Python for DataScience

Guillaume Wisniewski

Nov 07, 2025

CONTENTS

These notes contain a not so short introduction to `pandas` main features. It is (as usual) a work in progress. Please let me know of any errors, inconsistencies, gaps or passages that are difficult to understand.

These notes are intended for people who have never programmed or who have programmed very little. But they also contain what might be described as "advanced" manipulations. Not all chapters are relevant to you.

The Gold Rules for Good Programmers

- The best code (by every conceivable criterion) is the code you do not write: use libraries as much as possible!
- whatever problem you are trying to solve, there is almost certainly someone who has already dealt with it (why is `ChatGPT` so good?)
- if the code you write is longer than 5 lines, your solution is not the right one.
- `for` loops and conditions (`if`) are tools from another era. I'm not even talking about `while` or `do` loops, which even I have never heard of.
- using high-level functions is the best way of avoiding errors and obtaining the most efficient code/
- if you are blindly following these advices, you have not understood their purpose at all

Introduction

In just a few years, Python has established itself as the reference language for data analysis and machine learning (or, in more pompous terms, *data science*). This is largely due to the qualities of the language (those who have never had to program in C, C++ or Java can't possibly understand) and to the presence of numerous powerful, well-documented libraries enabling complex operations to be carried out very easily.

The aim of this course (and consequently this accompanying support) is to introduce you to one of the fundamental libraries in the "python for data analysis" ecosystem: `pandas`. We'll be taking a look at how this library enables you to integrate data from a variety of sources, clean it up, select it, represent it and perform more or less complex statistical analyses on it. Basically, pandas is the ideal tool for transforming raw data into useful decision-making information — the goal of every data scientist.

Python Ecosystem

Before diving into `pandas`, it is important to understand the different libraries and tools that make up the "Python ecosystem". Starting from the most fundamental building block and moving up to the most abstract:

- Python – this refers both to the programming language itself and to its interpreter.
- Data storage libraries – the classic `NumPy` library for representing vectors and tensors, and the more recent `Arrow` library.
- Low-level manipulation libraries – libraries designed to handle these representations at a low level. This level include library such as `scipy` that implements many numerical algorithms.
- High-level libraries – libraries that encapsulate operations on these structures such as `pandas` and `scikit-learn`.

Alongside this "stack" there are several tools and libraries that make it easier to work with all of the above. Chief among these is the Jupyter project, which introduced the concept of "notebooks" (now supported in most editors), as well as numerous tools for installing libraries, managing virtual environments and handling dependencies. All of these topics will (hopefully in the near future) be covered in more detail in this document.

Part I

Pandas features

CHAPTER

ONE

FUNDAMENTAL OPERATIONS WITH PANDAS

1.1 Working with Tabular Data in Python

Almost all data of interest—whether experimental results, a corpus for processing, or information handled by any organisation—can be represented as a table (think of an Excel® worksheet), commonly referred to as tabular data: each row records an observation and each column records a variable or attribute (i.e., a characteristic of that observation). For example, in a dataset of students' marks, each observation corresponds to a mark obtained by a student in a lecture (or module) and is described by three attributes: the student's name, the lecture, and the mark obtained.

1.1.1 Storing the data: CSV

In this chapter, we will assume this data is stored in a file named `notes.csv`. CSV (Comma-Separated Values) is a plain-text format for representing tabular data. It is something of a universal format for exchanging tables: it is readable by humans and supported by most software.

However, CSV is not a robust format (even if it is the format we will use throughout this document). Inferring what type of data is in each column is indeed not straightforward, and loading a file can sometimes lead to subtle, hard-to-diagnose bugs. As we will see, `pandas` provides a wide range of methods to load data from almost any format you can think of.

1.1.2 Why pandas?

The `pandas` library contains the tools used to manipulate tabular data in Python. It is installed by default if you use Google Colab or Anaconda. If you are using another Python distribution, you will need to install it explicitly.

1.1.3 Import convention

To use the library, we first import it and agree to access its functions with the prefix pd. This alias is a near-universal convention: virtually all code using `pandas` begins with:

```
import pandas as pd
```

Here's the data we're going to use, in this first chapter:

```
# Warning: this cell will run only on *NIX systems.
# On windows, you should run the following PowerShell command:
#
# @'
# nom,matiere,note
# Ali,Physique,18
```

(continues on next page)

(continued from previous page)

```
# Mélio,Maths,3
# Oana,Physique,9
# Séverine,Physique,15
# Séverine,Maths,18
# Ali,Maths,3
# '@ | Set-Content -Path notes.csv -Encoding UTF8

!printf '%s\n' \
'name,subject,mark' \
'Ali,Physics,18' \
'Mélio,Maths,3' \
'Oana,Physics,9' \
'Séverine,Physics,15' \
'Séverine,Maths,18' \
'Ali,Maths,3' > notes.csv
```

The previous cell contains shell commands (or PowerShell on Windows) that create a CSV file by specifying its contents. The first line of the file contains the column names; all subsequent lines contain the table data.

Remember that, in a notebook, lines beginning with an exclamation mark (`!`) are not Python code: they execute shell commands.

We will now load the data:

```
df = pd.read_csv("notes.csv")
# ask jupyter to display the content of the variable `data`
df
```

```
       name  subject  mark
0       Ali  Physics    18
1     Mélio    Maths     3
2      Oana  Physics     9
3  Séverine  Physics    15
4  Séverine    Maths    18
5       Ali    Maths     3
```

The first line calls the `read_csv` function from the `pandas` library (as indicated by the `pd` prefix). This function takes a single required argument: the path to the file.

Because the path here is just the file name, `pandas` looks for the file in the current working directory. The file must therefore be located in the same directory as your code. If the file cannot be found, for example, because of a typo in the file name or because it does not exist, the function raises an error (an "exception" in Python).

In a Python traceback, the most informative technical message is usually found on the final line. The rest of the traceback mainly helps you locate where the error occurred which is often straightforward in a notebook, but potentially tricky in a large codebase.

Here is an example of an exception:

```
pd.read_csv("file_that_does_not_exist.csv")
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[4], line 1
----> 1 pd.read_csv("file_that_does_not_exist.csv")

File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
```

(continues on next page)

(continued from previous page)

```
↪13/site-packages/pandas/io/parsers/readers.py:1026, in read_csv(filepath_or_
↪buffer, sep, delimiter, header, names, index_col, usecols, dtype, engine,␣
↪converters, true_values, false_values, skipinitialspace, skiprows, skipfooter,␣
↪nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_
↪dates, infer_datetime_format, keep_date_col, date_parser, date_format, dayfirst,␣
↪cache_dates, iterator, chunksize, compression, thousands, decimal,␣
↪lineterminator, quotechar, quoting, doublequote, escapechar, comment, encoding,␣
↪encoding_errors, dialect, on_bad_lines, delim_whitespace, low_memory, memory_map,
↪ float_precision, storage_options, dtype_backend)
   1013 kwds_defaults = _refine_defaults_read(
   1014     dialect,
   1015     delimiter,
   (...)   1022     dtype_backend=dtype_backend,
   1023 )
   1024 kwds.update(kwds_defaults)
-> 1026 return _read(filepath_or_buffer, kwds)

File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/io/parsers/readers.py:620, in _read(filepath_or_buffer,␣
↪kwds)
    617 _validate_names(kwds.get("names", None))
    619 # Create the parser.
--> 620 parser = TextFileReader(filepath_or_buffer, **kwds)
    622 if chunksize or iterator:
    623     return parser

File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/io/parsers/readers.py:1620, in TextFileReader.__init__
↪(self, f, engine, **kwds)
   1617     self.options["has_index_names"] = kwds["has_index_names"]
   1619 self.handles: IOHandles | None = None
-> 1620 self._engine = self._make_engine(f, self.engine)

File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/io/parsers/readers.py:1880, in TextFileReader._make_
↪engine(self, f, engine)
   1878     if "b" not in mode:
   1879         mode += "b"
-> 1880 self.handles = get_handle(
   1881     f,
   1882     mode,
   1883     encoding=self.options.get("encoding", None),
   1884     compression=self.options.get("compression", None),
   1885     memory_map=self.options.get("memory_map", False),
   1886     is_text=is_text,
   1887     errors=self.options.get("encoding_errors", "strict"),
   1888     storage_options=self.options.get("storage_options", None),
   1889 )
   1890 assert self.handles is not None
   1891 f = self.handles.handle

File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/io/common.py:873, in get_handle(path_or_buf, mode,␣
↪encoding, compression, memory_map, is_text, errors, storage_options)
    868 elif isinstance(handle, str):
    869     # Check whether the filename is to be opened in binary mode.
    870     # Binary mode does not support 'encoding' and 'newline'.
```

(continues on next page)

(continued from previous page)

```
    871     if ioargs.encoding and "b" not in ioargs.mode:
    872         # Encoding
--> 873         handle = open(
    874             handle,
    875             ioargs.mode,
    876             encoding=ioargs.encoding,
    877             errors=errors,
    878             newline="",
    879         )
    880     else:
    881         # Binary mode
    882         handle = open(handle, ioargs.mode)

FileNotFoundError: [Errno 2] No such file or directory: 'file_that_does_not_exist.csv'
```

The most useful part of the traceback is the final line, which states the type of error that occurred (here, `FileNotFoundError`) and, in this case, a more detailed description (including the name of the file that was not found).

Note that `read_csv` takes a path to the file as its argument. A bare file name such as `my_filename.csv` is interpreted as `./my_filename.csv`, i.e. a path relative to the current working directory. The path may be absolute or relative.

