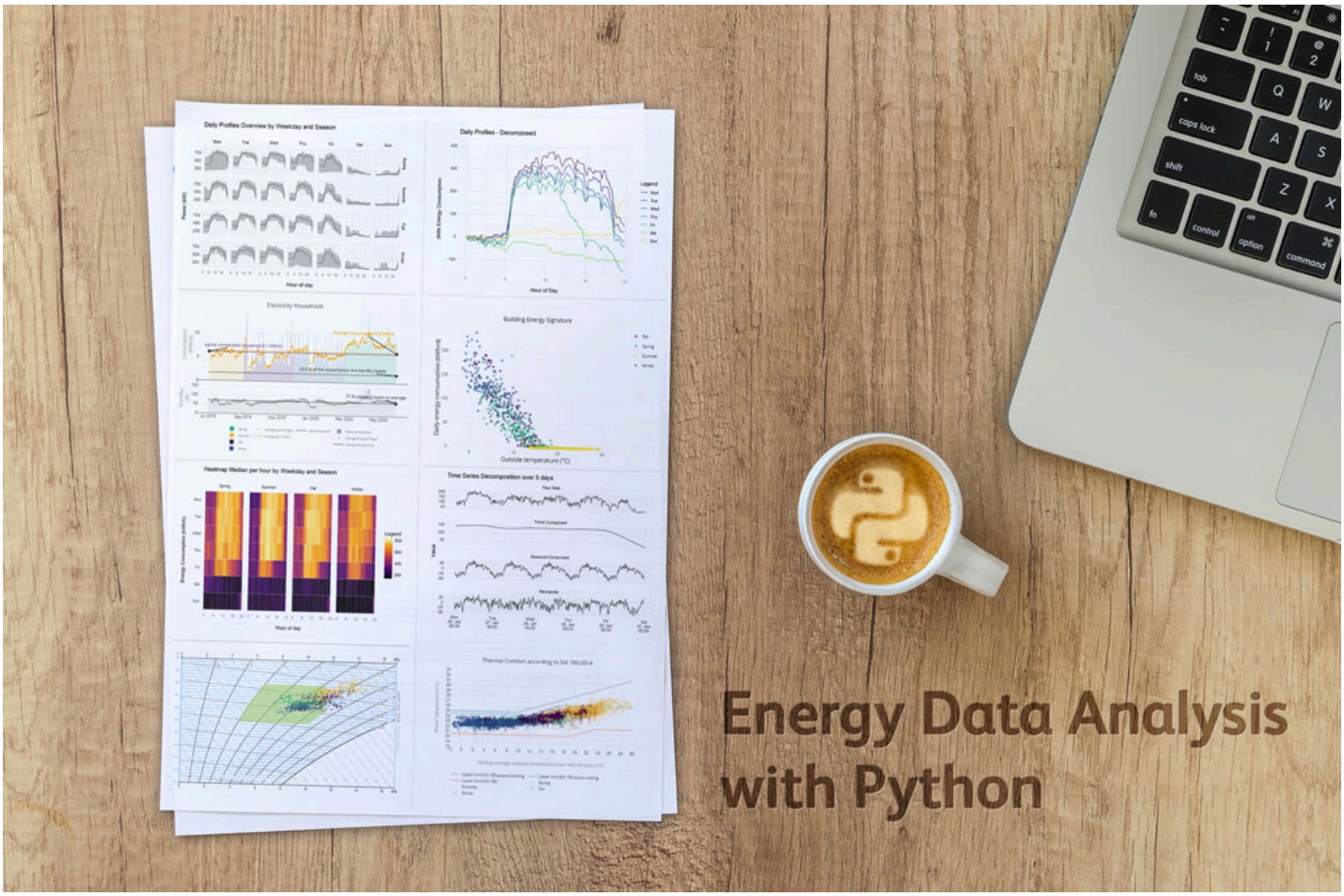
Energy Data Analysis with Python

Contents

- Preface
- Part I — Getting Started
- Part II — Python Basics
- Part III — Data Visualizations
- Appendix



> **About this book**
>
> This book provides practical recipes for analyzing and visualizing energy and comfort time series data using Python. Whether you are new to Python or an experienced user looking for domain-specific recipes, you will find practical examples and ready-to-use code.
>
> This book is also available as a PDF for offline reading: Download PDF

What you will learn

- How to use **pandas** to load, transform, and analyze time series data
- How to handle **datetime objects** and time zones correctly
- How to perform **exploratory data analysis** (EDA) on building data
- How to assess **data quality** — detect gaps, duplicates, and implausible values
- How to create **meaningful visualizations** of energy data using pyedautils
- How to judge **indoor comfort** with temperature and humidity, including the Mollier h-x diagram

Preface

Book structure

The book is organized in three main parts:

> **Part I — Getting Started**
>
> Set up your Python environment, learn about Jupyter Notebooks, and install the required packages including pyedautils.

Part II — Python Basics

Covers general data analysis tasks: pandas fundamentals, data loading, datetime handling, wrangling, time series operations, exploratory analysis, data quality assessment, and Swiss weather data.

Part III — Data Visualizations

Brings the numbers to life. Visualization recipes for statistical plots, decompositions, seasonal patterns, heat maps, daily profiles, comfort diagrams, and more — using Plotly for interactive plots and pyedautils for energy-domain-specific visualizations. Each visualization recipe follows a consistent pattern: goal, data basis, data preparation, and visualization. Simply copy the code, run it, and replace the sample data with your own.

Why Python?

In the EVISU study, researchers at the Lucerne University of Applied Sciences and Arts asked industry experts how and where they perform energy analyses. The result: many rely either on Excel or on expensive building monitoring applications.

Excel users often push the software to its limits as data volumes continue to grow. Users of monitoring applications, on the other hand, benefit from built-in functionality and visualizations. For more complex tasks, switching to an analysis environment such as Python or R becomes almost inevitable—but the learning curve is steep. At the same time, there have been very few specialist applications to date that are specifically tailored to the field of energy and building services engineering and enable efficient data analysis and visualization.

This book aims to bridge that gap and make it easier to get started with Python.

> **Tip**
>
> Wondering why Python and not R? I use both! Check out the R version: Energy Data Analysis with R. But to be honest, Python is much more widely used in the industry…

Acknowledgements

This book is inspired by the R Graphics Cookbook and Engineering Data Analysis in R, and it is the Python counterpart of Energy Data Analysis with R.

It was developed using Jupyter Book, an open-source project for building publication-quality books from computational material. The book combines Python code, Jupyter Notebooks, data, and text — all examples are re-executed every time the book is rebuilt.

Hosted on GitHub Pages. Source available on GitHub.

License

The code in this book is licensed under the MIT License, the text and images under CC BY 4.0. Some sample data comes from third parties, see data sources and credits.

Part I — Getting Started

The fastest way to use this book is with Google Colab — a free, cloud-based Jupyter environment that requires no installation.

> **What you will learn**
>
> - How to open and run notebooks from this book in Google Colab
> - What Jupyter Notebooks are and how they work
> - How to install and manage Python packages

If you prefer to work locally, see the Python Installation & IDE Setup appendix.

Jupyter Notebooks

What are Jupyter Notebooks?

Jupyter Notebooks (`.ipynb` files) are interactive documents that combine code, text, and visualizations. They are the standard format for data analysis in Python and are used throughout this book.

Each notebook consists of **cells**:

- **Code cells** — contain Python code that can be executed
- **Markdown cells** — contain formatted text, headings, lists, images, and equations

Using Google Colab

Google Colab is a free, cloud-based Jupyter environment — no local installation required.

Opening notebooks from this book in Colab:

The notebooks in this book can be opened directly in Google Colab. A rocket icon (🚀) appears in the top toolbar — click it and select **Colab** to open the notebook there.

Hello World in Colab

To get started, open a new notebook in Google Colab and try the following.

1. Code Cell — Click on the first cell (or press `+ Code`) and type the following code. Press `Shift+Enter` to run the cell. The output `Hello World!` appears directly below the cell.

```
print("Hello World!")
```

```
Hello World!
```

2. Markdown Cell — Click `+ Text` to add a text cell. Markdown cells support formatted text using Markdown syntax:

```
# My First Notebook

This is a **Markdown cell**. You can write:

- **Bold** and *italic* text
- Bullet lists
- [Links](https://colab.research.google.com/)
- Mathematical equations: $E = mc^2$
- Tables, images, and more
```

Press `Shift+Enter` to render the Markdown cell into formatted text. Double-click the cell to edit it again.

> **Tip**
>
> A typical notebook alternates between Markdown cells (explaining *what* and *why*) and Code cells (showing *how*). This combination of narrative and code is what makes notebooks so effective for data analysis.

> **Tip**
>
> For working with notebooks locally in VS Code or JupyterLab, see the Python Installation & IDE Setup appendix.

Python Packages

Virtual Environments

It is best practice to create a virtual environment for each project. This keeps dependencies isolated and avoids version conflicts.

First, create a project folder and navigate into it:

```
mkdir my-project
cd my-project
```

Then create and activate a virtual environment inside that folder:

```
# Create a virtual environment
python -m venv .venv

# Activate it
# Windows:
.venv\Scripts\activate
# macOS/Linux:
source .venv/bin/activate
```

The `.venv` folder contains the isolated Python installation. All packages you install with `pip` while the environment is active are stored there.

> **Tip**
>
> VS Code and PyCharm can automatically detect and activate virtual environments.

> **Anaconda**
>
> Anaconda uses its own environment system (`conda`) instead of `venv` . You can create environments via **Anaconda Navigator** (Environments tab) or via the command line:
>
> ```
> conda create -n my-project python=3.12
> conda activate my-project
> ```
>
> Do not mix `venv` and `conda` — use one or the other. If you installed Python via Anaconda, use `conda` environments.

Installing Packages

Use `pip` to install packages:

```
pip install pandas plotly pyedautils
```

Or install all packages used in this book at once:

```
pip install -r requirements.txt
```

> ⚠ **Anaconda users — use conda instead of pip!**
>
> If you are using **Anaconda**, always install packages via **Anaconda Navigator** or `conda install` instead of `pip`:
>
> ```
> conda install pandas plotly statsmodels
> ```
>
> Mixing `pip` and `conda` in the same environment can lead to broken dependencies — `conda` manages packages differently than `pip`, and installing with `pip` can overwrite or conflict with `conda`-managed packages. This may result in import errors, version mismatches, or a corrupted environment that needs to be recreated.
>
> If a package is not available via `conda` (e.g. `pyedautils`), install it with `pip` as a last resort, but do so **after** all `conda` packages are installed.

Key Packages

Package	Purpose
pandas	Data manipulation and analysis
numpy	Numerical computing
plotly	Interactive visualizations
matplotlib	Static visualizations
pyedautils	Energy data analysis utilities — ready-to-use analysis and visualization functions
statsmodels	Time series decomposition and statistics

Updating Packages

```
pip install --upgrade pyedautils
```

Part II — Python Basics

The following chapters give an overview of the basic principles needed to understand the recipes in the “Data Visualizations” part. The focus is on practical examples of how data can be loaded, transformed, and analyzed using Python and pandas.

> **Chapter overview**
>
> - **Pandas Introduction** — Series, DataFrames, indexing, and data types
> - **Loading Data** — Import data from CSV, Excel, JSON, and URLs
> - **Datetime Handling** — Parse timestamps, handle time zones, use the dt accessor
> - **Data Wrangling** — Reshape, filter, merge, and transform data
> - **Time Series** — Regular time grids, resampling, rolling windows, and gap filling
> - **Exploratory Data Analysis** — Descriptive statistics and visual exploration
> - **Data Quality** — Detect and handle missing values, gaps, and duplicates
> - **Meteo Data** — Find Swiss weather stations and download weather data

It is not the goal of this book to provide a complete introduction to Python. There are many excellent resources for learning the fundamentals:

- Python Tutorial — Official Python documentation
- Real Python — Practical tutorials for all levels
- Python Data Science Handbook — Free online book by Jake VanderPlas
- pandas documentation — Getting started tutorials

Pandas Introduction

pandas is the standard Python library for working with tabular (structured) data. It provides fast, flexible data structures — primarily **Series** and **DataFrame** — that make data manipulation, cleaning, and analysis straightforward.

> **Tip**
>
> If you have ever worked with spreadsheets (Excel, Google Sheets), think of a DataFrame as a programmable spreadsheet.

```
import pandas as pd
```

> ⚠ **ModuleNotFoundError?**
>
> If you get a `ModuleNotFoundError: No module named 'pandas'`, the package is not installed. Install it first:
>
> ```
> # With pip
> pip install pandas
>
> # With conda (Anaconda)
> conda install pandas
> ```
>
> Then restart the kernel and run the cell again.

Series vs DataFrame

A **Series** is a one-dimensional labeled array — essentially a single column of data with an index.

A **DataFrame** is a two-dimensional labeled data structure — a table with rows and columns, where each column is a Series.

```
# A Series
temperatures = pd.Series([22.1, 23.4, 21.8, 24.0], name="Temperature")
temperatures
```

```
0    22.1
1    23.4
2    21.8
3    24.0
Name: Temperature, dtype: float64
```

```
type(temperatures)
```

```
pandas.Series
```

Creating DataFrames

DataFrames can be created from Python dictionaries, lists of lists, or many other sources.

From a dictionary

```
data = {
    "room": ["Living Room", "Bedroom", "Kitchen", "Office"],
    "temperature": [22.1, 20.5, 23.4, 21.8],
    "humidity": [45.0, 50.2, 38.7, 42.1]
}

df = pd.DataFrame(data)
df
```

	room	temperature	humidity
0	Living Room	22.1	45.0
1	Bedroom	20.5	50.2
2	Kitchen	23.4	38.7
3	Office	21.8	42.1

From a list of lists

```
rows = [
    ["Living Room", 22.1, 45.0],
    ["Bedroom", 20.5, 50.2],
    ["Kitchen", 23.4, 38.7],
    ["Office", 21.8, 42.1]
]

df2 = pd.DataFrame(rows, columns=["room", "temperature", "humidity"])
df2
```

	room	temperature	humidity
0	Living Room	22.1	45.0
1	Bedroom	20.5	50.2
2	Kitchen	23.4	38.7
3	Office	21.8	42.1

Basic Indexing

There are several ways to select data from a DataFrame.

Select a single column (returns a Series)

```
df["temperature"]
```

```
0    22.1
1    20.5
2    23.4
3    21.8
Name: temperature, dtype: float64
```

> **Note**
>
> Selecting a single column with `df["col"]` returns a **Series**.

Select multiple columns (returns a DataFrame)

```
df[["room", "temperature"]]
```

	room	temperature
0	Living Room	22.1
1	Bedroom	20.5
2	Kitchen	23.4
3	Office	21.8

> **Note**
>
> Selecting multiple columns with `df[["col1", "col2"]]` returns a **DataFrame**.

Label-based indexing with `.loc[]`

`.loc[]` selects rows and columns by **label** (index value and column name).

```
# Select row with index label 0
df.loc[0]
```

```
room           Living Room
temperature           22.1
humidity              45.0
Name: 0, dtype: object
```

```
# Select rows 0 to 2, columns "room" and "temperature"
df.loc[0:2, ["room", "temperature"]]
```

	room	temperature
0	Living Room	22.1
1	Bedroom	20.5
2	Kitchen	23.4

Position-based indexing with `.iloc[]`

`.iloc[]` selects rows and columns by **integer position** (like array indexing).

```
# Select the first two rows, first two columns
df.iloc[0:2, 0:2]
```

	room	temperature
0	Living Room	22.1
1	Bedroom	20.5

```
# Select a specific cell: row 1, column 2
df.iloc[1, 2]
```

```
np.float64(50.2)
```

Data Types and Info

Understanding data types is important because certain operations only work with specific types.

```
df.dtypes
```

```
room              str
temperature    float64
humidity       float64
dtype: object
```

```
df.info()
```

```
<class 'pandas.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   room         4 non-null      str    
 1   temperature  4 non-null      float64
 2   humidity     4 non-null      float64
dtypes: float64(2), str(1)
memory usage: 228.0 bytes
```

Basic Properties

```
# Number of rows and columns
df.shape
```

```
(4, 3)
```

```
# Column names
df.columns
```

```
Index(['room', 'temperature', 'humidity'], dtype='str')
```

```
# Index
df.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

Loading a Real CSV File

Let’s load a real data set to explore. The file `flatTempHum.csv` contains hourly temperature and humidity measurements from four flats.

> **Note**
>
> The CSV file uses `;` as separator, which is common in European data files.

```
url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTe

df = pd.read_csv(url, sep=";")
df.head()
```

	time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
0	2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
1	2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2	2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
3	2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
4	2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

```
df.shape
```

```
(17181, 9)
```

```
df.dtypes
```

```
time              str
FlatA_Hum     float64
FlatA_Temp    float64
FlatB_Hum     float64
FlatB_Temp    float64
FlatC_Hum     float64
FlatC_Temp    float64
FlatD_Hum     float64
FlatD_Temp    float64
dtype: object
```

Note

The `time` column is text (dtype `object` in pandas 2, `str` in pandas 3). It should be converted to `datetime` for time-based operations — this is covered in the Datetime Handling chapter.

```
df.info()
```

```
<class 'pandas.DataFrame'>
RangeIndex: 17181 entries, 0 to 17180
Data columns (total 9 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   time        17181 non-null  str
 1   FlatA_Hum   16484 non-null  float64
 2   FlatA_Temp  16484 non-null  float64
 3   FlatB_Hum   16470 non-null  float64
 4   FlatB_Temp  16470 non-null  float64
 5   FlatC_Hum   16497 non-null  float64
 6   FlatC_Temp  16497 non-null  float64
 7   FlatD_Hum   16354 non-null  float64
 8   FlatD_Temp  16354 non-null  float64
dtypes: float64(8), str(1)
memory usage: 1.2 MB
```

```
df.describe()
```

	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_
count	16484.000000	16484.000000	16470.000000	16470.000000	16497.000000	16497.0
mean	54.336599	23.786531	49.666697	23.404749	48.895339	23.9646
std	6.300389	1.273444	6.903822	1.470354	6.487768	1.36001
min	23.500000	19.100000	25.000000	20.530000	22.500000	21.3800
25%	51.000000	22.800000	44.700000	22.320000	44.300000	22.8800
50%	54.800000	23.480000	50.000000	23.000000	49.200000	23.6000
75%	58.300000	24.600000	54.500000	24.100000	53.800000	24.8000
max	76.500000	28.880000	73.300000	29.120000	69.800000	28.8000

What do these statistics mean?

Statistic	Meaning
count	Number of non-missing values
mean	Average value (sum / count)
std	Standard deviation — how spread out the values are
min	Smallest value
25%	25th percentile — 25% of values are below this
50%	Median — the middle value (50th percentile)
75%	75th percentile — 75% of values are below this
max	Largest value

A large difference between **mean** and **50%** (median) indicates skewed data. A high **std** relative to the **mean** indicates high variability.

```
df.columns
```

```
Index(['time', 'FlatA_Hum', 'FlatA_Temp', 'FlatB_Hum', 'FlatB_Temp',
       'FlatC_Hum', 'FlatC_Temp', 'FlatD_Hum', 'FlatD_Temp'],
      dtype='str')
```

```
# Select only temperature columns
temp_cols = [col for col in df.columns if "Temp" in col]
df[temp_cols].head()
```

	FlatA_Temp	FlatB_Temp	FlatC_Temp	FlatD_Temp
0	24.43	22.40	24.5	24.43
1	24.40	22.40	24.5	24.40
2	24.40	22.40	24.5	24.38
3	24.40	22.40	24.5	24.33
4	24.40	22.37	24.5	24.30

```
# Access specific rows and columns
df.loc[0:4, ["time", "FlatA_Temp", "FlatB_Temp"]]
```

	time	FlatA_Temp	FlatB_Temp
0	2018-10-03 00:00:00	24.43	22.40
1	2018-10-03 01:00:00	24.40	22.40
2	2018-10-03 02:00:00	24.40	22.40
3	2018-10-03 03:00:00	24.40	22.40
4	2018-10-03 04:00:00	24.40	22.37

Loading Data

Before any analysis can begin, data must be loaded into Python. pandas supports a wide range of file formats and data sources out of the box.

```
import pandas as pd
```

CSV Files

CSV (Comma-Separated Values) is one of the most common data exchange formats. Use `pd.read_csv()` to load CSV files.

> **Tip**
>
> In many European countries, CSV files use `;` as separator instead of `,`. Always check and set the `sep` parameter accordingly.

```
# Load a CSV file with semicolon separator
df = pd.read_csv(
    "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTemp
    sep=";"
)
df.head()
```

	time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
0	2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
1	2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2	2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
3	2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
4	2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

Useful `read_csv` parameters

Parameter	Description	Example
`sep`	Column separator	`sep=";"`
`parse_dates`	Columns to parse as datetime	`parse_dates=["time"]`
`index_col`	Column to use as row index	`index_col="time"`
`usecols`	Only load specific columns	`usecols=["time", "FlatA_Temp"]`
`nrows`	Only read first N rows	`nrows=100`
`encoding`	File encoding	`encoding="utf-8"`

```
# Load with datetime parsing and index
df_parsed = pd.read_csv(
    "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTemp
    sep=";",
    parse_dates=["time"],
    index_col="time"
)
df_parsed.head()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

```
df_parsed.dtypes
```

```
FlatA_Hum     float64
FlatA_Temp    float64
FlatB_Hum     float64
FlatB_Temp    float64
FlatC_Hum     float64
FlatC_Temp    float64
FlatD_Hum     float64
FlatD_Temp    float64
dtype: object
```

Loading from URLs

`pd.read_csv()` works directly with URLs — no need to download the file first.

```
url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatEl

df_elec = pd.read_csv(url, sep=";")
df_elec.head()
```

	time	FlatA_Ele	FlatB_Ele	FlatC_Ele	FlatD_Ele
0	2018-11-30 23:45:00	5619.889	7246.254	5125.006	8387.030
1	2018-12-01 00:00:00	5619.904	7246.551	5125.039	8387.085
2	2018-12-01 00:15:00	5619.929	7246.753	5125.077	8387.137
3	2018-12-01 00:30:00	5619.952	7246.933	5125.117	8387.181
4	2018-12-01 00:45:00	5619.973	7247.005	5125.153	8387.222

Excel Files

Use `pd.read_excel()` to load Excel files (`.xlsx`, `.xls`).

openpyxl required

Reading Excel files requires the `openpyxl` package. Install it with:

```
pip install openpyxl
```

```
# Example: reading an Excel file (not executed here)
# df_excel = pd.read_excel("path/to/file.xlsx", sheet_name="Sheet1")
# df_excel.head()
```

Useful `read_excel` parameters

Parameter	Description	Example
`sheet_name`	Name or index of sheet to read	`sheet_name="Data"`
`header`	Row number for column headers	`header=0`
`skiprows`	Rows to skip at the start	`skiprows=2`
`usecols`	Columns to read	`usecols="A:D"`

JSON Files

JSON (JavaScript Object Notation) is commonly used for data exchange, especially with web APIs and IoT devices. Many building monitoring systems export sensor data as JSON.

A JSON file typically contains a list of records (like rows) or a nested structure.

Example: sensor readings as JSON

Imagine a building management system exports sensor readings in this format:

```
import json
from io import StringIO

# Simulated JSON export from a building management system
json_string = """[
    {"timestamp": "2024-01-15 08:00", "sensor": "Office_101", "temperature": 21.3,
    {"timestamp": "2024-01-15 09:00", "sensor": "Office_101", "temperature": 22.1,
    {"timestamp": "2024-01-15 10:00", "sensor": "Office_101", "temperature": 22.8,
    {"timestamp": "2024-01-15 11:00", "sensor": "Office_101", "temperature": 23.1,
]"""

# Load JSON string into a DataFrame
df_json = pd.read_json(StringIO(json_string))
df_json
```

	timestamp	sensor	temperature	co2
0	2024-01-15 08:00:00	Office_101	21.3	420
1	2024-01-15 09:00:00	Office_101	22.1	580
2	2024-01-15 10:00:00	Office_101	22.8	720
3	2024-01-15 11:00:00	Office_101	23.1	850

> **What happens here?**
>
> 1. The JSON string contains a **list of objects** — each object is one measurement with timestamp, sensor ID, temperature, and CO2 level
> 2. `pd.read_json()` automatically converts each object into a **row** and each key into a **column**
> 3. `StringIO()` wraps the string so pandas reads it like a file
>
> To load a JSON **file** from disk instead: `df = pd.read_json("sensor_data.json")`

Loading JSON from a web API

Many data providers offer REST APIs that return JSON. Use the requests library to fetch the data:

```
# Example: fetching sensor data from an API
# import requests
#
# response = requests.get("https://api.building-system.com/sensors/Office_101/readi
# data = response.json()  # returns a Python list/dict
# df_api = pd.DataFrame(data)
# df_api.head()
```

Saving Data

After processing, data can be saved back to various formats.

```
# Save to CSV
# df.to_csv("output.csv", sep=";", index=False)

# Save to Excel
# df.to_excel("output.xlsx", index=False)
```

> **Tip**
>
> Use `index=False` when saving to prevent pandas from writing the row index as an extra column.

Using pyedautils for Data I/O

I/O stands for **Input/Output** — reading data into Python (input) and writing data to files (output).

The `pyedautils` package provides `save_data()` and `load_data()` functions that simplify saving and loading DataFrames with preserved data types.

> **⚠ ModuleNotFoundError?**
>
> If you get a `ModuleNotFoundError: No module named 'pyedautils'`, install it first:
>
> ```
> pip install pyedautils
> ```
>
> Then restart the kernel and run the cell again.

```
import os
import tempfile
from pyedautils.data_io import save_data, load_data

# Create a sample DataFrame
df_sample = pd.DataFrame({
    "timestamp": pd.date_range("2024-01-01", periods=5, freq="h"),
    "temperature": [21.3, 21.5, 22.0, 22.4, 22.1],
    "humidity": [45.0, 44.8, 43.5, 42.0, 43.2]
})

# Save the DataFrame (here to the temp folder; any path works, e.g. "my_sensor_data
file_path = os.path.join(tempfile.gettempdir(), "my_sensor_data.pkl")
save_data(df_sample, file_path)
print("Data saved.")
```

```
Data saved.
```

save_data() and load_data() — full reference

`save_data(data, file_path, index=False)` saves a DataFrame to a file. The format is determined by the file extension:

Extension	Format	Best for
`.csv`	CSV (text)	Data exchange with Excel or other tools
`.pkl`	Pickle (binary)	Fast intermediate storage, preserves all data types
`.pklz`	Compressed Pickle	Same as pickle but smaller file size
`.json`	JSON (text)	Data exchange with web APIs or JavaScript

The `index` parameter controls whether the DataFrame index is written to the file (default: `False`).

`load_data(file_path)` loads data back from any of the above formats. The format is automatically detected from the file extension. For CSV files, the separator is auto-detected.

Examples:

```
# Save as CSV (readable in Excel)
save_data(df, "output/measurements.csv")

# Save as pickle (fast, preserves dtypes)
save_data(df, "output/measurements.pkl")

# Save as compressed pickle (smaller file)
save_data(df, "output/measurements.pklz")

# Save with index column
save_data(df, "output/measurements.csv", index=True)

# Load any format back
df = load_data("output/measurements.pkl")
```

Why pickle instead of CSV?

- **Data types preserved** — datetime, float, int, categorical all stay intact. CSV converts everything to text
- **Faster** — binary format is much quicker to read/write, especially for large datasets
- **Exact reproduction** — the loaded DataFrame is identical to the saved one

Use CSV for sharing data with others, pickle for your own intermediate storage.

```
# Load it back
df_loaded = load_data(file_path)
df_loaded
```

	timestamp	temperature	humidity
0	2024-01-01 00:00:00	21.3	45.0
1	2024-01-01 01:00:00	21.5	44.8
2	2024-01-01 02:00:00	22.0	43.5
3	2024-01-01 03:00:00	22.4	42.0
4	2024-01-01 04:00:00	22.1	43.2

```
# Verify: dtypes are preserved (timestamp is still datetime, not a string)
df_loaded.dtypes
```

```
timestamp      datetime64[us]
temperature           float64
humidity              float64
dtype: object
```

Datetime Handling

Working with dates and times is essential when analyzing time series data from building sensors and energy meters. pandas provides powerful tools for parsing, manipulating, and indexing datetime data.

```
import pandas as pd
```

Converting Strings to Datetime

When loading CSV files, date and time columns are read as text (dtype `object` in pandas 2, `str` in pandas 3). Use `pd.to_datetime()` to convert them.

```
url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatEl

df = pd.read_csv(url, sep=";")
df.head()
```

	time	FlatA_Ele	FlatB_Ele	FlatC_Ele	FlatD_Ele
0	2018-11-30 23:45:00	5619.889	7246.254	5125.006	8387.030
1	2018-12-01 00:00:00	5619.904	7246.551	5125.039	8387.085
2	2018-12-01 00:15:00	5619.929	7246.753	5125.077	8387.137
3	2018-12-01 00:30:00	5619.952	7246.933	5125.117	8387.181
4	2018-12-01 00:45:00	5619.973	7247.005	5125.153	8387.222

```
df.dtypes
```

```
time             str
FlatA_Ele    float64
FlatB_Ele    float64
FlatC_Ele    float64
FlatD_Ele    float64
dtype: object
```

Note

The `time` column is still text (dtype `object` in pandas 2, `str` in pandas 3). We need to convert it to `datetime64`.

```
df["time"] = pd.to_datetime(df["time"])
df.dtypes
```

```
time         datetime64[us]
FlatA_Ele           float64
FlatB_Ele           float64
FlatC_Ele           float64
FlatD_Ele           float64
dtype: object
```

Format Strings

If the date format is non-standard, you can specify a format string to help pandas parse it correctly.

Code	Meaning	Example
`%Y`	4-digit year	2023
`%m`	Month (zero-padded)	01, 12
`%d`	Day (zero-padded)	01, 31
`%H`	Hour (24h)	00, 23
`%M`	Minute	00, 59
`%S`	Second	00, 59

```
# Example with explicit format
dates = pd.Series(["03.10.2018 00:00", "03.10.2018 01:00", "03.10.2018 02:00"])
pd.to_datetime(dates, format="%d.%m.%Y %H:%M")
```

```
0   2018-10-03 00:00:00
1   2018-10-03 01:00:00
2   2018-10-03 02:00:00
dtype: datetime64[us]
```

> **Tip**
>
> Specifying the `format` parameter explicitly is faster than letting pandas infer it, especially for large data sets.

Setting a Datetime Index

Setting the datetime column as the DataFrame index enables powerful time-based slicing and resampling.

```
df = df.set_index("time")
df.head()
```

time	FlatA_Ele	FlatB_Ele	FlatC_Ele	FlatD_Ele
2018-11-30 23:45:00	5619.889	7246.254	5125.006	8387.030
2018-12-01 00:00:00	5619.904	7246.551	5125.039	8387.085
2018-12-01 00:15:00	5619.929	7246.753	5125.077	8387.137
2018-12-01 00:30:00	5619.952	7246.933	5125.117	8387.181
2018-12-01 00:45:00	5619.973	7247.005	5125.153	8387.222

```
df.index
```

```
DatetimeIndex(['2018-11-30 23:45:00', '2018-12-01 00:00:00',
               '2018-12-01 00:15:00', '2018-12-01 00:30:00',
               '2018-12-01 00:45:00', '2018-12-01 01:00:00',
               '2018-12-01 01:15:00', '2018-12-01 01:30:00',
               '2018-12-01 01:45:00', '2018-12-01 02:00:00',
               ...
               '2020-05-31 21:45:00', '2020-05-31 22:00:00',
               '2020-05-31 22:15:00', '2020-05-31 22:30:00',
               '2020-05-31 22:45:00', '2020-05-31 23:00:00',
               '2020-05-31 23:15:00', '2020-05-31 23:30:00',
               '2020-05-31 23:45:00', '2020-06-01 00:00:00'],
              dtype='datetime64[us]', name='time', length=52610, freq=None)
```

```
# Slice by date range
df.loc["2019-01-01":"2019-01-03"]
```

time	FlatA_Ele	FlatB_Ele	FlatC_Ele	FlatD_Ele
2019-01-01 00:00:00	5849.920	7576.754	5289.692	8742.399
2019-01-01 00:15:00	5849.925	7576.782	5289.714	8742.508
2019-01-01 00:30:00	5849.931	7576.803	5289.731	8742.626
2019-01-01 00:45:00	5849.939	7576.823	5289.746	8742.741
2019-01-01 01:00:00	5849.953	7576.844	5289.761	8742.844
...	...	...	...	...
2019-01-03 22:45:00	5852.718	7583.857	5295.587	8783.319
2019-01-03 23:00:00	5852.722	7583.877	5295.608	8783.358
2019-01-03 23:15:00	5852.727	7583.897	5295.620	8783.389
2019-01-03 23:30:00	5852.741	7583.917	5295.633	8783.417
2019-01-03 23:45:00	5852.754	7583.939	5295.645	8783.445

288 rows × 4 columns

The `.dt` Accessor

The `.dt` accessor provides convenient properties for extracting components from datetime columns.

