```
X.append(avg_color("fruit.jpg"))
labels.append("banana")
df = pd.DataFrame(X,
     columns=["R", "G", "B"])
df["label"] = labels
# Train the model
knn = KNeighborsClassifier()
knn.fit(df[["R", "G", "B"]],
        df["label"])
# Make prediction
knn.predict(test_df)
```

Python, Aut
Artificial
Data Scienc
Pandas, API
Flask, NumP
Visual Stud
Classificat

Python for AI and Data Analysis

The Practical Guide for Business and Science

Johannes Schildgen

Rheinwerk Computing

Contents

Chapter 5
Data Analysis

The two Python modules, NumPy and pandas, make it possible to load, filter, and transform data from files and other sources, and to perform analyses on this data.

One of the most common applications of the Python programming language, especially among non-computer scientists, is data analysis. For this purpose, data is usually read from files or accessed via the Internet. However, before you can begin the actual data analysis, you must first perform *preprocessing* steps, also known as *data cleaning*. This is because data is rarely available exactly as you need it. Typical preprocessing steps include:

- Filtering data: Retaining only relevant data; excluding the rest
- Identifying and deleting or filling in missing values
- Removing duplicates: Deleting or merging multiple entries with the same content
- Identifying and correcting errors and outliers in the data: Typos, incorrect date formats, extreme values, etc.
- Transforming data into the desired data format
- Encodinging/decodinging categories: for example, convert a `0` to the value `"iPhone,"` a `1` to `"Android"`
- Normalizing/making values comparable: Ensuring consistent units; scaling to the range `[0, 1]`.

These steps are necessary to bring the raw data into a clean, consistent, and analyzable form. Data cleaning is often the most time-consuming part of performing data analysis. Once it's done, the actual work can begin: You can then calculate statistics (means, medians, etc.), create charts for visualization, or prepare the data for further analysis steps or machine learning. Finally, the results and insights from the data can be evaluated and interpreted to draw conclusions and derive recommendations.

In this chapter, we'll look at the two modules NumPy and pandas. These two tools form the foundation for both simple and complex data analysis in Python. They allow you to load and save data, as well as perform data cleaning and calculations—even on very large datasets.

5.1 NumPy

The `numpy` module provides an important data type called a *NumPy array*. It's used to store vectors and matrices of numbers. We'll need this in later chapters, for example, to create charts.

```
import numpy as np
a = np.array([7, 8, 9])
b = np.array([[1, 2, 3], [4, 5, 6]])
```

When importing the `numpy` module, it has been agreed to write `import numpy as np` to give the module the shorter alias `np`. After that, you can simply write `np.array`.

Our variable `a` now contains a one-dimensional NumPy array, a vector with three numbers:

$$a = \begin{pmatrix} 7 \\ 8 \\ 9 \end{pmatrix}$$

The individual components of the vector can be accessed in the same way as list elements: `a[0]` returns the first number, that is, `7`. With `a[0:2]`, we obtain a NumPy array consisting only of the first two entries. The from position (`0`) is included, but the to position (`2`) isn't. It therefore returns the entries at positions 0 and 1, that is, `[7, 8]`.

The variable `b` is a two-dimensional NumPy array. It's initialized using a list of lists. We pass the values in the individual columns row by row, thereby defining our matrix:

$$b = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Now, we can access a value using `b[row_index, column_index]`. Here, `b[0, 0]` is the value at the top left, which in our case is the number 1. `b[0]` gives us the entire first row, and `b[:, 0]` gives the entire first column, each as a NumPy array.

One extremely useful feature is the ability to filter entries in NumPy arrays. This is because we can specify not only a direct index `[0]` or a range `[0:2]` within the square brackets but also a filter predicate. This returns a NumPy array containing only the entries that satisfy that predicate. In the following prompting example, our NumPy array contains the daily number of steps taken over the course of a week.

```
steps = np.array([8234, 11789, 3120, 10054, 7422, 4821, 15337])
```

I only want the entries where my step goal (7,500 steps per day) is met.

> You can easily formulate a condition and filter the array with NumPy.
>
> ```
> import numpy as np
> steps = np.array([8234, 11789, 3120, 10054, 7422, 4821, 15337])
> target = 7500
> met = steps[steps >= target]
> print(met)
> ```

The program outputs the NumPy array `[ 8234 11789 10054 15337]`. Using `len(met)`, we get the number of days on which the step goal was met, which is 4.

In NumPy, and later in pandas as well, it's almost never necessary to iterate over the entire dataset using a loop and perform a check or calculation element by element. This is because NumPy allows you to perform calculations on the entire array at once. For example, if we were to write `steps += 1` (stands for: `steps = steps + 1`), a new NumPy array would result in which every single number is one greater than in the original array. Both in this transformation and when filtering entries, NumPy creates a copy of the array with the new numbers. However, if we use `[0:2]` to slice the array into a smaller array, this is only a virtual copy. If the entries in the original array change afterward, they are also changed in this virtual copy:

```
a = np.array([7, 8, 9])
b = a[0:2]              # array([7, 8])
a[0] = 0
print(b)                # array([0, 8])
```

Two NumPy arrays can also be added together, and we can perform matrix multiplications, calculate determinants, and much more. But we won't do that now. If you need something like that, look it up online or ask a chatbot. With our current knowledge, however, we'll be able to create plots using Matplotlib in a later chapter.

5.2 Pandas

The name *pandas* has nothing to do with the animal by the same name; it comes from the term *panel data*. This refers to two-dimensional data, which is what we're mainly dealing with when we perform data analysis using pandas. The two most important data structures we need are called Series and DataFrames.

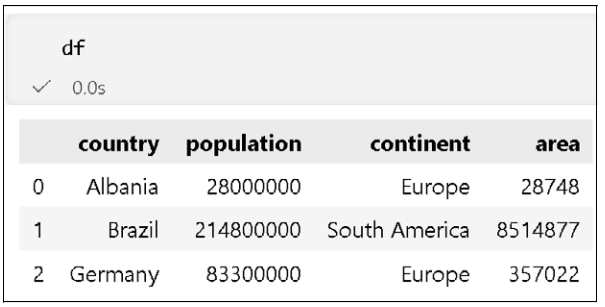
A pandas *Series* is a one-dimensional data structure, similar to a list or a NumPy array. However, the best way to visualize a Series is as a column in a table. For example, if we have a table listing countries of the world, each column is a separate Series: one containing country names, one containing population numbers, and so on. Unlike with a NumPy array, we're not necessarily dealing with numbers here; a Series can also contain strings, date values, and more. Working with a Series is still very similar to what we're used to with NumPy arrays:

```
import pandas as pd

steps = pd.Series([8234, 11789, 3120, 10054, 7422, 4821, 15337])

print(steps[0])             # first entry (8234)
print(steps[0:2])           # Series with the first
                            # two entries [(8234, 11789)]
print(steps[steps > 7500])  # Series containing the entries
                            # that are greater than 7500
```

In a Series, each entry has an index. By default, this starts at 0 and is numbered in ascending order. However, it's also possible to assign different indices to entries, even a string. This makes the series resemble a dictionary rather than a list:

```
data = {"Mon": 8234, "Tue": 11789, "Wed": 3120, "Thu": 10054,
        "Fri": 7422, "Sat": 4821, "Sun": 15337}
steps = pd.Series(data)
print(steps["Wed"])
```

However, the even more important data structure in pandas is the *DataFrame*. We can think of a DataFrame as a table. A table has fixed columns, each column has a name, and each row of the DataFrame contains a value of a specific data type in every column.



```
df
✓ 0.0s
```

	country	population	continent	area
0	Albania	28000000	Europe	28748
1	Brazil	214800000	South America	8514877
2	Germany	83300000	Europe	357022

Figure 5.1: Pandas DataFrame in Jupyter Notebooks

As with Series, each row in a DataFrame has an index that identifies the row; by default, this is 0, 1, 2, and so on. Strictly speaking, a DataFrame is a sequence of multiple Series with the same index.

```
import pandas as pd
df = pd.DataFrame([["Albania", 28000000, "Europe", 28748],
                   ["Brazil", 214800000, "South America", 8514877],
                   ["Germany", 83300000, "Europe", 357022]],
                  columns=["country", "population", "continent",
"area"])
```

Now that the DataFrame, which we've named `df`, has been created, we can access the data stored in it as follows:

- **`df`**
  DataFrame (displayed as a table in Jupyter Notebooks).
- **`df.head()`** or **`df.head(5)`**
  Only the first five rows.
- **`df.tail()`** or **`df.tail(5)`**
  Only the last five rows.