Be careful when specifying paths on Windows: Windows uses `\` as the directory separator (not `/`, as on Unix-like systems). When passing a path string to Python, you should either use forward slashes, escape backslashes, or mark the string as a raw string. For example:

```
pd.read_csv(r"C:\Users\Guillaume\my_file.csv")   # raw string
# or
pd.read_csv("C:\\Users\\Guillaume\\my_file.csv") # escaped backslashes
# or (forward slashes also work on Windows in Python)
pd.read_csv("C:/Users/Guillaume/my_file.csv")
```

Without the leading `r` (and without escaping), Python will interpret sequences like `\u` as escape codes, and the string may no longer refer to the intended file.

The instruction `data = pd.read_csv("notes.csv")` allows you to:

1. read the contents of the CSV file
2. store it in a variable that can be accessed and manipulated.

1.2 Understanding how DataFrame are working

The variable `df` is a `DataFrame` (technically, that is its type), a Python data structure used to represent tabular data. The name `df` is commonly used when no specific name suggests itself, but this is generally unwise: variable names are the first line of documentation for your code.

At first glance, a `DataFrame` is like an Excel worksheet in which the data are organised:

- in columns: columns have names and correspond to characteristics/attributes;
- in rows: rows have an index (a number or label) and describe an observation.

`DataFrame` is one of the fundamental types defined in the `pandas` library for tabular data. Another fundamental type is `Series`, which represents a single column.

Note that a `DataFrame`'s index is not a column (which is why it often appears in bold in many displays). You can access the index via `df.index` and the list of columns (also called the column index) via `df.columns`.

```
df.index
```

```
RangeIndex(start=0, stop=6, step=1)
```

```
df.columns
```

```
Index(['name', 'subject', 'mark'], dtype='object')
```

In pandas, row and column indices are chiefly a labelling mechanism that makes data access convenient and helps express certain operations more naturally. Unlike SQL indexes—which are engineered for performance, often enforce uniqueness, and are managed by the database engine—pandas indices impose no such constraints. Depending on the task, you can freely change the index, reset it, or construct a composite index (a `MultiIndex`) to simplify later steps. In the remainder of this document, we will review the main ways to modify indices and show how temporarily adjusting them can make some operations markedly easier to write.

`DataFrame` columns are typed: each column holds values of a single data type. The `info` method provides a full summary of a `DataFrame`, including the column types; alternatively, use `df.dtypes` (or `series.dtype` for a single column).

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   name     6 non-null      object
 1   subject  6 non-null      object
 2   mark     6 non-null      int64
dtypes: int64(1), object(2)
memory usage: 276.0+ bytes
```

```
df.dtypes
```

```
name        object
subject     object
mark         int64
dtype: object
```

In our example, pandas has inferred the column types automatically (CSV does not describe column types, so pandas relies on heuristics). It detected that the mark column contains integers, while the other two columns were read as `object`. A generic data type that, in this case, holds text strings.

1.2.1 Selecting part of `DataFrame`

The most fundamental operation on a `DataFrame` is selecting part of it, either:

- Columns: `data[column_name]` for a single column (returns a `Series`), or `data[[col1, col2, ..., coln]]` for several columns (returns a `DataFrame`);
- Rows: `data.loc[2:4]` to select rows by their index labels (note that `.loc` slices are inclusive of both ends).

```
df["mark"]
```

```
0    18
1     3
2     9
3    15
4    18
5     3
Name: mark, dtype: int64
```

The statement here returns a Series. Visually, the difference is mainly the absence of a column name (although this information is stored in the `name` attribute, shown at the end of the printed values). A `Series` can be viewed as a dictionary that maps index labels (the values in the first "column") to data values (the second "column"). To access the values directly, use the `values` attribute:

```
df["mark"].values
```

```
array([18,  3,  9, 15, 18,  3])
```

The last instruction returns a NumPy array (an `ndarray` from the numpy library), which shows that, in practice, `pandas` is a layer on top of NumPy. It offers higher-level methods for accessing and manipulating data whilst preserving NumPy's performance (NumPy was originally designed for scientific computing with a strong focus on speed).

In newer versions of `pandas` (from 2.0 onwards), you can choose different data back-ends, that is, alternative libraries that manage how DataFrame values are stored and accessed, each offering a distinct trade-off between memory usage and computational speed (e.g., NumPy-backed vs PyArrow-backed dtypes).

We'll come back to this point later. (XXX to be done!)

`pandas` raises an exception, specifically a `KeyError` when you attempt to access a non-existent column:

```
df["notes"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/core/indexes/base.py:3812, in Index.get_loc(self, key)
   3811 try:
-> 3812     return self._engine.get_loc(casted_key)
   3813 except KeyError as err:

File pandas/_libs/index.pyx:167, in pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/index.pyx:196, in pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/hashtable_class_helper.pxi:7088, in pandas._libs.hashtable.
↪PyObjectHashTable.get_item()

File pandas/_libs/hashtable_class_helper.pxi:7096, in pandas._libs.hashtable.
↪PyObjectHashTable.get_item()

KeyError: 'notes'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Cell In[11], line 1
----> 1 df["notes"]
```

(continues on next page)

(continued from previous page)

```
File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/core/frame.py:4107, in DataFrame.__getitem__(self, key)
   4105 if self.columns.nlevels > 1:
   4106     return self._getitem_multilevel(key)
-> 4107 indexer = self.columns.get_loc(key)
   4108 if is_integer(indexer):
   4109     indexer = [indexer]

File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/core/indexes/base.py:3819, in Index.get_loc(self, key)
   3814     if isinstance(casted_key, slice) or (
   3815         isinstance(casted_key, abc.Iterable)
   3816         and any(isinstance(x, slice) for x in casted_key)
   3817     ):
   3818         raise InvalidIndexError(key)
-> 3819     raise KeyError(key) from err
   3820 except TypeError:
   3821     # If we have a listlike key, _check_indexing_error will raise
   3822     #  InvalidIndexError. Otherwise we fall through and re-raise
   3823     #  the TypeError.
   3824     self._check_indexing_error(key)

KeyError: 'notes'
```

```
df[["subject", "mark"]]
```

```
   subject  mark
0  Physics    18
1    Maths     3
2  Physics     9
3  Physics    15
4    Maths    18
5    Maths     3
```

Here, the value returned is a DataFrame.

You can select specific rows by supplying a list of index labels:

```
df.loc[[2, 3, 4]]
```

```
       name  subject  mark
2      Oana  Physics     9
3  Séverine  Physics    15
4  Séverine    Maths    18
```

Consecutive index labels can be selected using a slice:

```
df.loc[2:4]
```

```
       name  subject  mark
2      Oana  Physics     9
3  Séverine  Physics    15
4  Séverine    Maths    18
```

Be careful: the previous instructions select rows whose index labels are passed to `loc`. Those labels do not necessarily correspond to the row positions. In practice, when a CSV is loaded without an explicit index, pandas creates a default

RangeIndex (0, 1, 2, ...) that initially matches the row positions. However, operations such as sort_values can reorder the rows without changing the index labels. As the following output shows, the left-most index column then no longer reflects the displayed row order:

```
df.sort_values(by="mark")
```

```
       name  subject  mark
1     Mélio    Maths     3
5       Ali    Maths     3
2      Oana  Physics     9
3  Séverine  Physics    15
0       Ali  Physics    18
4  Séverine    Maths    18
```

```
d = df.sort_values(by="mark")
d.loc[[0, 1, 2]]
```

```
    name  subject  mark
0    Ali  Physics    18
1  Mélio    Maths     3
2   Oana  Physics     9
```

In the preceding instruction, the rows selected correspond to the index labels, not their positions. To select rows by position, use .iloc.

```
df.iloc[1:3]
```

```
    name  subject  mark
1  Mélio    Maths     3
2   Oana  Physics     9
```

This time we're genuinely selecting the first two rows of the sorted DataFrame. Note that, unlike .loc, .iloc excludes the upper bound (the end index) of the slice.

1.2.2 Selecting rows that match a given criterion

It is also possible to select observations that satisfy a given condition. For example, you can select all Physics marks or all marks below 10 with the following commands:

```
df[df["subject"] == "Physics"]
```

```
       name  subject  mark
0       Ali  Physics    18
2      Oana  Physics     9
3  Séverine  Physics    15
```

```
df[df["mark"] < 10]
```

```
    name  subject  mark
1  Mélio    Maths     3
2   Oana  Physics     9
5    Ali    Maths     3
```

1.3 Aggregate column values

A number of methods are defined to "aggregate" the values of a column and, in particular, calculate certain statistics. The syntax is always the same: `data_frame[column].methode()`, and the method can take any number of arguments:

```
# mean value of a columns:
df["mark"].mean()
```

```
np.float64(11.0)
```

```
# standard deviatio
df["mark"].std()
```

```
np.float64(7.014271166700073)
```

```
df
```

```
       name  subject  mark
0       Ali  Physics    18
1     Mélio    Maths     3
2      Oana  Physics     9
3  Séverine  Physics    15
4  Séverine    Maths    18
5       Ali    Maths     3
```