> **Note**
>
> The `.dt` accessor works on **columns** (Series), not on a DatetimeIndex. If the datetime is the index, access the properties directly (e.g., `df.index.year`).

```
# Reset index to use .dt accessor on the column
df_reset = df.reset_index()
df_reset.head()
```

	time	FlatA_Ele	FlatB_Ele	FlatC_Ele	FlatD_Ele
0	2018-11-30 23:45:00	5619.889	7246.254	5125.006	8387.030
1	2018-12-01 00:00:00	5619.904	7246.551	5125.039	8387.085
2	2018-12-01 00:15:00	5619.929	7246.753	5125.077	8387.137
3	2018-12-01 00:30:00	5619.952	7246.933	5125.117	8387.181
4	2018-12-01 00:45:00	5619.973	7247.005	5125.153	8387.222

```
# Extract year
df_reset["time"].dt.year.head()
```

```
0    2018
1    2018
2    2018
3    2018
4    2018
Name: time, dtype: int32
```

```
# Extract month
df_reset["time"].dt.month.head()
```

```
0    11
1    12
2    12
3    12
4    12
Name: time, dtype: int32
```

```
# Extract day
df_reset["time"].dt.day.head()
```

```
0    30
1     1
2     1
3     1
4     1
Name: time, dtype: int32
```

```
# Extract hour
df_reset["time"].dt.hour.head()
```

```
0    23
1     0
2     0
3     0
4     0
Name: time, dtype: int32
```

```
# Day of the week (Monday=0, Sunday=6)
df_reset["time"].dt.day_of_week.head()
```

```
0    4
1    5
2    5
3    5
4    5
Name: time, dtype: int32
```

```
# Custom string formatting
df_reset["time"].dt.strftime("%A, %d %B %Y").head()
```

```
0      Friday, 30 November 2018
1    Saturday, 01 December 2018
2    Saturday, 01 December 2018
3    Saturday, 01 December 2018
4    Saturday, 01 December 2018
Name: time, dtype: str
```

Adding Derived Columns

The `.dt` accessor is useful for creating new columns that group data by time period.

```
df_reset["year"] = df_reset["time"].dt.year
df_reset["month"] = df_reset["time"].dt.month
df_reset["hour"] = df_reset["time"].dt.hour
df_reset["day_of_week"] = df_reset["time"].dt.day_of_week

df_reset.head()
```

	time	FlatA_Ele	FlatB_Ele	FlatC_Ele	FlatD_Ele	year	month	hour	day_of_week
0	2018-11-30 23:45:00	5619.889	7246.254	5125.006	8387.030	2018	11	23	4
1	2018-12-01 00:00:00	5619.904	7246.551	5125.039	8387.085	2018	12	0	5
2	2018-12-01 00:15:00	5619.929	7246.753	5125.077	8387.137	2018	12	0	5
3	2018-12-01 00:30:00	5619.952	7246.933	5125.117	8387.181	2018	12	0	5
4	2018-12-01 00:45:00	5619.973	7247.005	5125.153	8387.222	2018	12	0	5

Easier with pyedautils: adding a season column

The code above extracts year, month, hour, and day of week manually. For **season** detection (spring, summer, fall, winter), the logic is more complex — it depends on solstice and equinox dates that change every year.

The `pyedautils` package provides a `get_season()` function that handles this automatically:

```
from pyedautils.data_prep.season import get_season

# Add a season column — one line instead of complex date logic
df_reset["season"] = df_reset["time"].apply(get_season)

df_reset[["time", "year", "month", "season"]].drop_duplicates(subset=["month", "sea
```

	time	year	month	season
0	2018-11-30 23:45:00	2018	11	Fall
1	2018-12-01 00:00:00	2018	12	Fall
2011	2018-12-21 22:30:00	2018	12	Winter
2977	2019-01-01 00:00:00	2019	1	Winter
5953	2019-02-01 00:00:00	2019	2	Winter
8641	2019-03-01 00:00:00	2019	3	Winter
10553	2019-03-20 22:00:00	2019	3	Spring
11617	2019-04-01 00:00:00	2019	4	Spring
14497	2019-05-01 00:00:00	2019	5	Spring
17473	2019-06-01 00:00:00	2019	6	Spring
19457	2019-06-21 16:00:00	2019	6	Summer
20353	2019-07-01 00:00:00	2019	7	Summer

get_season() options

Parameter	Default	Description
`hemisphere`	`"north"`	`"north"` or `"south"` — seasons are reversed in the southern hemisphere
`labels`	`["Spring", "Summer", "Fall", "Winter"]`	Custom season names, e.g. `["Frühling", "Sommer", "Herbst", "Winter"]`
`tracking_type`	`"astronomical"`	`"astronomical"` (based on solstice/equinox) or `"meteorological"` (fixed months: Mar-May, Jun-Aug, Sep-Nov, Dec-Feb)

Example with German labels and meteorological seasons:

```
df["saison"] = df["time"].apply(
    get_season,
    labels=["Frühling", "Sommer", "Herbst", "Winter"],
    tracking_type="meteorological"
)
```

Time Zones

By default, datetime objects in pandas are timezone-naive. You can localize them to a specific timezone and convert between timezones.

```
# Create a small example
dates = pd.date_range("2023-06-15 12:00", periods=4, freq="h")
s = pd.Series([22.1, 22.5, 23.0, 22.8], index=dates)
s
```

```
2023-06-15 12:00:00    22.1
2023-06-15 13:00:00    22.5
2023-06-15 14:00:00    23.0
2023-06-15 15:00:00    22.8
Freq: h, dtype: float64
```

```
# Localize to a timezone (assign a timezone to naive datetimes)
s_zurich = s.tz_localize("Europe/Zurich")
s_zurich
```

```
2023-06-15 12:00:00+02:00    22.1
2023-06-15 13:00:00+02:00    22.5
2023-06-15 14:00:00+02:00    23.0
2023-06-15 15:00:00+02:00    22.8
dtype: float64
```

```
# Convert to another timezone
s_utc = s_zurich.tz_convert("UTC")
s_utc
```

```
2023-06-15 10:00:00+00:00    22.1
2023-06-15 11:00:00+00:00    22.5
2023-06-15 12:00:00+00:00    23.0
2023-06-15 13:00:00+00:00    22.8
dtype: float64
```

Daylight saving time

Local time has one hour per year that occurs twice (end of October) and one that does not exist (end of March). Meter exports in local time often show a repeated or a missing

hour there. Tell `tz_localize` how to handle them, or store and exchange data in UTC to avoid the problem.

```
# Autumn: 02:30 occurs twice -> "infer" assigns summer time, then winter time, from
autumn = pd.Series(pd.to_datetime(["2023-10-29 01:30", "2023-10-29 02:30", "2023-10
print(autumn.dt.tz_localize("Europe/Zurich", ambiguous="infer"))

# Spring: 02:30 does not exist -> NaT (or nonexistent="shift_forward")
spring = pd.Series(pd.to_datetime(["2023-03-26 01:30", "2023-03-26 02:30", "2023-03
print(spring.dt.tz_localize("Europe/Zurich", nonexistent="NaT"))
```

```
0   2023-10-29 01:30:00+02:00
1   2023-10-29 02:30:00+02:00
2   2023-10-29 02:30:00+01:00
3   2023-10-29 03:30:00+01:00
dtype: datetime64[us, Europe/Zurich]
0   2023-03-26 01:30:00+01:00
1                         NaT
2   2023-03-26 03:30:00+02:00
dtype: datetime64[us, Europe/Zurich]
```

Creating Date Ranges

`pd.date_range()` creates a sequence of evenly spaced datetime values. This is useful for creating reference time axes or filling gaps.

```
# Hourly timestamps for one week (7 x 24 hours)
date_range = pd.date_range(start="2023-01-01", periods=7 * 24, freq="h")
print(f"Number of timestamps: {len(date_range)}")
date_range[:10]
```

```
Number of timestamps: 168
```

```
DatetimeIndex(['2023-01-01 00:00:00', '2023-01-01 01:00:00',
               '2023-01-01 02:00:00', '2023-01-01 03:00:00',
               '2023-01-01 04:00:00', '2023-01-01 05:00:00',
               '2023-01-01 06:00:00', '2023-01-01 07:00:00',
               '2023-01-01 08:00:00', '2023-01-01 09:00:00'],
              dtype='datetime64[us]', freq='h')
```

```
# Daily timestamps for one year
daily_range = pd.date_range(start="2023-01-01", periods=365, freq="D")
daily_range[:10]
```

```
DatetimeIndex(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04',
               '2023-01-05', '2023-01-06', '2023-01-07', '2023-01-08',
               '2023-01-09', '2023-01-10'],
              dtype='datetime64[us]', freq='D')
```

```
# 15-minute intervals
quarter_hourly = pd.date_range(start="2023-01-01", periods=96, freq="15min")
quarter_hourly[:10]
```

```
DatetimeIndex(['2023-01-01 00:00:00', '2023-01-01 00:15:00',
               '2023-01-01 00:30:00', '2023-01-01 00:45:00',
               '2023-01-01 01:00:00', '2023-01-01 01:15:00',
               '2023-01-01 01:30:00', '2023-01-01 01:45:00',
               '2023-01-01 02:00:00', '2023-01-01 02:15:00'],
              dtype='datetime64[us]', freq='15min')
```

Tip

Common frequency strings: `"h"` (hourly), `"D"` (daily), `"W"` (weekly), `"MS"` (month start), `"15min"` (15 minutes), `"min"` (1 minute).

Practical Example

Let’s put it all together with the temperature and humidity data set.

```
url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTe

df = pd.read_csv(url, sep=";")

# Convert time column
df["time"] = pd.to_datetime(df["time"])

# Set as index
df = df.set_index("time")

df.head()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

```
# Check the time range of the data
print(f"Start: {df.index.min()}")
print(f"End:   {df.index.max()}")
print(f"Duration: {df.index.max() - df.index.min()}")
```

```
Start: 2018-10-03 00:00:00
End:   2020-09-17 20:00:00
Duration: 715 days 20:00:00
```

```
# Select data for a specific month
jan_2019 = df.loc["2019-01"]
jan_2019.head()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2019-01-01 00:00:00	50.0	22.85	45.0	22.85	44.3	22.42
2019-01-01 01:00:00	50.0	22.80	43.7	22.85	45.8	22.48
2019-01-01 02:00:00	50.0	22.80	42.2	22.80	47.0	22.50
2019-01-01 03:00:00	50.0	22.80	41.2	22.73	47.8	22.50
2019-01-01 04:00:00	50.0	22.70	40.2	22.65	48.0	22.50

```
# Resample to daily mean temperature
df_daily = df[["FlatA_Temp"]].resample("D").mean()
df_daily.head()
```

time	FlatA_Temp
2018-10-03	24.200435
2018-10-04	24.232083
2018-10-05	24.210435
2018-10-06	24.180435
2018-10-07	24.296667

When to use .mean() vs .sum() vs .last()?

The right aggregation method depends on **what the data represents**:

Data type	Example	Method	Why
Measurement	Temperature, humidity, CO2	`.mean()`	Average over the period
Cumulative counter	Electricity meter, gas meter	`.resample("D").last().diff()`	Consumption = difference between the readings at the end of each period
Event count	Number of door openings	`.sum()`	Total events per period
State	On/off, occupancy	`.last()` or `.mode()`	Last known state or most frequent

For counters you can also take `.diff()` first and then `.resample("D").sum(min_count=96)` (for 15-minute data): `min_count` drops days with missing readings instead of reporting them as days with low consumption.

Data Wrangling

This chapter gives an overview of the most important data manipulation functions used throughout this book.

It is a fact that the data import and manipulation part of analyses often takes more time than the actual analysis or visualization itself. This is because the exchange data formats are not standardized and meters and sensors record at different time intervals. Data quality, missing data, data imputation and data cleansing also play a major role.

```
# Load data set

import pandas as pd

df = pd.read_csv("https://raw.githubusercontent.com/retomarek/edap/main/edap/sample
                 sep = ";")
df.head()
```

	time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
0	2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
1	2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2	2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
3	2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
4	2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

Wide to Long

```
dfWide = df.copy()
dfWide.head()
```

	time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
0	2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
1	2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2	2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
3	2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
4	2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

```
dfLong = dfWide.melt(id_vars=["time"], var_name="dataPoint")
dfLong.head()
```

	time	dataPoint	value
0	2018-10-03 00:00:00	FlatA_Hum	53.0
1	2018-10-03 01:00:00	FlatA_Hum	53.0
2	2018-10-03 02:00:00	FlatA_Hum	53.0
3	2018-10-03 03:00:00	FlatA_Hum	53.0
4	2018-10-03 04:00:00	FlatA_Hum	53.3

Splitting a column into multiple columns

The `dataPoint` column contains combined information like `FlatA_Hum` — both the flat name and the sensor type. We can split this into two separate columns using `str.split()`:

```
# Split "FlatA_Hum" into "flat" = "FlatA" and "sensor" = "Hum"
dfLong[["flat", "sensor"]] = dfLong["dataPoint"].str.split("_", expand=True)
dfLong.head(10)
```

	time	dataPoint	value	flat	sensor
0	2018-10-03 00:00:00	FlatA_Hum	53.0	FlatA	Hum
1	2018-10-03 01:00:00	FlatA_Hum	53.0	FlatA	Hum
2	2018-10-03 02:00:00	FlatA_Hum	53.0	FlatA	Hum
3	2018-10-03 03:00:00	FlatA_Hum	53.0	FlatA	Hum
4	2018-10-03 04:00:00	FlatA_Hum	53.3	FlatA	Hum
5	2018-10-03 05:00:00	FlatA_Hum	53.7	FlatA	Hum
6	2018-10-03 06:00:00	FlatA_Hum	50.8	FlatA	Hum
7	2018-10-03 07:00:00	FlatA_Hum	NaN	FlatA	Hum
8	2018-10-03 08:00:00	FlatA_Hum	42.7	FlatA	Hum
9	2018-10-03 09:00:00	FlatA_Hum	46.0	FlatA	Hum

How does str.split() work here?

`"FlatA_Hum".split("_")` splits the string at the underscore and returns `["FlatA", "Hum"]`.

With `expand=True`, pandas creates a DataFrame with one column per part — which we assign directly to two new columns `flat` and `sensor`.

This is useful for filtering, e.g. only temperature sensors:

```
dfLong[dfLong["sensor"] == "Temp"]
```

```
# Remove the original combined column – no longer needed
dfLong = dfLong.drop(columns=["dataPoint"])
dfLong.head()
```

	time	value	flat	sensor
0	2018-10-03 00:00:00	53.0	FlatA	Hum
1	2018-10-03 01:00:00	53.0	FlatA	Hum
2	2018-10-03 02:00:00	53.0	FlatA	Hum
3	2018-10-03 03:00:00	53.0	FlatA	Hum
4	2018-10-03 04:00:00	53.3	FlatA	Hum

Long to Wide

```
# Pivot back to wide format using the new flat and sensor columns
dfWide = dfLong.pivot_table(index="time", columns=["flat", "sensor"], values="value
dfWide.head()
```

flat	FlatA		FlatB		FlatC		FlatD	
sensor	Hum	Temp	Hum	Temp	Hum	Temp	Hum	Temp
time								
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5	49.0	24.43
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5	49.0	24.40
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5	48.3	24.38
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5	48.0	24.33
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5	47.7	24.30

```
# Flatten the MultiIndex columns: (FlatA, Hum) -> "FlatA_Hum"
dfWide.columns = [f"{flat}_{sensor}" for flat, sensor in dfWide.columns]
dfWide.head()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

> **Why flatten MultiIndex columns?**
>
> MultiIndex columns (two levels) make simple operations harder: selecting one column needs a tuple, `df[("FlatA", "Temp")]`, instead of `df["FlatA_Temp"]`.
>
> Flat column names also work better with `.plot()`, `.to_csv()`, and most pandas functions that expect simple string column names.

Access Data

Get Values of Column(s)

```
df["FlatA_Temp"]
```

```
0        24.43
1        24.40
2        24.40
3        24.40
4        24.40
         ...
17176      NaN
17177      NaN
17178      NaN
17179      NaN
17180      NaN
Name: FlatA_Temp, Length: 17181, dtype: float64
```

Note

A Series is returned!

```
df[["FlatA_Temp", "FlatA_Hum"]]
```

	FlatA_Temp	FlatA_Hum
0	24.43	53.0
1	24.40	53.0
2	24.40	53.0
3	24.40	53.0
4	24.40	53.3
...	...	...
17176	NaN	NaN
17177	NaN	NaN
17178	NaN	NaN
17179	NaN	NaN
17180	NaN	NaN

17181 rows × 2 columns

Note

A DataFrame is returned!

Rename column name(s)

```
df.columns
```

```
Index(['time', 'FlatA_Hum', 'FlatA_Temp', 'FlatB_Hum', 'FlatB_Temp',
       'FlatC_Hum', 'FlatC_Temp', 'FlatD_Hum', 'FlatD_Temp'],
      dtype='str')
```

```
df1 = df.copy()
df1 = df1.rename({'FlatA_Temp': 'A_Temp', 'FlatA_Hum': 'A_Hum'}, axis=1)
df1.columns
```

```
Index(['time', 'A_Hum', 'A_Temp', 'FlatB_Hum', 'FlatB_Temp', 'FlatC_Hum',
       'FlatC_Temp', 'FlatD_Hum', 'FlatD_Temp'],
      dtype='str')
```

Select / Drop Column(s)

Select Column(s) to Keep

```
df1 = df[["FlatA_Temp", "FlatB_Temp"]]
df1.head()
```

	FlatA_Temp	FlatB_Temp
0	24.43	22.40
1	24.40	22.40
2	24.40	22.40
3	24.40	22.40
4	24.40	22.37

Drop Column(s) by Position

```
df2 = df.drop(columns=df.columns[[1, 2]])
df2.head()
```

	time	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp	FlatD_Hum	FlatD_Temp
0	2018-10-03 00:00:00	38.8	22.40	44.0	24.5	49.0	24.43
1	2018-10-03 01:00:00	38.8	22.40	44.0	24.5	49.0	24.40
2	2018-10-03 02:00:00	39.3	22.40	44.7	24.5	48.3	24.38
3	2018-10-03 03:00:00	40.3	22.40	45.0	24.5	48.0	24.33
4	2018-10-03 04:00:00	41.0	22.37	45.2	24.5	47.7	24.30

Drop Column(s) by Name

```
df3 = df.drop(columns=["FlatA_Hum", "FlatB_Hum", "FlatC_Hum", "FlatD_Hum"])
df3.head()
```

	time	FlatA_Temp	FlatB_Temp	FlatC_Temp	FlatD_Temp
0	2018-10-03 00:00:00	24.43	22.40	24.5	24.43
1	2018-10-03 01:00:00	24.40	22.40	24.5	24.40
2	2018-10-03 02:00:00	24.40	22.40	24.5	24.38
3	2018-10-03 03:00:00	24.40	22.40	24.5	24.33
4	2018-10-03 04:00:00	24.40	22.37	24.5	24.30

Merge / Join

```
# Two subsets; Flat B without the hours where its sensor has no reading
df1 = df[["time", "FlatA_Temp", "FlatA_Hum"]]
df2 = df[["time", "FlatB_Temp", "FlatB_Hum"]].dropna()

# Inner join: only times that exist in both DataFrames
df_inner = pd.merge(df1, df2, on="time", how="inner")
print(f"df1: {len(df1)} rows, df2: {len(df2)} rows, inner join: {len(df_inner)} row
```

```
df1: 17181 rows, df2: 16470 rows, inner join: 16470 rows
```

```
# Left join: keep all rows of df1; Flat B columns are NaN where df2 has no row
df_left = pd.merge(df1, df2, on="time", how="left")
print(f"left join: {len(df_left)} rows, {df_left['FlatB_Temp'].isna().sum()} withou
```

```
left join: 17181 rows, 711 without Flat B values
```

GroupBy

```
# Prepare data with datetime index
dfg = df.copy()
dfg["time"] = pd.to_datetime(dfg["time"], format="%Y-%m-%d %H:%M:%S")
dfg = dfg.set_index("time", drop=True)

# Group by month and compute the mean of all columns
dfg.groupby(dfg.index.month).mean()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp	FlatI
1	52.878582	22.507328	48.053329	21.937808	46.652538	22.494339	44.9
2	52.107090	22.598973	46.520015	22.072326	44.131758	22.665266	45.6
3	50.331246	23.022580	46.032542	22.357078	42.981369	23.191518	43.1
4	48.705345	23.671990	41.335835	23.126276	40.312334	23.753350	41.0
5	53.191935	23.729677	46.807392	22.978609	45.911749	23.868393	48.7
6	55.955282	24.848394	51.569353	24.442122	52.579097	25.188718	54.4
7	56.037059	25.754824	51.296904	25.727128	53.276453	26.162317	54.8
8	58.475666	25.541228	54.017340	26.107928	55.817599	26.123594	57.0
9	58.042985	24.726139	52.488072	24.399940	54.136400	24.830060	55.1
10	57.549599	23.756966	50.848219	23.022058	52.411807	23.838805	51.6
11	56.401848	22.876539	55.186107	22.789972	50.703417	23.166213	49.2
12	53.976319	22.855074	53.051560	22.518948	50.286090	22.864814	48.4

```
# Group by month with multiple aggregation functions
dfg.groupby(dfg.index.month).agg({"FlatA_Temp": ["mean", "min", "max"]})
```

	FlatA_Temp		
	mean	min	max
time			
1	22.507328	20.02	24.58
2	22.598973	19.10	24.17
3	23.022580	19.92	25.60
4	23.671990	21.82	25.00
5	23.729677	21.72	25.10
6	24.848394	21.88	28.07
7	25.754824	21.38	28.88
8	25.541228	21.85	28.17
9	24.726139	22.20	26.80
10	23.756966	21.33	25.27
11	22.876539	20.73	24.18
12	22.855074	21.07	25.42

Pipe

```
# Define simple processing functions
def drop_humidity(df):
    """Drop all humidity columns."""
    hum_cols = [c for c in df.columns if "Hum" in c]
    return df.drop(columns=hum_cols)

def add_temp_mean(df):
    """Add a column with the mean temperature across all flats."""
    temp_cols = [c for c in df.columns if "Temp" in c]
    return df.assign(Temp_Mean=df[temp_cols].mean(axis=1))   # new frame, input unc

# Method chaining with .pipe()
dfp = (
    df
    .pipe(drop_humidity)
    .pipe(add_temp_mean)
)
dfp.head()
```

	time	FlatA_Temp	FlatB_Temp	FlatC_Temp	FlatD_Temp	Temp_Mean
0	2018-10-03 00:00:00	24.43	22.40	24.5	24.43	23.9400
1	2018-10-03 01:00:00	24.40	22.40	24.5	24.40	23.9250
2	2018-10-03 02:00:00	24.40	22.40	24.5	24.38	23.9200
3	2018-10-03 03:00:00	24.40	22.40	24.5	24.33	23.9075
4	2018-10-03 04:00:00	24.40	22.37	24.5	24.30	23.8925

Time Series

A time series is a sequence of data points recorded over time — for example, a temperature sensor logging every 15 minutes or an electricity meter sending hourly readings.

In practice, time series data does not always come at **regular intervals**. IoT sensors, LoRaWAN devices, and event-based systems often transmit data at irregular timestamps

— depending on signal quality, battery level, or only when a value changes. This makes it essential to **align and regularize** timestamps before analysis.

pandas provides powerful tools for this: resampling to a fixed frequency, interpolating gaps, computing rolling statistics, and aligning multiple series to a common time axis.

```
# Load data set

import pandas as pd

df = pd.read_csv("https://raw.githubusercontent.com/retomarek/edap/main/edap/sample
                 sep = ";")
df['time'] = pd.to_datetime(df['time'], format='%Y-%m-%d %H:%M:%S')

df.head()
```

	time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
0	2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
1	2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2	2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
3	2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
4	2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

Irregular to Regular Time Series

Real-world sensor data often arrives at irregular intervals. Before you can compare, resample, or plot multiple sensors together, the timestamps need to be aligned to a **regular grid** (e.g. every 15 minutes).

```
import pandas as pd
import numpy as np

# Simulate irregular IoT sensor data (LoRaWAN-style)
np.random.seed(42)
irregular_times = pd.to_datetime([
    "2024-01-15 08:02:13", "2024-01-15 08:17:45", "2024-01-15 08:28:02",
    "2024-01-15 08:46:33", "2024-01-15 09:01:11", "2024-01-15 09:14:58",
    "2024-01-15 09:33:27", "2024-01-15 09:44:05", "2024-01-15 10:02:41",
    "2024-01-15 10:19:16", "2024-01-15 10:31:52", "2024-01-15 10:48:09",
])
df_irregular = pd.DataFrame({
    "temperature": [21.3, 21.5, 21.4, 21.8, 22.0, 22.1, 22.3, 22.2, 22.5, 22.4, 22.
}, index=irregular_times)

print("Irregular timestamps:")
df_irregular
```

```
Irregular timestamps:
```

	temperature
2024-01-15 08:02:13	21.3
2024-01-15 08:17:45	21.5
2024-01-15 08:28:02	21.4
2024-01-15 08:46:33	21.8
2024-01-15 09:01:11	22.0
2024-01-15 09:14:58	22.1
2024-01-15 09:33:27	22.3
2024-01-15 09:44:05	22.2
2024-01-15 10:02:41	22.5
2024-01-15 10:19:16	22.4
2024-01-15 10:31:52	22.6
2024-01-15 10:48:09	22.5

```
# Step 1: Resample to a regular 15-minute grid using the nearest value
df_regular = df_irregular.resample("15min").nearest()

print("Regular 15-min timestamps:")
df_regular
```

Regular 15-min timestamps:

	temperature
2024-01-15 08:00:00	21.3
2024-01-15 08:15:00	21.5
2024-01-15 08:30:00	21.4
2024-01-15 08:45:00	21.8
2024-01-15 09:00:00	22.0
2024-01-15 09:15:00	22.1
2024-01-15 09:30:00	22.3
2024-01-15 09:45:00	22.2
2024-01-15 10:00:00	22.5
2024-01-15 10:15:00	22.4
2024-01-15 10:30:00	22.6
2024-01-15 10:45:00	22.5

Alignment methods

There are several ways to align irregular data to a regular grid:

Method	Code	Best for
Nearest	`.resample("15min").nearest()`	Sensor values that change slowly (temperature, humidity)
Mean	`.resample("15min").mean()`	Multiple readings per interval — averages them
Interpolate	`.resample("15min").interpolate()`	Smooth signals, fills gaps with linear interpolation
Forward fill	`.resample("15min").ffill()`	State data (on/off) — carries last known value forward

Why is this important? Many analysis methods (resampling, rolling mean, merging multiple sensors) require a regular time index. If two sensors report at different irregular times, you cannot directly compare or correlate them without aligning first.

Datetime Index

Setting the datetime column as the DataFrame **index** unlocks powerful time-based features in pandas: slicing by date range (`df.loc["2024-01"]`), resampling (`.resample()`), and automatic alignment when combining multiple DataFrames. Without a datetime index, these operations require extra code and are slower.

```
# set index and remove column
df = df.set_index("time", drop=True)

# remove duplicates
df = df[~df.index.duplicated(keep='first')]

df.head()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

> **Note**
>
> The index column with 0, 1, 2 etc. has gone and now datetime is the index!

Upsampling

Increase the frequency of the samples, such as from hours to 15min

```
df15min = df.resample("15min").interpolate(method="linear")
df15min.head()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.4300	38.8	22.4	44.0	24.5
2018-10-03 00:15:00	53.0	24.4225	38.8	22.4	44.0	24.5
2018-10-03 00:30:00	53.0	24.4150	38.8	22.4	44.0	24.5
2018-10-03 00:45:00	53.0	24.4075	38.8	22.4	44.0	24.5
2018-10-03 01:00:00	53.0	24.4000	38.8	22.4	44.0	24.5

Upsample methods

When upsampling (e.g. hourly to 15-min), you need to fill the new empty rows. Available methods:

Method	Code	Description
Linear interpolation	`.interpolate(method="linear")`	Straight line between known points — good default
Spline interpolation	`.interpolate(method="spline", order=2)`	Smooth curve — more natural for sensor data
Forward fill	`.ffill()`	Carries the last known value forward
Backward fill	`.bfill()`	Uses the next known value
Nearest	`.nearest()`	Uses the closest value in time

Frequency aliases

Alias	Meaning	Example
`"s"`	seconds	`"30s"`
`"min"`	minutes	`"15min"`
`"h"`	hours	`"h"`, `"6h"`
`"D"`	calendar days	`"D"`
`"W"`	weeks (ending on Sunday)	`"W"`, `"W-MON"`
`"MS"` / `"ME"`	month start / month end	`"MS"`
`"QS"` / `"YS"`	quarter start / year start	`"YS"`

Older code uses `"T"`, `"H"`, `"M"` or `"A"`: these aliases were renamed in pandas 2.2 and are rejected by pandas 3.

Downsampling

Decrease the frequency of the samples, such as from hours to days

```
dfDaily = df.resample("D").mean()
dfDaily.head()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp	Fla
2018-10-03	50.547826	24.200435	43.379167	22.627917	46.070833	24.632500	50.(
2018-10-04	54.033333	24.232083	47.366667	22.797500	47.812500	24.655417	48.(
2018-10-05	52.682609	24.210435	47.858333	23.095417	47.962500	24.747500	50.5
2018-10-06	52.708696	24.180435	49.645833	23.322500	51.183333	24.600000	52.(
2018-10-07	56.058333	24.296667	51.662500	23.587083	53.541667	24.565417	50.9

Other downsample methods

Method	Description
`.mean()`	Average value per period
`.sum()`	Total per period
`.min()`	Minimum value per period
`.max()`	Maximum value per period
`.median()`	Median value per period
`.first()` / `.last()`	First or last value per period
`.count()`	Number of data points per period

Rolling Windows

A rolling window computes a statistic (mean, std, etc.) over a sliding window of fixed size. This smooths noisy sensor data and reveals underlying trends — for example, a 24-hour rolling mean removes short-term fluctuations caused by window openings (visible as sudden temperature drops in the plot) and reveals the actual room temperature trend.

```
import matplotlib.pyplot as plt

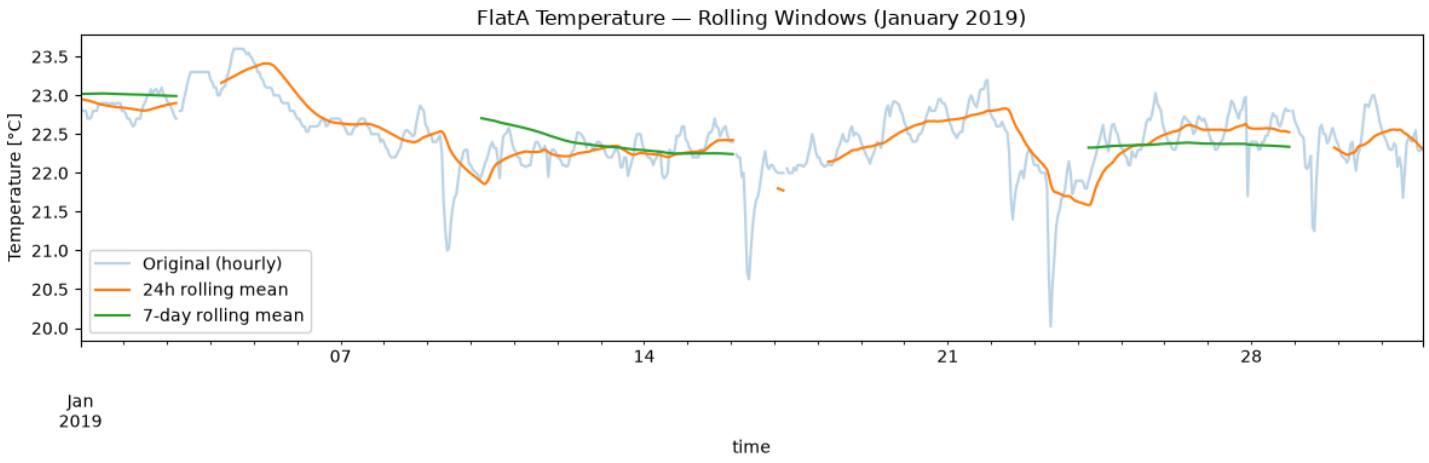
# 24-hour moving average (1 day). window=24 counts rows: 24 rows are 24 hours here
# every hour has a row (missing readings are NaN). For irregular data use rolling("
df["FlatA_Temp_24h"] = df["FlatA_Temp"].rolling(window=24).mean()

# 168-hour moving average (7 days)
df["FlatA_Temp_7d"] = df["FlatA_Temp"].rolling(window=168).mean()

# Plot one month to see the effect clearly
df_jan = df.loc["2019-01"]

fig, ax = plt.subplots(figsize=(12, 4))
df_jan["FlatA_Temp"].plot(ax=ax, alpha=0.3, label="Original (hourly)")
df_jan["FlatA_Temp_24h"].plot(ax=ax, label="24h rolling mean")
df_jan["FlatA_Temp_7d"].plot(ax=ax, label="7-day rolling mean")
ax.set_ylabel("Temperature [°C]")
ax.set_title("FlatA Temperature — Rolling Windows (January 2019)")
ax.legend()
plt.tight_layout()
plt.show()

# Clean up helper columns
df = df.drop(columns=["FlatA_Temp_24h", "FlatA_Temp_7d"])
```



Handling Missing Data in Rolling Windows

In the plot above, the rolling mean lines have **gaps**. This happens because `.rolling()` returns `NaN` whenever the window contains missing values — and real-world sensor data almost always has gaps (sensor offline, transmission errors, maintenance).

Before applying rolling windows, it is often useful to **fill these gaps** first.