- **`df["country"]`**
  The `"country"` column as a Series: (`0: "Albania"...`).
- **`df[["country", "population"]]`**
  DataFrame containing only these two columns.
- **`df[0:1]`**
  DataFrame containing only the first row.
- **`df[df["population"]>100000000]`**
  DataFrame of countries with more than 100 million inhabitants.
- **`df.loc[0]`**
  First row of the DataFrame as a Series.
- **`df.loc[0, "country"]`**
  String value in the `"country"` column in the first row.
- **`df.loc[df["population"]>100000000, "country"]`**
  Values from the `"country"` column of large countries as a Series.

Strictly speaking, `df.loc[0]` doesn't actually return the first row (that would be done with `df.iloc[0]`; the `i` stands for integer-based), but rather the row with index value 0. However, since we generate the indices automatically in our examples—they always start at 0 and are numbered consecutively—`df.loc[0]` and `df.iloc[0]` are the same in our case.

If you want to modify a DataFrame, for example, rename columns, there are corresponding methods that return a new DataFrame. We can then store this back into the original variable to overwrite the DataFrame:

```
df = df.rename(columns={"population": "population count"})
```

In later examples, you'll see that it's common to chain multiple methods together, for example, `df.rename(…).sort_values(…).drop(…)`. This is possible because each operation returns a new DataFrame that you can continue working with. In most cases, this DataFrame is then a modified copy of the original. So, if you subsequently make changes to the original DataFrame, this has no effect on the new DataFrame, where columns have been renamed and removed and values have been sorted.

A typical feature of DataFrames is that you rarely need to iterate over the entries using a loop. For almost everything, there are commands that allow you to perform tasks on the entire DataFrame in one go.

I have a pandas DataFrame with the columns `country`*,* `population`*,* `continent`*, and* `area`*. The area is given in km². Add two new columns:*

- `population_per_km2`
- `category` *("small" for fewer than 10 million inhabitants, "large" for 100 million or more inhabitants)*

You can easily calculate and add new columns to your DataFrame using pandas. Example:

```
import pandas as pd

# Example DataFrame
df = pd.DataFrame({
    "country": ["Germany", "USA", "China", "Switzerland"],
    "population": [84_000_000, 331_000_000, 1_400_000_000, 8_700_000],
    "continent": ["Europe", "North America", "Asia", "Europe"],
    "area": [357_000, 9_834_000, 9_597_000, 41_000]  # in km²
})
# New column: Population per km²
df["population_per_km2"] = df["population"] / df["area"]

# New column: Category by population
def category(population):
    if population < 10_000_000:
        return "small"
    elif population >= 100_000_000:
        return "large"
    else:
        return "medium"

df["category"] = df["population"].apply(category)

print(df)
```

The first thing you'll notice is that, in this program, the DataFrame was created column by column: first all country names, then all population figures, and so on. In our previous example, we did this row by row. Both approaches are possible. Additionally, we now have different countries because we didn't provide any data in the ChatGPT prompt. You may also have noticed the number written as `10_000_000`. In Python, you can freely use underscores in numbers to improve readability. They are simply ignored internally and have no real meaning.

We can easily add a new column by assigning a value to this nonexistent column: `df["population_per_km2"] = df["population"] / df["area"]`. So, we're dividing one entire series by another series. The result is a new series in which each value has been divided by the other value. This allows pandas to instantly calculate the population per square kilometer and store these numbers in the new column.

With the other column, the `category`, it's a bit more complicated. Since the value of this column can't be calculated simply with a mathematical formula but is determined using `if`,

`elif`, and `else`, we first create a function called `category` that handles this task. It takes a population number, for example, `84,000,000`, and returns the string `"medium"` in that case. pandas allows you to apply a function to all values in a series using the `apply` method. You pass the function to it—in our case, `apply(category)`—and the resulting Series forms our new `category` column. This column stores the corresponding category for each country: `"small"`, `"medium"`, or `"large"`.

5.3 Loading Data from Files into Pandas DataFrames

Pandas can load data from various file types and directly create a DataFrame from them: CSV, Excel, SQL, JSON, HTML, Pickle, Parquet, Feather, Optimized Row Columnar (ORC), Stata, Statistical Analysis System (SAS), Statistical Package for the Social Sciences (SPSS), OpenDocument Spreadsheet (ODS), and more. Here, we'll look at how importing from CSV and Excel files works; for everything else, you can simply look it up online or ask a chat AI. The principle is always the same, though: You call a function like `pd.read_xxx` (e.g., `pd.read_csv`) and pass the file path along with any additional options. For a CSV file, these options include, for example, specifying the column separator (comma, semicolon, tab, etc.) or whether the file has a header row with column labels at the top. You'll then receive a DataFrame containing all the data from your file. The column names and data types are usually taken from the file as well.

To save a DataFrame back to a file, pandas offers methods that start with `to_`. For example, if you call `df.to_excel("table.xlsx")`, pandas creates an Excel file and writes the data from the DataFrame into it.

The file climatedata.tsv contains the following data:

```
   Date       Location     Temperature_F     Precipitation_in
2026-01-01    Chicago      36.5              0.0
2026-01-01    Seattle      35.2              0.2
2026-01-02    Chicago      33.8              0.11
```

Provide me with general statistics using pandas.

All right! I'll show you how to generate general statistics from your *climatedata.tsv* file using pandas.

Here's some sample code:

```
import pandas as pd

# Read the TSV file
df = pd.read_csv("climatedata.tsv", sep="\t")
```

```
# General information about the DataFrame
print("General info:")
print(df.info())
print("\n")
# Basic statistics for numeric columns
print("Descriptive statistics:")
print(df.describe())
print("\n")
```

By the way, TSV stands for *tab-separated values*. This is a common term for CSV files in which the values are separated by tab characters. You pass the column separator to the `pd.read_csv` function via the `sep` parameter, for example, `sep=";"` or, as shown here, `sep="\t"`, which stands for *tab*. If you omit the `sep` parameter, pandas assumes that the values in the file are separated by commas.

Additionally, pandas assumes that the first row of a CSV or TSV file contains the column names. The example rows provided in the prompt demonstrate this as well. The first row doesn't contain any data yet, but rather `Date`, `Location`, `Temperature_F`, and `Precipitation_in`. Pandas automatically sets the column names for the DataFrame to these exact labels and then begins reading the actual data starting from the second row. If our file didn't have a header row, we would have to inform pandas of this and set the column names manually:

```
df = pd.read_csv("climatedata.tsv", sep="\t", header=None)
df.columns = ["Date", "Location", "Temperature_F", "Precipitation_in"]
```

In our program, `df.info()` and `df.describe()` were called. The former provides us with metadata about the DataFrame (column names, data types, etc.) and some general information, such as the number of entries. `df.describe()` calculates the mean, minimum, and maximum values, the standard deviation, and more for each numerical column.

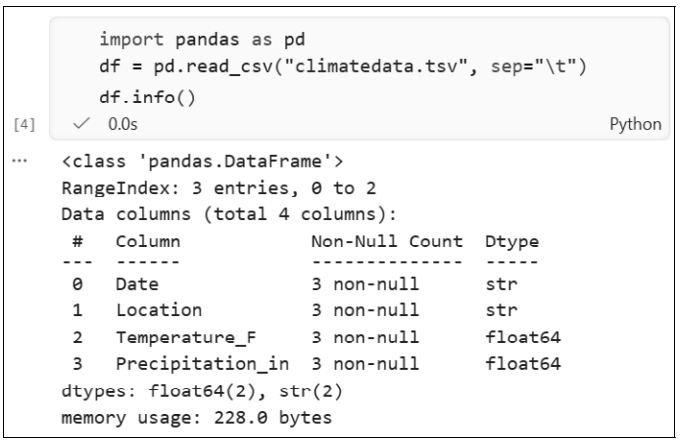
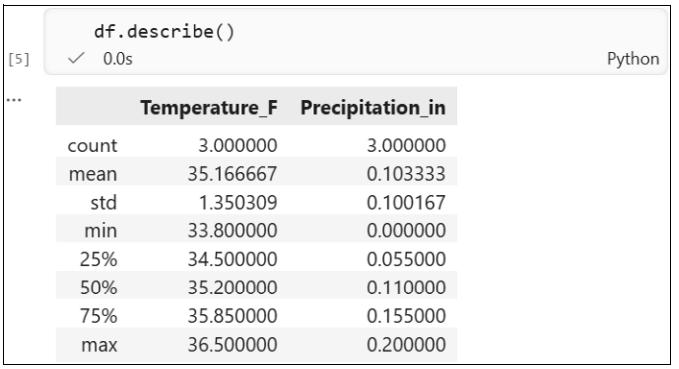