```
# distribution of a discrete value:
df["subject"].value_counts()
```

```
subject
Physics    3
Maths      3
Name: count, dtype: int64
```

```
# modality of an attribute (list of values an attribute can take)
df["name"].unique()
```

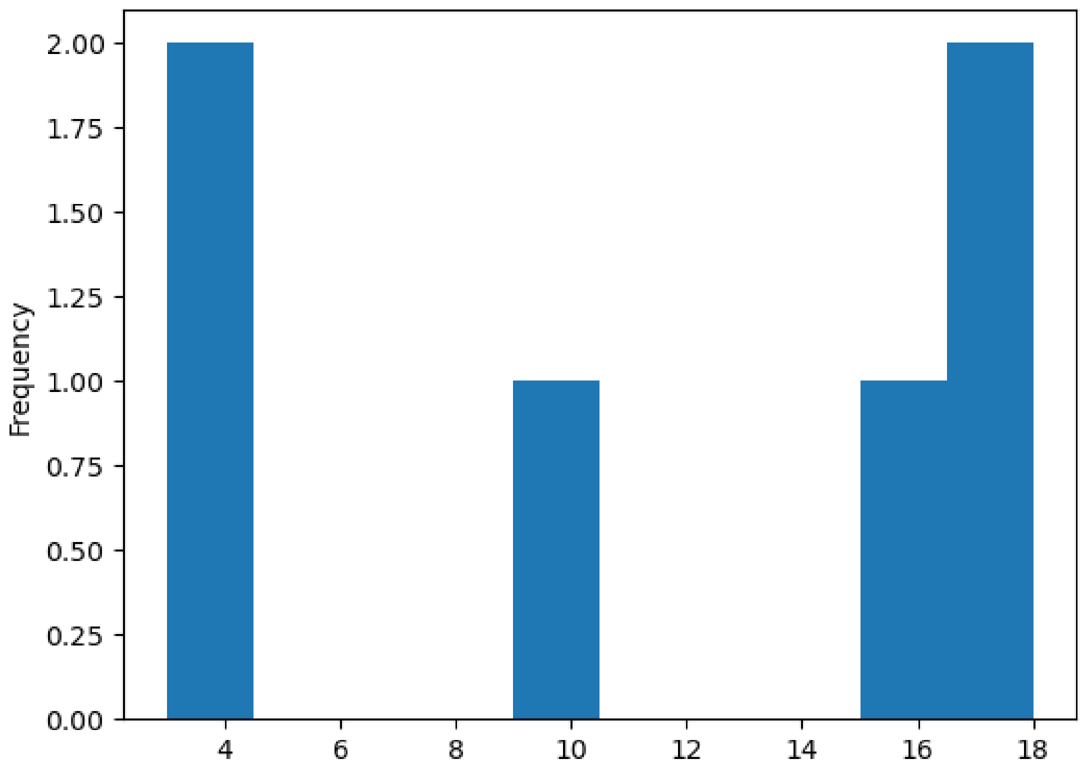
```
array(['Ali', 'Mélio', 'Oana', 'Séverine'], dtype=object)
```

1.4 Plotting

Plots can be created directly from `DataFrame`.

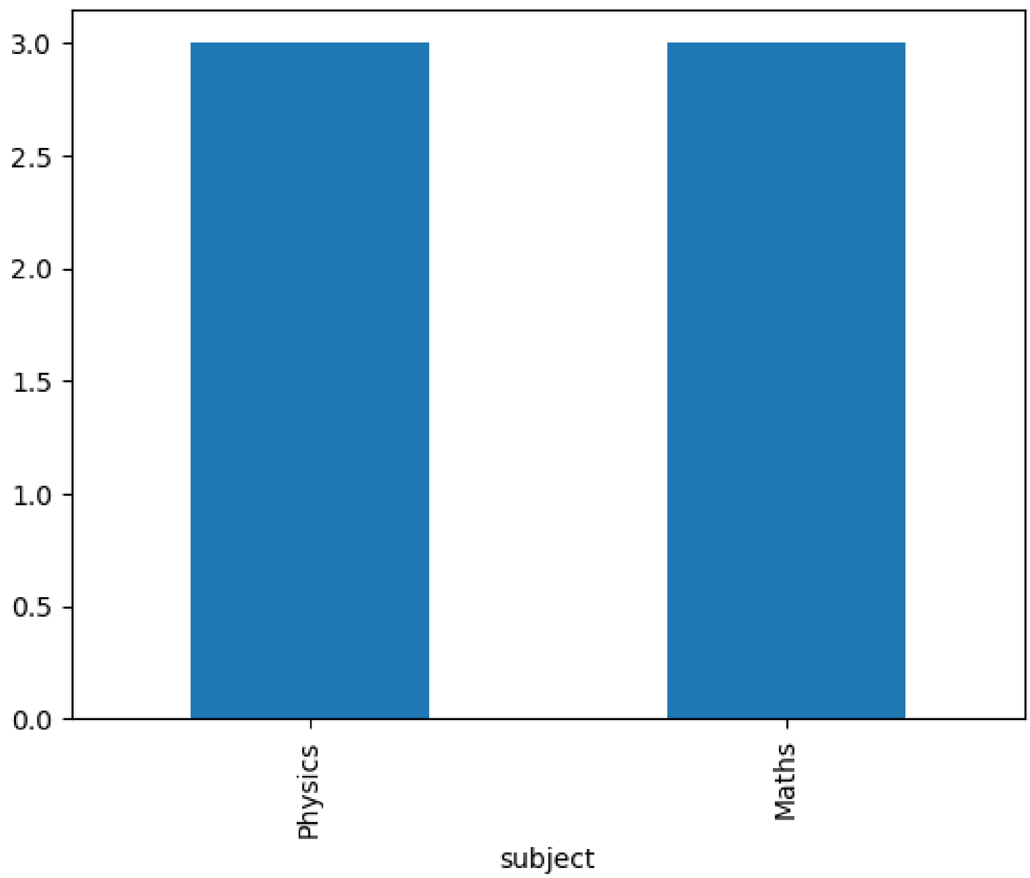
```
df["mark"].plot(kind="hist")
```

```
<Axes: ylabel='Frequency'>
```




```
df["subject"].value_counts().plot(kind="bar")
```

```
<Axes: xlabel='subject'>
```



The previous cell is a very good example of methods **chaining**: the plot is constructed from the result obtained by the `count_values()` method. The plot displayed result from three steps:

1. the values in the `subject` column are extracted;
2. the `count_values()` method is used to count the modalities of this attribute;
3. the count result is transformed into a graph.

CHAPTER

TWO

ANALYZING THE TITANIC DEATHS

The Titanic data set contains details about the ship's passengers—such as age, gender, ticket class and whether they survived. It is commonly used to demonstrate the capabilities of `pandas`.

```
import pandas as pd

from pathlib import Path
```

```
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [3.2, 2.4]
```

```
# download dataset (only once)
if not Path("titanic.csv").is_file():
    !curl -L https://lc.cx/WhWhEP -o titanic.csv
```

```
titanic_passengers = pd.read_csv("titanic.csv")
titanic_passengers.sample(5)
```

```
     PassengerId  Survived  Pclass                            Name     Sex  \
257          258         1       1            Cherry, Miss. Gladys  female
417          418         1       2   Silven, Miss. Lyyli Karoliina  female
310          311         1       1  Hays, Miss. Margaret Bechstein  female
827          828         1       2           Mallet, Master. Andre    male
116          117         0       3            Connors, Mr. Patrick    male

      Age  SibSp  Parch           Ticket     Fare Cabin Embarked
257  30.0      0      0           110152  86.5000   B77        S
417  18.0      0      2           250652  13.0000   NaN        S
310  24.0      0      0            11767  83.1583   C54        C
827   1.0      0      2  S.C./PARIS 2079  37.0042   NaN        C
116  70.5      0      0           370369   7.7500   NaN        Q
```

Each row of the DataFrame describes a passenger by specifying a number of attributes (the columns). In particular: the column `Survived` indicates whether the person survived the shipwreck or not, the column `Name` indicates the passenger's name, …

2.1 Questions

1. How many records (i.e. rows) has the dataset?
2. Select the `Age` column. How many different values are there?
3. Make a box plot of the `Fare` column.
4. Sort the rows of the DataFrame by the `Age` column, with the oldest passenger at the top.
5. Select all rows for male passengers and calculate the mean age of those passengers.
6. How many passengers older than 70 were on the Titanic?
7. Select the passengers that are between 30 and 40 years old?
8. Split the `Name` column on the `,` extract the first part (the surname), and add this as new column `Surname`. Do the same for the first name and the title (e.g. `Dr.`). How many different titles are there?
9. Select all passenger that have a surname starting with `Williams`
10. Select all rows for the passengers with a surname of more than 30 characters.
11. Using `groupby`, calculate the average age for each sex.
12. Calculate this survival ratio for all passengers younger/older than 25
13. Make a bar plot of the survival ratio for the different classes (`Pclass` column).
14. Make a pivot table with the survival rates for `Pclass` vs `Sex`.
15. Make a table of the median Fare payed by aged/underaged vs `Sex`.

2.2 Solutions

1. How many records (i.e. rows) has the dataset?

```
titanic_passengers.shape[0]
```

```
891
```

The `shape` attribute of a `DataFrame` is a tuple containing the number of rows and columns (in that order). Here we extract only the number of rows.

2. Select the `Age` column. How many different values are there?

```
titanic_passengers["Age"].nunique()
```

```
88
```

```
titanic_passengers["Age"].value_counts()
```

```
Age
24.00    30
22.00    27
18.00    26
19.00    25
28.00    25
```

(continues on next page)

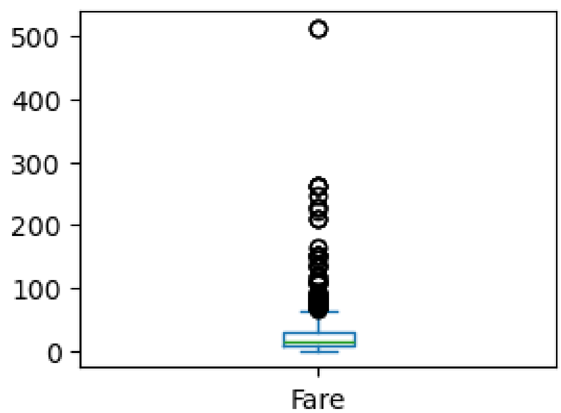
(continued from previous page)

```
         ..
36.50     1
55.50     1
0.92      1
23.50     1
74.00     1
Name: count, Length: 88, dtype: int64
```

3. Make a box plot of the `Fare` column.