```
# Check: how many missing values per column?
df_jan = df.loc["2019-01", ["FlatA_Temp"]].copy()

# Resample to explicit hourly grid to reveal hidden gaps
df_jan = df_jan.resample("h").mean()

print(f"Total rows:    {len(df_jan)}")
print(f"Missing (NaN): {df_jan['FlatA_Temp'].isna().sum()}")
print(f"Coverage:      {df_jan['FlatA_Temp'].notna().mean():.1%}")
```

```
Total rows:    744
Missing (NaN): 4
Coverage:      99.5%
```

```
# Fill gaps with linear interpolation
df_jan["FlatA_Temp_filled"] = df_jan["FlatA_Temp"].interpolate(method="linear")

print(f"Missing after fill: {df_jan['FlatA_Temp_filled'].isna().sum()}")
```

```
Missing after fill: 0
```

```
import matplotlib.pyplot as plt

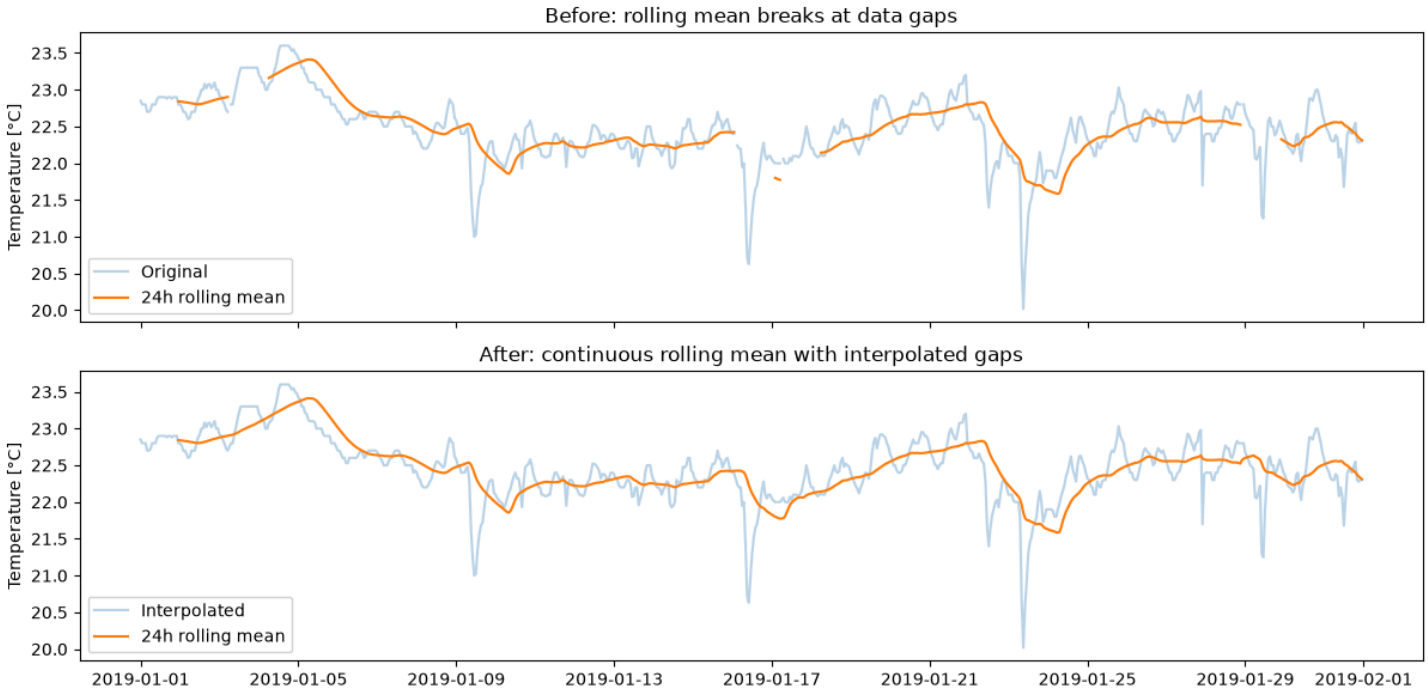
# Compare: rolling mean with and without gap filling
df_jan["rolling_with_gaps"] = df_jan["FlatA_Temp"].rolling(window=24).mean()
df_jan["rolling_filled"] = df_jan["FlatA_Temp_filled"].rolling(window=24).mean()

fig, axes = plt.subplots(2, 1, figsize=(12, 6), sharex=True)

# Top: with gaps
axes[0].plot(df_jan.index, df_jan["FlatA_Temp"], alpha=0.3, label="Original")
axes[0].plot(df_jan.index, df_jan["rolling_with_gaps"], label="24h rolling mean")
axes[0].set_ylabel("Temperature [°C]")
axes[0].set_title("Before: rolling mean breaks at data gaps")
axes[0].legend()

# Bottom: filled
axes[1].plot(df_jan.index, df_jan["FlatA_Temp_filled"], alpha=0.3, label="Interpola
axes[1].plot(df_jan.index, df_jan["rolling_filled"], label="24h rolling mean")
axes[1].set_ylabel("Temperature [°C]")
axes[1].set_title("After: continuous rolling mean with interpolated gaps")
axes[1].legend()

plt.tight_layout()
plt.show()
```



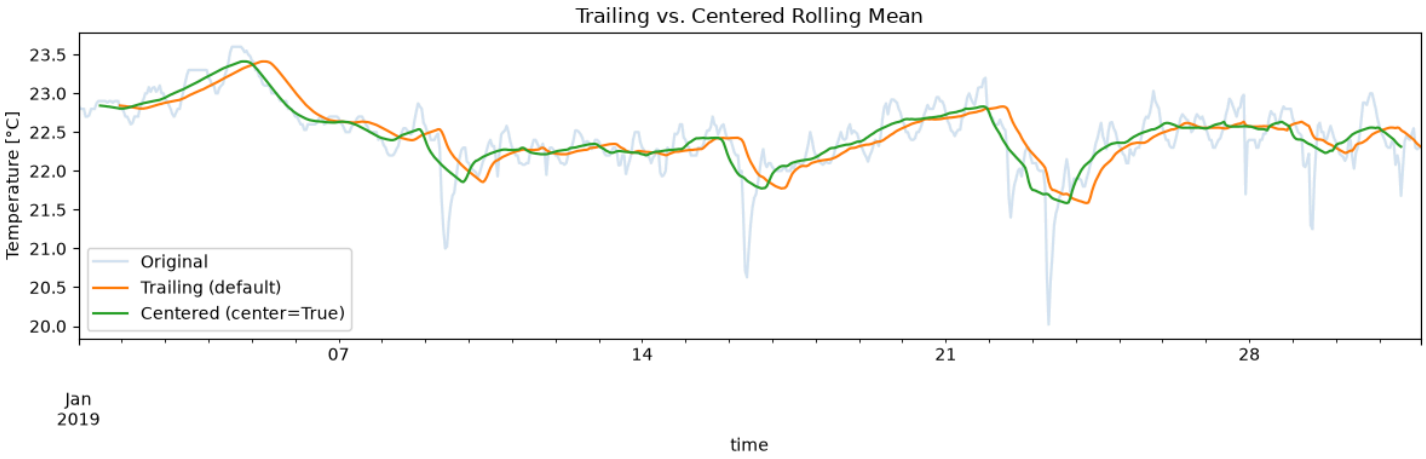
Centered vs. Trailing Rolling Mean

By default, `.rolling()` computes a **trailing** mean — it uses the current value and the N−1 values before it. This shifts the result to the right (it "lags" behind the data).

A **centered** rolling mean (`center=True`) uses values on both sides of each point, producing a symmetric smoothing without time shift.

```
# Trailing vs. centered rolling mean
df_jan["trailing_24h"] = df_jan["FlatA_Temp_filled"].rolling(window=24).mean()
df_jan["centered_24h"] = df_jan["FlatA_Temp_filled"].rolling(window=24, center=True

fig, ax = plt.subplots(figsize=(12, 4))
df_jan["FlatA_Temp_filled"].plot(ax=ax, alpha=0.2, label="Original")
df_jan["trailing_24h"].plot(ax=ax, label="Trailing (default)")
df_jan["centered_24h"].plot(ax=ax, label="Centered (center=True)")
ax.set_ylabel("Temperature [°C]")
ax.set_title("Trailing vs. Centered Rolling Mean")
ax.legend()
plt.tight_layout()
plt.show()
```



When to use which?

Method	Code	When to use
Centered	`.rolling(24, center=True).mean()`	**General smoothing and trend analysis** — no time shift, shows the true trend at each point in time. Best for visualizations and reports.
Trailing	`.rolling(24).mean()`	**Causal analysis and real-time applications** — only uses past data, which is physically meaningful. Use when modeling how outdoor temperature influences indoor temperature with a time lag (thermal inertia of the building).

Example — building thermal inertia: A building does not react immediately to outdoor temperature changes. A trailing rolling mean of 24h outdoor temperature better represents the “effective” outdoor temperature that the building actually “feels” at a given moment. This lagged signal correlates better with indoor temperature or heating demand than the instantaneous outdoor temperature.

For pure data visualization and trend detection, **centered** is usually the better choice.

Gap filling strategies

Method	Code	Best for
Linear interpolation	`.interpolate(method="linear", limit_area="inside")`	Short gaps in smooth signals (temperature, humidity)
Forward fill	`.ffill()`	State data (on/off) — carries the last known state forward
Spline	`.interpolate(method="spline", order=2)`	More natural curves for sensor data
Leave as NaN	—	Long gaps where interpolation would be misleading

`limit_area="inside"` fills only gaps between two valid values; without it, pandas also extends the last value over missing data at the end of a series.

Careful with `limit`: `interpolate(limit=6)` or `ffill(limit=6)` does *not* skip long gaps: it fills the first 6 values of every gap, also of a two-day gap. To fill only gaps of up to 6 hours and leave longer gaps as NaN, measure the length of each gap first:

```
s = df["FlatA_Temp"]
gap_len = s.isna().groupby(s.notna().cumsum()).transform("sum")   # length o
filled = s.interpolate(limit_area="inside").where(s.notna() | (gap_len <= 6)
```

Exploratory Data Analysis (EDA)

Exploratory Data Analysis is the most important step in any data project. Before building models or drawing conclusions, you need to **understand your data** — its structure, quality, patterns, and anomalies.

For sensor data, EDA typically answers questions like:

- Are there gaps or outliers in the sensor readings?
- What are the daily and seasonal patterns?
- How do different variables relate to each other (e.g., outdoor temperature vs. heating energy)?
- Has something changed over time (e.g., HVAC system replacement, occupancy changes)?

This chapter walks through a systematic EDA workflow using a real temperature and humidity dataset from four flats.

EDA Workflow Overview

The first four steps are shown here; the deeper analyses each have their own recipe in Part III, linked in the checklist at the end.

Step 1	Step 2	Step 3	Step 4
Load & Inspect	**Data Quality**	**Time Series Overview**	**Distribution Analysis**
`shape`, `info()`, `describe()`	Missing values, duplicates, gaps	Plot raw data, zoom in	Histograms

Step 1: Load and Inspect

Always start by loading the data, parsing dates, and getting a first overview. If you are unfamiliar with loading data in pandas, see the Loading Data chapter first.

```
import pandas as pd
import numpy as np

url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTe

df = pd.read_csv(url, sep=";")
df["time"] = pd.to_datetime(df["time"])
df = df.set_index("time")

print(f"Shape: {df.shape}")
print(f"Period: {df.index.min()} to {df.index.max()}")
print(f"Duration: {df.index.max() - df.index.min()}")
df.head()
```

```
Shape: (17181, 8)
Period: 2018-10-03 00:00:00 to 2020-09-17 20:00:00
Duration: 715 days 20:00:00
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

```
df.describe()
```

	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_T
count	16484.000000	16484.000000	16470.000000	16470.000000	16497.000000	16497.0
mean	54.336599	23.786531	49.666697	23.404749	48.895339	23.9646
std	6.300389	1.273444	6.903822	1.470354	6.487768	1.36001
min	23.500000	19.100000	25.000000	20.530000	22.500000	21.3800
25%	51.000000	22.800000	44.700000	22.320000	44.300000	22.8800
50%	54.800000	23.480000	50.000000	23.000000	49.200000	23.6000
75%	58.300000	24.600000	54.500000	24.100000	53.800000	24.8000
max	76.500000	28.880000	73.300000	29.120000	69.800000	28.8000

> **What to look for in describe()**
>
> - **count** differs between columns? → some columns have more missing values
> - **min/max** outside physically plausible range? → possible sensor error (e.g., temperature of -999 or 99)
> - **mean vs. 50%** (median) very different? → skewed data, possibly caused by outliers
> - **std** very large relative to mean? → high variability or mixed data

Step 2: Data Quality

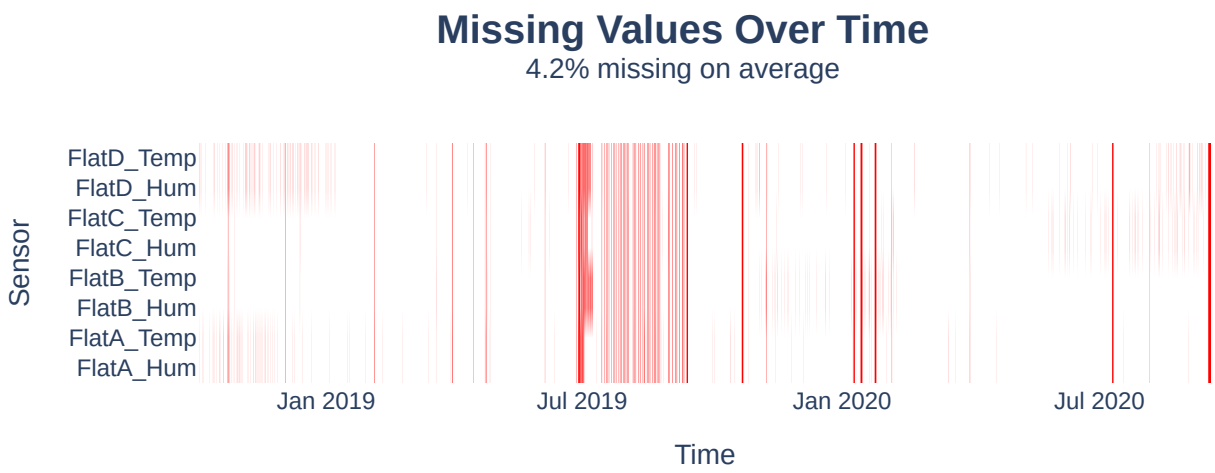
Real-world sensor data almost always has quality issues: missing values, duplicate timestamps, or irregular intervals. Identifying these early prevents wrong conclusions later.

```
# Missing values per column
missing = pd.DataFrame({
    "missing": df.isna().sum(),
    "% missing": (df.isna().mean() * 100).round(1)
})
missing
```

	missing	% missing
FlatA_Hum	697	4.1
FlatA_Temp	697	4.1
FlatB_Hum	711	4.1
FlatB_Temp	711	4.1
FlatC_Hum	684	4.0
FlatC_Temp	684	4.0
FlatD_Hum	827	4.8
FlatD_Temp	827	4.8

```
from pyedautils.data_quality import plot_missing_values_heatmap

fig = plot_missing_values_heatmap(df)
fig.show()
```



Interpreting the missing values heatmap

Red bands indicate periods where sensors reported no data. Common causes:

- **Vertical red band across all sensors** → system-wide outage (network, gateway, power)
- **Single sensor red band** → individual sensor failure or battery empty
- **Red at the end** → sensors stopped reporting (decommissioned or data export cutoff)

```
# Check for duplicate timestamps and irregular intervals
print(f"Duplicate timestamps: {df.index.duplicated().sum()}")
print(f"Index is monotonic:  {df.index.is_monotonic_increasing}")

# Most common interval
time_diffs = df.index.to_series().diff().dropna()
print(f"Most common interval: {time_diffs.mode()[0]}")
print(f"Min interval: {time_diffs.min()}")
print(f"Max interval: {time_diffs.max()}")
```

```
Duplicate timestamps: 0
Index is monotonic:  True
Most common interval: 0 days 01:00:00
Min interval: 0 days 01:00:00
Max interval: 0 days 01:00:00
```

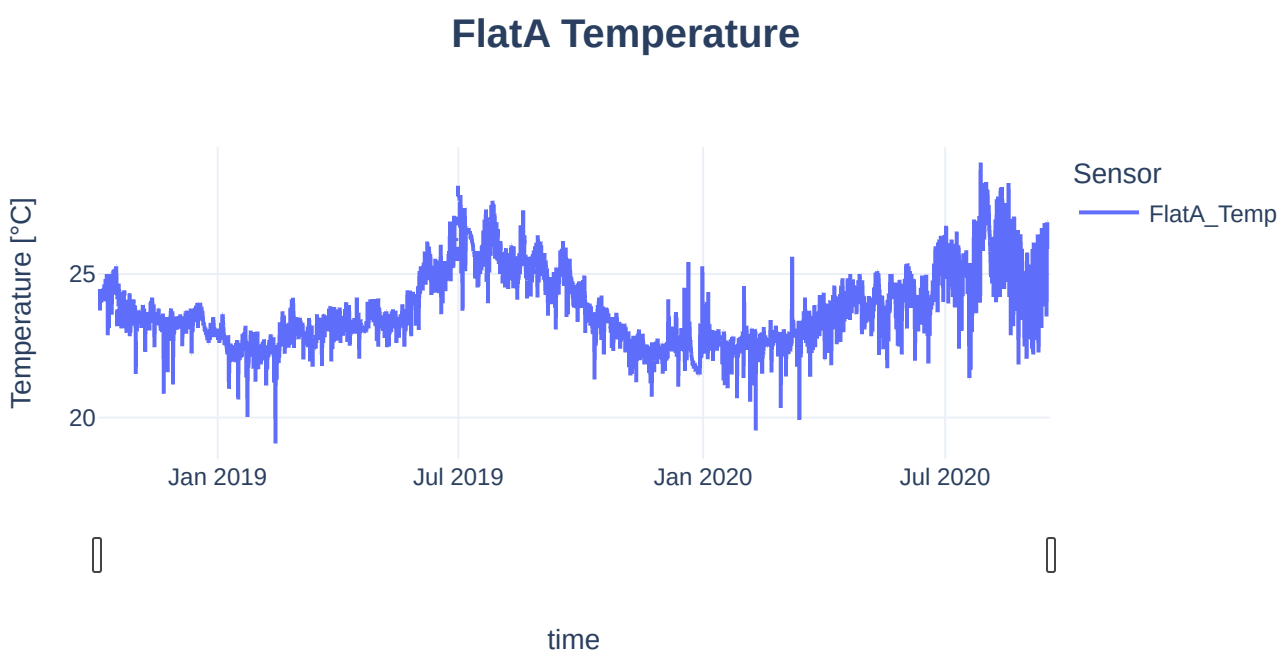
Step 3: Time Series Overview

Plot the raw data to get a visual overview. Interactive plots allow zooming into specific periods.

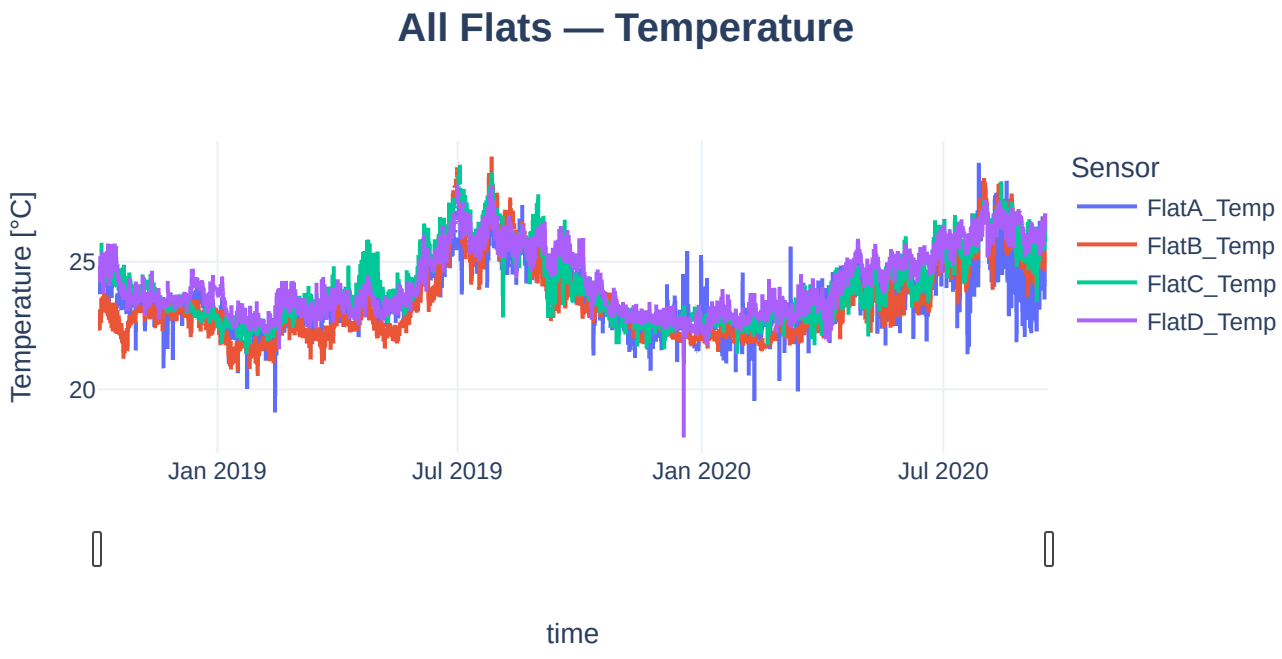
```
from pyedautils.plots import plot_timeseries

fig = plot_timeseries(df, columns=["FlatA_Temp"], title="FlatA Temperature", ylab="
fig.show()

# Column list reused in later cells
temp_cols = [col for col in df.columns if "Temp" in col]
```




```
fig = plot_timeseries(df, columns=temp_cols, title="All Flats — Temperature", ylab=
fig.show()
```

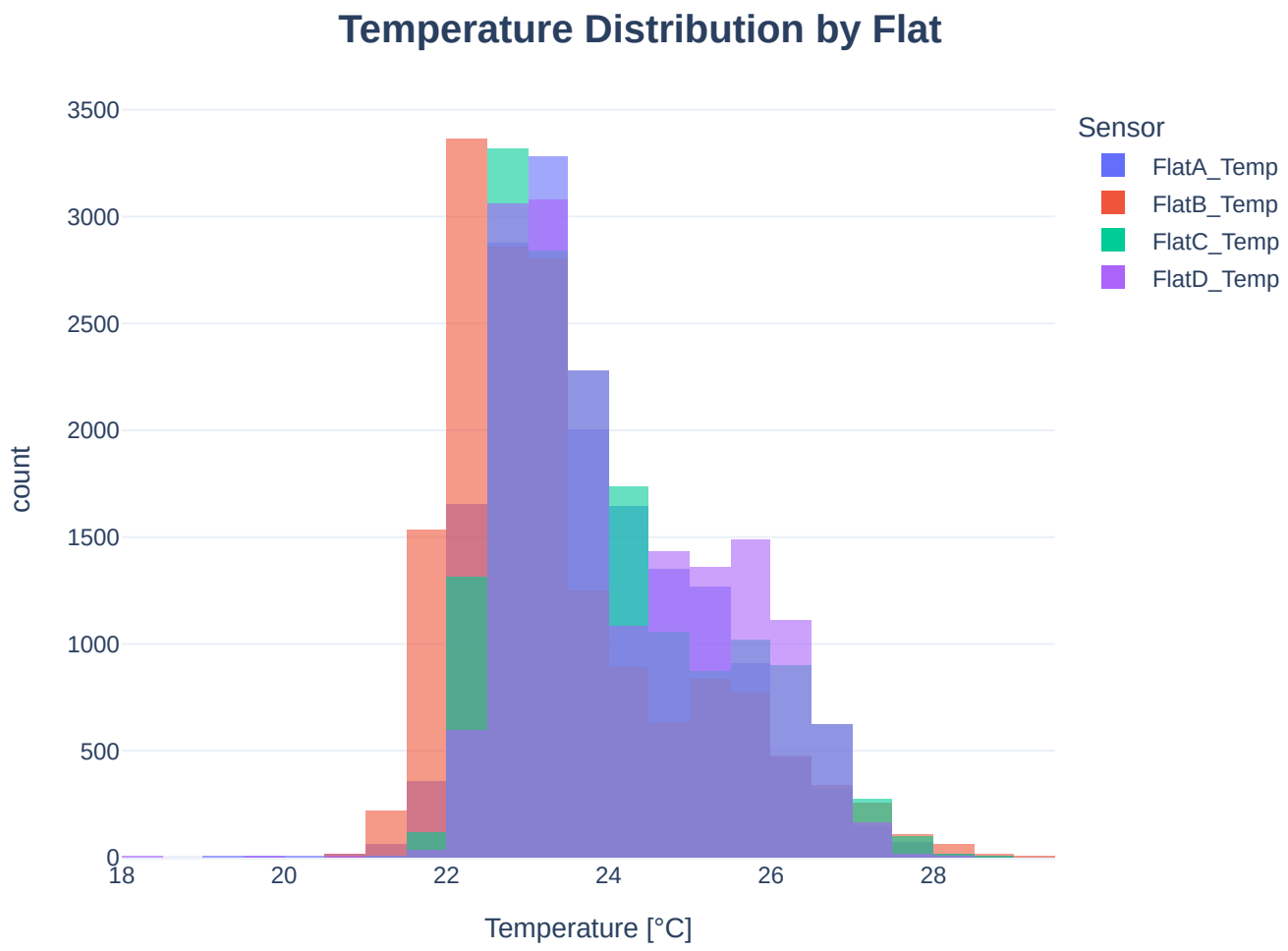


Step 4: Distribution Analysis

A histogram reveals the **distribution** of values — how often each temperature occurs and whether there are unusual clusters. Box plots per month or hour show how the distribution changes over time (see Boxplot).

```
from pyedautils.plots import plot_distribution

fig = plot_distribution(df, columns=temp_cols, title="Temperature Distribution by F
fig.show()
```



> **What distribution patterns tell you**
>
> - **Two peaks** (bimodal) → different operating modes (e.g., heating season and summer, occupied/unoccupied)
> - **Long tail on one side** → occasional events (window openings, sun) or sensor problems
> - **One flat with a very narrow distribution** → heavily controlled room or a stuck sensor

Steps 5 to 8 (outliers, correlation, decomposition, autocorrelation) each have their own recipe in Part III; the checklist below links them.

EDA Checklist

Use this checklist as a starting point for every new dataset:

Step	Question	Where in this book
1. Inspect	How big? What columns? What types?	`shape`, `dtypes`, `info()`, `describe()` (above)
2. Quality	Missing values? Duplicates? Gaps?	Data Quality
3. Visualize	What does the raw data look like?	`plot_timeseries()` (above)
4. Distribute	How are values distributed?	Histogram, Boxplot
5. Outliers	Are there suspicious values?	Outlier Detection
6. Correlate	How do variables relate?	Correlation & Scatter, Cross-Correlation
7. Decompose	What are the trend and seasonal patterns?	Time Series Decomposition
8. Cycles	What are the dominant cycles?	Autocorrelation

Data Quality

Before performing any analysis, it is essential to assess and improve the quality of the data. Real-world sensor data often contains missing values, gaps and duplicates.

This chapter covers common techniques for detecting and handling these issues. Implausible values (outliers) are covered in the recipe Outlier Detection.

```
import pandas as pd
import numpy as np
import plotly.express as px
import plotly.graph_objects as go
```

```
url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTe

df = pd.read_csv(url, sep=";")
df["time"] = pd.to_datetime(df["time"])
df = df.set_index("time")
df.head()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

Missing Values

Missing values (`NaN`) are common in sensor data due to communication failures, sensor outages, or maintenance periods.

```
# Count missing values per column
df.isna().sum()
```

```
FlatA_Hum     697
FlatA_Temp    697
FlatB_Hum     711
FlatB_Temp    711
FlatC_Hum     684
FlatC_Temp    684
FlatD_Hum     827
FlatD_Temp    827
dtype: int64
```

```
# Percentage of missing values
missing_pct = (df.isna().sum() / len(df)) * 100
missing_pct.round(2)
```

```
FlatA_Hum     4.06
FlatA_Temp    4.06
FlatB_Hum     4.14
FlatB_Temp    4.14
FlatC_Hum     3.98
FlatC_Temp    3.98
FlatD_Hum     4.81
FlatD_Temp    4.81
dtype: float64
```

```
# Detailed info including non-null counts
df.info()
```

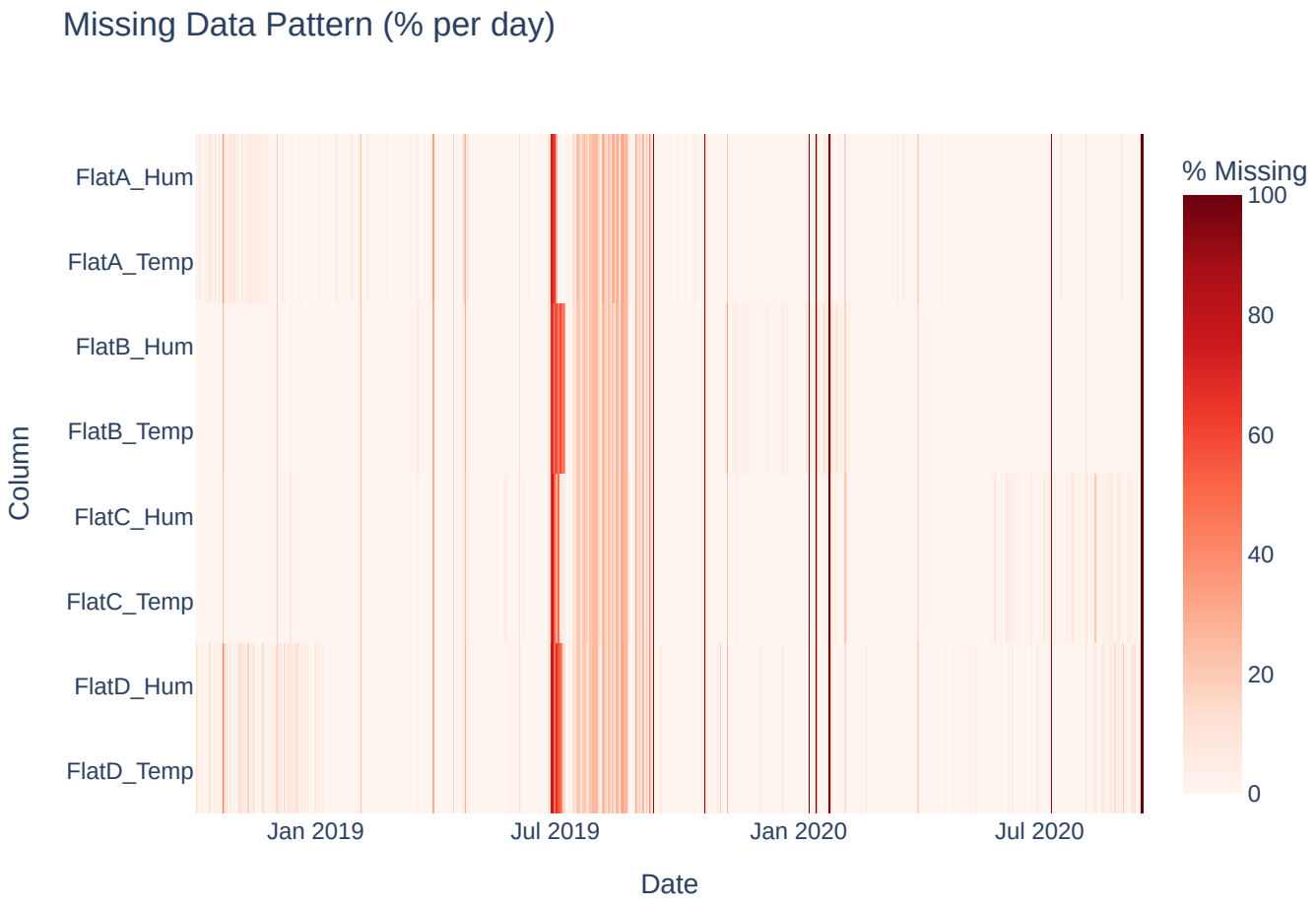
```
<class 'pandas.DataFrame'>
DatetimeIndex: 17181 entries, 2018-10-03 00:00:00 to 2020-09-17 20:00:00
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   FlatA_Hum   16484 non-null  float64
 1   FlatA_Temp  16484 non-null  float64
 2   FlatB_Hum   16470 non-null  float64
 3   FlatB_Temp  16470 non-null  float64
 4   FlatC_Hum   16497 non-null  float64
 5   FlatC_Temp  16497 non-null  float64
 6   FlatD_Hum   16354 non-null  float64
 7   FlatD_Temp  16354 non-null  float64
dtypes: float64(8)
memory usage: 1.2 MB
```

Visualizing Missing Data

A heatmap of missing values helps identify patterns — for example, whether all sensors fail at the same time.