Figure 5.2: General Information About the DataFrame in the Jupyter Notebook Using "df.info()"



```
df.describe()
```

[5] ✓ 0.0s Python

	Temperature_F	Precipitation_in
count	3.000000	3.000000
mean	35.166667	0.103333
std	1.350309	0.100167
min	33.800000	0.000000
25%	34.500000	0.055000
50%	35.200000	0.110000
75%	35.850000	0.155000
max	36.500000	0.200000

Figure 5.3: Statistics About the Columns in the DataFrame in a Jupyter Notebook Using "df.describe()"

The message `"3 non-null"` or `"count 3.000000"` indicates that there are three values in each of the temperature and precipitation columns in the DataFrame. It's also possible that the temperature is missing for some days or cities because the corresponding cells were left blank in the file or set to `"None"` or `"NaN"` in the DataFrame. The rows **25%**, **50%**, and **75%** indicate the quartiles of the values. The fact that the 25% quartile in our example shows **34.5** in the temperature column means that 25% of the rows contain temperatures below 34.5 °F. The 50% quartile is also called the *median*. Half of the measured temperatures are below 35.2 °F, and the other half are above.

While `df.info()` and `df.describe()` are designed to provide a quick overview of the DataFrame, there are also special methods that calculate the statistics shown, as well as much more:

```
average_temperature = df["Temperature_F"].mean()
```

In our case, the data was stored in a CSV file (or TSV file). If we had an Excel file instead, pandas also offers a function for that:

```
df = pd.read_excel("climatedata.xlsx")
```

Here, we could use the `sheet_name` option to specify that a sheet other than the first one should be opened: `sheet_name=1` would be the second sheet (since we start counting from 0), and with `sheet_name="Climate 2026"`, we can specify it by the sheet name. The `usecols` option is also very handy for selecting columns. With this option, you can specify when reading the Excel table that you only need certain columns. The irrelevant columns are then simply skipped and don't end up in the DataFrame:

```
df = pd.read_excel("climatedata.xlsx",
                   usecols=["Date", "Location", "Temperature_F"])
```

If we now modify the DataFrame, we can finally write it back to a file. To do this, we use the appropriate `to_` method depending on the desired file type:

```
df.to_csv("climatedata.csv")
df.to_csv("climatedata.tsv", sep="\t")
df.to_excel("climatedata.xlsx")
df.to_excel("climatedata.xlsx", index=False)
```

If we omit the `index=False` option, there will be a column on the far left of the file containing the row index, that is, `0`, `1`, `2`, `3`, and so on. This is part of the DataFrame and is therefore exported by default. If you don't need this column, simply set `index=False`.

5.4 Data Cleaning with Pandas

After we've loaded the data from a file into a pandas DataFrame, the next step is to perform data cleaning. This involves filtering, preparing, and transforming the data so that it can be easily analyzed, visualized, and processed further. While this is sometimes time-consuming because it usually requires many steps, pandas offers practical tools for everything, which we'll now take a look at.

5.4.1 Filtering Data

I have a pandas DataFrame containing `Date`*,* `Location`*,* `Temperature_F`*, and* `Precipitation_in`*. I only need the data from Seattle.*

> You can easily filter the DataFrame for Seattle. Assuming your DataFrame is named `df` and the column containing the location is named `Location`, do the following:
>
> ```
> df_seattle = df[df["Location"] == "Seattle"]
> ```

Filtering is very easy with pandas, as you can use the syntax with a filter predicate in square brackets. This even allows you to combine multiple conditions using `&` (and) and `|` (or). It's important with such complex predicates that they are always enclosed in parentheses:

```
df = df.loc[(df["Location"] == "Seattle") & ((df["Temperature_F"] < 50)
                                           | (df["Temperature_F"] > 80)) ]
```

Earlier, we assigned the DataFrame to a new variable `df_seattle`; in this example, we've simply overwritten the original DataFrame `df`.

Since we now only have climate data from Seattle after filtering, we no longer actually need the `Location` column. We could now delete it using `df.drop(columns="Location")`; alternatively, we can also specify which columns we want to keep directly during filtering:

```
df = df.loc[df["Location"] == "Seattle",
            ["Date", "Temperature_F", "Precipitation_in"]]
```

5.4.2 Identifying and Handling Missing Values

When we looked at general information and statistics about the DataFrame earlier using `df.info()` and `df.describe()`, we could already see how many records actually had a value in a given column.

For the following examples, let's simply set the temperature in the first row to "unknown" (`None`):

```
df.loc[0, "Temperature_F"] = None
```

If we examine the DataFrame using `df` or `print(df)`, we now see the value `NaN` (not a number) in the first row of the `Temperature_F` column. Such missing data occurs quite frequently in practice. Therefore, the task now is to identify it and then consider how to handle it.

The `isna()` method can be applied to a Series (i.e., a single column) or to an entire DataFrame. It returns a Series or a DataFrame where `True` appears wherever a value was previously missing; the other entries now show `False`. Using the `sum` method, we then count the number of `True` values, that is, the missing values:

```
print(df.isna().sum())             # Number of missing values per column
print(df["Temperature_F"].isna().sum())  # … or just in this column
```

One possible solution for handling missing values is to simply delete the entire rows containing gaps:

```
df = df.dropna()
```

After that, every row where a value is missing in any column is gone. However, with `df.dropna(subset=["Temperature_F"])`, you can also specify that only certain columns should be checked.

Instead of deleting the rows, we can also simply fill in the missing values:

```
df = df.fillna({
    "Temperature_F": df["Temperature_F"].mean(),
    "Precipitation_in": 0,
    "Date": "2026-01-01"
})
```

Wherever the temperature is missing in our sample dataset, it's set to the average value of the entire column. If a precipitation value is missing anywhere, we set it to `0`. For missing values in the date column, we simply choose January 1, 2026. Whether this makes sense or not is open to debate. In any case, we now have complete data, both after deleting the corresponding rows and after filling in the missing values.

5.4.3 Removing Duplicates

Duplicates are duplicate entries in a DataFrame that describe the same real-world object or the same observation. For our climate data, for example, this means that we want only one row for each day in each city. The first step is again to identify duplicates in the first place, and then we need to decide what to do with them:

```
duplicates = df.duplicated(["Date", "Location"])
print("Number of duplicates:", duplicates.sum())
```

The `duplicated` method checks whether each row is a duplicate or not. With `duplicates.sum()`, we then get the number of such rows. If this returns `0`, it means there are no duplicates. The combination of date and location is therefore unique in every row.

For the experiments that follow, we'll add a new row to the DataFrame with the date January 2, 2026, and the location Chicago. Now we have two rows with this exact date-location combination. To add a row, we first create a new DataFrame consisting of this single new row and append it to our original DataFrame using the `pd.concat` function:

```
df = pd.concat([df, pd.DataFrame({"Date": ["2026-01-02"], "Location":
 ["Chicago"], "Temperature_F": [32.9], "Precipitation_in": [0.5]})],
 ignore_index=True)
```

If we now run `df.duplicated(["Date", "Location"])`, our new row will be identified as a duplicate. The `sum()` method returns the number of duplicates: `1`. The following command now removes those duplicates:

```
df = df.drop_duplicates(["Date", "Location"])
```

The command doesn't delete all rows where the date and location match, but instead keeps the first occurrence and removes the other duplicates. After this deduplication, each date-location combination appears only once—exactly as it should!

Sometimes, you want a bit more control over which duplicate rows to keep and which to delete. In our case, we always keep the first one. This is because `drop_duplicates` has a `keep` option with the default behavior `keep="first"`. You can also set this to `keep="last"` or `keep=False`. Then, the last duplicate row—or none at all—will be kept.

