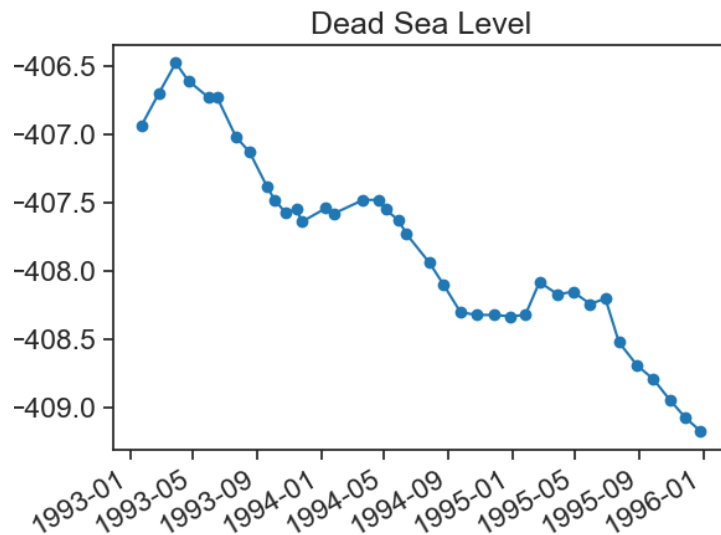
The data seems ok, until we take a closer look. Data points are not evenly spaced in time.

```

fig, ax = plt.subplots()
ax.plot(df.loc["1993":"1995", 'level'], color="tab:blue", marker="o")
ax.set(title="Dead Sea Level",
        ylabel="level (m)")
plt.gcf().autofmt_xdate() # makes slanted dates

```

/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_3777/934261896.py:2: FutureWarning:
 ax.plot(df.loc["1993":"1995", 'level'], color="tab:blue", marker="o")



We can resample by day (a much higher rate than the original), and linearly interpolate:

```
df2 = df['level'].resample('D').interpolate('time').to_frame()
df2['level_sm'] = df2['level'].rolling('30D', center=True).mean()
df3 = df2['level'].resample('W').mean().to_frame()
```

```
fig, ax = plt.subplots()
ax.plot(df2.loc["1993":"1995", 'level_sm'],
        color="tab:red",
        label="daily resampled")
ax.plot(df3.loc["1993":"1995", 'level'],
        color="black",
        label="daily resampled")
ax.plot(df2.loc["1993":"1995", 'level'],
        color="tab:orange",
        label="daily resampled")
ax.plot(df.loc["1993":"1995", 'level'],
        color="tab:blue",
        marker="o",
        linestyle="None",
```



```

        label="original")
    ax.set(title="Dead Sea Level",
           ylabel="level (m)")
    plt.gcf().autofmt_xdate() # makes slanted dates
    ax.legend(frameon=False)

```

```

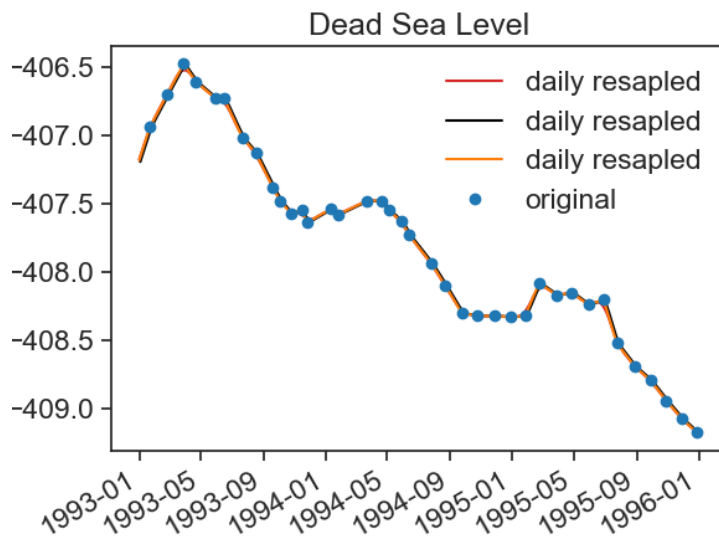
/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_3777/2583247388.py:11: FutureWarning
    ax.plot(df.loc["1993":"1995", 'level'],