```
# Resample to daily resolution for a cleaner visualization
missing_daily = df.isna().resample("D").mean() * 100

fig = px.imshow(
    missing_daily.T,
    labels=dict(x="Date", y="Column", color="% Missing"),
    title="Missing Data Pattern (% per day)",
    color_continuous_scale="Reds",
    aspect="auto"
)
fig.show()
```



Detecting Gaps in Time Series

A gap shows up in one of two ways: as **missing timestamps** (no row at all), or as rows whose values are `NaN`. Check both.

```
# 1) Missing timestamps: expected vs. actual number of hourly rows
expected = pd.date_range(df.index.min(), df.index.max(), freq="h")
print(f"Expected timestamps: {len(expected)}")
print(f"Actual timestamps:   {len(df)}")
```

```
Expected timestamps: 17181
Actual timestamps:   17181
```

```
# 2) Gaps as NaN rows: length of each run of missing values
s = df["FlatA_Temp"]
gap_len = s.isna().groupby(s.notna().cumsum()).sum()  # NaN rows after each value
gap_len = gap_len[gap_len > 0]
print(f"FlatA_Temp: {len(gap_len)} gaps, longest {gap_len.max()} hours, "
      f"{(gap_len > 6).sum()} longer than 6 hours")
```

```
FlatA_Temp: 171 gaps, longest 47 hours, 23 longer than 6 hours
```

Duplicates

Duplicate rows can occur when data is merged from multiple sources or exported incorrectly.

```
# df.duplicated() compares the values only, not the timestamp
dup_rows = df.duplicated()
empty_rows = df.isna().all(axis=1)
print(f"Rows equal to an earlier row: {dup_rows.sum()}")
print(f"... of which completely empty (all sensors NaN): {(dup_rows & empty_rows).s
```

```
Rows equal to an earlier row: 544
... of which completely empty (all sensors NaN): 538
```

```
# Duplicate timestamps are what matters for time series
print(f"Number of duplicate timestamps: {df.index.duplicated().sum()}")
```

```
Number of duplicate timestamps: 0
```

```
# Remove duplicates (if any)
df_clean = df[~df.index.duplicated(keep="first")]
print(f"Rows before: {len(df)}, after: {len(df_clean)}")
```

```
Rows before: 17181, after: 17181
```

> **Tip**
>
> Use `keep="first"` to keep the first occurrence, `keep="last"` to keep the last, or `keep=False` to remove all duplicates.

Interpolation and Filling Missing Values

Once missing values are identified, they can be filled using various strategies.

Linear Interpolation

```
col = "FlatA_Temp"

# Original data with NaN values
print(f"NaN values before: {df[col].isna().sum()}")

# Interpolate linearly, only between two valid values (limit_area="inside")
df_interp = df.copy()
df_interp[col] = df_interp[col].interpolate(method="linear", limit_area="inside")

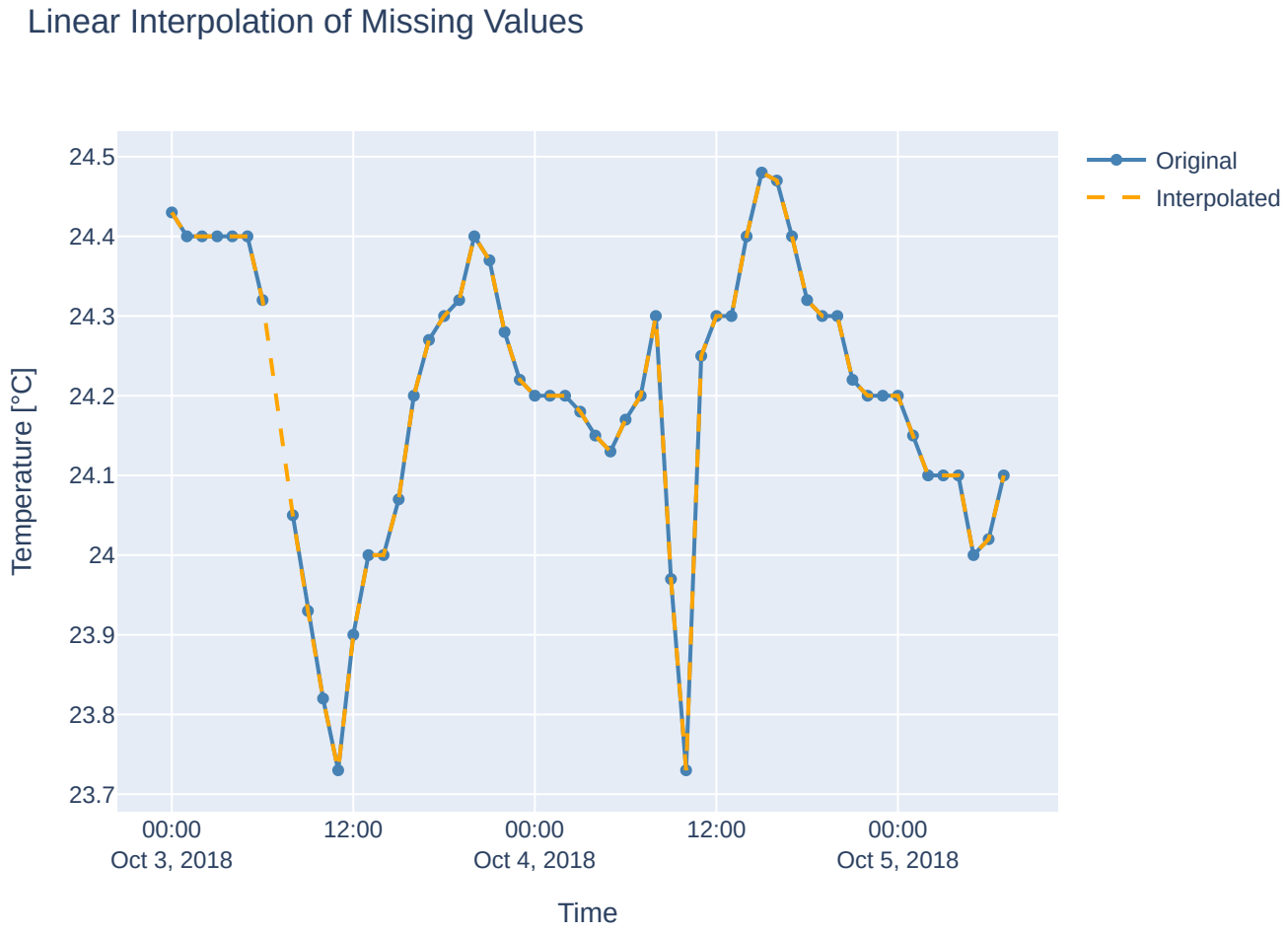
# The NaN values left are at the end of the series, where nothing can be interpolat
print(f"NaN values after:  {df_interp[col].isna().sum()}")
```

```
NaN values before: 697
NaN values after:  47
```

Interpolation also bridges long gaps with a straight line. To fill only short gaps, see *Gap filling strategies* in Time Series.

```
# Visualize a section with interpolated values
# Find a region with NaN values
nan_mask = df[col].isna()
if nan_mask.any():
    first_nan = df[nan_mask].index[0]
    start = first_nan - pd.Timedelta(days=2)
    end = first_nan + pd.Timedelta(days=2)

    fig = go.Figure()
    fig.add_trace(go.Scatter(
        x=df.loc[start:end].index, y=df.loc[start:end, col],
        mode="lines+markers", name="Original", line=dict(color="steelblue")
    ))
    fig.add_trace(go.Scatter(
        x=df_interp.loc[start:end].index, y=df_interp.loc[start:end, col],
        mode="lines", name="Interpolated", line=dict(color="orange", dash="dash")
    ))
    fig.update_layout(title="Linear Interpolation of Missing Values", xaxis_title="
    fig.show()
```



Forward and Backward Fill

```
# Forward fill: carry the last valid observation forward
df_ffill = df.copy()
df_ffill[col] = df_ffill[col].ffill()

# Backward fill: carry the next valid observation backward
df_bfill = df.copy()
df_bfill[col] = df_bfill[col].bfill()

print(f"Forward fill NaN remaining: {df_ffill[col].isna().sum()}")
print(f"Backward fill NaN remaining: {df_bfill[col].isna().sum()}")
```

```
Forward fill NaN remaining: 0
Backward fill NaN remaining: 47
```

Fill with a Constant

```
# Fill with the column mean
df_mean_fill = df.copy()
df_mean_fill[col] = df_mean_fill[col].fillna(df[col].mean())

print(f"NaN remaining: {df_mean_fill[col].isna().sum()}")
```

```
NaN remaining: 0
```

> **Note**
>
> Choose the imputation method carefully based on the nature of the data. Linear interpolation works well for slowly changing sensor data. Forward fill is appropriate when the last known value is the best estimate. Filling with the mean is generally not recommended for time series data.

Summary Statistics for Quality Assessment

A quick quality overview can be compiled by combining several checks.

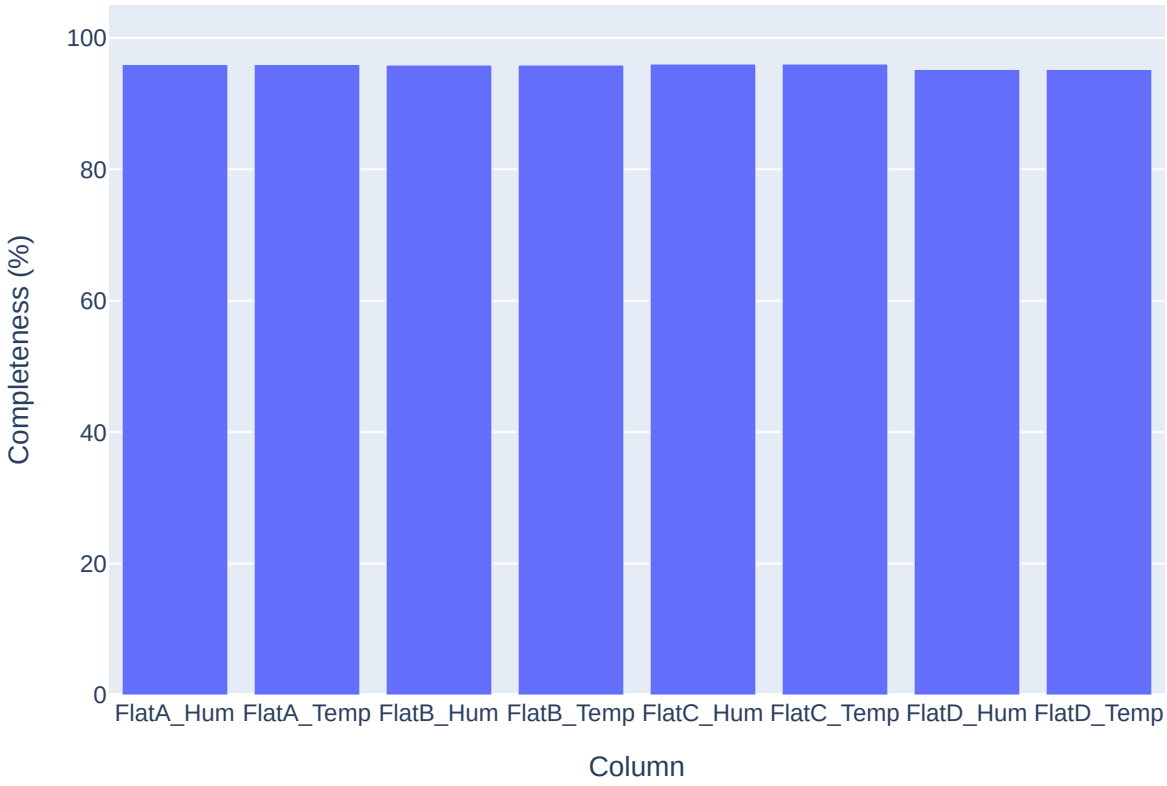
```
quality_summary = pd.DataFrame({
    "total_rows": len(df),
    "non_null": df.count(),
    "null_count": df.isna().sum(),
    "null_pct": (df.isna().sum() / len(df) * 100).round(2),
    "min": df.min(),
    "max": df.max(),
    "mean": df.mean().round(2),
    "std": df.std().round(2)
})

quality_summary
```

	total_rows	non_null	null_count	null_pct	min	max	mean	std
FlatA_Hum	17181	16484	697	4.06	23.50	76.50	54.34	6.30
FlatA_Temp	17181	16484	697	4.06	19.10	28.88	23.79	1.27
FlatB_Hum	17181	16470	711	4.14	25.00	73.30	49.67	6.90
FlatB_Temp	17181	16470	711	4.14	20.53	29.12	23.40	1.47
FlatC_Hum	17181	16497	684	3.98	22.50	69.80	48.90	6.49
FlatC_Temp	17181	16497	684	3.98	21.38	28.80	23.96	1.36
FlatD_Hum	17181	16354	827	4.81	20.50	69.80	49.30	6.52
FlatD_Temp	17181	16354	827	4.81	18.12	28.03	24.13	1.31

```
# Visualize data completeness
completeness = (1 - df.isna().mean()) * 100

fig = px.bar(
    x=completeness.index,
    y=completeness.values,
    labels={"x": "Column", "y": "Completeness (%)"},
    title="Data Completeness per Column"
)
fig.update_layout(yaxis_range=[0, 105])
fig.show()
```



Meteo Data

Weather data is essential for energy data analysis — outdoor temperature drives heating demand, solar radiation affects PV production, and humidity influences cooling loads. In Switzerland, two main **open data sources** provide free weather station data:

Provider	Description	Data Access	Resolution
MeteoSwiss	Federal Office of Meteorology and Climatology, official network	Station lookup + data download (open data)	10 min, hourly, daily
Agrometeo	Agricultural weather network	Station lookup + data download	Hourly

The `pyedautils` package provides a unified interface to both providers. This chapter shows how to:

1. Find the nearest weather station to any Swiss location
2. Download hourly weather data
3. Compare both station networks

> **Note**
>
> Both providers require an internet connection. All API calls go to public endpoints — no API key needed.

```
import pandas as pd
```

Station Lookup

Both providers let you find the nearest weather station to a given location. The typical workflow is:

1. Convert a **postal code (PLZ)** to GPS coordinates
2. Find the **nearest station** that has the sensor you need

Helper: PLZ to Coordinates

The `geopy` module converts any Swiss postal code to latitude/longitude and altitude.

```
from pyedautils.geopy import get_coordindates_ch_plz, get_altitude_lat_long

plz = 6048  # Horw
coord = get_coordindates_ch_plz(plz)
altitude = get_altitude_lat_long(coord[0], coord[1])

print(f"PLZ {plz}: lat={coord[0]:.4f}, lon={coord[1]:.4f}, altitude={altitude:.0f}
```

```
PLZ 6048: lat=47.0108, lon=8.3039, altitude=455 m
```

MeteoSwiss

MeteoSwiss operates Switzerland's official automatic weather stations. The `find_nearest_station()` function returns the **station abbreviation** (e.g. `"LUZ"` for Luzern) of the closest station that has the requested sensor and is within ±150 m altitude.

```
from pyedautils.weather.meteo_swiss import find_nearest_station as find_meteoswiss

station_ms = find_meteoswiss(coord[0], coord[1], altitude, sensor="temp")
print(f"Nearest MeteoSwiss station for temperature: {station_ms}")
```

```
Nearest MeteoSwiss station for temperature: LUZ
```

You can search for different sensor types: `"temp"`, `"globrad"` (global radiation), `"relhum"` (relative humidity), `"rain"`.

```
for sensor in ["temp", "globrad", "relhum", "rain"]:
    station = find_meteoswiss(coord[0], coord[1], altitude, sensor=sensor)
    print(f"  {sensor:8s} → {station}")
```

```
  temp     → LUZ
  globrad  → LUZ
  relhum   → LUZ
  rain     → LUZ
```

Agrometeo

Agrometeo is a Swiss agricultural weather network with a public API for station lookup and data download.

```
from pyedautils.weather.agroweather import find_nearest_station as find_agro

station_agro = find_agro(coord[0], coord[1], sensor="temp")
print(f"Nearest Agrometeo station for temperature: {station_agro}")
```

```
Nearest Agrometeo station for temperature: 239
```

Station Networks

Both providers expose a function to list all available stations. This is useful for understanding coverage and available sensors.

```
from pyedautils.weather.meteo_swiss import get_current_station_data

stations_ms = get_current_station_data()
print(f"MeteoSwiss: {len(stations_ms)} stations")
stations_ms[["Abk.", "Station", "Stationshöhe m ü. M."]].head()
```

```
MeteoSwiss: 300 stations
```

	Abk.	Station	Stationshöhe m ü. M.
0	TAE	Aadorf / Tänikon	539.0
1	ABE	Aarberg	444.0
2	COM	Acquarossa / Comprovasco	575.0
3	ABO	Adelboden	1321.0
4	AIE	Affoltern i. E.	755.0

```
from pyedautils.weather.agroweather import get_station_data

stations_agro = get_station_data()
print(f"Agrometeo: {len(stations_agro)} stations")
stations_agro[["id", "name", "lat", "lon", "altitude"]].head()
```

```
Agrometeo: 218 stations
```

	id	name	lat	lon	altitude
0	189	AESCH-BL	47.466863	7.575196	382
1	360	AGNO-Progetto ViSo-KS	45.997000	8.893944	364
2	94	AIGLE	46.314537	6.979451	442
3	78	AMBURNEX	46.536925	6.215849	1321
4	79	AMBURNEX-COMBE	46.535419	6.218099	1285

Downloading Weather Data

Both providers offer **hourly weather data** through public interfaces. MeteoSwiss publishes its measurements as open data; `download_data()` takes the station abbreviation and returns the data in UTC (or the time zone given with `tz`).

The examples use **fixed stations and a fixed past week**, so the results stay the same even if a new station opens closer to Horw. Here: MeteoSwiss station Luzern (`LUZ`) and Agrometeo station Meggen (ID `239`).

```
from pyedautils.weather import meteo_swiss

df_luz = meteo_swiss.download_data("LUZ", "2026-06-01", "2026-06-07", tz="Europe/Zu
df_luz.head()
```

datetime	temp	globrad	relhum	rain
2026-06-01 00:00:00+02:00	16.1	0.0	91.4	0.0
2026-06-01 01:00:00+02:00	15.6	0.0	94.6	0.0
2026-06-01 02:00:00+02:00	15.2	0.0	95.8	0.0
2026-06-01 03:00:00+02:00	15.2	0.0	95.8	0.0
2026-06-01 04:00:00+02:00	15.5	0.0	93.7	0.0

Agrometeo can be queried in two ways:

1. **By station ID** — if you already know the station
2. **By postal code (PLZ)** — automatically finds the nearest station(s)

Download by Station ID

Use `download_data()` when you know the station ID, e.g. from `find_nearest_station()` above or from `get_station_data()`.

```
from pyedautils.weather.agroweather import download_data

df_agro = download_data(
    station_id=239,  # Meggen, nearest Agrometeo station to Horw
    start_date="2026-06-01",
    end_date="2026-06-07",
    sensors=["temp", "globrad", "relhum", "rain"]
)

print(f"Shape: {df_agro.shape}")
print(f"Period: {df_agro.index.min()} to {df_agro.index.max()}")
df_agro.head()
```

```
Shape: (168, 4)
Period: 2026-06-01 00:00:00 to 2026-06-07 23:00:00
```

datetime	temp	globrad	relhum	rain
2026-06-01 00:00:00	17.1	0.1	84.0	0.1
2026-06-01 01:00:00	16.9	0.4	82.0	0.0
2026-06-01 02:00:00	16.9	NaN	83.0	0.0
2026-06-01 03:00:00	16.9	NaN	84.0	0.0
2026-06-01 04:00:00	17.2	NaN	83.0	0.0

Download by Postal Code

The convenience function `download_data_by_plz()` handles the full workflow: PLZ → coordinates → nearest station(s) → download. If different sensors are closest to different stations, it automatically merges the results. Because the nearest station can change over time, use it for quick looks and pin the station ID for reproducible analyses:

```
from pyedautils.weather.agroweather import download_data_by_plz

df_plz = download_data_by_plz(plz=6048, start_date="2026-06-01", end_date="2026-06-0
```

```
df_agro.describe()
```

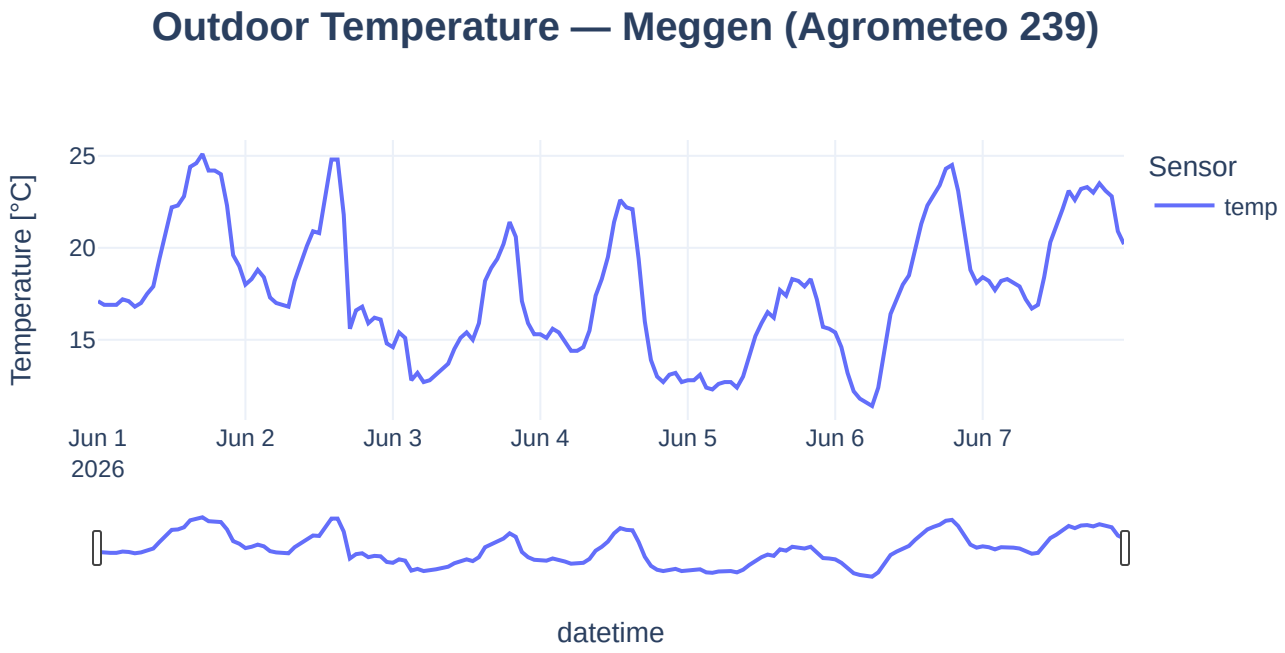
	temp	globrad	relhum	rain
count	168.000000	125.000000	168.000000	168.000000
mean	17.729167	273.253600	67.273810	0.244643
std	3.573344	272.144148	12.144688	0.779664
min	11.400000	0.000000	44.000000	0.000000
25%	15.100000	25.500000	56.000000	0.000000
50%	17.250000	189.600000	69.000000	0.000000
75%	20.375000	474.700000	77.250000	0.000000
max	25.100000	931.600000	85.000000	4.600000

Quick Visualization

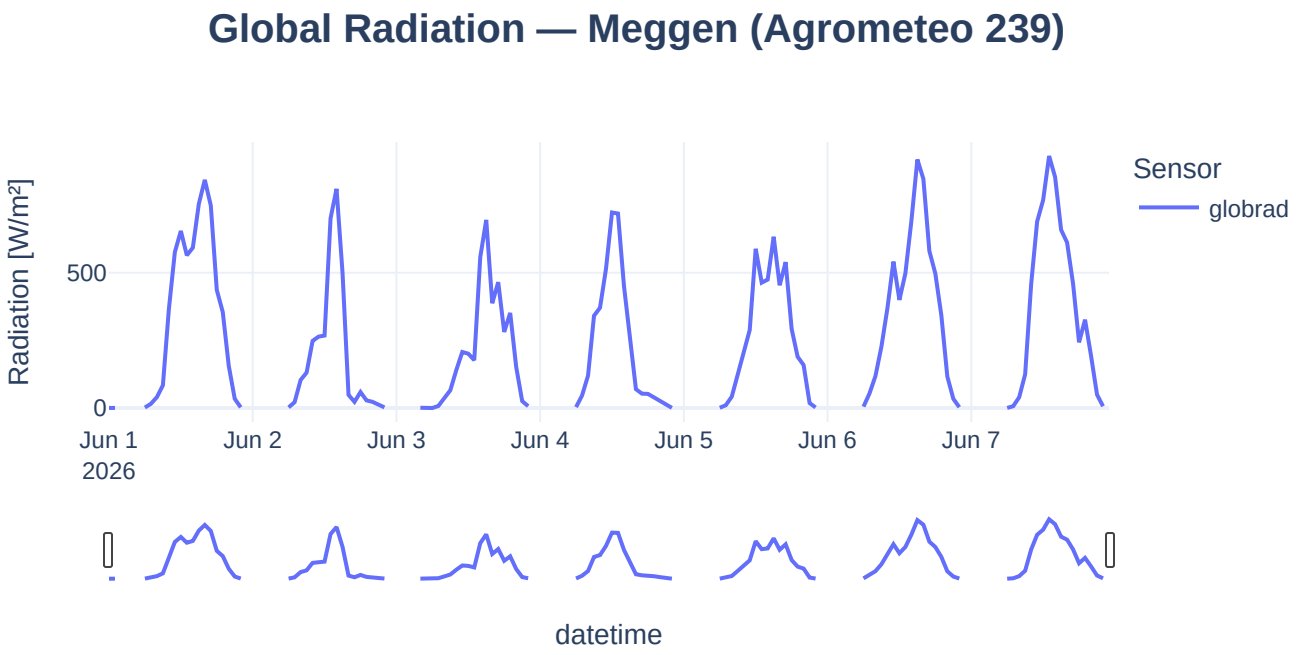
Let’s plot the downloaded data to verify it looks reasonable.

```
from pyedautils.plots import plot_timeseries

fig = plot_timeseries(
    df_agro,
    columns=["temp"],
    title="Outdoor Temperature — Meggen (Agrometeo 239)",
    ylab="Temperature [°C]",
)
fig.show()
```




```
fig = plot_timeseries(
    df_agro,
    columns=["globrad"],
    title="Global Radiation — Meggen (Agrometeo 239)",
    ylab="Radiation [W/m²]",
)
fig.show()
```



Available Sensors

Both providers support the same four sensor types:

Sensor key	Description	Unit
`"temp"`	Air temperature	°C
`"globrad"`	Global radiation	W/m²
`"relhum"`	Relative humidity	%
`"rain"`	Precipitation	mm

Which provider should I use?

Criteria	MeteoSwiss	Agrometeo
Data download	Yes, open data (10 min, hourly, daily, back to the start of the measurements)	Yes, via API (hourly)
Altitude matching	Yes (±150 m filter)	No
Best for	Official reference data, long time series	A station close to rural sites

Recommendation: Start with the nearest **MeteoSwiss** station at a similar altitude; use **Agrometeo** when it is much closer to your building.

Summary

Task	MeteoSwiss	Agrometeo
Find nearest station	`weather.meteo_swiss.find_nearest_station(lat, lon, alt, sensor)`	`weather.agroweather.fi lon, sensor)`
List all stations	`weather.meteo_swiss.get_current_station_data()`	`weather.agroweather.ge`
Download data by station	`weather.meteo_swiss.download_data(station, start, end)`	`weather.agroweather.do start, end)`
Download data by PLZ	`weather.meteo_swiss.download_data_by_plz(plz, start, end)`	`weather.agroweather.do start, end)`
PLZ → coordinates	`geopy.get_coordindates_ch_plz(plz)`	(built into `download_data`
Get altitude	`geopy.get_altitude_lat_long(lat, lon)`	—

Part III — Data Visualizations

In the following chapters, visualizations are presented that are frequently used in the field of building energy monitoring. Sometimes not the visualization itself is challenging, but the preceding data preparation. Therefore, all recipes show the complete workflow from raw data to final plot.

Plotly and pyedautils

All visualizations in this book use Plotly for interactive plots — every chart is zoomable, hoverable, and exportable. Many recipes use pyedautils, a Python package that provides

ready-to-use visualization functions for energy data analysis.

Recipe Pattern

Each visualization notebook follows a consistent structure:

> **Goal**
>
> What the visualization shows and when to use it.

> **Data Basis**
>
> Which data is needed — format, columns, and resolution.

Data Preparation — Load and transform the raw data.

Visualization — Create the plot using pyedautils or manual Plotly code.

> **Customization**

Categories

The visualizations are organized into the following categories:

- **Statistical** — Boxplots, histograms, density by season, correlation, outliers, autocorrelation, cross-correlation
- **Decomposition** — Long-term and short-term time series decomposition
- **Seasonal** — Overlapping years, miniplots, polar charts, before/after comparisons
- **Heat Maps** — Calendar heat maps and median week patterns
- **Daily Profiles** — Overview, overlaid, mean, and decomposed profiles
- **Sum & Frequency** — Daily and hourly duration curves
- **Comfort** — Temperature and humidity over time, temperature vs. humidity, Mollier h-x, comfort donuts, comfort compass, SIA 180
- **Miscellaneous** — Energy signatures, household electricity, multi-axis plots

Statistical Plots

Statistical visualizations help to understand the distribution of values, identify outliers, and explore relationships between variables.

Recipes

- **Boxplot** — Compare distributions across flats or months
- **Histogram** — Show how often each value occurs
- **Density Plot by Season** — Compare the distribution of a variable across the seasons
- **Correlation & Scatter** — Find variables that move together
- **Outlier Detection** — Flag implausible values with the IQR method
- **Autocorrelation** — Find repeating cycles (daily, weekly) in a time series
- **Cross-Correlation** — Find the time lag between two signals

Boxplot

Compare distributions across multiple groups using interactive boxplots.

> **Goal**
>
> Compare temperature distributions across four flats using interactive Plotly boxplots. Boxplots reveal the median, quartiles, and outliers of each distribution at a glance.

> **Data Basis**
>
> **Dataset:** `flatTempHum.csv` — Hourly indoor temperature and humidity measurements from four residential flats (A–D).
> **Columns used:** `FlatA_Temp`, `FlatB_Temp`, `FlatC_Temp`, `FlatD_Temp`
> **Time range:** October 2018 – September 2020

Data Preparation

```
import pandas as pd

url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTe
df = pd.read_csv(url, sep=";")
df["time"] = pd.to_datetime(df["time"])
df = df.set_index("time", drop=True)
df.head()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

```
# Select temperature columns and melt to long format
temp_cols = ["FlatA_Temp", "FlatB_Temp", "FlatC_Temp", "FlatD_Temp"]
df_long = df[temp_cols].melt(var_name="flat", value_name="value")

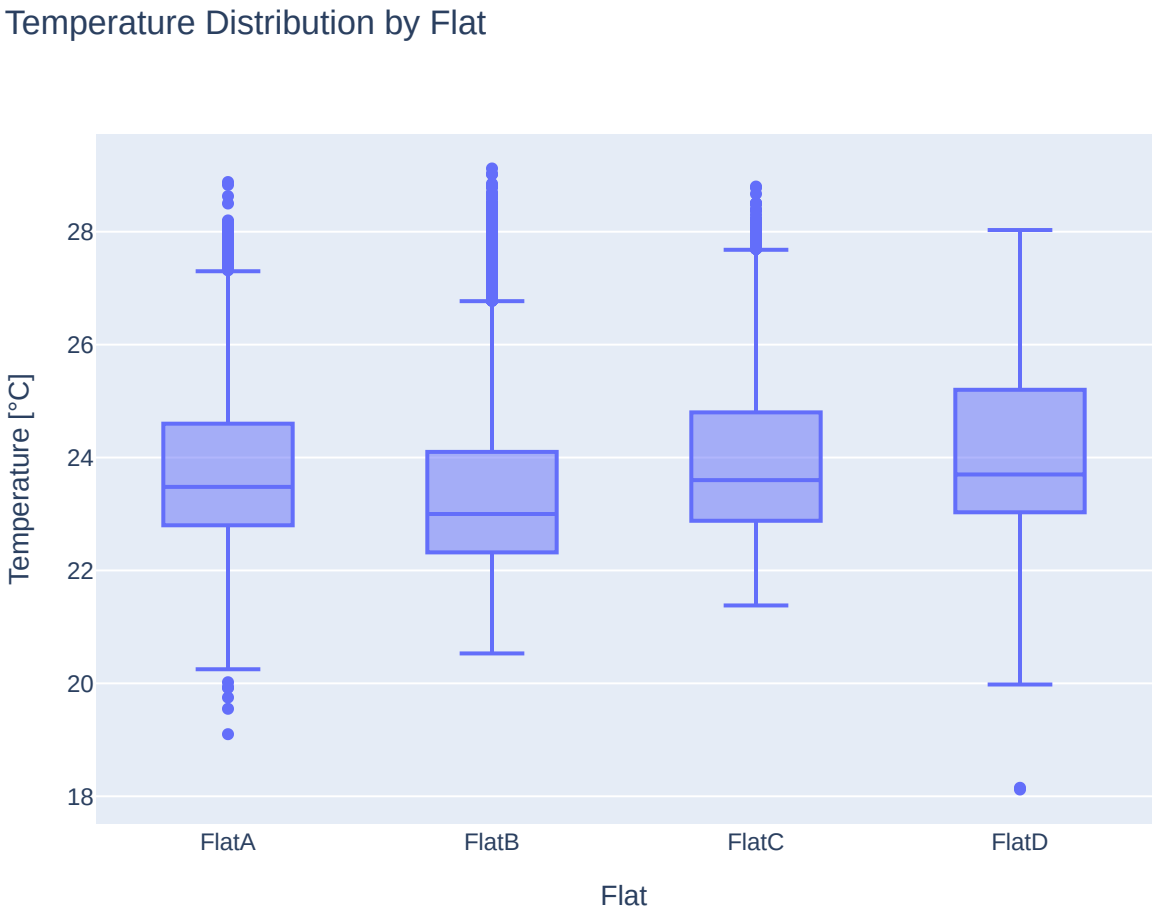
# Clean up flat names for display
df_long["flat"] = df_long["flat"].str.replace("_Temp", "")
df_long.head()
```

	flat	value
0	FlatA	24.43
1	FlatA	24.40
2	FlatA	24.40
3	FlatA	24.40
4	FlatA	24.40

Visualization

```
import plotly.express as px

fig = px.box(
    df_long,
    x="flat",
    y="value",
    title="Temperature Distribution by Flat",
    labels={"value": "Temperature [°C]", "flat": "Flat"},
)
fig.show()
```