```
titanic_passengers["Fare"].plot(kind="box")
```

```
<Axes: >
```



4. Sort the rows of the DataFrame by the `Age` column, with the oldest passenger at the top.

```
# Display only the first three rows and the key columns to keep the output tidy
titanic_passengers.sort_values(by="Age", ascending=False)\
    .head(3)[["Name", "Age", "PassengerId"]]
```

```
                                    Name   Age  PassengerId
630  Barkworth, Mr. Algernon Henry Wilson  80.0          631
851                  Svensson, Mr. Johan  74.0          852
493              Artagaveytia, Mr. Ramon  71.0          494
```

5. Select all rows for male passengers and calculate the mean age of those passengers.

```
titanic_passengers[titanic_passengers["Sex"] == "male"]["Age"].mean()
```

```
np.float64(30.72664459161148)
```

6. How many passengers older than 70 were on the Titanic?

```
titanic_passengers[titanic_passengers["Age"] > 70].shape[0]
```

```
5
```

7. Select the passengers that are between 30 and 40 years old?

```
titanic_passengers[titanic_passengers["Age"].between(30, 40)]\
    .sample(5)[["Name", "Age", "PassengerId"]]
```

```
                                              Name   Age  PassengerId
705     Morley, Mr. Henry Samuel ("Mr Henry Marshall")  39.0          706
719                          Johnson, Mr. Malkolm Joackim  33.0          720
670  Brown, Mrs. Thomas William Solomon (Elizabeth ...  40.0          671
308                                   Abelson, Mr. Samuel  30.0          309
346                             Smith, Miss. Marion Elsie  40.0          347
```

```
titanic_passengers[(titanic_passengers["Age"] >= 30) &
    (titanic_passengers["Age"] <= 40)]\
    .sample(5)[["Name", "Age", "PassengerId"]]
```

```
                          Name   Age  PassengerId
534          Cacic, Miss. Marija  30.0          535
263        Harrison, Mr. William  40.0          264
814  Tomlin, Mr. Ernest Portage  30.5          815
70   Jenkin, Mr. Stephen Curnow  32.0           71
671      Davidson, Mr. Thornton  31.0          672
```

8. Split the `Name` column on the `,` extract the first part (the surname), and add this as new column `Surname`. Do the same for the first name and the title (e.g. `Dr.`). How many different titles are there?

```
titanic_passengers["Surname"] = titanic_passengers["Name"].str.split(",").str[0]
titanic_passengers[["Name", "Surname"]].head()
```

```
                                                Name    Surname
0                            Braund, Mr. Owen Harris     Braund
1  Cumings, Mrs. John Bradley (Florence Briggs Th...    Cumings
2                             Heikkinen, Miss. Laina  Heikkinen
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)   Futrelle
4                           Allen, Mr. William Henry      Allen
```

```
titanic_passengers["Name"]\
    .str.split(", ").str[1]\
    .str.split(" ").str[0]\
    .value_counts()
```

```
Name
Mr.         517
Miss.       182
Mrs.        125
Master.      40
Dr.           7
Rev.          6
Mlle.         2
Major.        2
Col.          2
the           1
Capt.         1
Ms.           1
Sir.          1
Lady.         1
Mme.          1
```

(continues on next page)

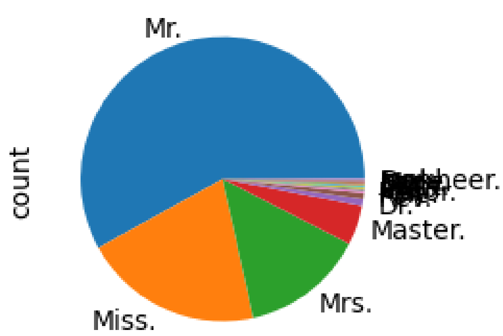
(continued from previous page)

```
Don.            1
Jonkheer.       1
Name: count, dtype: int64
```

It is possible to plot the result of the command:

```
titanic_passengers["Name"]\
    .str.split(", ").str[1]\
    .str.split(" ").str[0]\
    .value_counts()\
    .plot(kind="pie")
```

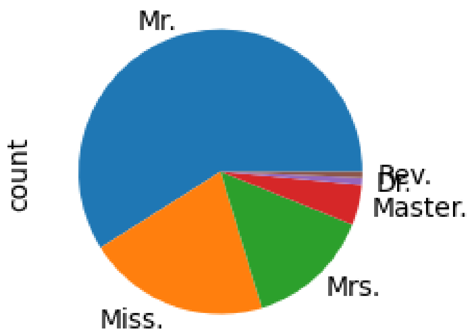
```
<Axes: ylabel='count'>
```



It is also possible select only the most title that appears at least five times in the corpus:

```
most_frequent_titles = titanic_passengers["Name"]\
    .str.split(", ").str[1]\
    .str.split(" ").str[0]\
    .value_counts()\
    .rename("count")\
    .to_frame()\
    .query("count > 5").index

titanic_passengers["Title"] = titanic_passengers["Name"].str.split(", ").str[1].str.
↪split(" ").str[0]
titanic_passengers[titanic_passengers["Title"].isin(most_frequent_titles)]["Title"].
↪value_counts().plot(kind="pie")
```

```
<Axes: ylabel='count'>
```



The last two lines of the cell filter out any rows that do not correspond to the most commonly used titles.

First, the top line identifies the most frequent titles; its opening part is the same as the one we saw earlier. The next two steps (`rename` and `to_frame`) convert the resulting column into a `DataFrame`, which is both easier and more conventional to filter. Finally, `query` and `.index` remove titles that occur only once and extract the corresponding list.

9. Select all passenger that have a surname starting with `Williams`

```
titanic_passengers[titanic_passengers["Name"].str.startswith("Williams")]\
    [["PassengerId", "Survived", "Name"]]
```

```
     PassengerId  Survived                                   Name
17            18         1           Williams, Mr. Charles Eugene
155          156         0            Williams, Mr. Charles Duane
304          305         0        Williams, Mr. Howard Hugh "Harry"
351          352         0  Williams-Lambert, Mr. Fletcher Fellows
735          736         0                   Williams, Mr. Leslie
```

10. Select all rows for the passengers with a surname of more than 30 characters.

```
titanic_passengers[titanic_passengers["Name"].str.len() > 30].shape[0]
```

```
222
```

11. Using `groupby`, calculate the average age for each sex.

```
titanic_passengers.groupby("Sex")["Age"].mean()
```

```
Sex
female    27.915709
male      30.726645
Name: Age, dtype: float64
```

12. Calculate this survival ratio for all passengers younger/older than 25

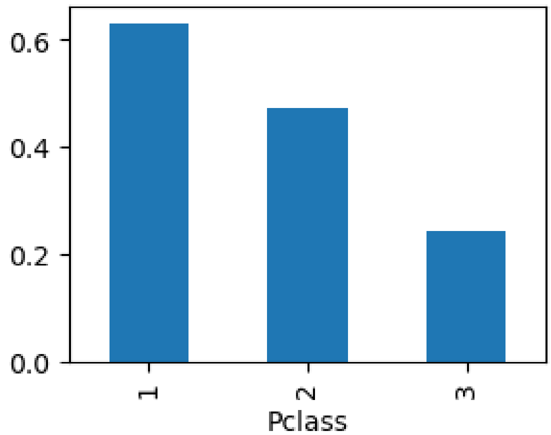
```
titanic_passengers.groupby(titanic_passengers["Age"] <= 25)["Survived"].mean()
```

```
Age
False    0.369492
True     0.411960
Name: Survived, dtype: float64
```

13. Make a bar plot of the survival ratio for the different classes (`Pclass` column).

```
titanic_passengers.groupby("Pclass")["Survived"].mean().plot(kind="bar")
```

```
<Axes: xlabel='Pclass'>
```



14. Make a pivot table with the survival rates for `Pclass` vs `Sex`.

```
titanic_passengers.pivot_table(index="Pclass", columns="Sex", values="Survived",
 ↪aggfunc="mean")
```

```
Sex       female      male
Pclass
1       0.968085  0.368852
2       0.921053  0.157407
3       0.500000  0.135447
```

15. Make a table of the median Fare payed by aged/underaged vs `Sex`.

```
titanic_passengers.pivot_table(index="Sex",
                               columns=titanic_passengers["Age"] < 18,
                               values="Fare",
                               aggfunc="median")
```

```
Age       False    True
Sex
female  23.2500  22.025
male    10.1708  26.000
```

CHAPTER

THREE

ANALYZING IMDB MOVIES

In this exercise we will analyze data collected on the *Internet Movie Database* website (imbd). We will consider two files:

- `movies.csv` that contains a list of movies referenced on imbd;
- `casting.csv` that contains information about the casting of movies, namely for each character, the name of the actor or actress who played the role. This information is extracted from the film credits and also contains the order of appearance of the character in the credits (column `n`).