Let's sort the DataFrame before removing the duplicates. This way, we can apply specific rules and more precisely determine which rows to keep, for example, always the row with the highest temperature:

```
df = (df.sort_values("Temperature_F")
        .drop_duplicates(["Date", "Location"], keep="last"))
```

If we want even more control over the deduplication process, we can also perform it manually. Of course, this only makes sense if there are a small number of duplicates. First, we flag

these duplicates by adding a new column to the DataFrame that contains the value `True` for all duplicates:

```
df["is_duplicate"] = df.duplicated(["Date", "Location"], keep=False)
df[df["is_duplicate"]]
```

Here, `keep=False` ensures that every row in which the combination of date and location appears multiple times in the DataFrame is marked with `True`, including the first occurrence. The second command, `df[df["is_duplicate"]]`, outputs the rows that have been marked as duplicates, as shown in Figure 5.4.

```
df["is_duplicate"] = df.duplicated(["Date", "Location"], keep=False)
df[df["is_duplicate"]]
```

✓ 0.0s Python

	Date	Location	Temperature_F	Precipitation_in	is_duplicate
2	2026-01-02	Chicago	33.8	0.11	True
3	2026-01-02	Chicago	32.9	0.50	True

Figure 5.4: Mark and Output Duplicates

Since the original index is also visible in this output (duplicates are present in rows 2 and 3, for example), we could now manually make changes to these rows and delete them:

```
df.loc[2, "Temperature_F"] = 32.9
df = df.drop(index=3)
```

The `combine_first` method is also useful, as it combines two rows in a DataFrame into one. If we apply this method to a row from our DataFrame `df.loc[2]` and pass it a second row, the result is similar to the first row, but the values from the other row are used in places where `NaN` appears. For example, if we had one missing temperature value and one missing precipitation value, `combine_first` would combine these two rows into one, where the precipitation value from one row and the temperature value from the other row were used:

```
df.loc[2] = df.loc[2].combine_first(df.loc[3])
df = df.drop(index=3)
```

Here, too, we delete the row with index 3 after the combination process, since it's the duplicate and is no longer needed.

You've now seen how to easily delete any of these rows when dealing with duplicates, how to merge them into a single row when dealing with missing values, and how to keep the rows with the largest or smallest value in a specific column by sorting and deduplicating. Of course, we can now come up with even more complex rules, such as combining where we have `NaN` values, taking the temperature value from the row where it's highest, taking the lowest precipitation value, and, on Friday the 13th in leap years, always taking the precipitation value from the warmest location. There are, of course, no limits to your imagination

here, but if you really need something like that, it's best to ask a chat AI. Here is just a quick final example (more on the `groupby` method will follow later when we cover data analysis):

```
df = df.groupby(["Date", "Location"], as_index=False).agg({
    "Temperature_F": "mean",
    "Precipitation_in": "sum"
})
```

The rows in our DataFrame that share the same date and location are combined into a single row. The average is calculated from the individual temperature values, and the precipitation values are summed.

5.4.4 Identifying and Correcting Errors and Outliers

We've already looked at two types of errors in the data: missing values and duplicates. However, there are many other types of errors hiding in the data. Before we can begin troubleshooting and correcting errors, though, we first need to know what kinds of errors might even occur in our data. Unfortunately, there's no magic command that will give us the erroneous rows if it isn't clear in the first place what "erroneous" actually means. For an email address, it would be an error if an @ symbol is missing; temperature values should always fall within a certain range; and the date a person got married should always be later than their date of birth. Only after we've defined such error cases, we can specifically search for rows containing these errors and then look at how to fix them.

For the examples that follow, we'll first create a DataFrame that contains a few errors:

```
df = pd.DataFrame({
    'Date': ['2026-01-01', '2026-01-01', '2026-02-30'],
    'Location': ['Chicago', 'Seattle ', 'chicago'],
    'Temperature_F': [36.5, 35.2, 525.47],
    'Precipitation_in': [0.0, 0.2, 0.1]
})
```

I have a DataFrame `df` *with, among other things, a* `'Location'` *column. Find the rows where the location is written in lowercase and correct the error by capitalizing the first letter, for example, "chicago" → "Chicago".*

You can do this quite easily in pandas. Here's an example of how to find the rows where the location is in lowercase and correct them:

```
import pandas as pd

df = pd.DataFrame({
    'Date': ['2026-01-01', '2026-01-01', '2026-02-30'],
    'Location': ['Chicago', 'Seattle ', 'chicago'],
```

```
    'Temperature_F': [36.5, 35.2, 525.47],
    'Precipitation_in': [0.0, 0.2, 0.1]
})

# Find rows where the location is written in lowercase
mask = df['Location'].str[0].str.islower()
print("Rows with the location in lowercase:")
print(df[mask])

# Correct the lowercase locations (capitalize the first character)
df.loc[mask, 'Location'] = df.loc[mask, 'Location'].str.capitalize()

print("\nDataFrame after correction:")
print(df)
```

The filter predicate, which we always wrote directly in square brackets in previous examples, is stored here in a variable `mask`. We could also have written:

```
df[df['Location'].str[0].str.islower()]
```

This finds rows where the city is in lowercase. `df['City']` is the entire city column. `str` gives us access to string methods; in our case, `[0]` gives us the first character, and `islower()` allows us to check if it's in lowercase. The `capitalize` method converts a string so that the first letter is now uppercase; `"chicago"` becomes `"Chicago"`.

Did you actually notice that there's a space after "Seattle" in our data ("`Seattle `")? Unfortunately, this is something that's very often overlooked in practice. Especially when data is loaded from TSV, Excel, or similar files. You then wonder in later steps why a duplicate wasn't detected and why there are still two rows with the location "Seattle" on the same day. After a long troubleshooting session, you eventually notice that one instance is the city `"Seattle"` and the other is `"Seattle "` with an invisible space. The best way to fix this is right at the start using the `strip()` method. `strip()` removes spaces at the beginning and end of a string:

```
df["Location"] = df["Location"].str.strip()
```

Now let's address outliers in numerical columns. We want temperatures to always fall within a specific range and precipitation values to never be negative. If we do find such values, we'll replace them with `NaN` so they don't skew our calculations in later data analyses:

```
df.loc[(df["Temperature_F"] < -60) | (df["Temperature_F"] > 140),
           "Temperature_F"] = np.nan
df.loc[df["Precipitation_in"] < 0, "Precipitation_in"] = np.nan
```

The value `np.nan` comes from the NumPy package. So, at the beginning of the Python program or at the top of the Jupyter Notebook, we must first import it using `import numpy as np`.

One final error that has crept into our sample dataset is the invalid date value `"2026-02-30"`. Dates and times shouldn't be stored as strings in a DataFrame anyway; instead, it's better to use the `datetime` data type. This also allows us to perform calculations on date values, for example, to add a week to a date or to check whether a date falls within a certain range. And incidentally, when converting strings containing a date, we can also see whether that date is invalid or not:

```
df["Date"] = pd.to_datetime(df["Date"], errors="coerce")
```

After this, the values in the date column no longer have the string data type, but rather the `datetime` data type. The option `errors="coerce"` ensures that invalid date values are converted to *NaT* (not a timestamp). Without this option, an error would be raised if there were invalid date values in the DataFrame, and the program would terminate.

5.4.5 Transforming Data into the Desired Format

So far, we've focused on removing errors from the data. From now on, we can say that we have error-free data. However, it's still not in the format we'd like for performing adequate and convenient data analysis. For example, a temperature in degrees Celsius isn't an error, but we'd much rather convert it to Fahrenheit:

```
df = pd.DataFrame({
    'Date': ['2026-01-01', '2026-01-01', '2026-01-02'],
    'Location': ['Chicago', 'Seattle', 'Chicago'],
    'Temperature_C': [2.5, 1.8, 1],
    'Precipitation_in': [0.0, 0.2, 0.11]
})

df["Temperature_F"] = df["Temperature_C"] * 9/5 + 32
```

After this transformation, our DataFrame contains both the old `Temperature_C` column with temperature values in degrees Celsius and the calculated Fahrenheit values in the `Temperature_F` column. We can keep the former column or remove it from the DataFrame using `df = df.drop(columns=["Temperature_C"])`.