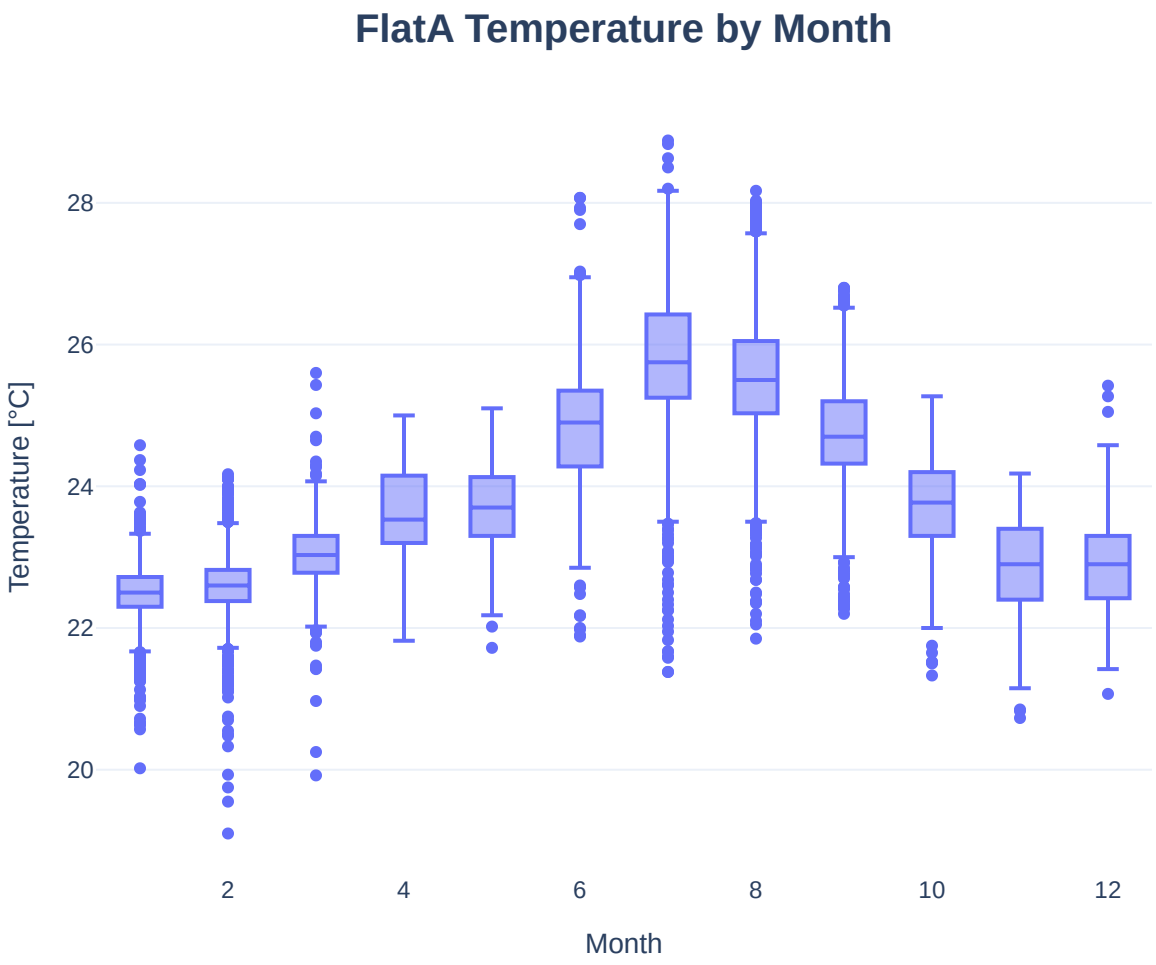


Boxplot by Time Group

Use `plot_boxplot()` from pyedautils to group values by a time unit (month, hour, weekday, quarter). This reveals seasonal or daily patterns in the data.

```
from pyedautils.plots import plot_boxplot

fig = plot_boxplot(
    df,
    column="FlatA_Temp",
    groupby="month",
    title="FlatA Temperature by Month",
    ylab="Temperature [°C]",
)
fig.show()
```



Customization

Histogram

Visualize the distribution of a single variable with a histogram.

Goal

Visualize the distribution of indoor temperature for Flat A. A histogram counts how many hours fall into each temperature bin and reveals whether temperatures are symmetric, skewed, or have several peaks (e.g. heating season and summer).

Data Basis

Dataset: `flatTempHum.csv` — Hourly indoor temperature and humidity measurements.
Column used: `FlatA_Temp`
Time range: October 2018 – September 2020

Data Preparation

```
import pandas as pd

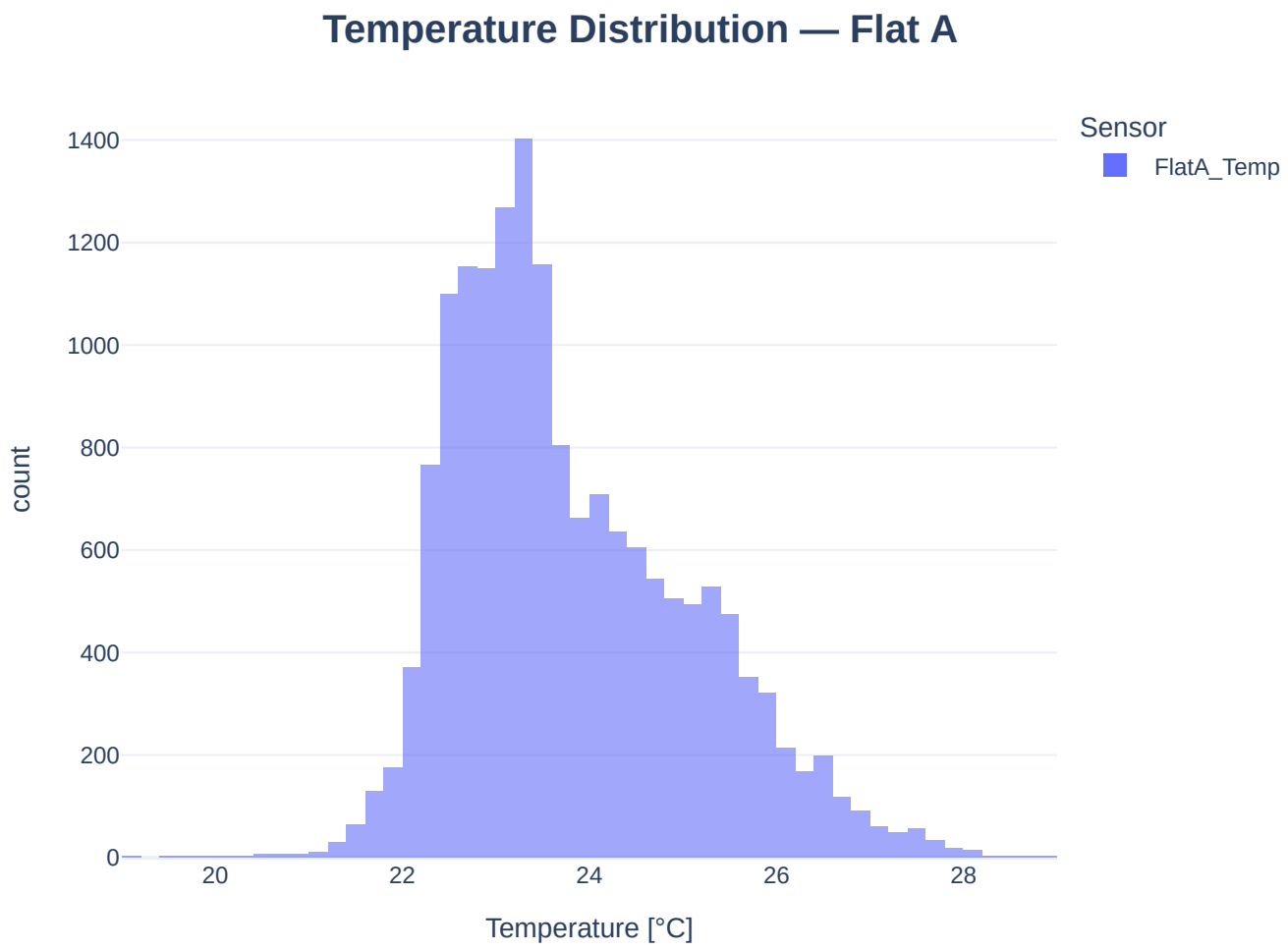
url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTe
df = pd.read_csv(url, sep=";")
df["time"] = pd.to_datetime(df["time"])
df = df.set_index("time", drop=True)
df.head()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

Visualization

```
from pyedautils.plots import plot_distribution

fig = plot_distribution(
    df,
    columns=["FlatA_Temp"],
    nbins=50,
    title="Temperature Distribution — Flat A",
    xlab="Temperature [°C]",
)
fig.show()
```



Customization

Density Plot by Season

Compare how a variable’s distribution shifts across seasons using overlaid density plots.

Goal

Compare the temperature distribution of Flat A across the four seasons. This reveals how indoor temperatures shift between winter and summer and whether the building maintains consistent conditions year-round.

Data Basis

Dataset: `flatTempHum.csv` — Hourly indoor temperature and humidity measurements.
Columns used: `time`, `FlatA_Temp`
Time range: October 2018 – September 2020
Library: `pyedautils.plots.plot_density_seasons` — expects a 2-column DataFrame (timestamp column + value column).

Data Preparation

```
import pandas as pd

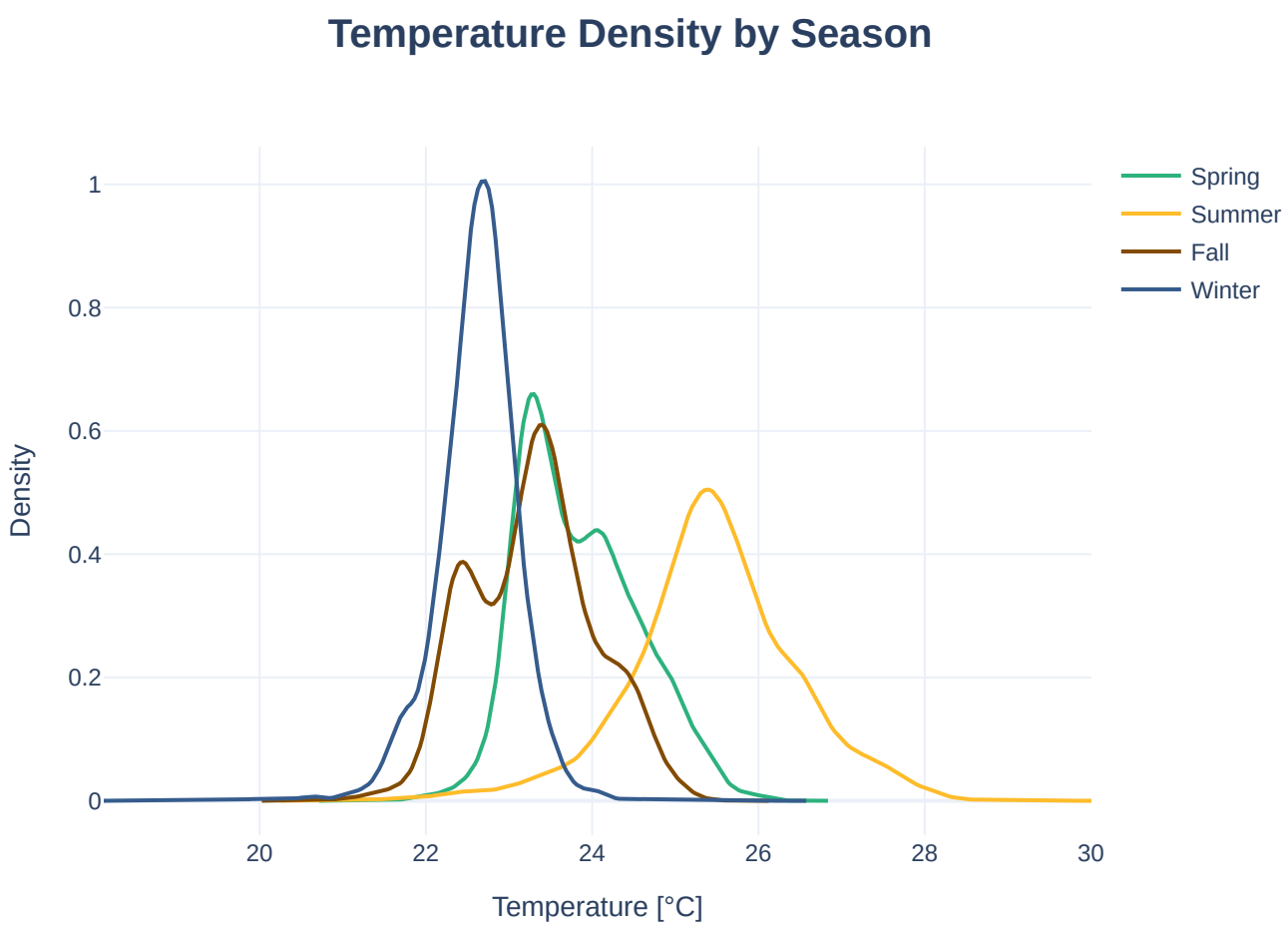
url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTe
df = pd.read_csv(url, sep=";", usecols=["time", "FlatA_Temp"])
df["time"] = pd.to_datetime(df["time"])
df.head()
```

	time	FlatA_Temp
0	2018-10-03 00:00:00	24.43
1	2018-10-03 01:00:00	24.40
2	2018-10-03 02:00:00	24.40
3	2018-10-03 03:00:00	24.40
4	2018-10-03 04:00:00	24.40

Visualization

```
from pyedautils.plots import plot_density_seasons

fig = plot_density_seasons(
    df,
    title="Temperature Density by Season",
    xlab="Temperature [°C]",
)
fig.show()
```



Customization

Correlation & Scatter

Goal

Explore linear relationships between variables using a correlation matrix heatmap and scatter plots. Correlation analysis reveals which sensors move together, which are independent, and whether relationships are positive or negative.

Data Basis

Dataset: `flatTempHum.csv` — Hourly indoor temperature and humidity measurements from four residential flats (A–D).
Columns used: All temperature and humidity columns
Time range: October 2018 – September 2020

Data Preparation

```
import pandas as pd

url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTe
df = pd.read_csv(url, sep=";")
df["time"] = pd.to_datetime(df["time"])
df = df.set_index("time", drop=True)
df.head()
```

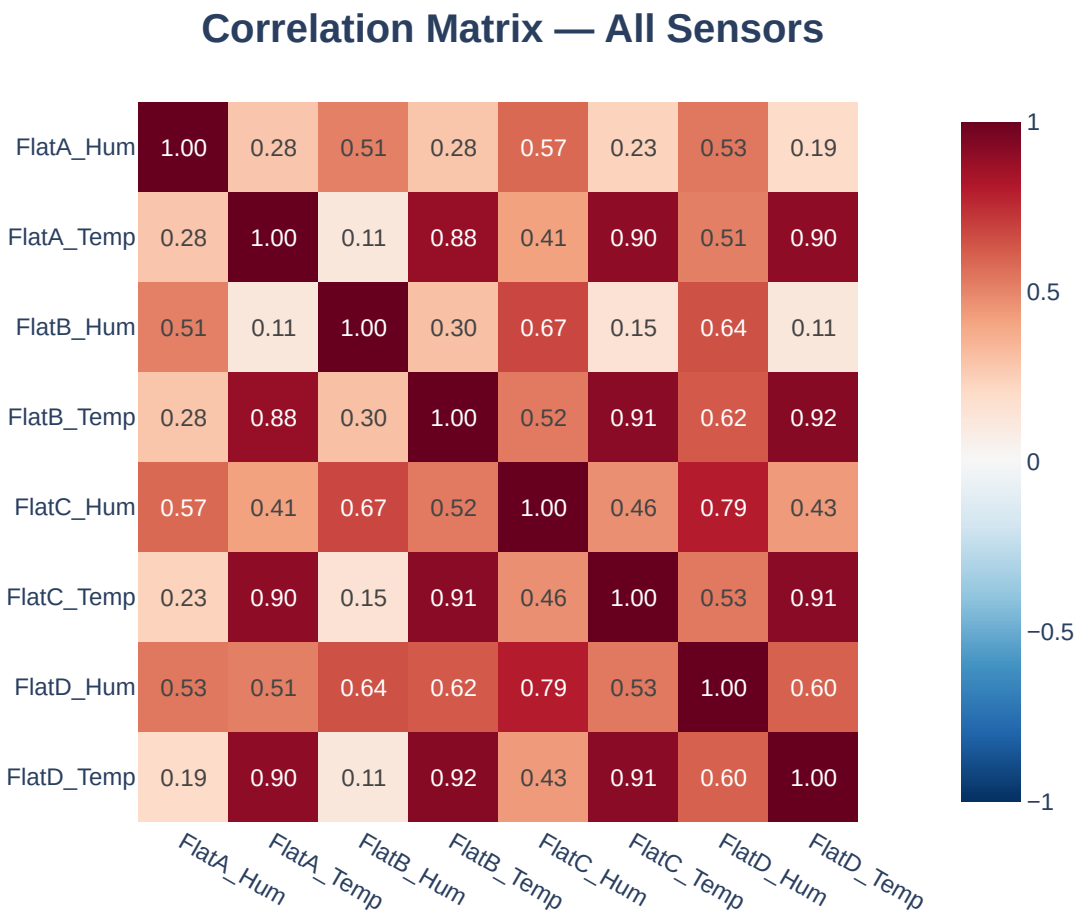
time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

Correlation Matrix

The correlation matrix shows the Pearson correlation coefficient between every pair of variables. Values range from −1 (perfect negative correlation) to +1 (perfect positive correlation).

```
from pyedautils.plots import plot_correlation

fig = plot_correlation(df, title="Correlation Matrix — All Sensors")
fig.show()
```



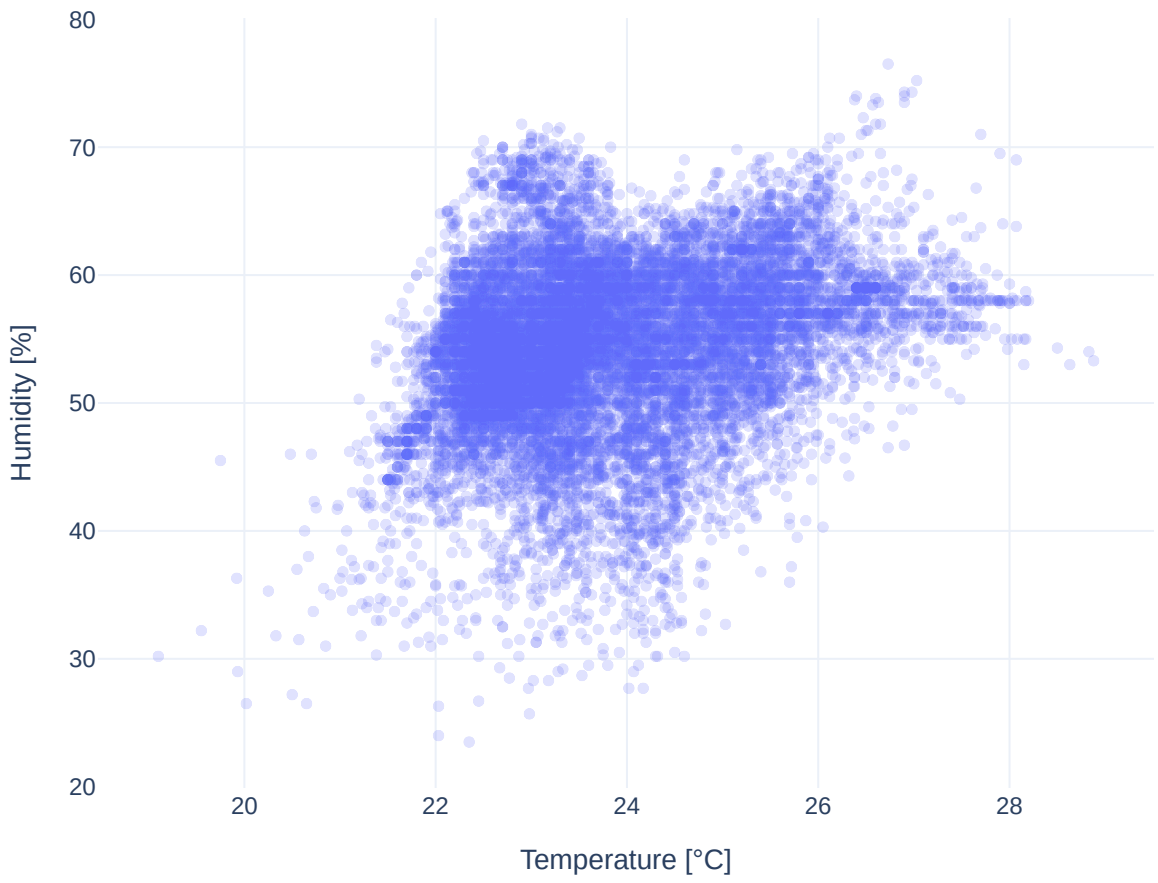
Interpreting the correlation matrix

Scatter Plot

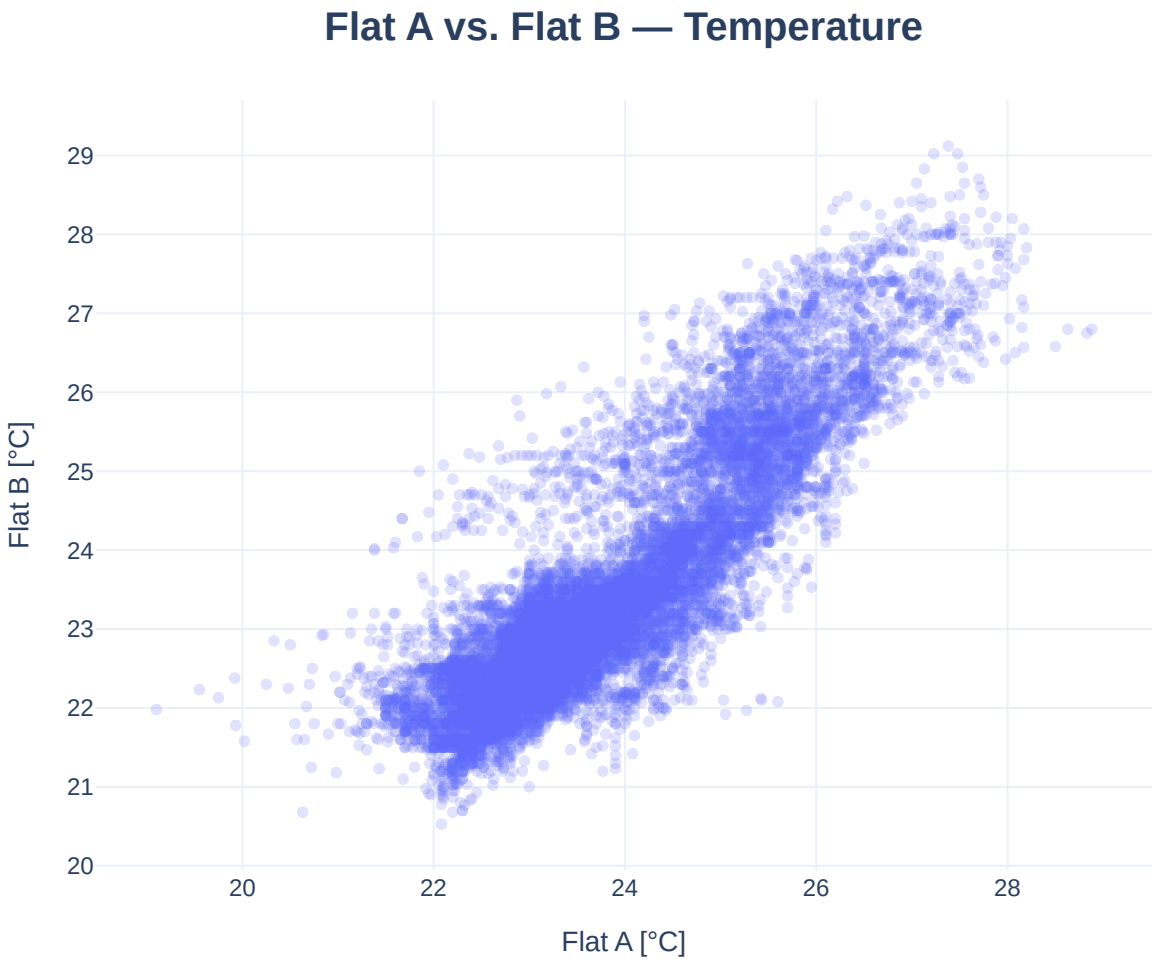
A scatter plot reveals the shape and density of the relationship between two variables. Low opacity makes dense regions visible.

```
from pyedautils.plots import plot_scatter

fig = plot_scatter(
    df,
    x_column="FlatA_Temp",
    y_column="FlatA_Hum",
    title="Flat A: Temperature vs. Humidity",
    xlab="Temperature [°C]",
    ylab="Humidity [%]",
)
fig.show()
```




```
fig = plot_scatter(
    df,
    x_column="FlatA_Temp",
    y_column="FlatB_Temp",
    title="Flat A vs. Flat B – Temperature",
    xlab="Flat A [°C]",
    ylab="Flat B [°C]",
)
fig.show()
```



Customization

Outlier Detection

Goal

Detect and visualize outliers in sensor data using the Interquartile Range (IQR) method. Outliers can indicate sensor errors, unusual events (e.g. window openings), or data quality issues.

Data Basis

Dataset: `flatTempHum.csv` — Hourly indoor temperature and humidity measurements from four residential flats (A–D).
Column used: `FlatA_Temp`
Time range: October 2018 – September 2020

Data Preparation

```
import pandas as pd

url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTe
df = pd.read_csv(url, sep=";")
df["time"] = pd.to_datetime(df["time"])
df = df.set_index("time", drop=True)
df.head()
```

time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

Outlier Calculation

The IQR method flags values outside [Q1 − k×IQR, Q3 + k×IQR] as outliers, where k is typically 1.5.

```
from pyedautils.data_quality import calc_outliers

result = calc_outliers(df, column="FlatA_Temp", multiplier=1.5)

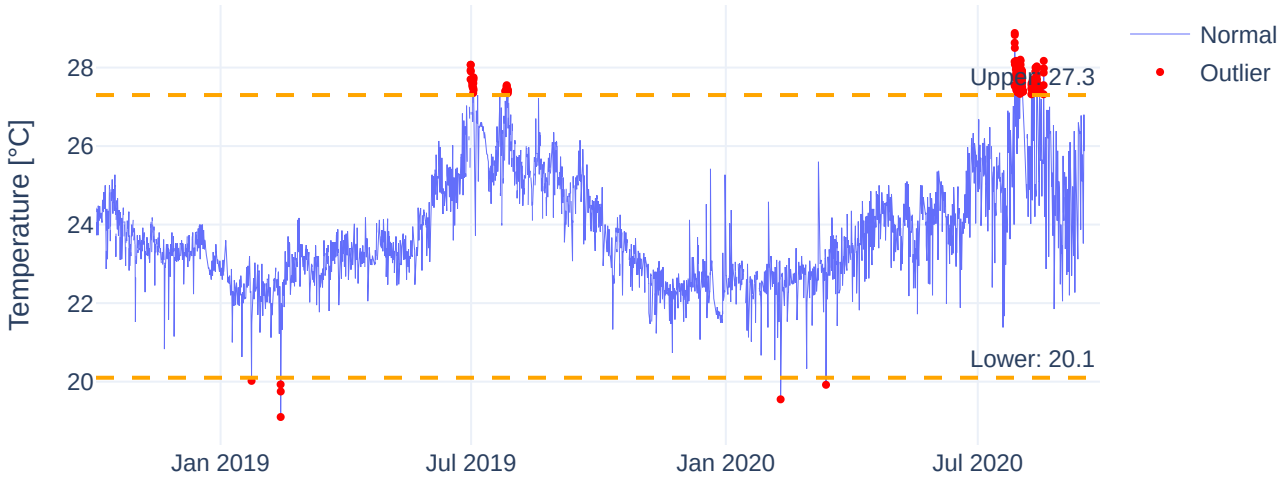
print(f"IQR fences: [{result['lower']:.1f}, {result['upper']:.1f}] °C")
print(f"Outliers:   {result['count']} of {len(df)} ({result['percentage']}%)")
```

```
IQR fences: [20.1, 27.3] °C
Outliers:   148 of 17181 (0.86%)
```

Visualization

```
from pyedautils.plots import plot_outliers

fig = plot_outliers(
    df,
    column="FlatA_Temp",
    title="FlatA Temperature — Outlier Detection (IQR)",
    ylab="Temperature [°C]",
)
fig.show()
```



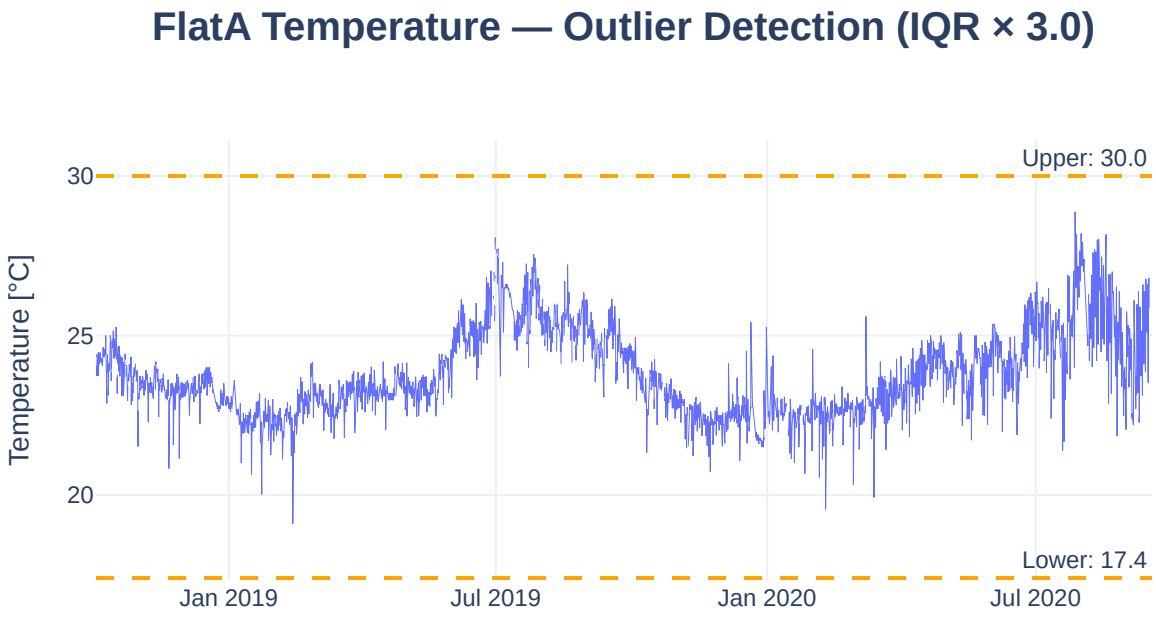
Adjusting Sensitivity

The `multiplier` parameter controls how aggressive the detection is. A lower value flags more outliers.