```
from pathlib import Path

import pandas as pd
```

```
# download the files if they have not been downloaded yet
# this cel will only work on *nix systems
if not Path("cast.csv").is_file():
    !curl -L https://short.do/Co-p5y -o cast.csv
    !curl -L https://short.do/klo1RK -o movies.csv
```

```
cast = pd.read_csv("cast.csv")
movies = pd.read_csv("movies.csv")
```

3.1 Questions

1. How many movies have the title “Hamlet”?
2. What are the earliest/latest two films listed in the `movies.csv` file?
3. List all of the “Treasure Island” movies from earliest to most recent.
4. How many movies were made from 1950 through 1959?
5. How many roles in the movie “Inception” are NOT ranked by an “n” value?
6. Display the cast of the “Titanic” (the most famous 1997 one) in their correct “n”-value order, ignoring roles that did not earn a numeric `n` value.
7. List the supporting roles (having n=2) played by Brad Pitt in the 1990s, ordered by year.
8. Using `groupby`, plot the number of films that have been released each decade in the history of cinema
9. List the 10 actors and the 10 actresses that have the most leading roles (n = 1) since the 1990’s
10. How many leading (n = 1) roles were available to actors, and how many to actresses, in each year of the 1950s?
11. What are the 11 most common character names in movie history?

12. Plot how many roles Brad Pitt has played in each year of his career.
13. List, in order by year, each of the movies in which “Frank Oz” has played more than 1 role.
14. List each of the characters that Frank Oz has portrayed at least twice.

3.2 Answers

1. How many movies have the title “Hamlet”?

```
movies[movies["title"] == "Hamlet"].shape[0]
```

```
20
```

The same result can be obtained from the `cast` dataframe; you just need to pay attention to one detail: film titles are repeated for each entry in the cast. Therefore, you need to remove duplicates using the `drop_duplicates` command. This command takes as a parameter the list of columns that make it possible to determine whether two rows describe the same individual or not:

```
cast.drop_duplicates(subset=["title", "year"]).query("title == 'Hamlet'").shape[0]
```

```
20
```

This time, we have to use the query method to filter the titles in order to be able to perform chaining.

2. What are the earliest/latest two films listed in the `movies.csv` file?

```
movies.sort_values(by="year").head(2)
```

```
                      title  year
237862           Miss Jerry  1894
69961    The Startled Lover  1898
```

```
movies.sort_values(by="year").tail(2)
```

```
                            title  year
221962  The Zero Century: Maetel  2026
33654                   100 Years  2115
```

```
# put everything in a single `DataFrame`(see the “DataFrame merging” chapter)
pd.concat([movies.sort_values(by="year").tail(2),
           movies.sort_values(by="year").head(2)])
```

```
                            title  year
221962  The Zero Century: Maetel  2026
33654                   100 Years  2115
237862                 Miss Jerry  1894
69961          The Startled Lover  1898
```

3. List all of the “Treasure Island” movies from earliest to most recent.

```
movies[movies["title"] == "Treasure Island"].sort_values(by="year", ascending=True)
```

```
                    title  year
92837   Treasure Island  1918
110254  Treasure Island  1920
91235   Treasure Island  1934
239429  Treasure Island  1950
188828  Treasure Island  1972
100165  Treasure Island  1973
209124  Treasure Island  1985
241322  Treasure Island  1999
```

4. How many movies were made from 1950 through 1959?

```
movies[(movies["year"] >= 1950) & (movies["year"] <= 1959)].shape[0]
```

```
12934
```

```
# more “compact” / “readable” version using a predefined function
movies[movies["year"].between(1950, 1959)].shape[0]
```

```
12934
```

5. How many roles in the movie “Inception” are NOT ranked by an “n” value?

```
cast[(cast["title"] == "Inception") & ~cast["n"].isna()].shape[0]
```

```
37
```

6. Display the cast of the “Titanic” (the most famous 1997 one) in their correct “n”-value order, ignoring roles that did not earn a numeric `n` value.

```
cast[(cast["title"] == "Titanic") & (cast["year"] == 1997) & (~cast["n"].isna())].
↪sort_values(by="n")
```

```
           title  year                name     type               character  \
590596   Titanic  1997  Leonardo DiCaprio    actor            Jack Dawson
2514391  Titanic  1997         Billy Zane    actor            Cal Hockley
2609960  Titanic  1997        Kathy Bates  actress            Molly Brown
1763447  Titanic  1997        Bill Paxton    actor           Brock Lovett
997604   Titanic  1997       Bernard Hill    actor          Captain Smith
...          ...   ...                ...      ...                    ...
957117   Titanic  1997      Lorenz Hasler    actor     I Salonisti Violin
782046   Titanic  1997       Thomas F?ri    actor     I Salonisti Violin
2221876  Titanic  1997      Ferenc Szedl?k    actor      I Salonisti Cello
2221875  Titanic  1997         B?la Szedl?k    actor  I Salonisti Double Bass
827916   Titanic  1997       Werner Giger    actor      I Salonisti Piano

             n
590596     1.0
2514391    3.0
2609960    4.0
1763447    7.0
997604     8.0
...        ...
957117   105.0
782046   106.0
```

(continues on next page)

(continued from previous page)

```
2221876  107.0
2221875  108.0
827916   109.0

[87 rows x 6 columns]
```

7. List the supporting roles (having n=2) played by Brad Pitt in the 1990s, ordered by year.

```
cast[(cast["name"] == "Brad Pitt") & (cast["n"] == 2) & cast["year"].between(1990,
↪1999)]\
    .sort_values(by="year")
```

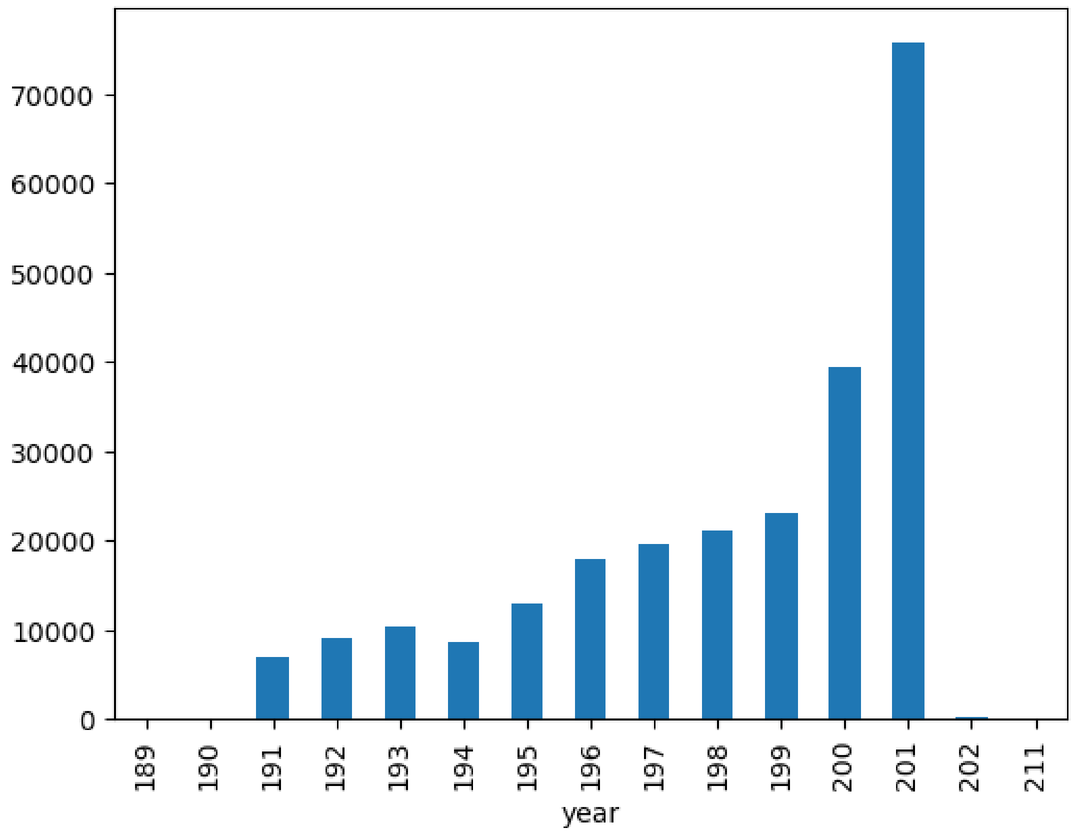
```
                              title  year       name   type  \
1808796         Across the Tracks  1990  Brad Pitt  actor
1808794   A River Runs Through It  1992  Brad Pitt  actor
1808839  The Dark Side of the Sun  1997  Brad Pitt  actor
1808840            The Devil's Own  1997  Brad Pitt  actor
1808841            The Devil's Own  1997  Brad Pitt  actor
1808806                Fight Club  1999  Brad Pitt  actor

                       character    n
1808796              Joe Maloney  2.0
1808794             Paul Maclean  2.0
1808839                     Rick  2.0
1808840             Rory Devaney  2.0
1808841  Francis Austin McGuire  2.0
1808806             Tyler Durden  2.0
```

8. Using `groupby`, plot the number of films that have been released each decade in the history of cinema

```
def get_decade(year):
    # can also be written “return year // 10" (Euclidian division)
    return int(year / 10)
```

```
movies.groupby(movies["year"].apply(get_decade)).size().plot(kind="bar")
```

```
<Axes: xlabel='year'>
```



9. List the 10 actors and the 10 actresses that have the most leading roles (n = 1) since the 1990's

```
cast[(cast["year"] >= 1990) & (cast["n"] == 1)].groupby("name").size().sort_values().
↪tail(10)
```

```
name
Nagarjuna Akkineni     65
Eric Roberts           67
Amitabh Bachchan       70
Dileep (III)           72
Andy Lau               73
Ajay Devgn             77
Jayaram                81
Akshay Kumar           97
Mammootty             130
Mohanlal              141
dtype: int64
```

```
cast[(cast["year"] >= 1990) & (cast["n"] == 1)].groupby("name").size().nlargest(10)
```

```
name
Mohanlal              141
Mammootty             130
Akshay Kumar           97
Jayaram                81
Ajay Devgn             77
Andy Lau               73
```

(continues on next page)

(continued from previous page)

```
Dileep (III)           72
Amitabh Bachchan       70
Eric Roberts           67
Nagarjuna Akkineni     65
dtype: int64
```

10. How many leading (n = 1) roles were available to actors, and how many to actresses, in each year of the 1950s?