Not only can calculations using `+`, `-`, `*`, and so on, be performed on entire columns, but mathematical functions can also be applied, for example, to round numbers. Using the `round` method, we can round the values in the column `Precipitation_in` to one decimal place:

```
df["Precipitation_in"] = df["Precipitation_in"].round(1)
```

This converts `0.11` to `0.1`. Such data type–specific methods are available for numbers (e.g., `round`) and strings (e.g., `strip()`, which removes leading and trailing spaces), but also for columns containing dates or timestamps. The following command calculates the corresponding day of the week from the date and writes it to a new column:

```
df["Day of Week"] = pd.to_datetime(df["Date"]).dt.day_name()
```

More complex transformations can be performed by creating a function in which we define exactly how values should be converted. We already did this earlier in the Section 5.2 to assign countries to the categories `"small"`, `"medium"`, or `"large"` based on their population. For simpler cases, however, it's sufficient to simply create a dictionary that specifies which values should be mapped to which new values:

```
state_map = {
    "Chicago": "IL",
    "Seattle": "WA",
    "New York": "NY",
    "Los Angeles": "CA",
    "Houston": "TX"
}
df["State"] = df["Location"].map(state_map)
```

If a place name appeared in our data that isn't present in the `state_map`, these rows would show a `NaN` in the `State` column.

The transformations shown are intended only as examples and are meant to serve as inspiration. Of course, many other transformations are possible, including significantly more complex ones. Chat-based AI tools are a great help here. It's best to include the column names in the prompt—or even the entire output of `df.info()`—and describe exactly what should be transformed and how. Examples in the prompt are always helpful here as well, for example, "For Houston, the state should be TX." Then the AI knows what typical values in the data look like and that states should be abbreviated.

5.4.6 (De)coding Categories

Decoding means that we have a kind of code or abbreviation in the data and want to convert it into what this abbreviation stands for. In the following DataFrame, the location `0` stands for Chicago and `1` for Seattle:

```
df = pd.DataFrame({
    'Date': ['2026-01-01', '2026-01-01', '2026-01-02'],
    'Location': [0, 1, 0],
    'Temperature_F': [36.5, 35.2, 33.8],
    'Precipitation_in': [0.0, 0.2, 0.11]
})
```

If we now want to have the full place names in the DataFrame, we can do this using the `map` method by passing it a dictionary. This dictionary specifies which abbreviation corresponds to which location:

```
location_map = {0: "Chicago", 1: "Seattle"}
df["Location"] = df["Location"].map(location_map)
```

The reverse process is called *encoding*. So, we have place names in the `"Location"` column but want to generate numbers instead—for example, `0` for Chicago and `1` for Seattle. In this case, we don't need to pass a map; instead, we can have the codes generated automatically. The first value that appears in the DataFrame is then assigned the code `0`. In our case, that would be Chicago, as before. All subsequent occurrences of Chicago are then assigned a `0` as well.

```
df["Location"] = df["Location"].astype("category").cat.codes
```

Now, of course, you might ask why you should do this at all. The goal of data cleaning and data transformations is, after all, to convert data into a format that allows you to work with it optimally, and there are some use cases where it's easier to work with category numbers or numerical codes than with full-text values. This is particularly useful with machine learning algorithms. These algorithms often don't care whether someone lives in Chicago or in Location `0`. Many algorithms even require that the data be in numerical form. So, you have to perform such encoding beforehand, and when the algorithm later delivers a result, it may be necessary to decode it again. For example, if an AI algorithm predicts that a customer is highly likely to live in Location `0`, you must convert this value back to the location name "Chicago" to make sense of the statement. Additionally, numbers save storage space compared to strings. This allows for more complex analyses to be performed with less storage requirements and thus potentially faster. Another use case is data anonymization. Once the data is encoded, it's no longer possible to convert the numerical codes back to their original values.

5.4.7 Normalizing: Making Values Comparable

To avoid comparing apples to oranges, we must transform values so that they can be compared with one another at all, thereby enabling meaningful calculations. For example, if we merge two DataFrames from different sources and one uses temperatures in °C while the other uses °F, we must decide on a uniform format before merging them. We thus overcome the heterogeneity present in our data and make the data homogeneous.

Normalization is similar. We bring the data into a normal form—that is, a form that's standardized in a certain way—so that it becomes comparable without changing the actual meaning of the values. A well-known example is min-max normalization. Behind this is an algorithm that identifies the smallest and largest values in a column, and then replaces every instance of that smallest value with a `0` and the largest value with a `1`. All values in between are assigned a value between 0 and 1 accordingly. The formula behind it might look a bit strange, but ultimately it does exactly that:

```
df["Temp_minmax"] = (df["Temperature_F"] - df["Temperature_F"].min()) /
(df["Temperature_F"].max() - df["Temperature_F"].min())
```

If the lowest temperature in our dataset were 31 °F and the highest 105 °F, 31 °F would become `0`, 68 °F would become `0.5` (because (68 – 31) / (105 – 31) = 0.5), and 105 °F would become `1`.

Such normalizations are also frequently performed when AI algorithms are used. Many machine learning algorithms require that the data be normalized beforehand. Furthermore, values between 0 and 1 are often easier to understand. For example, is 103 °F a lot? That depends on the context. Is 0.98 a lot (on a scale from 0 to 1)? Yes, definitely! Outliers—both below and above the mean—are easier to spot in normalized values, and when creating charts and visualizations, normalized values provide a useful and consistent scale right out of the box.

In addition to min-max normalization, there are other normalization methods, such as the Z-score, the L2 norm, log normalization, calculating the proportion of the total sum, and many more. Each has its pros and cons. With log normalization, for example, outliers stand out more. But I don't need to go into detail here—if you want to do something like this, you probably already know what you're doing.

5.5 Calculations and Analyses with Pandas

This section focuses on analyzing the data we've loaded into a DataFrame and cleaned up to extract new insights from it. We'll start with simple calculations of statistical measures across the entire dataset and then explore how to group the data and perform calculations on individual groups.

5.5.1 Row-by-Row Calculations

You can perform many data analyses using the methods you learned in the previous subsections:

> *I have a pandas DataFrame with the columns* `Date`*,* `Location`*,* `Temperature_F`*, and* `Precipitation_in`*. I want to calculate a score for each row. The higher the temperature and the less precipitation, the higher the score should be. A higher score therefore indicates better weather.*

You can calculate this directly using pandas and normalization, for example:

```
import pandas as pd
df = pd.DataFrame({
    'Date': ['2026-01-01', '2026-01-01', '2026-01-02'],
    'Location': ['Chicago', 'Seattle', 'Chicago'],
```

```
    'Temperature_F': [36.5, 35.2, 33.8],
    'Precipitation_in': [0.0, 0.2, 0.11]
})

# Normalize between 0 and 1
df['Temp_norm'] = ((df['Temperature_F'] - df['Temperature_F'].min()) /
                   (df['Temperature_F'].max() -
                    df['Temperature_F'].min()))
df['Rain_norm'] = 1 - ((df['Precipitation_in'] -
                        df['Precipitation_in'].min()) /
                       (df['Precipitation_in'].max() -
                        df['Precipitation_in'].min()))

# Calculate score (simple average)
df['Score'] = (df['Temp_norm'] + df['Rain_norm']) / 2

print(df[['Date','Location','Score']])
```

Here, both the temperature values and the precipitation values are normalized to a range between 0 and 1. For precipitation, the formula starts with `1 -` so that the lowest precipitation value receives a value of `1`. After all, less precipitation is supposed to be better, and the higher the value, the better the weather. The final score, which incorporates both temperature and precipitation, is simply calculated here as the average of the two values. We could also use a weighted average instead, for example, if we want temperature to have a greater influence on the score:

```
df['Score'] = df['Temp_norm'] * 0.7 + df['Rain_norm'] * 0.3
```

Since the sum of the weights is 0.7 + 0.3 = 1, the score again falls within the range of 0 to 1.