```

```

<matplotlib.legend.Legend at 0x7fa71e8b0bb0>

```



```

df2['naive'] = df2['level'].diff()
df2['gradient'] = np.gradient(df2['level'])

```

```

df3['naive'] = df3['level'].diff()
df3['gradient'] = np.gradient(df3['level'])

```

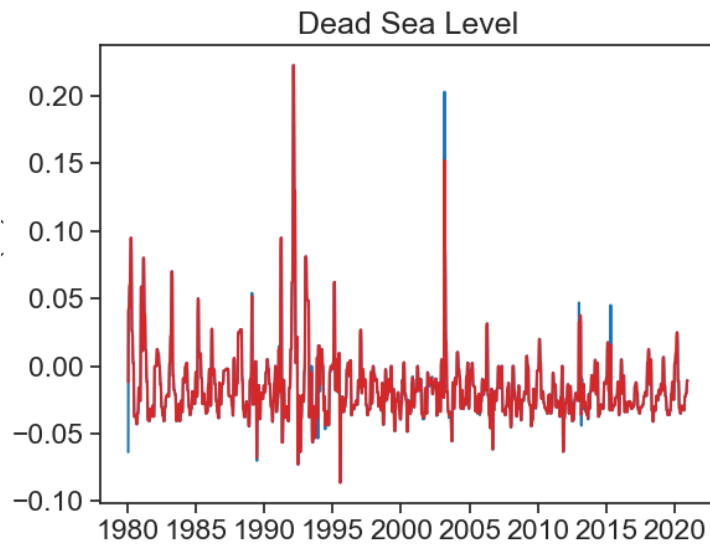
```

fig, ax = plt.subplots()
ax.plot(df3.loc["1980":"2020", 'naive'], color="tab:blue")
ax.plot(df3.loc["1980":"2020", 'gradient'], color="tab:red")

```

```
ax.set(title="Dead Sea Level",
       ylabel="level (m)")
```

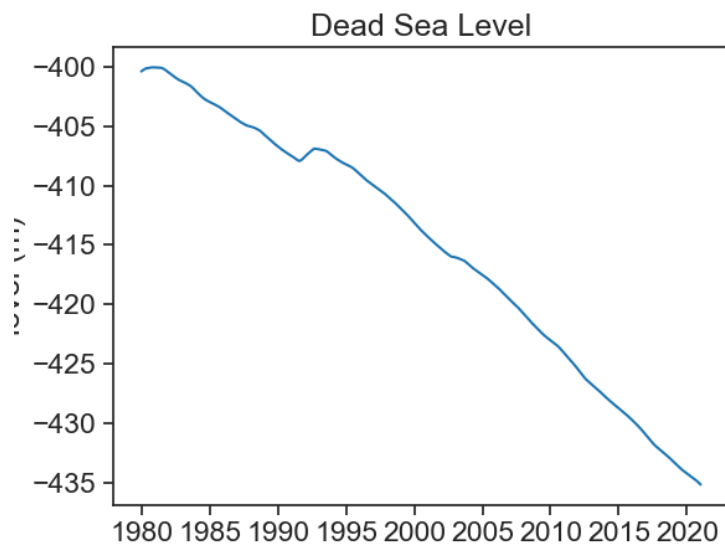
```
[Text(0.5, 1.0, 'Dead Sea Level'), Text(0, 0.5, 'level (m)')]
```



```
df3 = df2["level"].rolling('365.24D', center=True).mean().to_frame()
```