```
cast[cast["year"].between(1950, 1959) & (cast["n"] == 1)].groupby(["year", "type"]).
↪size()
```

```
year  type
1950  actor      625
      actress     47
1951  actor      651
      actress     63
1952  actor      613
      actress     68
1953  actor      664
      actress     55
1954  actor      636
      actress     61
1955  actor      648
      actress     54
1956  actor      668
      actress     61
1957  actor      739
      actress     60
1958  actor      715
      actress     63
1959  actor      733
      actress     63
dtype: int64
```

11. What are the 11 most common character names in movie history?

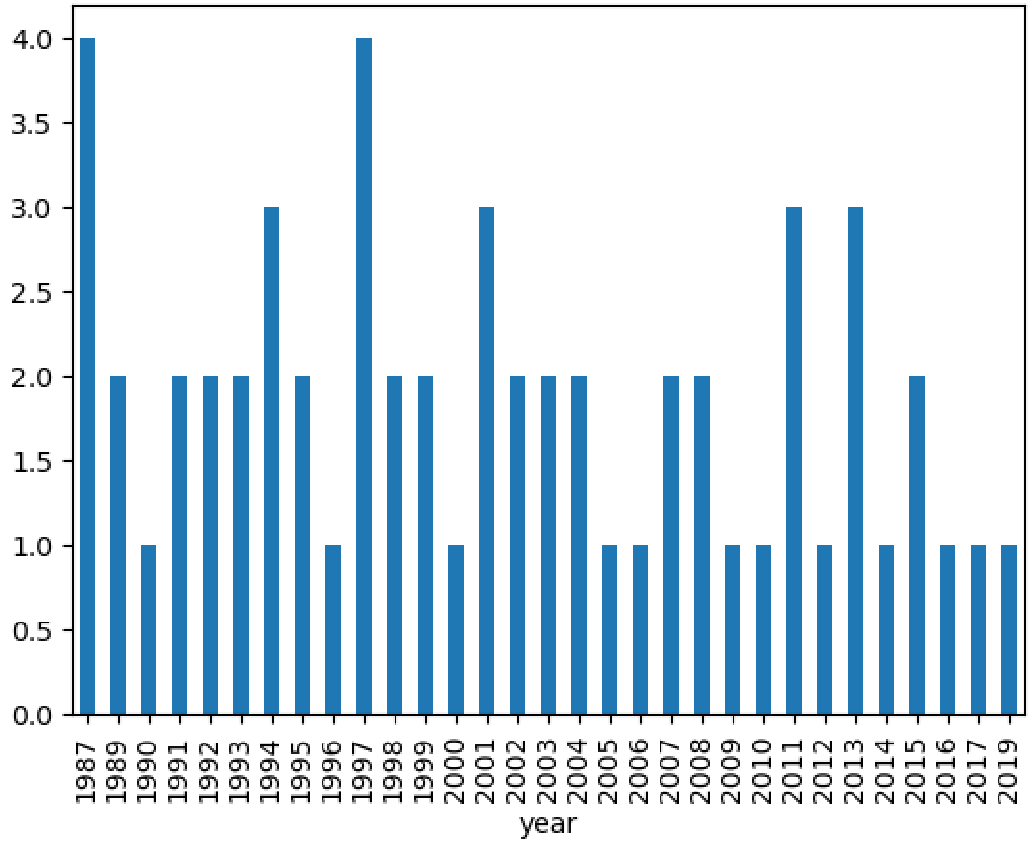
```
cast.groupby(["character"]).size().nlargest(11)
```

```
character
Himself        20719
Extra           7864
Policeman       7019
Doctor          6995
Reporter        6754
Dancer          6413
Bartender       6062
Townsman        5873
Waiter          5389
Party Guest     5099
Henchman        5064
dtype: int64
```

12. Plot how many roles Brad Pitt has played in each year of his career.

```
cast[cast["name"] == "Brad Pitt"].groupby("year").size().plot(kind="bar")
```

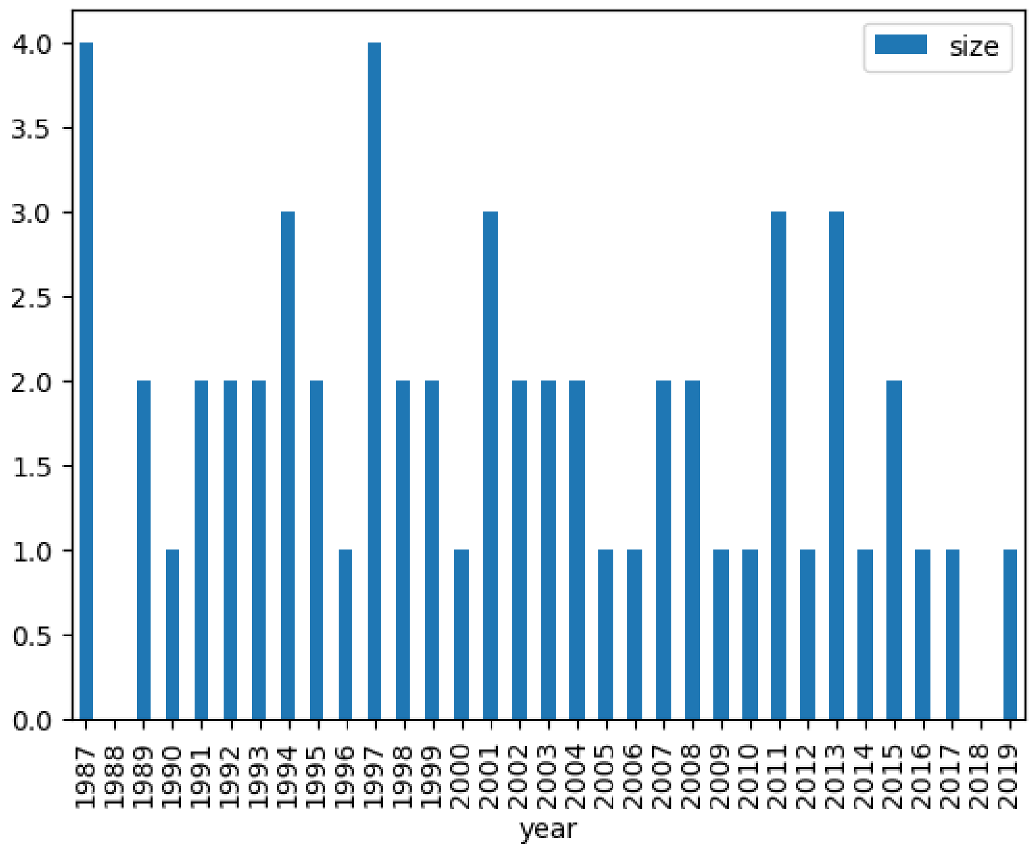
```
<Axes: xlabel='year'>
```



This graph, although obtained very easily, must unfortunately be regarded as false (or at least inaccurate). Without carefully reading the labels on the x-axis, one does not notice that there are years in which Brad Pitt did not play any role. To produce a more readable graph, it is necessary to ensure explicitly that each year appears in the dataframe, created for example in the following way:

```
df = cast[cast["name"] == "Brad Pitt"].groupby("year", as_index=False).size()
all_years = pd.DataFrame({"year": range(df["year"].min(), df["year"].max() + 1)})
pd.merge(all_years, df, how="left", on="year").plot(x="year", y="size", kind="bar")
```

```
<Axes: xlabel='year'>
```



Using the parameter `as_index=False` in the `groupby` function prevents the grouping criterion (or criteria) from becoming an index in the created DataFrame or Series. It is entirely equivalent to

```
cast[cast["name"] == "Brad Pitt"].groupby("year").size().reset_index()
```

(but much longer to write).

Using a time-based index greatly simplifies this code.

13. List, in order by year, each of the movies in which "Frank Oz" has played more than 1 role.

```
df = cast[cast["name"] == "Frank Oz"].groupby("title", as_index=False).size()
df[df["size"] > 1]
```

```
                                    title  size
0          An American Werewolf in London     2
2                         Follow That Bird     3
7                    Muppet Treasure Island     4
8                       Muppets from Space     4
18   The Adventures of Elmo in Grouchland     3
20                         The Dark Crystal     2
22                   The Great Muppet Caper     6
23               The Muppet Christmas Carol     7
24                         The Muppet Movie     8
25               The Muppets Take Manhattan     7
```

We need to create a temporary `DataFrame` to be able to filter the result of the `size` method. Alternatively, we can use the `query` method, which allows us to apply a filter when chaining operations.