```
for k in [1.0, 1.5, 2.0, 3.0]:
    r = calc_outliers(df, column="FlatA_Temp", multiplier=k)
    print(f"k={k:.1f}  →  fences [{r['lower']:.1f}, {r['upper']:.1f}]  →  {r['count
```

```
k=1.0  →  fences [21.0, 26.4]  →   639 outliers (3.72%)
k=1.5  →  fences [20.1, 27.3]  →   148 outliers (0.86%)
k=2.0  →  fences [19.2, 28.2]  →     5 outliers (0.03%)
k=3.0  →  fences [17.4, 30.0]  →     0 outliers (0.0%)
```

```
fig = plot_outliers(
    df,
    column="FlatA_Temp",
    multiplier=3.0,
    title="FlatA Temperature — Outlier Detection (IQR × 3.0)",
    ylab="Temperature [°C]",
)
fig.show()
```



Customization

Autocorrelation

Goal

Identify dominant periodicities in a time series using autocorrelation. Peaks in the autocorrelation function (ACF) reveal repeating cycles — daily, weekly, or seasonal — which are fundamental to understanding building energy and climate data.

Data Basis

Dataset: `flatTempHum.csv` — Hourly indoor temperature and humidity measurements from four residential flats (A–D).
Column used: `FlatA_Temp`
Time range: October 2018 – September 2020

Data Preparation

```
import pandas as pd

url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTe
df = pd.read_csv(url, sep=";")
df["time"] = pd.to_datetime(df["time"])
df = df.set_index("time", drop=True)
df.head()
```

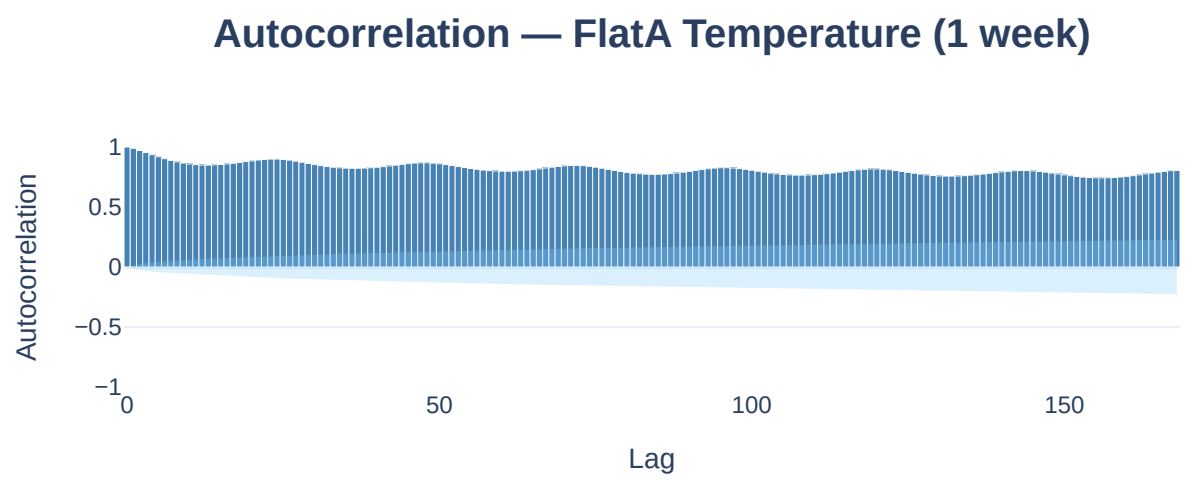
time	FlatA_Hum	FlatA_Temp	FlatB_Hum	FlatB_Temp	FlatC_Hum	FlatC_Temp
2018-10-03 00:00:00	53.0	24.43	38.8	22.40	44.0	24.5
2018-10-03 01:00:00	53.0	24.40	38.8	22.40	44.0	24.5
2018-10-03 02:00:00	53.0	24.40	39.3	22.40	44.7	24.5
2018-10-03 03:00:00	53.0	24.40	40.3	22.40	45.0	24.5
2018-10-03 04:00:00	53.3	24.40	41.0	22.37	45.2	24.5

Visualization

The autocorrelation plot shows how a signal correlates with itself at increasing time lags. The blue shaded area marks the 95% confidence interval — values inside the band are not statistically significant.

```
from pyedautils.plots import plot_autocorrelation

fig = plot_autocorrelation(
    df,
    column="FlatA_Temp",
    lags=168,
    title="Autocorrelation — FlatA Temperature (1 week)",
)
fig.show()
```

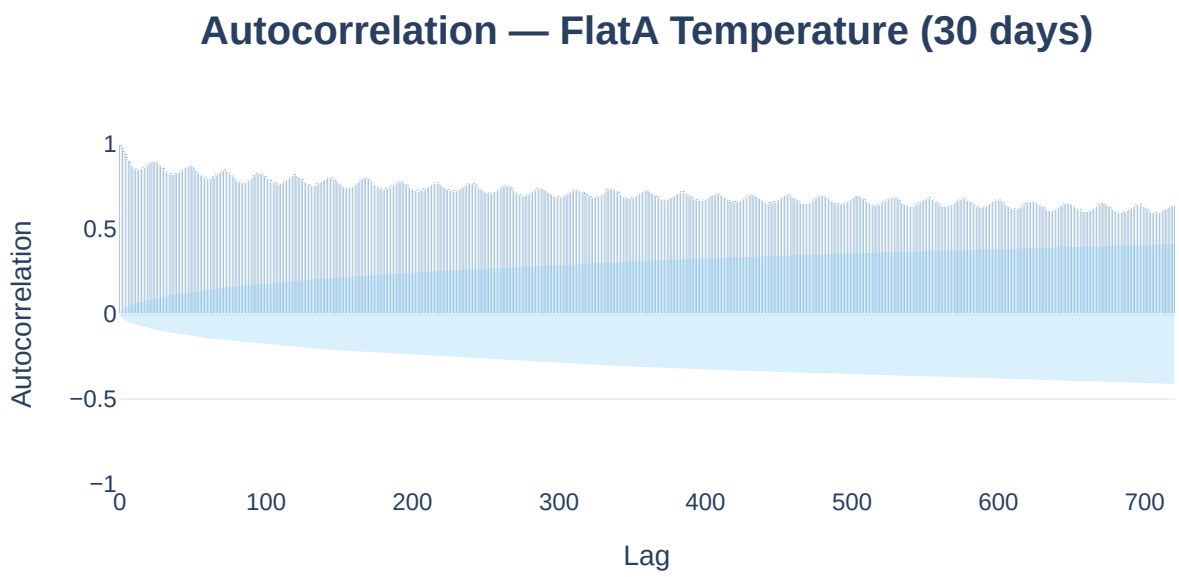


Reading the autocorrelation plot

Longer Lag Window

Increase the number of lags to look for longer-period cycles.

```
fig = plot_autocorrelation(
    df,
    column="FlatA_Temp",
    lags=720,
    title="Autocorrelation — FlatA Temperature (30 days)",
    height=350,
)
fig.show()
```



Customization

Cross-Correlation

Analyze the time-lagged relationship between two signals using cross-correlation.

Goal

Analyze the time-lagged correlation between outdoor temperature and indoor temperature (Flat A). Cross-correlation reveals how strongly and with what delay the outdoor climate influences the indoor environment.

Data Basis

Datasets:

- `flatTempHum.csv` — Hourly indoor temperature (`FlatA_Temp`)
- `centralOutsideTemp.csv` — Hourly outdoor temperature (`centralOutsideTemp`)

Time range: Overlapping period of both datasets
Lag range: -48 to +48 hours

Data Preparation

```
import pandas as pd

base_url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/"

# Load indoor temperature
df_indoor = pd.read_csv(base_url + "flatTempHum.csv", sep=";", usecols=["time", "Fl
df_indoor["time"] = pd.to_datetime(df_indoor["time"])
df_indoor = df_indoor.set_index("time", drop=True)

# Load outdoor temperature
df_outdoor = pd.read_csv(base_url + "centralOutsideTemp.csv", sep=";")
df_outdoor["time"] = pd.to_datetime(df_outdoor["time"])
df_outdoor = df_outdoor.set_index("time", drop=True)

df_indoor.head()
```

time	FlatA_Temp
2018-10-03 00:00:00	24.43
2018-10-03 01:00:00	24.40
2018-10-03 02:00:00	24.40
2018-10-03 03:00:00	24.40
2018-10-03 04:00:00	24.40

```
# Join on the time index and keep a regular hourly grid: missing hours stay NaN,
# so that .shift(1) below really moves the data by one hour
df = df_indoor.join(df_outdoor, how="inner").asfreq("h")
df = df.rename(columns={"centralOutsideTemp": "outdoor", "FlatA_Temp": "indoor"})
print(f"Combined dataset: {len(df)} hours, {df['indoor'].isna().sum()} without indo
df.head()
```

```
Combined dataset: 17172 hours, 688 without indoor value
```

time	indoor	outdoor
2018-10-03 00:00:00	24.43	11.80
2018-10-03 01:00:00	24.40	11.25
2018-10-03 02:00:00	24.40	11.45
2018-10-03 03:00:00	24.40	11.40
2018-10-03 04:00:00	24.40	11.10

Visualization

Both temperatures follow the annual cycle (warm in summer, cold in winter), so their raw values correlate strongly at every lag. To see the delay of the building, we correlate the **hour-to-hour changes** (`.diff()`) instead. `indoor.corr(outdoor.shift(lag))` compares the indoor change with the outdoor change `lag` hours earlier: a peak at a **positive lag** means the indoor temperature follows the outdoor temperature with that delay.

```
import plotly.express as px

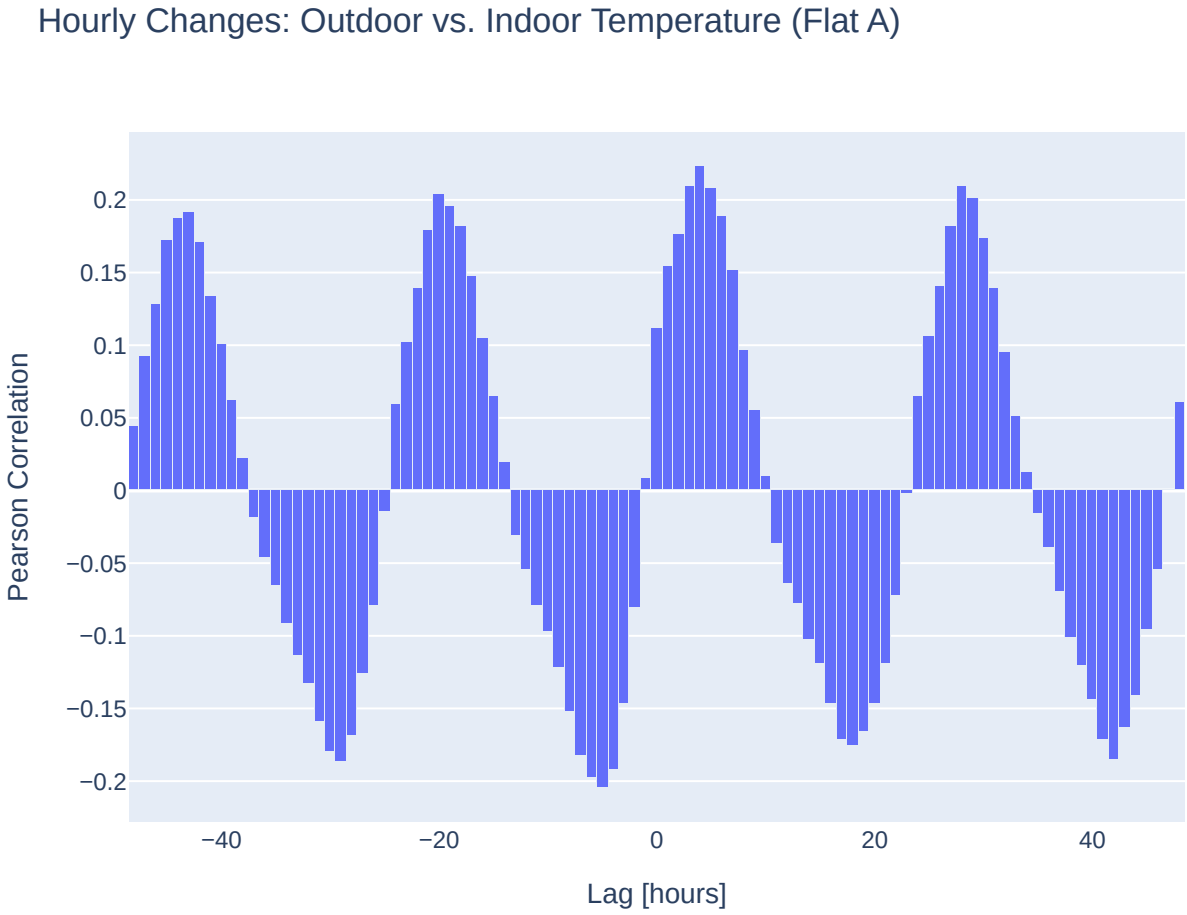
changes = df.diff()   # hour-to-hour changes, NaN next to missing hours

max_lag = 48
lags = range(-max_lag, max_lag + 1)
correlations = [changes["indoor"].corr(changes["outdoor"].shift(lag)) for lag in la
df_corr = pd.DataFrame({"lag_hours": list(lags), "correlation": correlations})

best = df_corr.loc[df_corr["correlation"].idxmax()]
print(f"Maximum correlation: {best['correlation']:.2f} at lag = {int(best['lag_hour
```

```
Maximum correlation: 0.22 at lag = 4 hours
```

```
fig = px.bar(
    df_corr,
    x="lag_hours",
    y="correlation",
    title="Hourly Changes: Outdoor vs. Indoor Temperature (Flat A)",
    labels={"lag_hours": "Lag [hours]", "correlation": "Pearson Correlation"},
)
fig.update_layout(bargap=0)
fig.show()
```



The indoor temperature follows the outdoor temperature with a delay of a few hours (thermal inertia of the building). The correlation is weak because heating and occupants change the indoor temperature much more than the weather does from one hour to the next.

Customization

Time Series Decomposition

Time series decomposition separates a signal into its components: trend, seasonal pattern, and residuals. This helps to understand the underlying structure of the data.

Recipes

- **Long-Term Decomposition** — Decompose multi-year heating or energy data into yearly seasonality and trend
- **Short-Term Decomposition** — Decompose shorter time periods to reveal daily or weekly patterns

Long-Term Decomposition

> **Goal**
>
> Decompose multi-year energy consumption data into trend, seasonal, and residual components to understand underlying patterns over a 10-year period.

> **Data Basis**
>
> Monthly energy consumption data (`decomposition_long.csv`) bundled with the `pyedautils` package. 119 monthly values over ~10 years.

Data Preparation

```
import pandas as pd
from importlib import resources

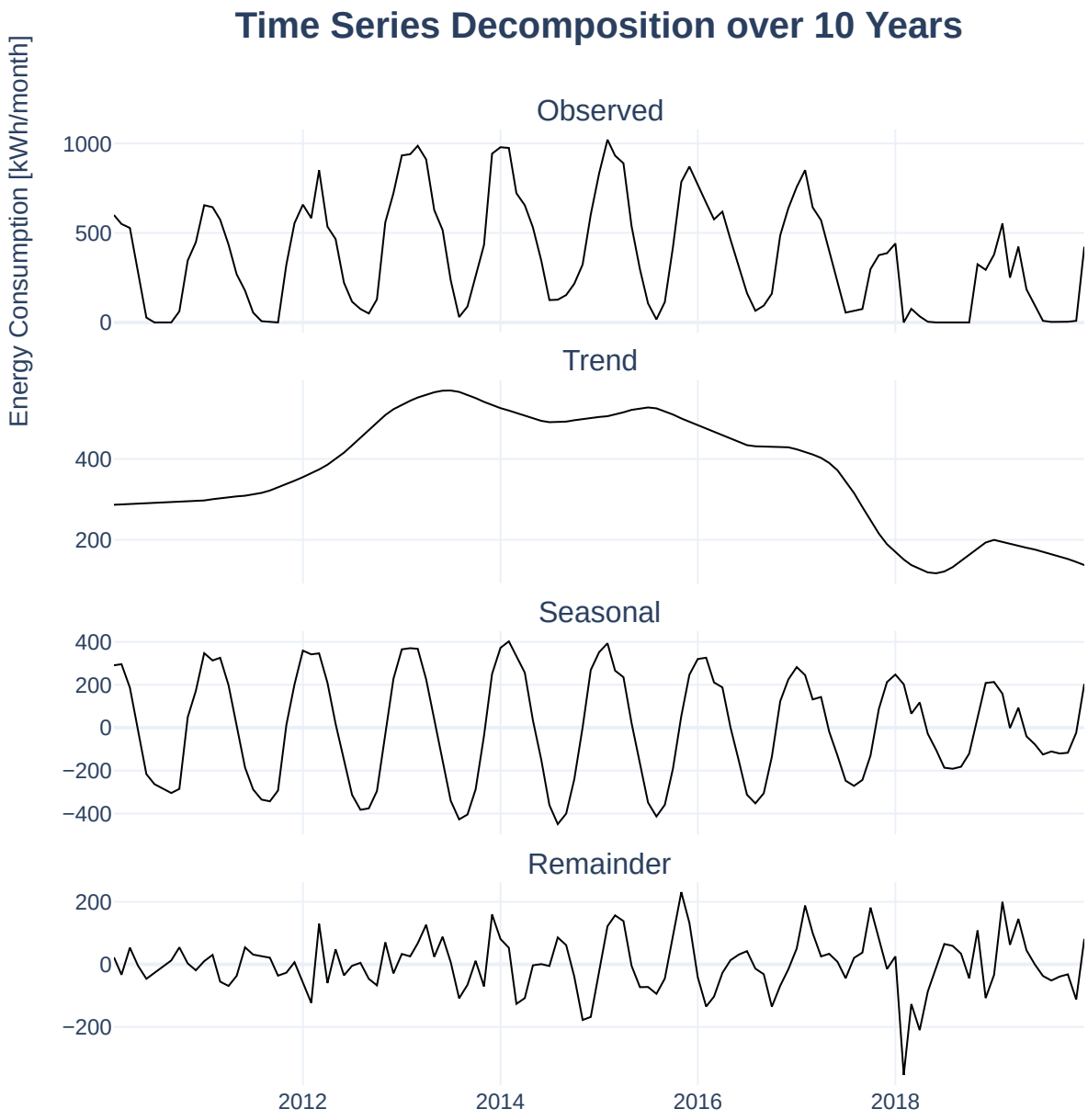
data_path = resources.files("pyedautils") / "data" / "decomposition_long.csv"
df = pd.read_csv(data_path, sep=";")
df.head()
```

	timestamp	value
0	2010-02-01	600.0
1	2010-03-01	550.0
2	2010-04-01	528.0
3	2010-05-01	259.0
4	2010-06-01	27.0

Visualization

```
from pyedautils.plots import plot_decomposition

fig = plot_decomposition(
    data=df,
    period=12,
    s_window=7,
    title="Time Series Decomposition over 10 Years",
    ylab="Energy Consumption [kWh/month]",
)
fig.show()
```



Short-Term Decomposition

Goal

Decompose high-resolution power data over a short period to reveal daily consumption patterns and short-term trends.

Data Basis

Power measurements at 15-minute intervals (`decomposition_short.csv`) bundled with the `pyedautils` package. 480 data points covering 5 days.

Data Preparation

```
import pandas as pd
from importlib import resources

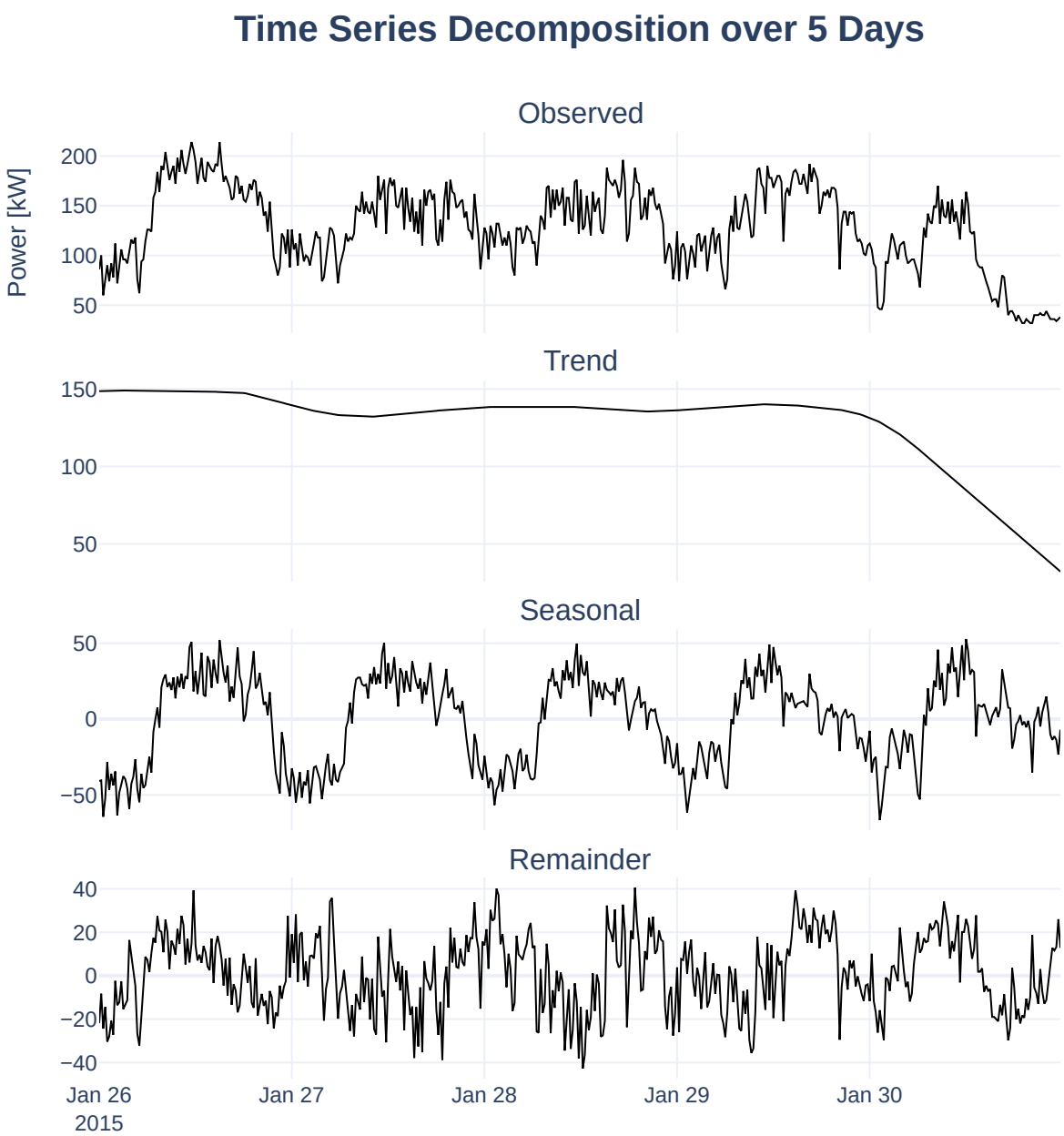
data_path = resources.files("pyedautils") / "data" / "decomposition_short.csv"
df = pd.read_csv(data_path, sep=";")
df.head()
```

	timestamp	value
0	2015-01-26 00:00:00	86.0
1	2015-01-26 00:15:00	100.0
2	2015-01-26 00:30:00	60.0
3	2015-01-26 00:45:00	82.0
4	2015-01-26 01:00:00	90.0

Visualization

```
from pyedautils.plots import plot_decomposition

fig = plot_decomposition(
    data=df,
    period=96,
    s_window=193,
    title="Time Series Decomposition over 5 Days",
    ylab="Power [kW]",
)
fig.show()
```



Seasonal Visualizations

Seasonal visualizations overlay or compare data across different time periods (e.g., years, months) to reveal recurring patterns and changes over time.

Recipes

- **Overlapping Years** — Overlay multiple years on a single axis to compare seasonal patterns
- **Seasonal Miniplots** — Small multiples showing each year or month side by side
- **Polar Chart** — Circular visualization of seasonal data (e.g., heating demand by month)
- **Before/After Comparison** — Compare a variable before and after an intervention or event

Overlapping Years

> **Goal**
>
> Overlay multiple years of energy data on the same axis to compare seasonal patterns across years.

> **Data Basis**
>
> Monthly energy consumption data (`seasonal_monthly.csv`) bundled with the `pyedautils` package. 10 years of monthly values.

Data Preparation

```
import pandas as pd
from importlib import resources

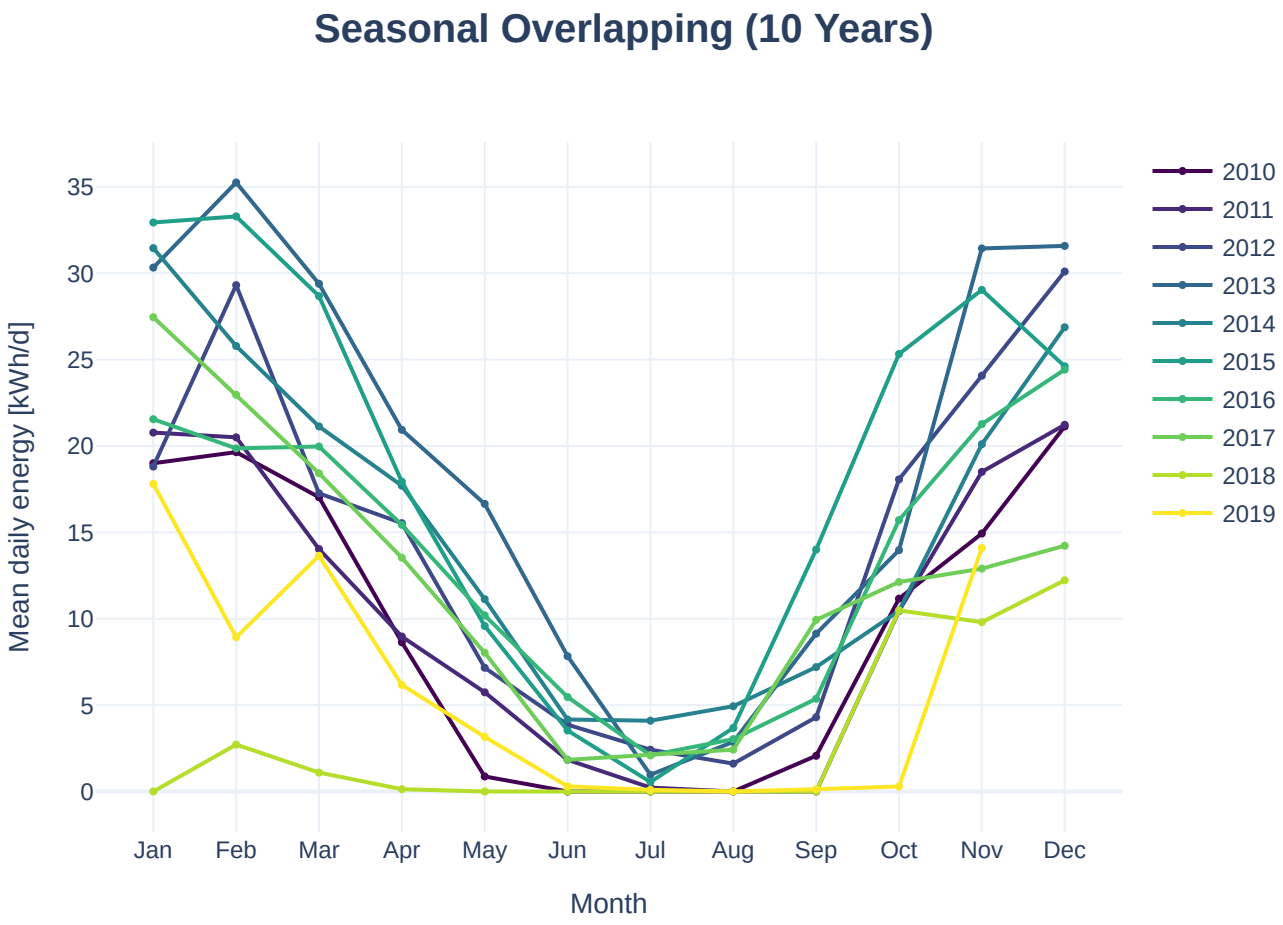
data_path = resources.files("pyedautils") / "data" / "seasonal_monthly.csv"
df = pd.read_csv(data_path, sep=";")
df.head()
```

	timestamp	value
0	2010-01-01	19.000000
1	2010-02-01	19.642857
2	2010-03-01	17.032258
3	2010-04-01	8.633333
4	2010-05-01	0.870968

Visualization

```
from pyedautils.plots import plot_seasonal_overlapping

fig = plot_seasonal_overlapping(
    data=df,
    title="Seasonal Overlapping (10 Years)",
    ylab="Mean daily energy [kWh/d]",
)
fig.show()
```



Miniplots

Goal

Show each month as a mini panel to compare energy values across years, making it easy to spot month-by-month trends.

Data Basis

Monthly energy consumption data (`seasonal_monthly.csv`) bundled with the `pyedautils` package. 10 years of monthly values.

Data Preparation

```
import pandas as pd
from importlib import resources

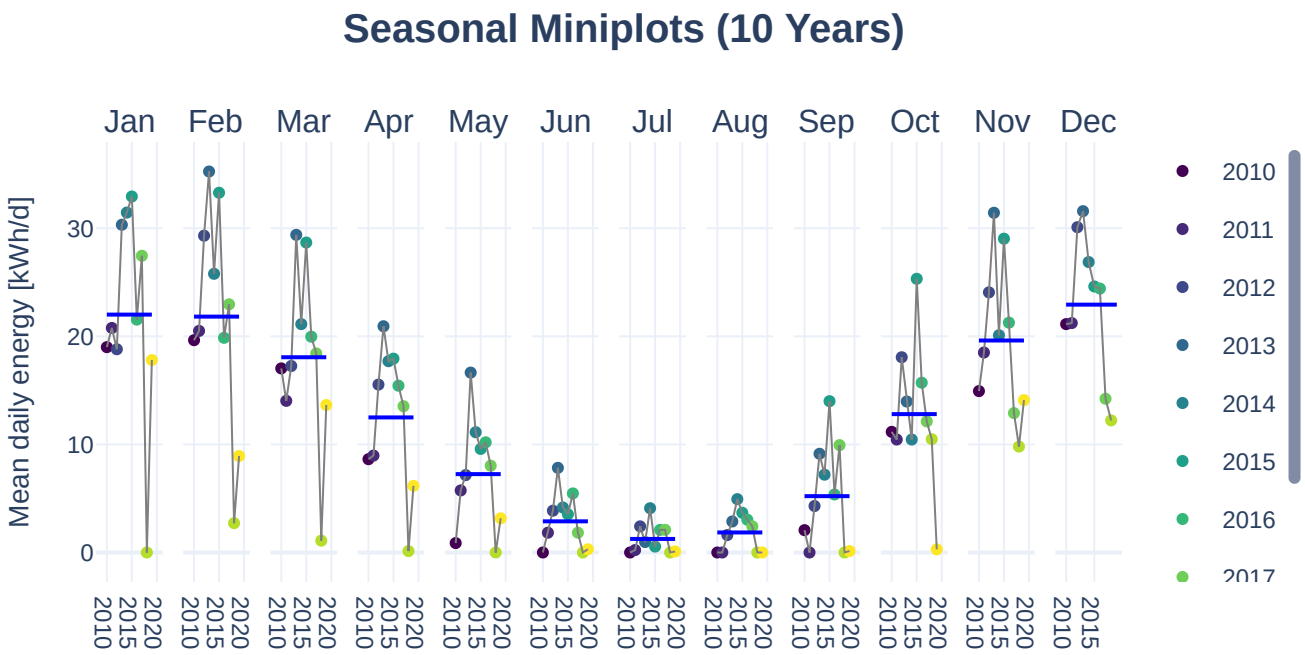
data_path = resources.files("pyedautils") / "data" / "seasonal_monthly.csv"
df = pd.read_csv(data_path, sep=";")
df.head()
```

	timestamp	value
0	2010-01-01	19.000000
1	2010-02-01	19.642857
2	2010-03-01	17.032258
3	2010-04-01	8.633333
4	2010-05-01	0.870968

Visualization

```
from pyedautils.plots import plot_seasonal_miniplots

fig = plot_seasonal_miniplots(
    data=df,
    title="Seasonal Miniplots (10 Years)",
    ylab="Mean daily energy [kWh/d]",
)
fig.show()
```



Polar Chart

> **Goal**
>
> Circular visualization of monthly energy demand, providing an intuitive view of the annual cycle.

> **Data Basis**
>
> Monthly energy consumption data (`seasonal_monthly.csv`) bundled with the `pyedautils` package. 10 years of monthly values.

Data Preparation