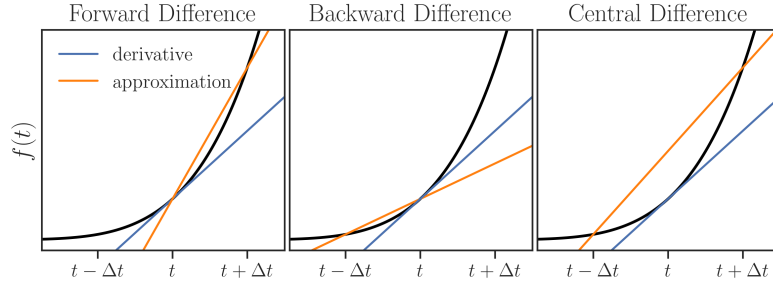
```
fig, ax = plt.subplots()
ax.plot(df3.loc["1980":"2020", 'level'], color="tab:blue")
ax.set(title="Dead Sea Level",
       ylabel="level (m)")
```

```
[Text(0.5, 1.0, 'Dead Sea Level'), Text(0, 0.5, 'level (m)')]
```



34 derivatives

35 finite differences



Definition of a **derivative**:

$$\underbrace{\dot{f} = f'(t) = \frac{df(t)}{dt}}_{\text{same thing}} = \lim_{\Delta t \rightarrow 0} \frac{f(t + \Delta t) - f(t)}{\Delta t}.$$

Numerically, we can approximate the derivative $f'(t)$ of a time series $f(t)$ as

$$\frac{df(t)}{dt} = \frac{f(t + \Delta t) - f(t)}{\Delta t} + \mathcal{O}(\Delta t). \quad (35.1)$$

The expression above is called the *two-point forward difference formula*. Likewise, we can define the *two-point backward difference formula*:

$$\frac{df(t)}{dt} = \frac{f(t) - f(t - \Delta t)}{\Delta t} + \mathcal{O}(\Delta t). \quad (35.2)$$

If we sum together Equation 35.1 and Equation 35.2 we get:

$$\begin{aligned} 2 \frac{df(t)}{dt} &= \frac{f(t + \Delta t) - \cancel{f(t)}}{\Delta t} + \frac{\cancel{f(t)} - f(t - \Delta t)}{\Delta t} \\ &= \frac{f(t + \Delta t) - f(t - \Delta t)}{\Delta t}. \end{aligned} \quad (35.3)$$

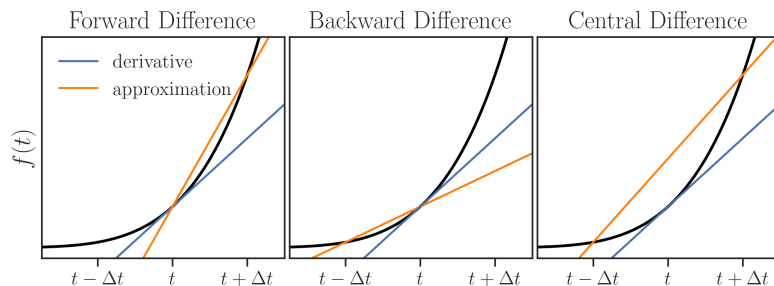
The expression $\mathcal{O}(\Delta t)$ means that the error associated with the approximation is proportional to Δt . This is called “**Big O notation**”.

Dividing both sides by 2 gives the *two-point central difference formula*:

$$\frac{df(t)}{dt} = \frac{f(t + \Delta t) - f(t - \Delta t)}{2\Delta t} + \mathcal{O}(\Delta t^2). \quad (35.4)$$

Two things are worth mentioning about the approximation above:

1. it is balanced, that is, there is no preference of the future over the past.
2. its error is proportional to Δt^2 , it is a lot more precise than the unbalanced approximations :)



To understand why the error is proportional to Δt^2 , one can subtract the Taylor expansion of $f(t - \Delta t)$ from the Taylor expansion of $f(t + \Delta t)$. [See this, pages 3 and 4.](#)

The function `np.gradient` calculates the derivative using the central difference for points in the interior of the array, and uses the forward (backward) difference for the derivative at the beginning (end) of the array.

Check out this [nice example](#).

The “gradient” usually refers to a first derivative with respect to space, and it is denoted as $\nabla f(x) = \frac{df(x)}{dx}$. However, it doesn’t really matter if we call the independent variable x or t , the derivative operator is exactly the same.