```
cast[cast["name"] == "Frank Oz"].sort_values(by="year").groupby("title", as_
↪index=False).size().query("size > 1")
```

```
                               title  size
0         An American Werewolf in London     2
2                    Follow That Bird     3
7               Muppet Treasure Island     4
8                   Muppets from Space     4
18  The Adventures of Elmo in Grouchland     3
20                    The Dark Crystal     2
22              The Great Muppet Caper     6
23          The Muppet Christmas Carol     7
24                    The Muppet Movie     8
25          The Muppets Take Manhattan     7
```

```
cast[cast["name"] == "Frank Oz"].sort_values(by="year")\
    .groupby("title")\
    .size()\
    .reset_index()\
    .rename(columns={0: "size"})\
    .query("size > 1")
```

```
                               title  size
0         An American Werewolf in London     2
2                    Follow That Bird     3
7               Muppet Treasure Island     4
8                   Muppets from Space     4
18  The Adventures of Elmo in Grouchland     3
20                    The Dark Crystal     2
22              The Great Muppet Caper     6
23          The Muppet Christmas Carol     7
24                    The Muppet Movie     8
25          The Muppets Take Manhattan     7
```

14. List each of the characters that Frank Oz has portrayed at least twice.

```
cast[cast["name"] == "Frank Oz"].sort_values(by="year")\
    .groupby("character")\
    .size()\
    .reset_index()\
    .rename(columns={0: "size"})\
    .query("size > 1")
```

```
         character  size
0           Animal     6
2             Bert     3
5   Cookie Monster     5
10     Fozzie Bear     4
15          Grover     2
18      Miss Piggy     6
25   Sam the Eagle     5
34            Yoda     6
```

The code is almost the same as for the previous question. This perfectly illustrates one of the key features of pandas: since the functions operate at the "right" level of abstraction, their combination makes it easy to obtain answers to questions and to adapt them to other ones.

CHAPTER

FOUR

DEALING WITH NAN IN A DATAFRAME

4.1 Les valeurs manquantes

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

from pathlib import Path

import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

from matplotlib import pyplot as plt
plt.style.use('ggplot')
```

```
if not Path("nba.csv").is_file():
    !curl -L bit.ly/3J4LDH2 -o nba.csv
```

```
# we only consider some columns to be able to display dataframes compactly
df = pd.read_csv("nba.csv")[["Name", "Team", "College", "Salary"]]
df.head(10)
```

```
            Name            Team            College      Salary
0  Avery Bradley  Boston Celtics              Texas   7730337.0
1    Jae Crowder  Boston Celtics          Marquette   6796117.0
2   John Holland  Boston Celtics  Boston University         NaN
3    R.J. Hunter  Boston Celtics      Georgia State   1148640.0
4  Jonas Jerebko  Boston Celtics                NaN   5000000.0
5   Amir Johnson  Boston Celtics                NaN  12000000.0
6  Jordan Mickey  Boston Celtics                LSU   1170960.0
7   Kelly Olynyk  Boston Celtics            Gonzaga   2165160.0
8   Terry Rozier  Boston Celtics         Louisville   1824360.0
9   Marcus Smart  Boston Celtics     Oklahoma State   3431040.0
```

Les `DataFrame` peuvent contenir des `NaN` *(Not a Number)* correspondant soit à des valeurs manquantes, soit à des échecs lors de calculs.

La plupart des opérations sur les `DataFrame` ignore ces valeurs :

```
print(f'{df["Salary"].mean():.2f}')
print(f'{df["Salary"].sum() / len(df):.2f}')
```

```
4842684.11
4715801.55
```

La différence entre les deux valeurs s'explique par le fait que les fonctions `mean()` et `sum()` ignorent toutes les deux les lignes contant des `NaN` mais pas la fonction `len` qui renvoie le nombre de lignes de la `DataFrame` indépendamment des valeurs de celles-ci.

Il est en général préférable de ne pas conserver de NaN dans ses `DataFrame` pour éviter des incohérences de ce type.

La plupart des opérations sur les `DataFrame` ont un paramètre `skipna` permettant d'ignorer ou non les valeurs `nan`:

```
print(df["Salary"].mean())
print(df["Salary"].mean(skipna=False))
```

```
4842684.105381166
nan
```

4.2 Détecter les `NaN`

Pour afficher les lignes dont qui contiennent un `NaN` sur une colonne donnée :

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 458 entries, 0 to 457
Data columns (total 4 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Name     457 non-null    object
 1   Team     457 non-null    object
 2   College  373 non-null    object
 3   Salary   446 non-null    float64
dtypes: float64(1), object(3)
memory usage: 14.4+ KB
```

La colonne « Non-Null Count » montre que les colonnes `College` et `Salary` contiennent des valeurs manquantes.

La méthode `isna` permet de construire un masque pour sélectionner les lignes contenant des valeurs manquantes.

```
df[df["Salary"].isna()]
```

```
                Name                    Team                College  Salary
2       John Holland          Boston Celtics      Boston University     NaN
46       Elton Brand      Philadelphia 76ers                   Duke     NaN
171    Dahntay Jones     Cleveland Cavaliers                   Duke     NaN
264    Jordan Farmar       Memphis Grizzlies                   UCLA     NaN
269     Ray McCallum       Memphis Grizzlies                Detroit     NaN
270   Xavier Munford       Memphis Grizzlies           Rhode Island     NaN
273   Alex Stepheson       Memphis Grizzlies                    USC     NaN
350    Briante Weber              Miami Heat  Virginia Commonwealth     NaN
353    Dorell Wright              Miami Heat                    NaN     NaN
397      Axel Toupane          Denver Nuggets                    NaN     NaN
409       Greg Smith  Minnesota Timberwolves           Fresno State     NaN
457              NaN                     NaN                    NaN     NaN
```

Dans l'exemple précédent, la méthode `isna` est utilisée pour sélectionner les éléments manquant dans une colonne. Il est possible de l'appliquer directement à une `DataFrame`:

```
df.isna()
```

```
      Name   Team  College  Salary
0    False  False    False   False
1    False  False    False   False
2    False  False    False    True
3    False  False    False   False
4    False  False     True   False
..     ...    ...      ...     ...
453  False  False    False   False
454  False  False     True   False
455  False  False     True   False
456  False  False    False   False
457   True   True     True    True

[458 rows x 4 columns]
```

Il faut alors utiliser une fonction d'aggrégation pour convertir cette `DataFrame` en un masque (rappel: un masque et une **colonne** contenant des booléans). On peut, par exemple, utiliser la fonction `any` pour détecter les lignes contenant au moins un élément manquant.

```
df[df.isna().any(axis=1)]
```

```
                 Name            Team             College      Salary
2        John Holland  Boston Celtics  Boston University         NaN
4       Jonas Jerebko  Boston Celtics                NaN   5000000.0
5        Amir Johnson  Boston Celtics                NaN  12000000.0
15   Bojan Bogdanovic   Brooklyn Nets                NaN   3425510.0
20     Sergey Karasev   Brooklyn Nets                NaN   1599840.0
..                ...             ...                ...         ...
447       Rudy Gobert       Utah Jazz                NaN   1175880.0
450        Joe Ingles       Utah Jazz                NaN   2050000.0
454         Raul Neto       Utah Jazz                NaN    900000.0
455      Tibor Pleiss       Utah Jazz                NaN   2900000.0
457               NaN             NaN                NaN         NaN

[94 rows x 4 columns]
```

4.3 Gestion des `NaN`

La manière la plus simple de traiter les `NaN` est de supprimer toutes les lignes contenant un `NaN` :

```
#syntaxe alternative: df.dropna(inplace=True)
df = df.dropna()
df
```

```
              Name            Team         College     Salary
0    Avery Bradley  Boston Celtics           Texas  7730337.0
1      Jae Crowder  Boston Celtics       Marquette  6796117.0
3      R.J. Hunter  Boston Celtics   Georgia State  1148640.0
6    Jordan Mickey  Boston Celtics             LSU  1170960.0
```

(continues on next page)

(continued from previous page)

```
7     Kelly Olynyk  Boston Celtics        Gonzaga  2165160.0
..             ...             ...            ...        ...
449     Rodney Hood       Utah Jazz           Duke  1348440.0
451   Chris Johnson       Utah Jazz         Dayton   981348.0
452      Trey Lyles       Utah Jazz       Kentucky  2239800.0
453    Shelvin Mack       Utah Jazz         Butler  2433333.0
456     Jeff Withey       Utah Jazz         Kansas   947276.0

[364 rows x 4 columns]
```

Comme toutes les méthodes de `pandas` (ou presque), la méthode `dropna` crée une nouvelle `DataFrame` sans modifier la `DataFrame` initiale (sauf si l'argument `inplace` est `True`). Il est donc nécessaire de (ré-)affecter le résultat de l'appel comme dans l'exemple précédent pour garder trace de la modification.

Une solution alternative consiste à remplacer les `NaN` par une valeur données (ici : "No College"). L'intérêt de cette manipulation est de garantir que la ligne ne sera plus ignorée dans les traitements qui suivront.