```
import pandas as pd
from importlib import resources

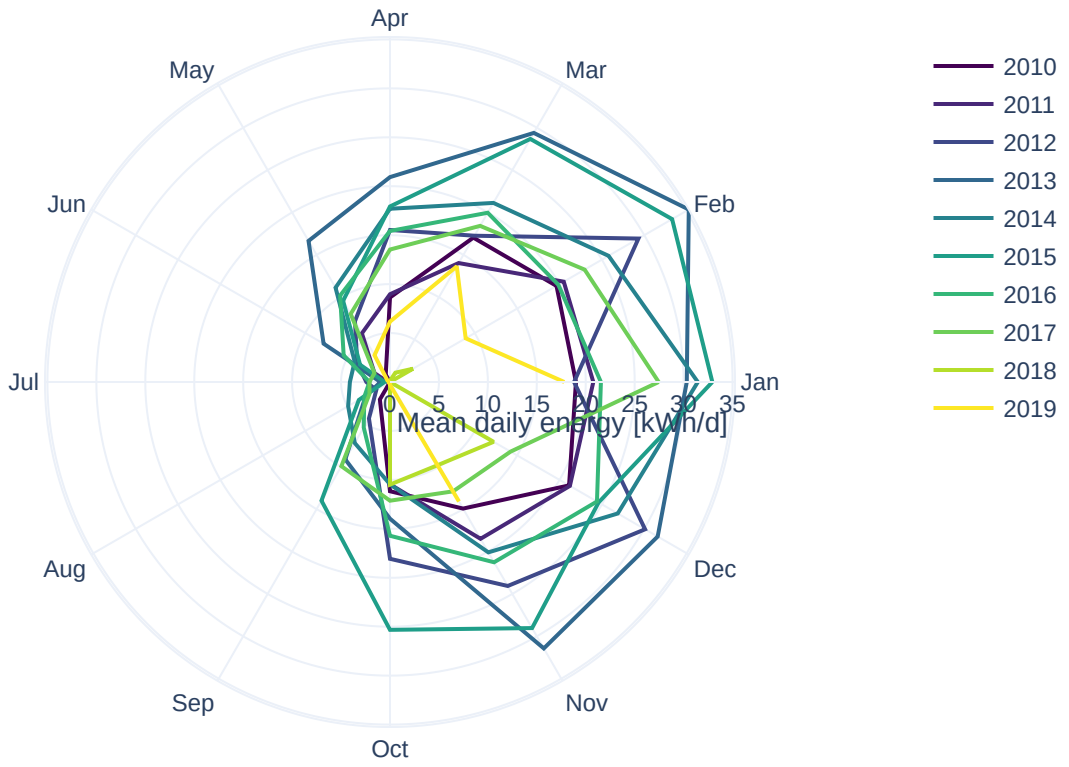
data_path = resources.files("pyedautils") / "data" / "seasonal_monthly.csv"
df = pd.read_csv(data_path, sep=";")
df.head()
```

	timestamp	value
0	2010-01-01	19.000000
1	2010-02-01	19.642857
2	2010-03-01	17.032258
3	2010-04-01	8.633333
4	2010-05-01	0.870968

Visualization

```
from pyedautils.plots import plot_seasonal_polar

fig = plot_seasonal_polar(
    data=df,
    title="Seasonal Polar (10 Years)",
    ylab="Mean daily energy [kWh/d]",
)
fig.show()
```



Before / After Comparison

Goal

Compare energy consumption before and after an optimization event to evaluate its impact.

Data Basis

Monthly energy consumption data (`seasonal_monthly.csv`) bundled with the `pyedautils` package. 10 years of monthly values.

Data Preparation

```
import pandas as pd
from importlib import resources

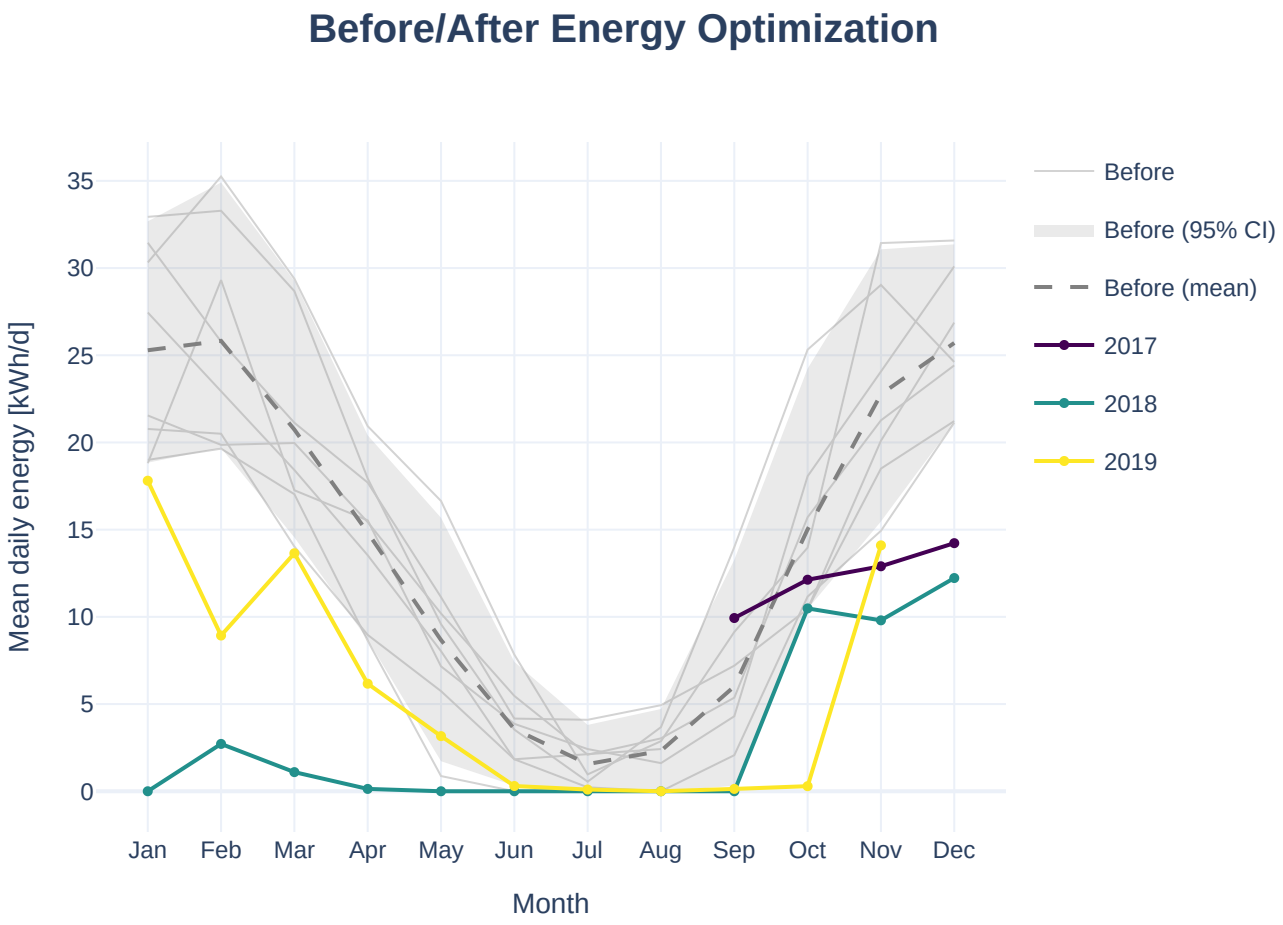
data_path = resources.files("pyedautils") / "data" / "seasonal_monthly.csv"
df = pd.read_csv(data_path, sep=";")
df.head()
```

	timestamp	value
0	2010-01-01	19.000000
1	2010-02-01	19.642857
2	2010-03-01	17.032258
3	2010-04-01	8.633333
4	2010-05-01	0.870968

Visualization

```
from pyedautils.plots import plot_seasonal_before_after

fig = plot_seasonal_before_after(
    data=df,
    date_optimization="2017-09-01",
    title="Before/After Energy Optimization",
    ylab="Mean daily energy [kWh/d]",
)
fig.show()
```



Customization

Heat Maps

Heat maps use color to represent values in a matrix, making it easy to spot patterns across two dimensions (e.g., time of day vs. day of year).

Recipes

- **Calendar Heat Map** — Show daily values over a full year in a calendar layout
- **Median Weeks** — Display the median weekly pattern as a heat map (hour of day vs. day of week)

Calendar Heat Map

> **Goal**
>
> Visualize daily electricity consumption as a calendar heatmap to identify patterns and anomalies over time.

> **Data Basis**
>
> The dataset `flatElectricity.csv` contains cumulative electricity meter readings at 15-minute intervals for multiple flats. We use the column `FlatA_Ele`, convert cumulative readings to consumption by computing differences and sum them per day. Days with missing readings stay empty in the calendar.

Data Preparation

```
import pandas as pd

url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatEl
raw = pd.read_csv(url, sep=";")

# Parse timestamp and set as index
raw["time"] = pd.to_datetime(raw["time"])
raw = raw.set_index("time")

# Consumption per 15 minutes from the cumulative meter readings (kWh)
consumption = raw["FlatA_Ele"].diff()

# Daily sums of complete days only (all 96 quarter-hours have a reading):
# without min_count, days with missing data would show up as low or zero
# consumption
daily = consumption.resample("D").sum(min_count=96).dropna()

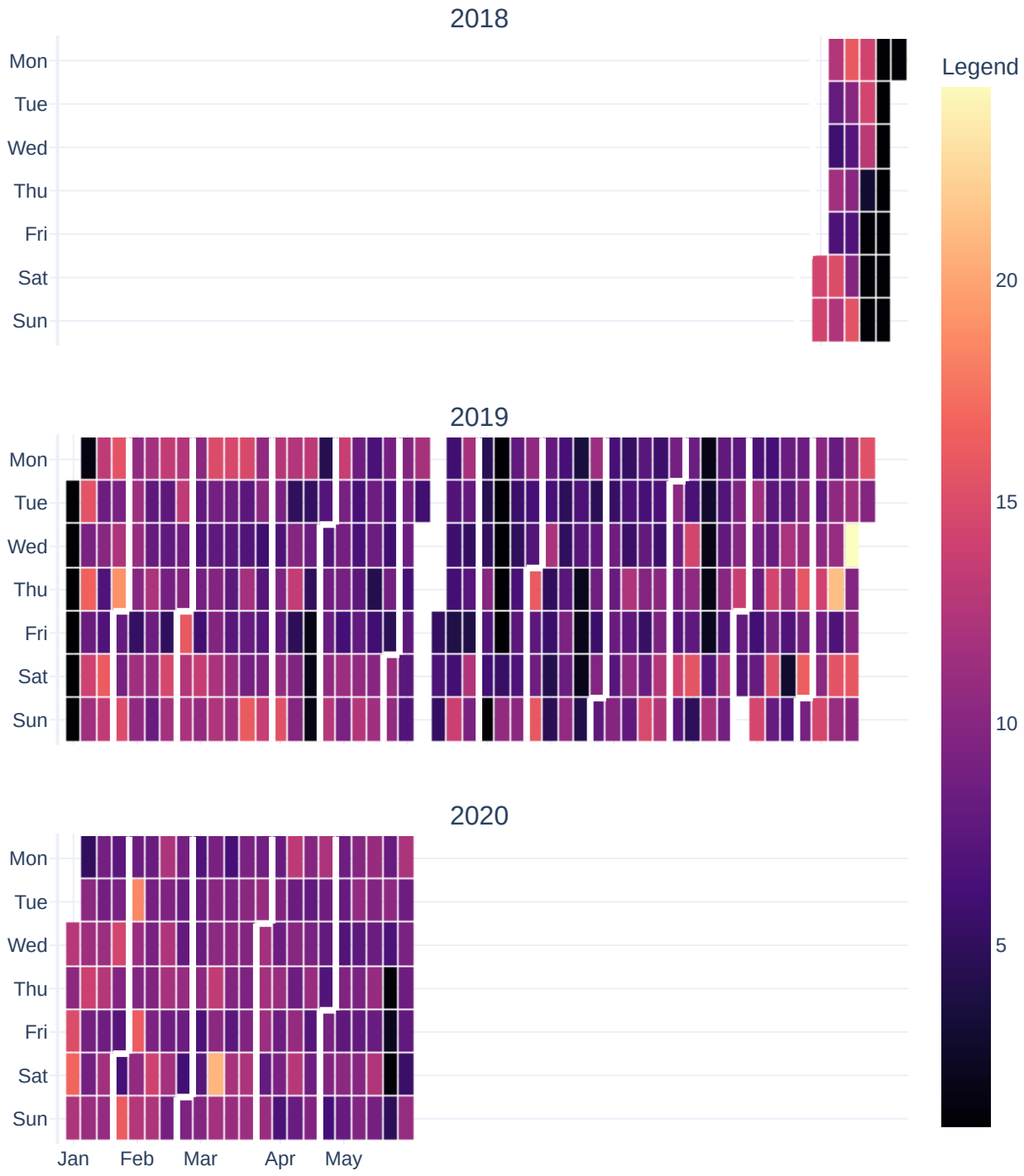
# Build clean 2-column DataFrame [timestamp, value]
df = daily.reset_index()
df.columns = ["timestamp", "value"]
df.head()
```

	timestamp	value
0	2018-12-01	14.324
1	2018-12-02	14.141
2	2018-12-03	12.444
3	2018-12-04	7.954
4	2018-12-05	5.732

Visualization

```
from pyedautils.plots import plot_heatmap_calendar

fig = plot_heatmap_calendar(df, title="Electricity – Calendar Heat Map",
                            ylab="Electricity [kWh/d]")
fig.show()
```



Customization

Median Weeks

Goal

Show the median weekly electricity consumption pattern as a heatmap with hours on one axis and weekdays on the other.

Data Basis

The dataset `flatElectricity.csv` contains cumulative electricity meter readings at 15-minute intervals. We convert cumulative readings to consumption values for `FlatA_Ele` and sum them per hour (complete hours only).

Data Preparation

```
import pandas as pd

url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatEl
raw = pd.read_csv(url, sep=";")

# Parse timestamp and set as index
raw["time"] = pd.to_datetime(raw["time"])
raw = raw.set_index("time")

# Consumption per 15 minutes from the cumulative meter readings (kWh)
consumption = raw["FlatA_Ele"].diff()

# Hourly sums of complete hours only (all 4 quarter-hours have a reading)
hourly = consumption.resample("h").sum(min_count=4).dropna()

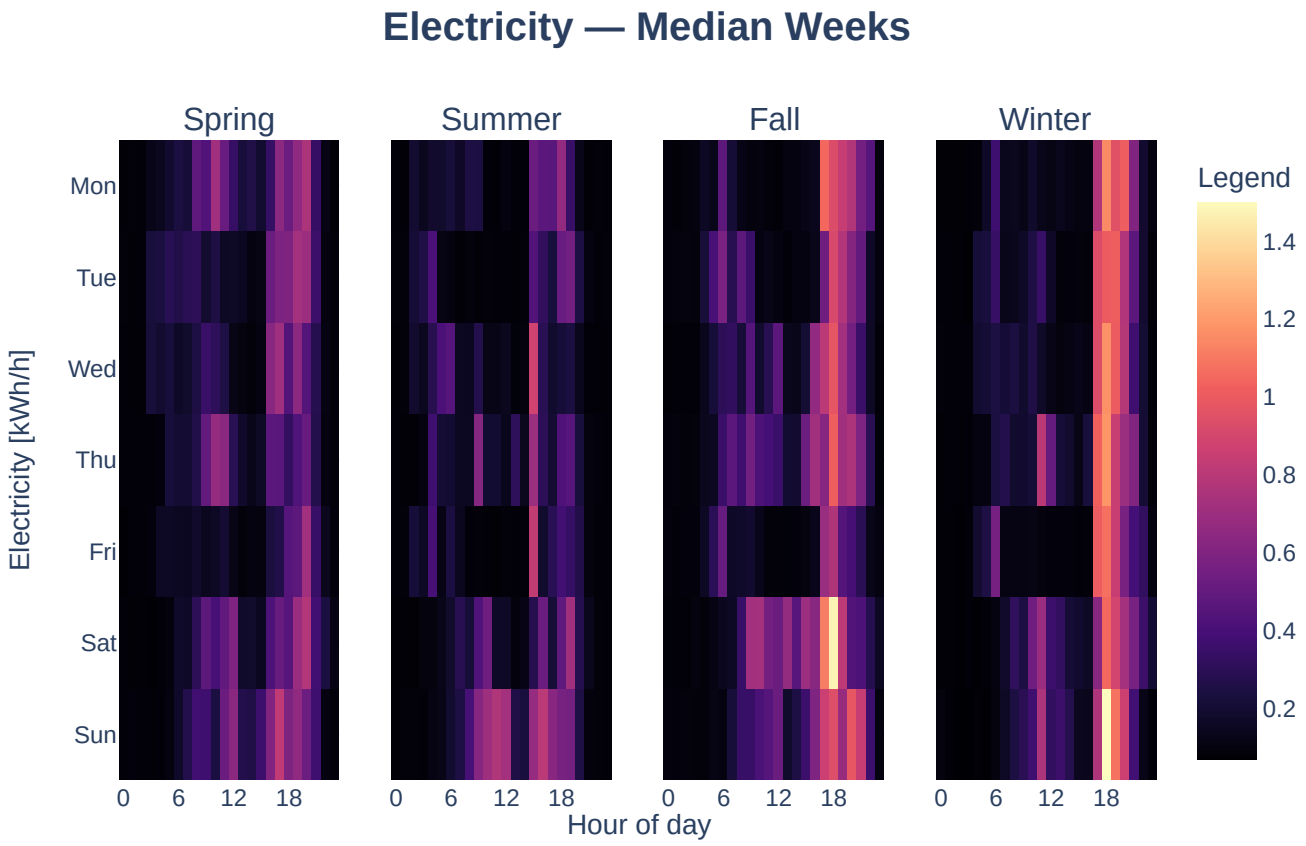
# Build clean 2-column DataFrame [timestamp, value]
df = hourly.reset_index()
df.columns = ["timestamp", "value"]
df.head()
```

	timestamp	value
0	2018-12-01 00:00:00	0.084
1	2018-12-01 01:00:00	0.073
2	2018-12-01 02:00:00	0.084
3	2018-12-01 03:00:00	0.075
4	2018-12-01 04:00:00	0.086

Visualization

```
from pyedautils.plots import plot_heatmap_median_weeks

fig = plot_heatmap_median_weeks(df, title="Electricity — Median Weeks",
                                ylab="Electricity [kWh/h]")
fig.show()
```



Customization

Daily Profiles

Daily profile visualizations show how a variable changes over the course of a day. They are essential for understanding usage patterns, occupancy behavior, and system operation.

Recipes

- **Overview** — Mean daily profile with confidence band for each season and weekday
- **Overlaid Profiles** — All days drawn on top of each other
- **Mean Profile** — One mean daily profile per weekday
- **Decomposed Profile** — Weekly pattern with the slow trend removed

Overview

> **Goal**
>
> Display a 4x7 grid of daily electricity consumption profiles, organized by season (rows) and weekday (columns), to reveal seasonal and weekly patterns.

> **Data Basis**
>
> Electricity data (`ele_meter.csv`) bundled with the `pyedautils` package: the mean electric power of a household in W per 15-minute interval (the 15-minute consumption in Wh × 4), for 2015.

Data Preparation

```
import pandas as pd
from importlib import resources

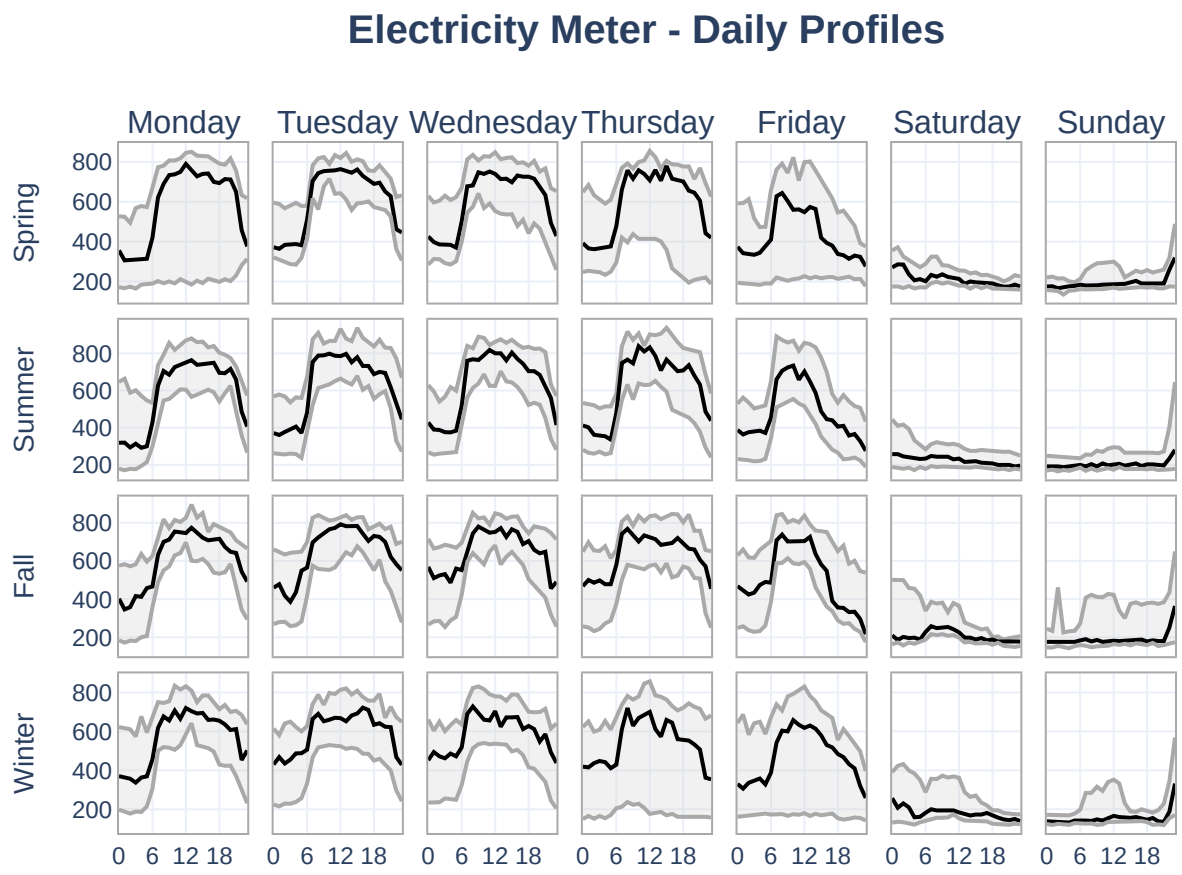
data_path = resources.files("pyedautils") / "data" / "ele_meter.csv"
df = pd.read_csv(data_path, sep=";")
df.head()
```

	timestamp	value
0	2015-01-01 00:00:00	40.0
1	2015-01-01 00:15:00	32.0
2	2015-01-01 00:30:00	32.0
3	2015-01-01 00:45:00	34.0
4	2015-01-01 01:00:00	34.0

Visualization

```
from pyedautils.plots import plot_daily_profiles_overview

fig = plot_daily_profiles_overview(
    data=df,
    title="Electricity Meter - Daily Profiles",
    ylab="Power [W]",
    confidence=95.0,
)
fig.show()
```



Customization

Overlaid Profiles

> **Goal**
>
> Overlay all individual daily electricity consumption profiles. Hovering highlights the entire day-line in orange for comparison.

> **Data Basis**
>
> Electricity data (`ele_meter.csv`) bundled with the `pyedautils` package: the mean electric power of a household in W per 15-minute interval (the 15-minute consumption in Wh × 4), for 2015.

Data Preparation

```
import pandas as pd
from importlib import resources

data_path = resources.files("pyedautils") / "data" / "ele_meter.csv"
df = pd.read_csv(data_path, sep=";")
df.head()
```

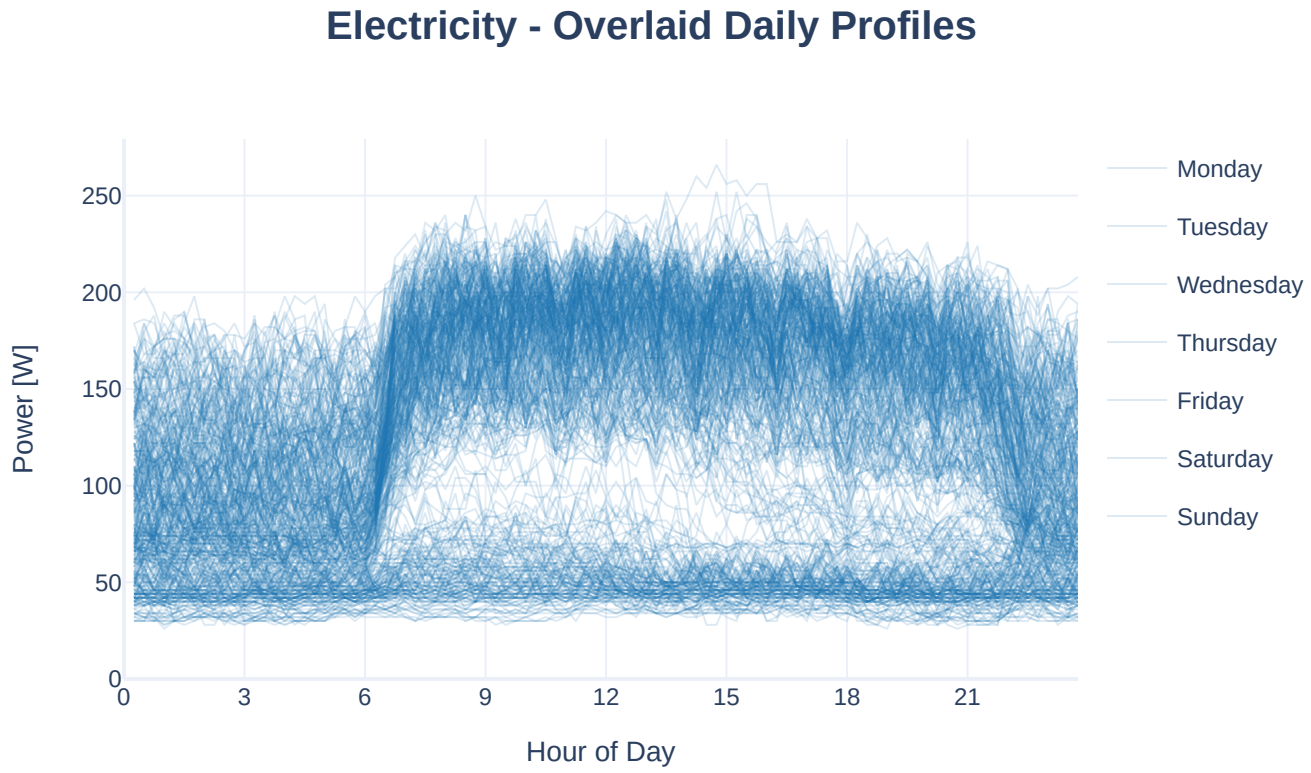
	timestamp	value
0	2015-01-01 00:00:00	40.0
1	2015-01-01 00:15:00	32.0
2	2015-01-01 00:30:00	32.0
3	2015-01-01 00:45:00	34.0
4	2015-01-01 01:00:00	34.0

Visualization

Note: The `method="overlayed"` option returns an HTML string instead of a Plotly Figure.

```
from pyedautils.plots import plot_daily_profiles
from IPython.display import HTML

html = plot_daily_profiles(
    data=df,
    method="overlayed",
    title="Electricity - Overlaid Daily Profiles",
    ylab="Power [W]",
)
HTML(html)
```



Mean Profile

> **Goal**
>
> Display the average daily electricity consumption profile with one line per weekday to compare typical usage across the week.

Data Basis

Electricity data (`ele_meter.csv`) bundled with the `pyedautils` package: the mean electric power of a household in W per 15-minute interval (the 15-minute consumption in Wh × 4), for 2015.

Data Preparation

```
import pandas as pd
from importlib import resources

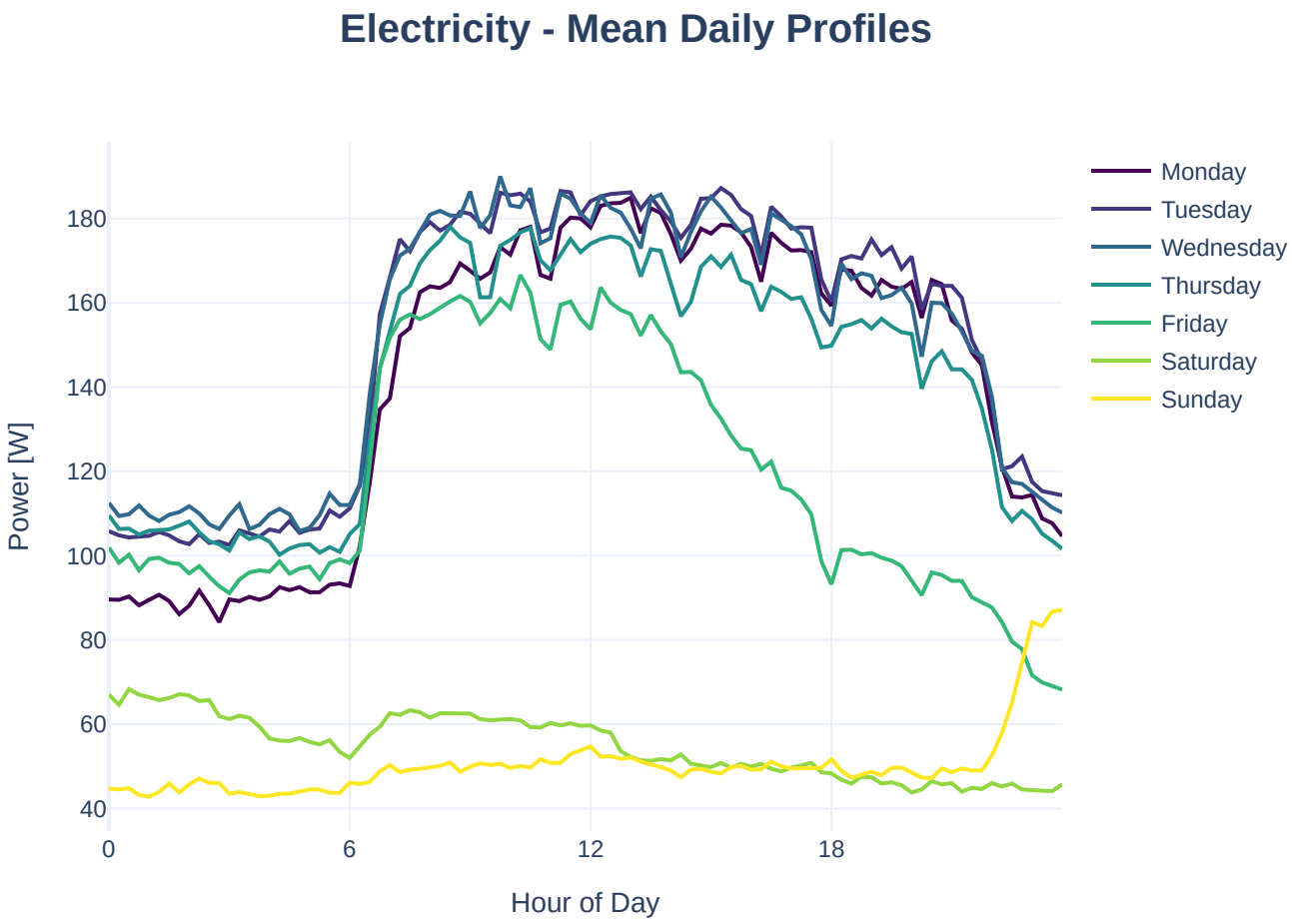
data_path = resources.files("pyedautils") / "data" / "ele_meter.csv"
df = pd.read_csv(data_path, sep=";")
df.head()
```

	timestamp	value
0	2015-01-01 00:00:00	40.0
1	2015-01-01 00:15:00	32.0
2	2015-01-01 00:30:00	32.0
3	2015-01-01 00:45:00	34.0
4	2015-01-01 01:00:00	34.0

Visualization

```
from pyedautils.plots import plot_daily_profiles

fig = plot_daily_profiles(
    data=df,
    method="mean",
    loc_time_zone="Europe/Zurich",
    title="Electricity - Mean Daily Profiles",
    ylab="Power [W]",
)
fig.show()
```



Decomposed Profile

Goal

Display daily electricity consumption profiles with the trend component removed, isolating the intra-day pattern from long-term drift.

Data Basis

Electricity data (`ele_meter.csv`) bundled with the `pyedautils` package: the mean electric power of a household in W per 15-minute interval (the 15-minute consumption in Wh × 4), for 2015.

Data Preparation