5.5.2 Differences from the Previous Value

The `diff()` method calculates the difference between the values of consecutive rows within a column. For example, if we write the difference between temperature values to a new column, we can use this to determine whether the temperature has risen or fallen compared to the previous day. If this difference is positive, it has risen. If it's negative, the temperature was higher the previous day. To obtain meaningful values, we first filter the data so that we only look at the temperature values for the city of Chicago, and sort them in ascending order by date:

```
df_chicago = df[df['Location'] == 'Chicago'].sort_values('Date')
df_chicago['Temp_diff'] = df_chicago['Temperature_F'].diff()
```

In the first row, the new **temp_diff** column contains the value **NaN** because there's no comparison value yet.

```
df_chicago = df[df['Location'] == 'Chicago'].sort_values('Date')
df_chicago['Temp_diff'] = df_chicago['Temperature_F'].diff()
df_chicago
```

✓ 0.0s Python

	Date	Location	Temperature_F	Precipitation_in	Temp_diff
0	2026-01-01	Chicago	36.5	0.00	NaN
2	2026-01-02	Chicago	33.8	0.11	-2.7

Figure 5.5: Calculating the Difference from the Value in the Previous Row

5.5.3 Aggregating Values

Aggregation means calculating a single value from many values. An example is calculating the average or the sum from all the values in a column as follows:

```
average_temperature = df['Temperature_F'].mean()
total_precipitation = df['Precipitation_in'].sum()
```

The `mean()` and `sum()` calculations are performed on the entire column. The result is a single number in each case, for example, the average temperature `54.8` and the total precipitation `369.27`. The latter number is probably fairly meaningless because it simply sums up many values from many days and locations. However, if we had, for example, filtered for a specific city beforehand and narrowed down the time period, the total precipitation would make sense again:

```
df_jan2026_chicago = df[(df['Location'] == 'Chicago') &
                        (df['Date'].dt.year == 2026) &
                        (df['Date'].dt.month == 1)]
total_precipitation = df_jan2026_chicago['Precipitation_in'].sum()
```

To extract the year and month from the date column, the date must first be converted from string values to values of the `datetime` data type using `pd.to_datetime`, as described in Section 5.4.

5.5.4 Cumulative Sums

Instead of calculating the total sum of all precipitation values all at once, we can also perform this calculation step-by-step for each individual row. The result is called a *cumulative sum*, and it can be interpreted as the total precipitation up to the day in question. For the first row, the cumulative sum is simply the precipitation value for that day itself; for the second row, it's the value of the first row plus the value in the second row. For the third row, its

value is added on top, and so on. In pandas, the `cumsum` method is used for this purpose. It creates a new column containing these cumulative sums:

```
df_chicago['Precipitation_cumsum'] = df_chicago['Precipitation_in'].cumsum()
```

```
df_chicago = df[df['Location'] == "Chicago"].sort_values('Date')
df_chicago['Precipitation_cumsum'] = df_chicago['Precipitation_in'].cumsum()
df_chicago
```

✓ 0.0s Python

	Date	Location	Temperature_F	Precipitation_in	Precipitation_cumsum
0	2026-01-01	Chicago	36.5	0.00	0.00
2	2026-01-02	Chicago	33.8	0.11	0.11

Figure 5.6: Cumulative Sum: Sum of All Values in the "Precipitation_in" Column Up to and Including the Respective Row

5.5.5 Grouping Data

I have a pandas DataFrame with the columns `Date`*,* `Location`*,* `Temperature_F`*, and* `Precipitation_in`*. Calculate the average temperature for each individual location.*

Perfect! To calculate the average temperature per location in a pandas DataFrame, you can use the `groupby` function together with `mean()`. Here's an example:

```
import pandas as pd

df = pd.DataFrame({
    'Date': ['2026-01-01', '2026-01-01', '2026-01-02'],
    'Location': ['Chicago', 'Seattle', 'Chicago'],
    'Temperature_F': [36.5, 35.2, 33.8],
    'Precipitation_in': [0.0, 0.2, 0.11]
})

# Calculate average temperature per location
avg_temp = df.groupby("Location")["Temperature_F"].mean()

print(avg_temp)
```

The `groupby` method doesn't return a DataFrame, but rather a Series where the index consists of the values from the column used for grouping (i.e., city names), and the corresponding values are the average temperatures in those cities, as shown in Figure 5.7.

```
# Calculate the average temperature by location
avg_temp = df.groupby("Location")["Temperature_F"].mean()
print(avg_temp)
```

✓ 0.0s Python

```
Location
Chicago    35.15
Seattle    35.20
Name: Temperature_F, dtype: float64
```

Figure 5.7: With "groupby": The DataFrame Divided into Groups and Calculations Performed for Each Individual Group

How do I convert this series back into a DataFrame with the two columns `"Location"` *and* `"Average Temperature"`*?*

This is very easy to do with `reset_index()` and, if necessary, a `rename`:

```
# Convert to a DataFrame
df_average = avg_temp.reset_index()
# Rename column (optional)
df_average = df_average.rename(columns={"Temperature_F":
                                        "Average_Temperature"})
print(df_average)
```

The `reset_index()` method converts the index back into a normal column named `Location`. The newly created DataFrame thus has two columns: `Location` and `Temperature_F`. We can rename the latter using `rename`.

In our grouping, we called `mean()` to calculate average temperatures. Other aggregate functions include, for example, `sum`, `min`, `max`, or `count`. The latter function counts the non-empty values within the column in the corresponding group.

It's also possible to calculate multiple aggregate functions at once, for example, the highest and lowest temperatures per city. You can also group by multiple columns. But we won't be doing any of that now; search for it online or ask a chatbot if you need to.

5.5.6 Sorting and Ranking Data

You can easily sort the data in a DataFrame using the `sort_values` method:

```
df = df.sort_values('Temperature_F')
```

By default, the data is sorted in ascending order. The first row in the DataFrame is now the one with the lowest temperature. Using the option `ascending=False`, we can sort the dataset in descending order instead. If we pass the column names not as a string but as a list of strings, we can also sort by multiple columns:

```
df = df.sort_values(['Date', 'Temperature_F'],
                    ascending=[True, False])
```

First, everything is sorted in ascending order by date. If multiple rows have the same date, the row with the higher temperature appears higher in the list.

Sorting works not only on numeric columns but also on string columns, such as the location name, or on columns containing date or time data.

The `rank()` method calculates a *rank* for each row. A rank of `1` means that the value in the specified column for that row is the smallest of all values. Rank `2` corresponds to the second-smallest value, and so on.

```
df['Temp_Rank'] = df['Temperature_F'].rank()
```

Here, too, the `ascending=False` option can be used to reverse the ranking so that rank `1` corresponds to the largest value. If the ranking is performed after grouping, a separate rank is calculated for each individual group. In this case, a `1` no longer represents the lowest temperature, but rather the lowest temperature in Chicago or the lowest temperature in Seattle:

```
df['Rank_Location'] = df.groupby('Location')['Temperature_F'].rank()
```

However, calling the rank method only ranks the data; it doesn't sort it. Therefore, it usually makes sense to sort by the column used to calculate the rank either before or after applying the rank. Alternatively, you can of course sort by the rank itself, as shown in Figure 5.8.

```
df['Rank_Location'] = df.groupby('Location')['Temperature_F'].rank()
df = df.sort_values(['Location', 'Rank_Location'])
df
```

✓ 0.0s Python

	Date	Location	Temperature_F	Precipitation_in	Rank_Location
2	2026-01-02	Chicago	33.8	0.11	1.0
0	2026-01-01	Chicago	36.5	0.00	2.0
1	2026-01-01	Seattle	35.2	0.20	1.0

Figure 5.8: A Ranking with Sorting: Rank 1 Represents the Day with the Lowest Temperature in the Respective Location

5.6 Merging Data from Multiple Sources

Data is rarely available error-free in exactly the format you need. That's why we do data cleaning. It's also rare for a single file to contain all the data you need. Often, you have to combine, merge, or link the contents of multiple files or from various other data sources. Here are three examples:

- **Example 1**
  We want to analyze weather data spanning several years. We have individual files for

each year. However, the format is the same every time: same columns, same data types, same units.

- **Example 2**
  We want to compare weather data from different cities. The data comes from different sources, attributes have different names, and different units and data types are used for the same type of measurement. Additionally, the files use different formats, for example, one is in Excel, the other in CSV. Nevertheless, both data sources deal with the same thing: that a certain temperature was measured in a specific city on a specific day.
- **Example 3**
  One file contains weather data, but it doesn't include the location where the respective values were measured. Instead, the data contains only IDs of weather stations, for example, `13667`. A second file contains details about each weather station, for example, that the station with ID `13667` is in Freiburg, Germany.

In all of these cases, we ultimately want to have a single pandas DataFrame in which all data from the various sources has been integrated. That's what we'll do now.