36 Fourier-based derivatives

This tutorial is based on Pelliccia (2019).

nice trick: <https://math.stackexchange.com/questions/430858/fourier-transform-of-derivative>

37 LOESS-based derivatives

Part XI

forecasting

38 motivation

39 ARIMA

Part XII

assignments

40 assignment 1

This assignment comes right after the first session, where we discussed resampling. Read the whole instructions.

40.1 Task

Go to the [IMS website](#), and choose another weather station we have not worked with yet. Download 10-minute data for a full year, any year.

Make 3 graphs:

1. Daily maximum humidity. Bonus: add another line to the graph, the daily minimum humidity.
2. The number of rainy days for each month
3. For each day of the year, show the number of hours when global solar radiation was above, on average, the threshold 10 W/m^2 . Now add another line, for the threshold 500 W/m^2 .

Make 5 more graphs (total of 8 graphs) of whatever you find interesting. You have the liberty to explore various facets of your dataset that capture your interest. It's essential, however, to maintain a focus on resampling. Each of your plots should effectively showcase and emphasize different aspects or techniques of resampling in your data analysis. To ensure diversity in your visualizations, avoid repetitive themes; for instance, if your first plot illustrates daily wind speed, then your second plot should not simply be a monthly resampling of wind speed. Aim for variety and innovation in each plot to fully explore the potential of resampling in data visualization.

You must download this Jupyter Notebook template. Create a zip file with your Jupyter notebook and with the `.csv` you used. Upload this zip file to the moodle task we created.

40.2 Guidelines

1. Always name the axes and add units when relevant.
2. Always give a title to the plot.
3. Make sure that all axis tick labels (the numbers/dates on the axes) are readable.
4. Include a legend if you have multiple lines, colors, or groups.
5. Use appropriate scales for the axes (linear, logarithmic, etc.) depending on the data's nature.
6. Ensure that the plot is adequately sized for all elements to be clear and visible.
7. Choose colors and markers that are distinguishable, especially for plots with multiple elements.
8. If applicable, include error bars to indicate the variability or uncertainty in the data.
9. Use grid lines sparingly; they should not overshadow the data.

40.3 Evaluation

All your assignments will be evaluated according to the following criteria:

- 40% Presentation. How the graphs look, labels, general organization, markdown, clean code.
- 30% Discussion. This is where you explain what you did, what you found out, etc.
- 15% Depth of analysis. You can analyze/explore the data with different levels of complexity, this is where we take that into consideration.
- 10% Replicability: Your code runs flawlessly.
- 5%: Code commenting. Explain in your code what you are doing, this is good for everyone, especially for yourself!

- Bonus: 10% for originality, creative problem solving, or notable analysis.

41 assignment 2

42 final assignment

43 technical stuff

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](#).

43.1 software

[Anaconda's Python distribution](#)

[VSCode](#)

43.2 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.
- Eilers, Paul HC. 2003. “A Perfect Smoother.” *Analytical Chemistry* 75 (14): 3631–36. <https://doi.org/10.1021/ac034173t>.
- McDonald, Andy. 2022. “Creating Boxplots with the Seaborn Python Library.” *Medium*. Towards Data Science. <https://towardsdatascience.com/creating-boxplots-with-the-seaborn-python-library-f0c20f09bd57>.
- Pelliccia, Daniel. 2019. “Fourier Spectral Smoothing Method.” 2019. <https://nirpyresearch.com/fourier-spectral-smoothing-method/>.
- Zhang, Ou. 2020. “Outliers-Part 3:outliers in Regression.” ouzhang.me. <https://ouzhang.me/blog/outlier-series/outliers-part3/>.

stuff

Hey! What are you doing here? This page is not meant for you

- This is not a coding course. Neither a programming course. Our goal is to learn how to **think** about time series. Of course, we want to tell the computer how to do stuff, and that requires programming.
- “How do you do?” *How do you do what?!*
Welche Sprachen sprechen Sie?
First do, then get use to it.
Once you got used to doing, go back and understand better.