```
import pandas as pd
from importlib import resources

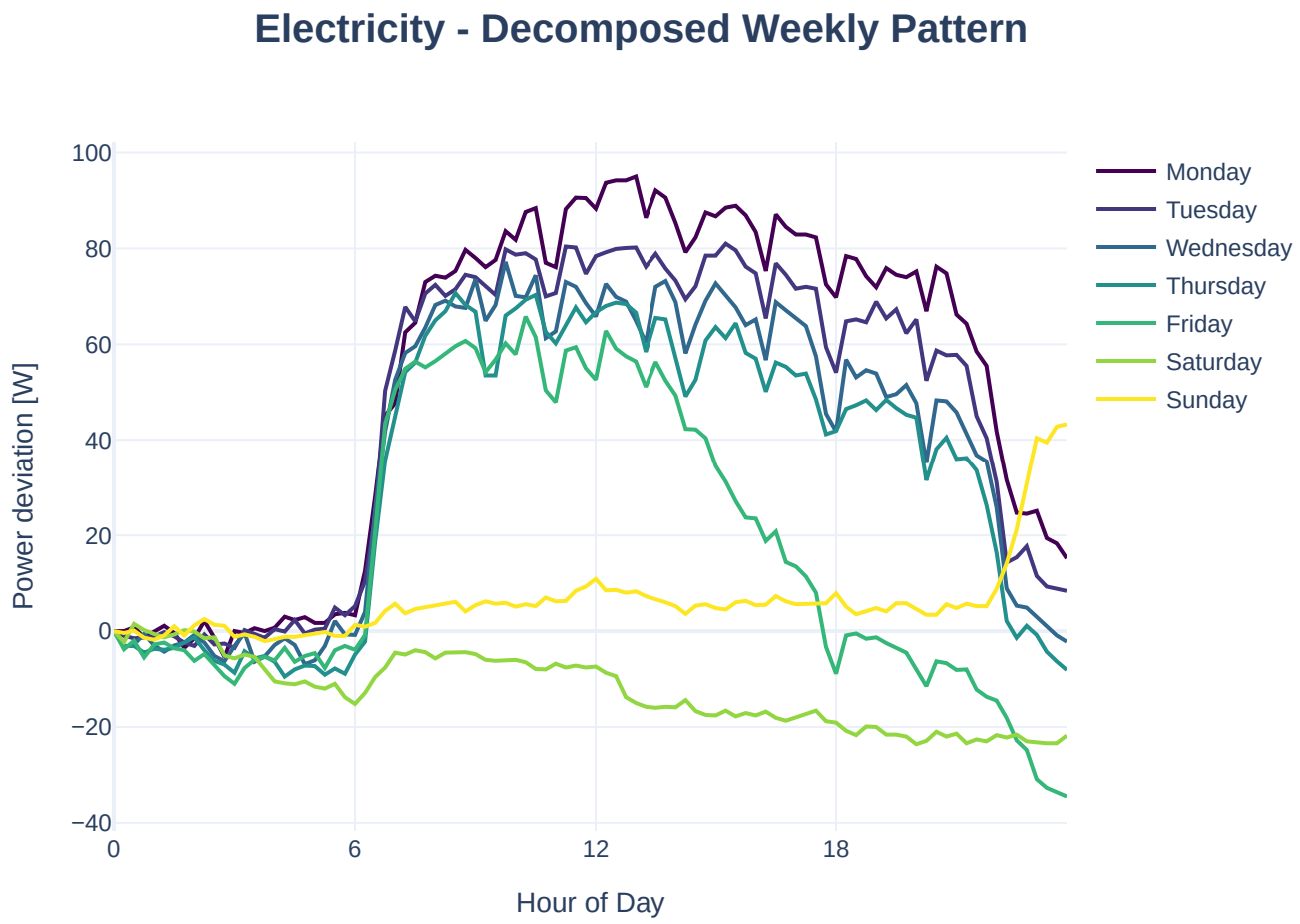
data_path = resources.files("pyedautils") / "data" / "ele_meter.csv"
df = pd.read_csv(data_path, sep=";")
df.head()
```

	timestamp	value
0	2015-01-01 00:00:00	40.0
1	2015-01-01 00:15:00	32.0
2	2015-01-01 00:30:00	32.0
3	2015-01-01 00:45:00	34.0
4	2015-01-01 01:00:00	34.0

Visualization

```
from pyedautils.plots import plot_daily_profiles

fig = plot_daily_profiles(
    data=df,
    method="decomposed",
    loc_time_zone="Europe/Zurich",
    title="Electricity - Decomposed Weekly Pattern",
    ylab="Power deviation [W]",
)
fig.show()
```



Customization

Sum & Frequency

Duration curves sort the consumption of each day or hour from highest to lowest. They show how often a consumption level is exceeded, how high the peaks are and how much of the time the base load runs.

Recipes

- **Daily Duration Curve** — Daily consumption sorted from highest to lowest
- **Hourly Duration Curve** — Hourly consumption sorted from highest to lowest

Daily Duration Curve

> **Goal**
>
> Display a duration curve (sum frequency) of daily electricity consumption. The curve ranks daily totals from highest to lowest, revealing how many days exceed a given consumption level.

> **Data Basis**
>
> The dataset `flatElectricity.csv` contains cumulative electricity meter readings at 15-minute intervals (columns: `time`, `FlatA_Ele`, `FlatB_Ele`, `FlatC_Ele`, `FlatD_Ele`; separator `;`). We use `FlatA_Ele`, compute consumption via `diff()`, and sum it per day, keeping only days without missing readings.

Data Preparation

```
import pandas as pd

url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatEl
raw = pd.read_csv(url, sep=";")

# Parse timestamp and set as index
raw["time"] = pd.to_datetime(raw["time"])
raw = raw.set_index("time")

# Consumption per 15 minutes from the cumulative meter readings (kWh)
consumption = raw["FlatA_Ele"].diff()

# Daily sums of complete days only (all 96 quarter-hours have a reading):
# without min_count, days with missing data would show up as low or zero
# consumption
daily = consumption.resample("D").sum(min_count=96).dropna()

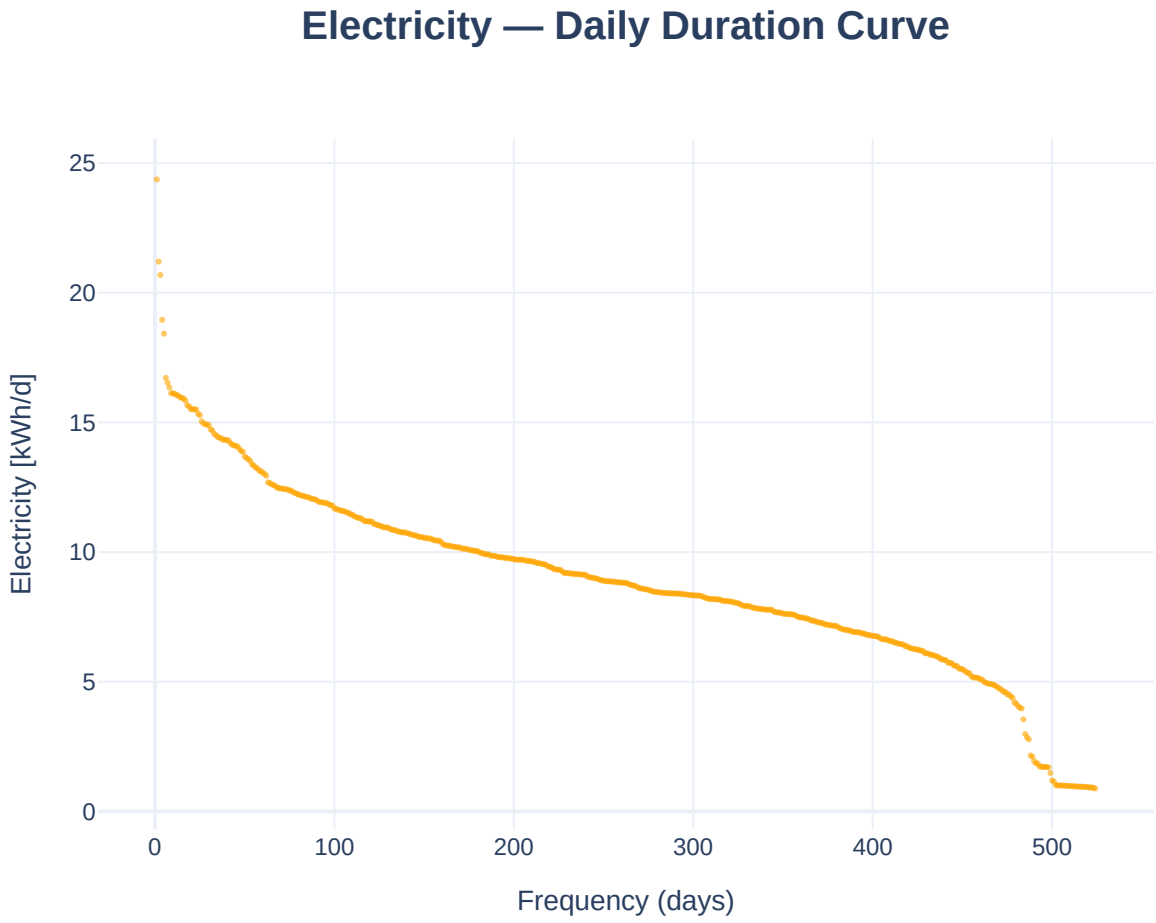
# Build clean 2-column DataFrame [timestamp, value]
df = daily.reset_index()
df.columns = ["timestamp", "value"]
df.head()
```

	timestamp	value
0	2018-12-01	14.324
1	2018-12-02	14.141
2	2018-12-03	12.444
3	2018-12-04	7.954
4	2018-12-05	5.732

Visualization

```
from pyedautils.plots import plot_sum_frequency

fig = plot_sum_frequency(df, resolution="daily", reverse=True,
                         title="Electricity — Daily Duration Curve",
                         ylab="Electricity [kWh/d]")
fig.show()
```



Customization

Hourly Duration Curve

Goal

Display a duration curve (sum frequency) of hourly electricity consumption. The curve ranks hourly totals from highest to lowest, showing the distribution of consumption intensities.

Data Basis

The dataset `flatElectricity.csv` contains cumulative electricity meter readings at 15-minute intervals (columns: `time`, `FlatA_Ele`, `FlatB_Ele`, `FlatC_Ele`, `FlatD_Ele`; separator `;`). We use `FlatA_Ele`, compute consumption via `diff()`, and sum it per hour, keeping only hours without missing readings.

Data Preparation

```
import pandas as pd

url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatEl
raw = pd.read_csv(url, sep=";")

# Parse timestamp and set as index
raw["time"] = pd.to_datetime(raw["time"])
raw = raw.set_index("time")

# Consumption per 15 minutes from the cumulative meter readings (kWh)
consumption = raw["FlatA_Ele"].diff()

# Hourly sums of complete hours only (all 4 quarter-hours have a reading)
hourly = consumption.resample("h").sum(min_count=4).dropna()

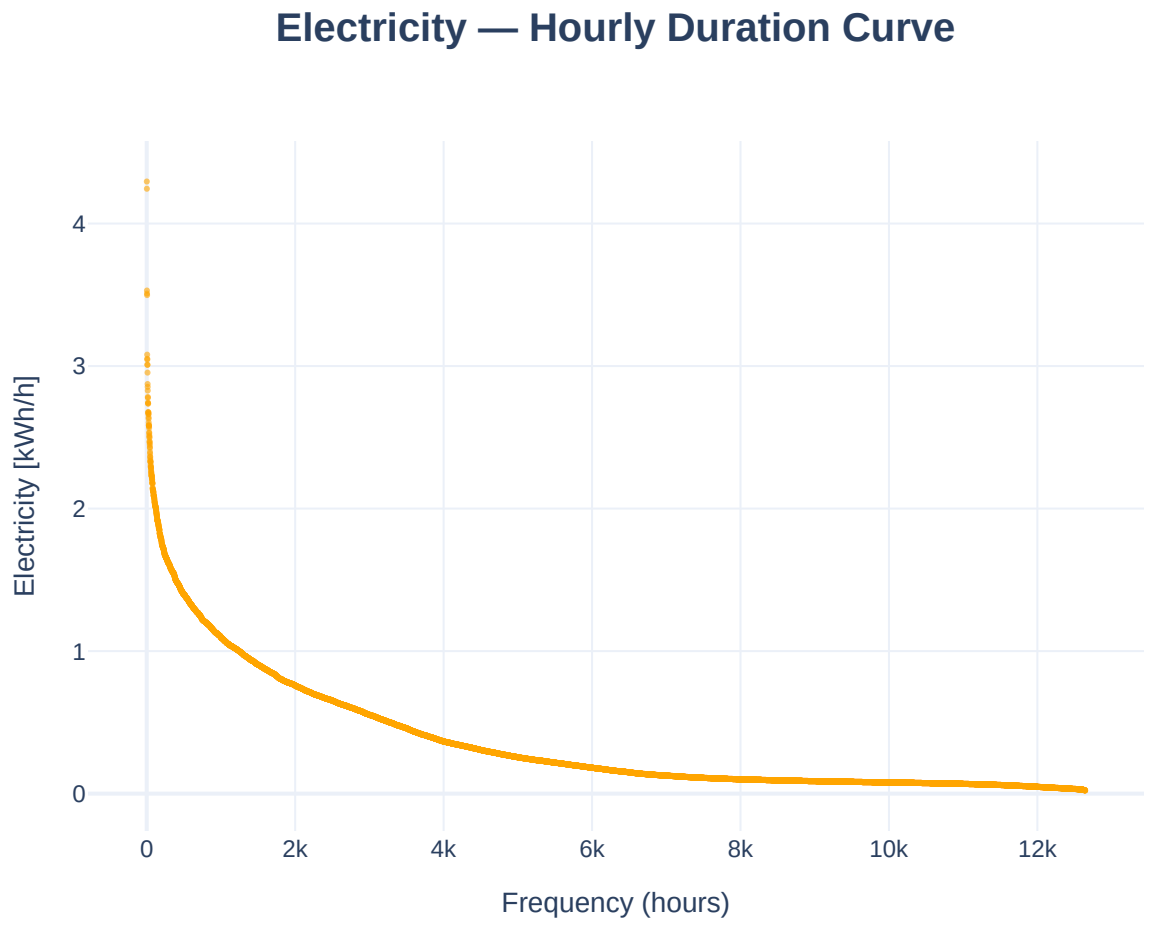
# Build clean 2-column DataFrame [timestamp, value]
df = hourly.reset_index()
df.columns = ["timestamp", "value"]
df.head()
```

	timestamp	value
0	2018-12-01 00:00:00	0.084
1	2018-12-01 01:00:00	0.073
2	2018-12-01 02:00:00	0.084
3	2018-12-01 03:00:00	0.075
4	2018-12-01 04:00:00	0.086

Visualization

```
from pyedautils.plots import plot_sum_frequency

fig = plot_sum_frequency(df, resolution="hourly", reverse=True,
                         title="Electricity — Hourly Duration Curve",
                         ylab="Electricity [kWh/h]")
fig.show()
```



Comfort Diagrams

Comfort diagrams visualize indoor climate conditions and assess whether they meet comfort standards. These plots are essential for building performance evaluation.

Recipes

- **Temperature vs. Humidity** — Scatter plot of indoor temperature and humidity with comfort zone overlay
- **Temperature & Humidity Over Time** — Stacked time series with the comfort target band plus moderate and severe warning limits
- **Mollier h-x Diagram** — Psychrometric chart showing the relationship between temperature, humidity, and enthalpy, with warning bands and the outside air with its ventilation line
- **Comfort Donut** — Share of time temperature and humidity spend below / within / above the comfort band
- **Comfort Compass** — Area-true polar glyph of the days spent in each comfort state and severity stage
- **SIA 180 Adaptive Comfort** — Evaluate indoor temperatures against the SIA 180:2014 adaptive comfort model

Reading the diagrams

Two short pyedautils guides explain how to read the h-x diagram and the comfort compass, what the comfort zone means and how to save energy, with worked examples.

Temperature & Humidity Over Time

Goal

Plot indoor temperature and humidity as two stacked **time series**. Each shows the hourly readings as faint dots and the daily mean as a line that **breaks at data gaps**, on top of the green **comfort target band** plus the **moderate** and **severe** warning limits. Use it to see *when* — and how far — the indoor climate leaves the comfort range over the year.

Data Basis

We use the bundled dataset `mollier_sample.csv` shipped with **pyedautils** — anonymized hourly indoor temperature and humidity from a real sensor. It is loaded directly from the installed package (`importlib.resources`), so no download is needed.

Data Preparation

```
import pandas as pd
from importlib import resources

data_path = resources.files("pyedautils") / "data" / "mollier_sample.csv"
df = pd.read_csv(data_path, parse_dates=["timestamp"])

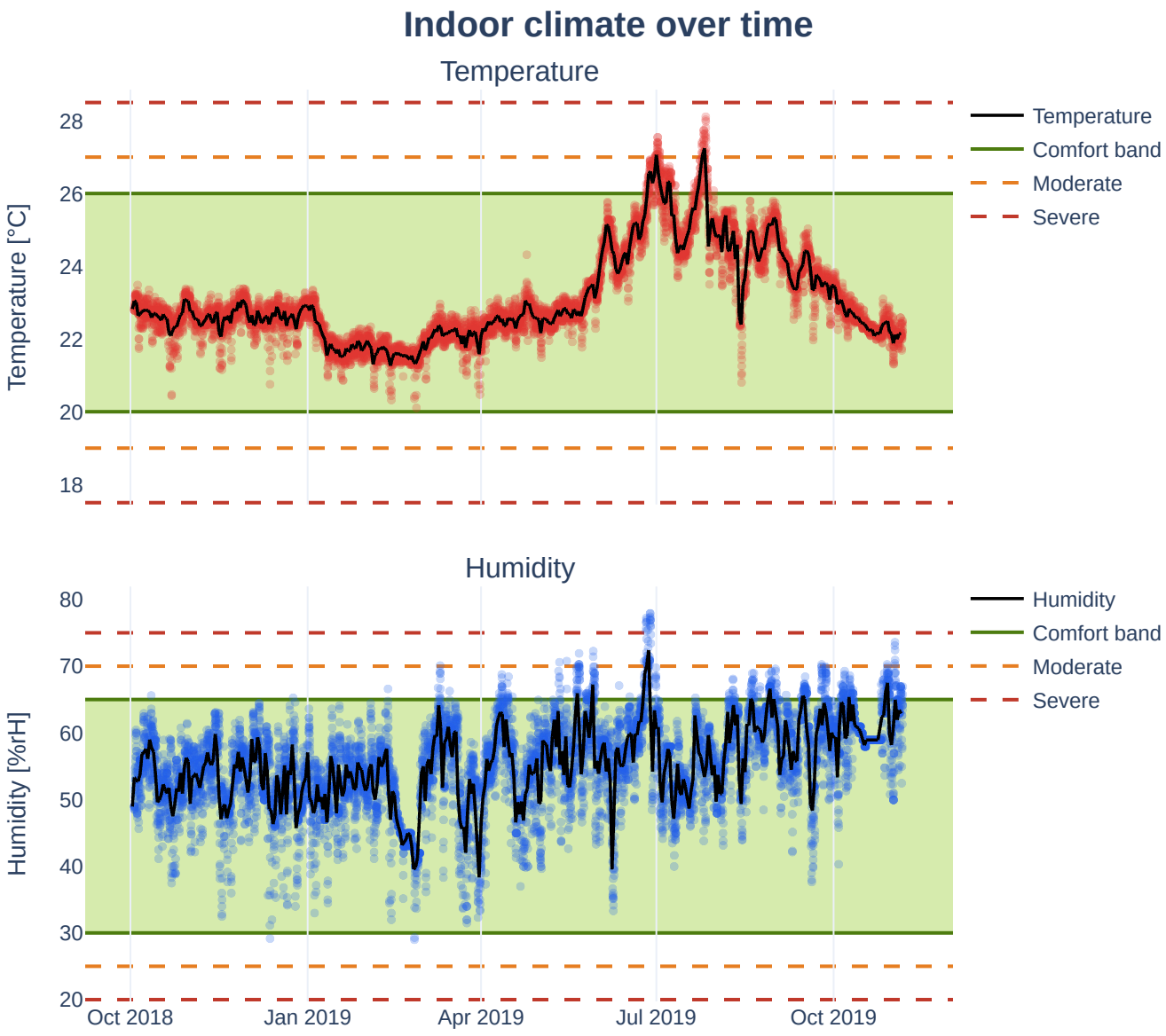
df.head()
```

	timestamp	humidity	temperature
0	2018-10-02 22:00:00+00:00	49.000000	22.866667
1	2018-10-02 23:00:00+00:00	49.000000	22.816667
2	2018-10-03 00:00:00+00:00	49.000000	22.800000
3	2018-10-03 01:00:00+00:00	48.500000	22.800000
4	2018-10-03 02:00:00+00:00	48.666667	22.800000

Visualization

```
from pyedautils.plots import plot_temp_humidity_timeseries

# The bands default to the Mollier h,x comfort zone
# (T 20-26 deg C, phi 30-65 %, with +/- 1 K / 5 % and +/- 2.5 K / 10 % limits).
fig = plot_temp_humidity_timeseries(
    df,
    temp_title="Temperature",
    hum_title="Humidity",
    title="Indoor climate over time",
)
fig.show()
```



Customization

Temperature vs. Humidity

Goal

Scatter plot of indoor temperature against relative humidity with comfort zone boundaries. This helps evaluate whether indoor conditions fall within acceptable thermal comfort ranges.

Data Basis

The dataset `flatTempHum.csv` contains hourly indoor temperature and humidity measurements for multiple flats (columns: `time`, `FlatA_Hum`, `FlatA_Temp`, `FlatB_Hum`, `FlatB_Temp`, ...; separator `;`). We use `FlatA_Temp` and `FlatA_Hum` and build a 3-column DataFrame [timestamp, temperature, humidity].

Data Preparation

```
import pandas as pd

url = "https://raw.githubusercontent.com/retomarek/edap/main/edap/sampleData/flatTe
raw = pd.read_csv(url, sep=";")

# Parse timestamp
raw["time"] = pd.to_datetime(raw["time"])

# Select [timestamp, temperature, humidity] — 3-column DataFrame
df = raw[["time", "FlatA_Temp", "FlatA_Hum"]].copy()
df.columns = ["timestamp", "temperature", "humidity"]
df = df.dropna()

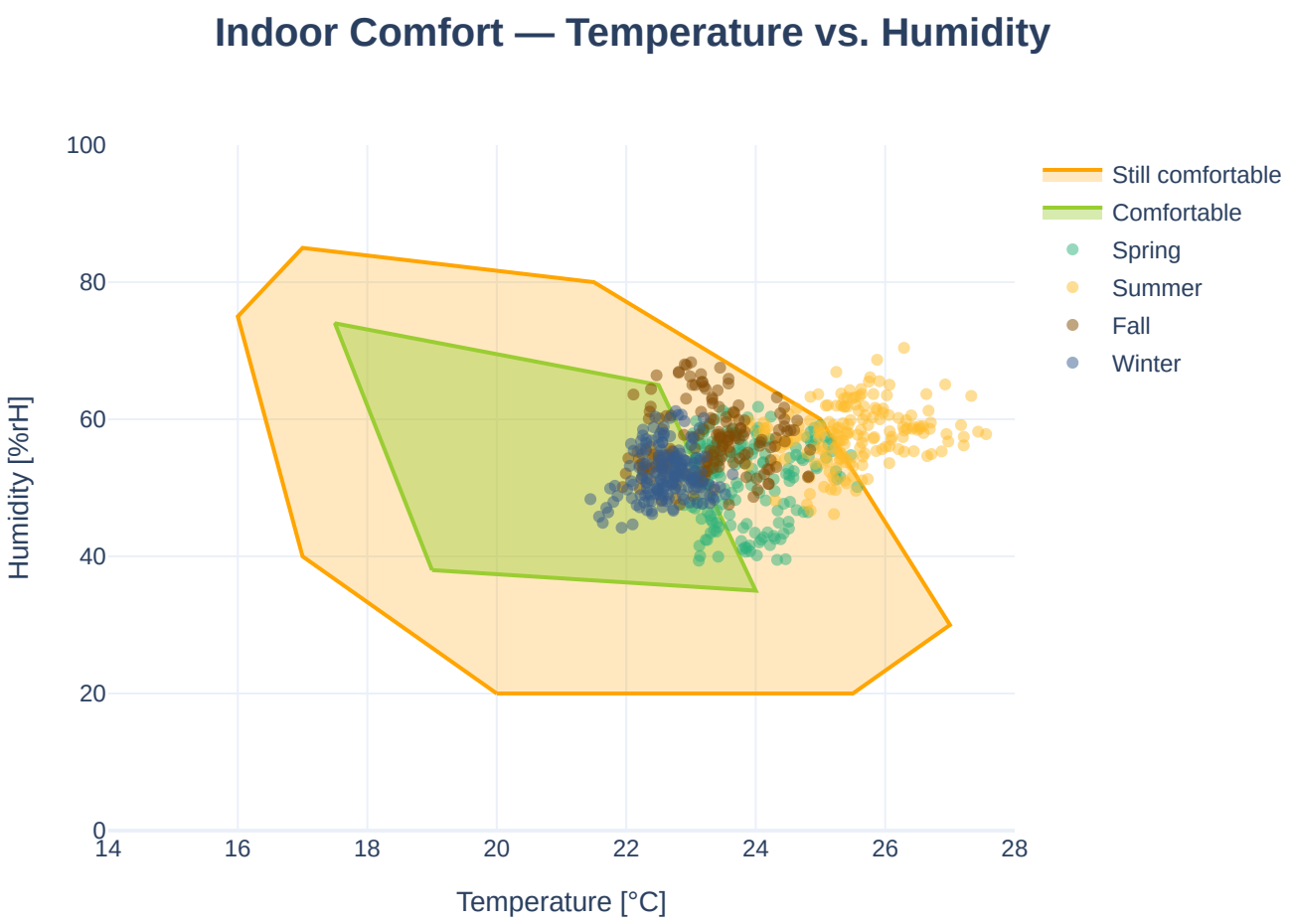
df.head()
```

	timestamp	temperature	humidity
0	2018-10-03 00:00:00	24.43	53.0
1	2018-10-03 01:00:00	24.40	53.0
2	2018-10-03 02:00:00	24.40	53.0
3	2018-10-03 03:00:00	24.40	53.0
4	2018-10-03 04:00:00	24.40	53.3

Visualization

```
from pyedautils.plots import plot_comfort_temp_humidity

fig = plot_comfort_temp_humidity(df, title="Indoor Comfort — Temperature vs. Humidi
fig.show()
```



Customization

Mollier h-x Diagram

Goal

Display measured indoor conditions on a Mollier h-x (psychrometric) diagram — the relationship between temperature, humidity and enthalpy of moist air, with a comfort zone, dashed warning bands, the outside air with its ventilation line and frequency contour lines.

Data Basis

We use the bundled dataset `room_climate_sample.csv` shipped with **pyedautils** — anonymized hourly temperature and humidity from a real sensor in a cool, humidity-controlled room. It is loaded directly from the installed package (`importlib.resources`), so no download is needed.

Reading the diagram

New to the h-x diagram? The pyedautils guide The h,x diagram in five minutes explains how to read it, the dew point, the comfort zone and how to save energy.

Data Preparation

```
import pandas as pd
from importlib import resources

data_path = resources.files("pyedautils") / "data" / "room_climate_sample.csv"
df = pd.read_csv(data_path)   # columns: timestamp, humidity, temperature
df.head()
```

	timestamp	humidity	temperature
0	2025-05-01 00:00:00	51.0	19.1
1	2025-05-01 01:00:00	51.0	19.1
2	2025-05-01 02:00:00	51.0	19.1
3	2025-05-01 03:00:00	51.0	19.1
4	2025-05-01 04:00:00	50.2	19.1

Visualization

Pass the DataFrame and the room's target band to `plot_mollier_hx`. This cool, humidity-controlled room is kept at 8–18 °C and 40–70 % — the same band as in the Comfort Donuts and Comfort Compass recipes — drawn as the green **comfort zone**. Measured points are colored by season. All charts are interactive — hover a point for details.

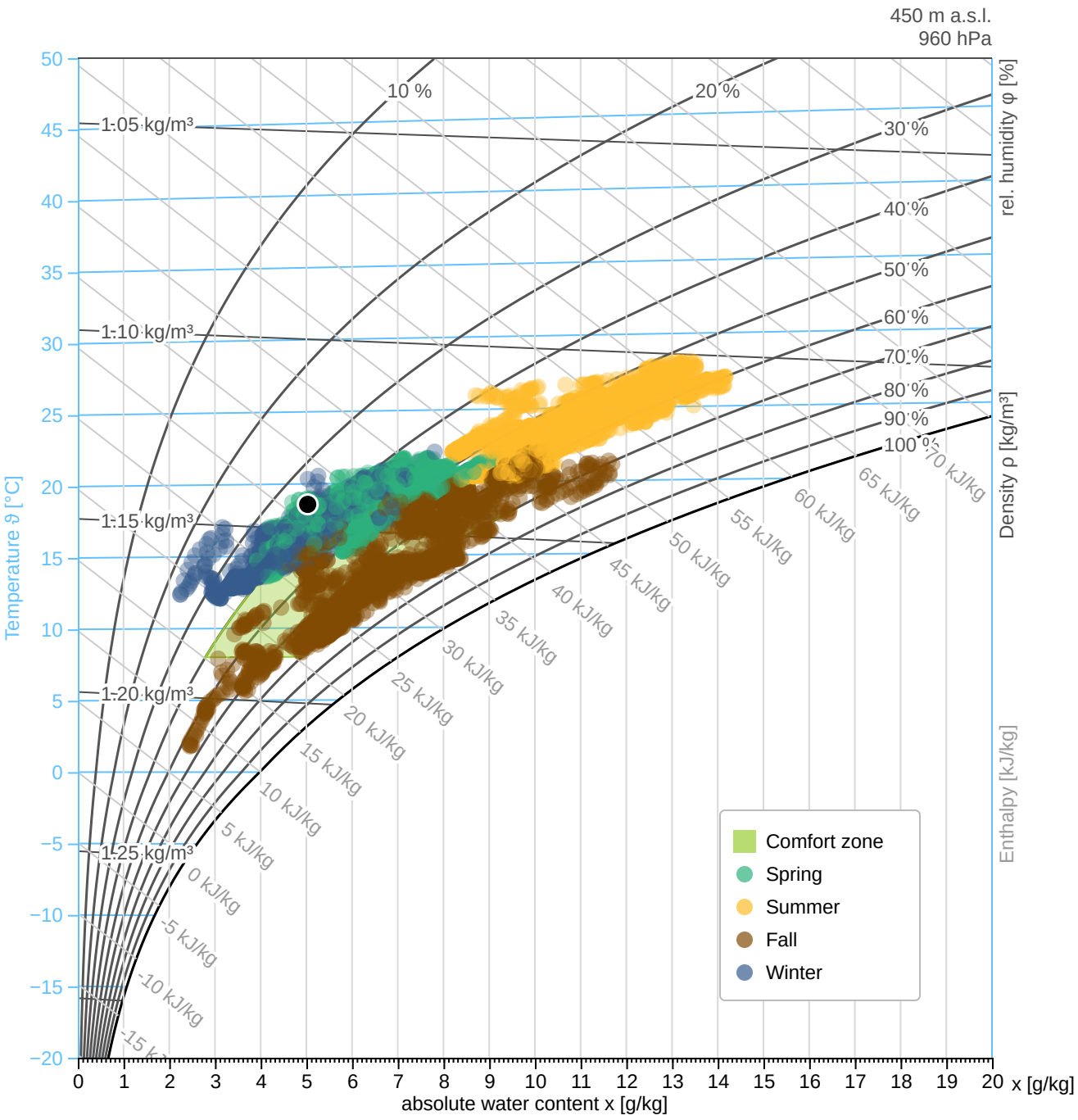
Every diagram below is drawn for the building's **site altitude** (here 450 m a.s.l.): `plot_mollier_hx` derives the air pressure from it, so the psychrometric relations are correct for the thinner air. Altitude and pressure are written in the top-right corner.

```
from pyedautils.plots import plot_mollier_hx
from IPython.display import HTML, display

altitude = 450  # m a.s.l., the air pressure follows from it; reused below

# Target band of this room: 8-18 °C, 40-70 % rH, no limit on the
# absolute humidity (as in the Comfort Compass recipe)
zone = {"temperature": (8, 18), "rel_humidity": (0.40, 0.70),
        "abs_humidity": None}

html = plot_mollier_hx(df, altitude=altitude, comfort_zone=zone, height=700)
display(HTML(html))
```


450 m a.s.l.
960 hPa
10 %
20 %
30 %
40 %
50 %
60 %
70 %
80 %
90 %
100 %
1.05 kg/m³
1.10 kg/m³
1.15 kg/m³
1.20 kg/m³
1.25 kg/m³
70 kJ/kg
65 kJ/kg
60 kJ/kg
55 kJ/kg
50 kJ/kg
45 kJ/kg
40 kJ/kg
35 kJ/kg
30 kJ/kg
25 kJ/kg
20 kJ/kg
15 kJ/kg
10 kJ/kg
5 kJ/kg
0 kJ/kg
-5 kJ/kg
-10 kJ/kg
-15 kJ/kg
Comfort zone
Spring
Summer
Fall
Winter
Temperature ϑ [°C]
rel. humidity φ [%]
Density ρ [kg/m³]
Enthalpy [kJ/kg]
absolute water content x [g/kg]
x [g/kg]