The procedure for Example 1 and Example 2 is essentially the same, except that there isn't much to do in Step 2 of the first example:

1. The data is loaded into separate DataFrames (e.g., `df1` and `df2`).
2. We perform data cleaning so that `df1` and `df2` are homogeneous afterward.
3. We merge the DataFrames into one: `df = pd.concat([df1, df2], ignore_index=True)`.

The `pd.concat` function creates a new DataFrame with `len(df1)+len(df2)` rows. It's important that the two DataFrames to be concatenated have the same columns; otherwise, the result will contain all columns from both DataFrames, and many cells will be filled up with NaN values. The option `ignore_index=True` is important when merging so that the row numbering is discarded. Without it, we would have multiple row 0s, and so on. With this option, the rows are renumbered after merging.

We'll address the third example later. In that case, the DataFrames contain completely different types of data. We can't merge them directly but must combine the data in an appropriate way. For now, however, let's stick with the scenario where we're combining data from different sources or from different datasets.

I have 12 files from the German Weather Service in the dwd folder[1]: regional_averages_tm_01.txt, and so on, one file for each month. I need a pandas DataFrame that contains all the data.

1 The files are sourced from the German Weather Service website and are freely available: *https://opendata.dwd.de/climate_environment/CDC/regional_averages_DE/monthly/air_temperature_mean/*.

The first lines of each file look like this:

```
Time series for regional averages for federal states and combina-
tions of federal states, created on: 20250904
Year;Month;Brandenburg/Berlin;Brandenburg;Baden-Wuerttemberg;
Bavaria; Hesse;Mecklenburg-Western Pomerania;Lower Saxony;Lower
Saxony/Hamburg/Bremen;North Rhine-Westphalia;Rhineland-Palatinate;
Schleswig-Holstein; Saarland;Saxony;Saxony-Anhalt;Thuringia/Saxony-
Anhalt;Thuringia; Germany;
1881;01;    -5.54;    -5.56;    -4.89;    -6.51;    -5.68;    -5.07;
  -4.55;    -4.55;    -4.21;    -4.49;    -4.06;    -4.15;    -6.22;
  -5.89;    -6.28;    -6.76;    -5.36;
```

All right! We can load the 12 monthly files into a pandas DataFrame by loading each file, skipping the header rows, and then merging all the DataFrames together.

Here's an example of how you can do that:

```
import pandas as pd
from pathlib import Path

# Folder containing the DWD files
folder = Path("dwd")

# Select all relevant files (sorted by month)
files = sorted(folder.glob("regional_averages_tm_*.txt"))

# Read all files and collect them into a list of DataFrames
dfs = []
for f in files:
    df = pd.read_csv(f, sep=";", skiprows=1, na_values=["-"])
    # Last column is empty due to the trailing semicolon -> delete
    df = df.drop(columns=df.columns[-1])
    dfs.append(df)

# Combine everything into a single DataFrame
df = pd.concat(dfs, ignore_index=True)

# Adjust data types
df["Year"] = df["Year"].astype(int)
df["Month"] = df["Month"].astype(int)

print(df.head())
print(df.shape)
```

In a loop, each of the CSV files is read using `pd.read_csv`. Because we included sample lines in the ChatGPT prompt, the AI correctly sets the column separator to a semicolon, skips the first line because it contains no column names and no data, and converts values consisting solely of the character `"-"` to `NaN`. The latter was a guess, since there was no such `"-"` value in the provided example. However, ChatGPT noticed one thing: There's a semicolon at the end of every line, even though no value follows it. This would result in a completely empty column in the DataFrame. Therefore, we remove them. The individual DataFrames, which are collected in a list, are merged again using `pd.concat`.

```
df
```

[23] ✓ 0.1s Python

...

	year	month	Brandenburg/Berlin	Brandenburg	Baden-Wuerttemberg
0	1881	1	-5.54	-5.56	-4.89
1	1882	1	1.43	1.42	-0.65
2	1883	1	-0.41	-0.42	-0.09
3	1884	1	3.35	3.34	2.34
4	1885	1	-2.15	-2.16	-3.88
...	...	...	...	...	...

Figure 5.9: New DataFrame Containing All the Data from the 12 Files

The DataFrame ultimately produced by our program code has columns for `Year` (e.g., `1881`), `Month` (e.g., `1`), and for each federal state one column containing the temperature for that state. For analysis, however, it would be much better if, first, the year and month weren't in two separate columns but in a single date column (e.g., `"1881-01-01"`), and second, the states weren't individual columns but rather had a separate row for each state. Such a conversion isn't easy, but we can let ChatGPT do the work:

> *The DataFrame should look like this:* `Date`, `State`, `Temperature`.
>
> *Show me only the lines of code I need to add.*

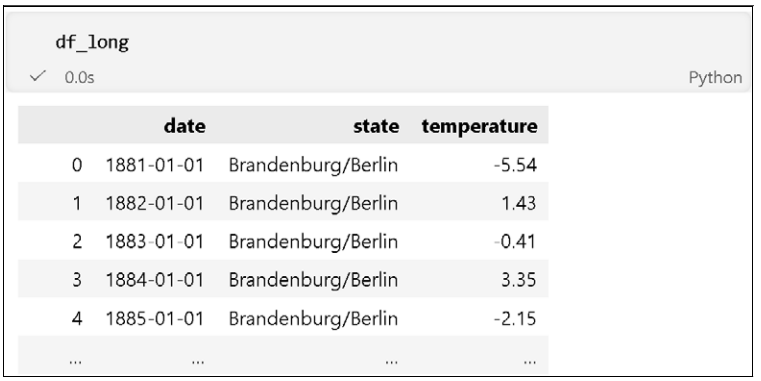
All right. So you want to convert the data to a "long" format. Then you need to reshape (`melt`) the DataFrame and create a date field.

Just add these lines after the code you've written so far:

```
# Wide → Long: one row per date and state
df_long = df.melt(
    id_vars=["Year", "Month"],
    var_name="State",
```

```
    value_name="Temperature"
)

# Create a date column from year + month
df_long["Date"] = pd.to_datetime({"year": df_long["Year"],
                                  "month": df_long["Month"],
                                  "day": 1})

# Keep only the desired columns
df_long = df_long[["Date", "State", "Temperature"]]
```

Okay, once you know how, it's not that hard to transform the whole thing so that the states are now in the rows by using the `melt` method. The columns `id_vars=["Year", "Month"]` are the ones that should remain in the DataFrame. The two columns `var_name="State"` and `value_name="Temperature"` are the ones that should be added. The `State` column will then contain what was the column name in the original DataFrame, for example, `Bavaria`. And the new `Temperature` column will contain the value that was in that very column.


df_long

✓ 0.0s Python

	date	state	temperature
0	1881-01-01	Brandenburg/Berlin	-5.54
1	1882-01-01	Brandenburg/Berlin	1.43
2	1883-01-01	Brandenburg/Berlin	-0.41
3	1884-01-01	Brandenburg/Berlin	3.35
4	1885-01-01	Brandenburg/Berlin	-2.15
...	...	...	...

Figure 5.10: After the "Melt" Operation: States and Temperatures Now in the Two Columns Designated for Them

If we now had additional files with weather data from other sources covering different time periods, federal states, or cities that we wanted to combine with our DataFrame, we would first need to bring them into the same format using appropriate data transformation steps (renaming columns, converting units, etc.). After that, we could combine the separate DataFrames back into a single DataFrame using `pd.concat`.

Now let's look at the case from the third example described earlier. For this, we'll use the following two files, which also come from the German Weather Service[1]:

```
Station_id;Reference_period;Data_source;Jan;Feb;Mar;…
1;          1981-2010;      46;          -.4; .7;  4.7;…
…
```

Listing 5.1: File Temperature_1981-2010.txt

```
Station_id;Station_name;Latitude;Longitude;Elevation;State;
1;          Aach;        47.841299;    8.849299;     478;  Baden-Württemberg;
```

Listing 5.2: File Temperature_1981-2010_StationList.txt

First, both are read into two separate DataFrames. The practical thing about using Jupyter Notebooks is that we can always display what the DataFrame looks like up to that point between the individual steps. So, for example, we load the first DataFrame, display the first few rows using `head()`, and then do the same with the second DataFrame. This allows us to immediately see if everything is working as intended. Additionally, we can perform transformations, such as directly removing unnecessary columns and renaming columns.

```
import pandas as pd
df_temperature = pd.read_csv("dwd/Temperature_1981-2010.txt", sep=";",
                             skipinitialspace=True, encoding="latin1")
df_temperature = df_temperature[["Station_id", 'Jan', 'Feb', 'Mar',
'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']]
df_temperature.head()
```

	Station_id	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
0	1	-0.4	0.7	4.7	8.7	13.3	16.5	18.5	17.9	13.8	9.2	3.7	0.7
1	3	3.0	3.2	6.4	9.5	13.6	16.2	18.4	18.0	14.6	11.0	6.6	3.7
2	44	1.7	1.9	4.9	8.7	12.9	15.3	17.6	17.2	13.6	9.6	5.4	2.3
3	52	1.2	1.5	4.2	8.1	12.5	15.1	17.6	17.1	13.4	9.3	5.0	1.9
4	61	-1.4	-0.3	4.3	8.8	13.9	16.5	18.5	18.0	13.6	8.7	3.4	-0.2

Figure 5.11: Temperature Data Read from the File into the DataFrame "df_temperature"

The data is again in an unfavorable format—with a separate column for each month—so we use `melt` to convert it back into separate rows:

```
df_temperature = df_temperature.melt(id_vars=["Station_id"],
                 var_name="Month", value_name="Temperature")
df_temperature.head()
```

1 The temperature and station data can be found at *https://opendata.dwd.de/climate_environment/CDC/observations_germany/climate/multi_annual/mean_81-10/*.

	Station_id	Month	Temperature
0	1	Jan	-0.4
1	3	Jan	3.0
2	44	Jan	1.7
3	52	Jan	1.2
4	61	Jan	-1.4

Figure 5.12: After the "Melt" Step: Separate Rows for Each Month

We'll perform one final transformation on this temperature DataFrame. Specifically, month names (`Jan.`, etc.) are difficult to handle and hard to sort. Therefore, we'll convert them to month numbers:

```
df_temperature["Month"] = df_temperature["Month"].map({'Jan': 1,
 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6, 'Jul': 7,
 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12})
df_temperature.head()
```

	Station_id	Month	Temperature
0	1	1	-0.4
1	3	1	3.0
2	44	1	1.7
3	52	1	1.2
4	61	1	-1.4

Figure 5.13: Using "map" to Convert the Month Names into Numbers

Now it's time for the station file. This DataFrame requires significantly fewer transformation steps. We simply remove columns that aren't needed, rename one column, and remove spaces:

```
df_stations = pd.read_csv("dwd/Temperature_1981-2010_StationList.txt",
                          sep=";", skipinitialspace=True,
                          encoding="latin1")
df_stations = df_stations[["Station_id", "Station_name",
                           "State"]]
df_stations = df_stations.rename(columns={ "Station_name": "Location"})
df_stations["Location"] = df_stations["Loca-
tion"].astype(str).str.strip()
df_stations.head()
```

	Station_id	Location	State
0	1	Aach	Baden-Württemberg
1	3	Aachen	Nordrhein-Westfalen
2	44	Großenkneten	Niedersachsen
3	52	Ahrensburg-Wulfsdorf	Schleswig-Holstein
4	61	Aiterhofen	Bayern

Figure 5.14: DataFrame "df_stations" Containing Weather Station Data

Now comes the actual merging of the two DataFrames. In pandas, this is done using `pd.merge`. The function takes two DataFrames and the name of the column used to find the matching rows in the other DataFrame; in our case, that is `Station_id` (see Figure 5.15):

```
df = pd.merge(df_temperature, df_stations, on="Station_id", how="left")
df.head()
```

	Station_id	Month	Temperature	Location	State
0	1	1	-0.4	Aach	Baden-Württemberg
1	3	1	3.0	Aachen	Nordrhein-Westfalen
2	44	1	1.7	Großenkneten	Niedersachsen
3	52	1	1.2	Ahrensburg-Wulfsdorf	Schleswig-Holstein
4	61	1	-1.4	Aiterhofen	Bayern

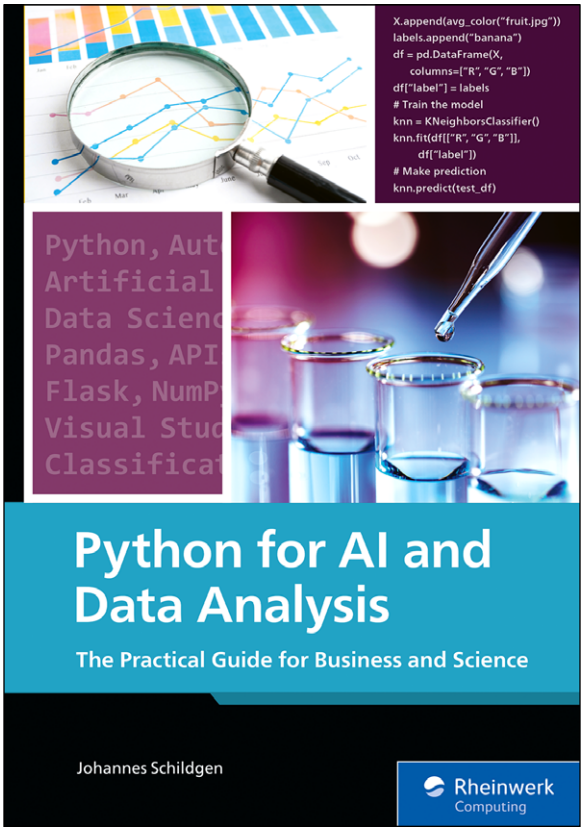
Figure 5.15: Two DataFrames Joined Using "merge"

The easiest way to use `merge` is when there's only a single column used for the join and when this column has the same name in both DataFrames. Then, we can simply write `on="Station_id"`. The option `how="left"` specifies that the left DataFrame (`df_temperature`) is the main table. It checks what is in the `Station_id` column and then searches the right table (`df_stations`) for the row containing that exact value in its `Station_id` column. Finally, the resulting row is assembled from these two rows. If the station with the found `Station_id` didn't exist in the right DataFrame, the `Location` and `State` columns would simply be filled with `NaN`.

So now, in our final DataFrame, each row contains the month, the city name, and the measured temperature. This is perfect for our purposes. We'll use this DataFrame in the next chapter to generate charts using Matplotlib. Before that, we'll expand our DataFrame to include a column with precipitation values, which also come from a file provided by the German Weather Service. The transformation steps are the same as for the temperature data. Finally, we'll save our DataFrame to the file *weather.csv*:

```
# Read file
df_precipitation = pd.read_csv("dwd/Precipitation_1981-2010.txt",
                  sep=";", skipinitialspace=True, encoding="latin1")
```

```
# Remove unnecessary columns
df_precipitation = df_precipitation[["Station_id", 'Jan.', 'Feb.', 'Mar.',
  'Apr.', 'May', 'Jun.', 'Jul.', 'Aug.', 'Sept.', 'Oct.', 'Nov.', 'Dec.']]

# Before: Columns for each month; now columns for month and precipitation
df_precipitation = df_precipitation.melt(id_vars=["Station_id"],
                 var_name="Month", value_name="Precipitation")

# Convert month names to numbers
df_precipitation["Month"] = df_precipitation["Month"].map({'Jan.': 1,
 'Feb.': 2, 'Mar.': 3, 'Apr.': 4, 'May': 5, 'Jun.': 6, 'Jul.': 7,
 'Aug.': 8, 'Sept.': 9, 'Oct.': 10, 'Nov.': 11, 'Dec.': 12})

# Append precipitation data to our DataFrame
df = pd.merge(df, df_precipitation, on=["Station_id", "Month"],
              how="left")

# Save to file
df.to_csv("weather.csv", index=False)
```



Johannes Schildgen

Python for AI and Data Analysis

The Practical Guide for Business and Science

- A practical guide to Python for AI and data projects
- Learn to analyze and visualize data with NumPy, Pandas, and Matplotlib
- Use AI and machine learning for predictive modeling, text analysis, and image recognition

www.sap-press.com/6385

The Author

Prof. Dr. Johannes Schildgen is a professor of databases, specializing in big data, at Ostbayerische Technische Hochschule (OTH) Regensburg (the Regensburg University of Applied Sciences).

ISBN 978-1-4932-2892-8 • 282 pages • 09/2026

E-book: $44.99 • **Print book:** $49.95 • **Bundle:** $59.99

Rheinwerk Publishing