```
df.fillna("No College").head(10)
```

```
               Name            Team         College     Salary
0     Avery Bradley  Boston Celtics           Texas  7730337.0
1       Jae Crowder  Boston Celtics       Marquette  6796117.0
3       R.J. Hunter  Boston Celtics   Georgia State  1148640.0
6     Jordan Mickey  Boston Celtics             LSU  1170960.0
7      Kelly Olynyk  Boston Celtics         Gonzaga  2165160.0
8      Terry Rozier  Boston Celtics      Louisville  1824360.0
9      Marcus Smart  Boston Celtics  Oklahoma State  3431040.0
10  Jared Sullinger  Boston Celtics      Ohio State  2569260.0
11    Isaiah Thomas  Boston Celtics      Washington  6912869.0
12      Evan Turner  Boston Celtics      Ohio State  3425510.0
```

L'appel précédent modifie tous les `NaN` de la `DataFrame`. Il possible de n'appliquer le remplacement qu'à une seule colonne :

```
df["College"] = df["College"].fillna("No College")
df.head(10)
```

```
               Name            Team         College     Salary
0     Avery Bradley  Boston Celtics           Texas  7730337.0
1       Jae Crowder  Boston Celtics       Marquette  6796117.0
3       R.J. Hunter  Boston Celtics   Georgia State  1148640.0
6     Jordan Mickey  Boston Celtics             LSU  1170960.0
7      Kelly Olynyk  Boston Celtics         Gonzaga  2165160.0
8      Terry Rozier  Boston Celtics      Louisville  1824360.0
9      Marcus Smart  Boston Celtics  Oklahoma State  3431040.0
10  Jared Sullinger  Boston Celtics      Ohio State  2569260.0
11    Isaiah Thomas  Boston Celtics      Washington  6912869.0
12      Evan Turner  Boston Celtics      Ohio State  3425510.0
```

Remarquons que dans l'exemple précédent, la `DataFrame` initiale est modifiée.

Un cas d'utilisation classique est de remplacer les valeurs manquantes par la valeur moyenne de la colonne comme dans l'exemple suivant :

```
df["Salary"] = df["Salary"].fillna(df["Salary"].mean())
df.head(10)
```

```
                Name            Team          College     Salary
0      Avery Bradley  Boston Celtics            Texas  7730337.0
1        Jae Crowder  Boston Celtics        Marquette  6796117.0
3        R.J. Hunter  Boston Celtics    Georgia State  1148640.0
6      Jordan Mickey  Boston Celtics              LSU  1170960.0
7       Kelly Olynyk  Boston Celtics          Gonzaga  2165160.0
8       Terry Rozier  Boston Celtics       Louisville  1824360.0
9       Marcus Smart  Boston Celtics   Oklahoma State  3431040.0
10  Jared Sullinger  Boston Celtics       Ohio State  2569260.0
11     Isaiah Thomas  Boston Celtics       Washington  6912869.0
12       Evan Turner  Boston Celtics       Ohio State  3425510.0
```

```
# recharge la dataframe initiale : celle-ci a été modifiée par les dernières␣
↪opérations et ne contient plus de NaN

df = pd.read_csv("nba.csv")[["Name", "Team", "College", "Salary"]]
```

La méthode `fillna` peut également être utilisée pour remplacer les `NaN` en « propageant » les valeurs des lignes voisines. L'argument `method` permet de remplacer les `NaN` :

- par la valeur de la dernière ligne ne contenant pas un `NaN` (`ffill` pour `forward fill`)
- par la valeur de la prochaine ligne ne contenant pas un `NaN` (`bfill` pour `backward fill`)

```
df.ffill().head(10)
```

```
            Name            Team            College      Salary
0  Avery Bradley  Boston Celtics              Texas   7730337.0
1    Jae Crowder  Boston Celtics          Marquette   6796117.0
2   John Holland  Boston Celtics  Boston University   6796117.0
3    R.J. Hunter  Boston Celtics      Georgia State   1148640.0
4  Jonas Jerebko  Boston Celtics      Georgia State   5000000.0
5   Amir Johnson  Boston Celtics      Georgia State  12000000.0
6  Jordan Mickey  Boston Celtics                LSU   1170960.0
7   Kelly Olynyk  Boston Celtics            Gonzaga   2165160.0
8   Terry Rozier  Boston Celtics         Louisville   1824360.0
9   Marcus Smart  Boston Celtics     Oklahoma State   3431040.0
```

Dans ce dernier exemple, les valeurs de `College` des lignes 4 et 5 ont été fixées à `LSU`, la prochaine valeur de la colonne qui n'est pas `NaN`. De la même manière, le salaire de la ligne 2 a été fixé à 1148640.0.

4.3.1 Interpolation

Il est également possible de remplacer les `NaN` en interpolant les valeurs. Considérons l'exemple suivant :

```
df = pd.DataFrame({"A":[12, 4, 5, None, 1],
                   "B":[None, 2, 54, 3, None],
                   "C":[20, 16, None, 3, 8],
                   "D":[14, 3, None, None, 6]})
df
```

```
      A     B     C     D
0  12.0   NaN  20.0  14.0
1   4.0   2.0  16.0   3.0
2   5.0  54.0   NaN   NaN
```

(continues on next page)

(continued from previous page)

```
3    NaN    3.0    3.0    NaN
4    1.0    NaN    8.0    6.0
```

```
df.interpolate(method ='linear')
```

```
      A     B     C     D
0  12.0   NaN  20.0  14.0
1   4.0   2.0  16.0   3.0
2   5.0  54.0   9.5   4.0
3   3.0   3.0   3.0   5.0
4   1.0   3.0   8.0   6.0
```

La méthode `interpolate` permet de faire une interpolation (ici linéaire, voir la documentation pour les différents types d'interpolation qu'il est possible de réaliser) pour remplacer les valeurs manquantes : la fonction va considérer la valeur de la colonne juste avant un `NaN` et la valeur juste après le `NaN` et faire une interpolation en supposant que les lignes sont espacées de manière régulière.

Cette méthode est surtout utile lorsque l'on manipule des séries temporelles.

4.3.2 Comparaison des méthodes d'interpolation sur des séries temporelles

Génère une `DataFrame` artificielle : cours des deux actions en fonction du temps. La valeur des actions est déterminée selon une sinusoide et seule la moitié des valeurs est observée.

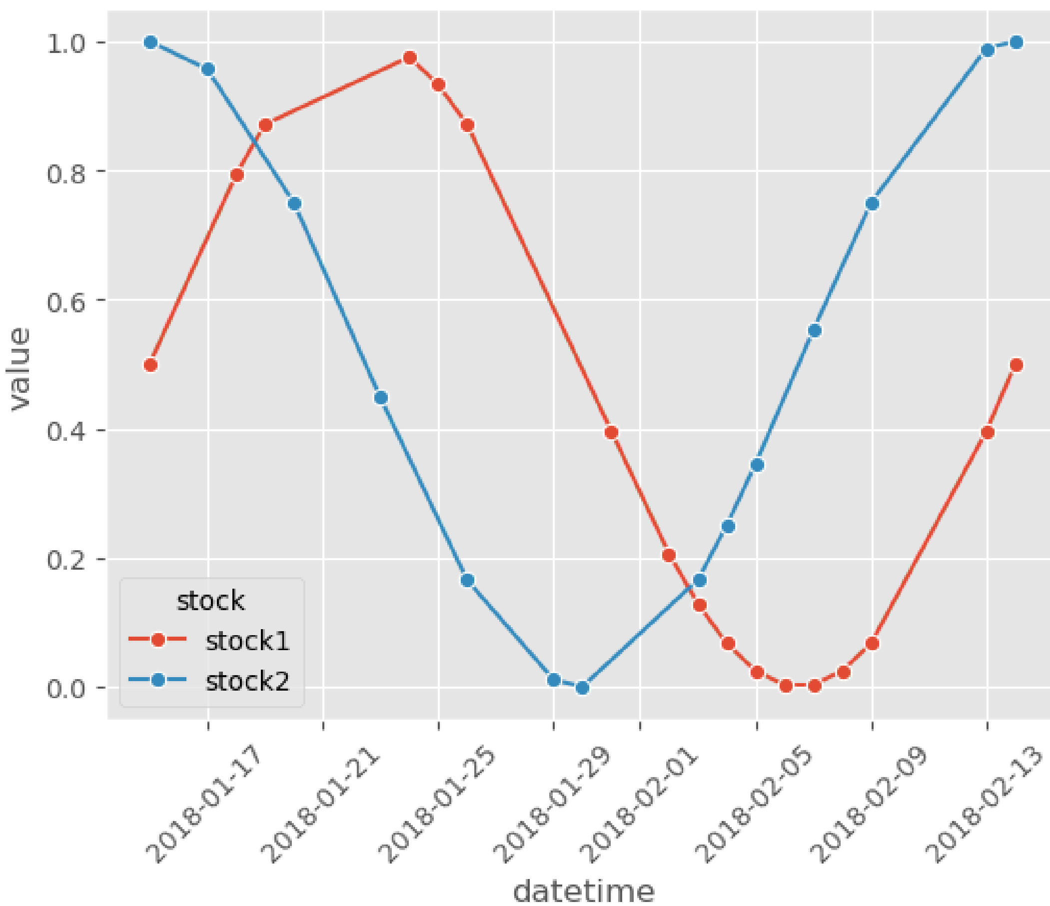
```
data = {'datetime': pd.DatetimeIndex(pd.date_range(start='1/15/2018', end='02/14/2018
↪', freq='D').append(pd.date_range(start='1/15/2018', end='02/14/2018', freq='D'))),
        'stock' : ['stock1' for i in range(31)] + ['stock2' for i in range(31)],
        'value':  [0.5 + 0.5 * np.sin(2 * np.pi / 30 * i) for i in range(31)]\
                   + [0.5 + 0.5 * np.cos(2 * np.pi / 30 * i) for i in range(31)]}
full_df = pd.DataFrame(data, columns = ['datetime', 'stock', 'value'])

# Randomly drop half the values
random.seed(42)
observed_df = full_df.drop(random.sample(range(full_df.shape[0]), k=full_df.shape[0] /
↪/ 2))

# use temporal indexes rather than row numbers --> easy to find out which values are␣
↪missing
observed_df.index = observed_df['datetime']
```

4.3.3 Visualisation

```
ax = sns.lineplot(data=observed_df,
                  x="datetime",
                  y="value",
                  hue="stock",
                  marker="o")
ax.set_xticklabels(ax.get_xticklabels(), rotation=45)
# a scatter plot would be better, but there is a bug in seaborn: we are not able to␣
↪use a scatterplot with DateTimeIndex
```

```
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71728/3300769383.py:6:
↪UserWarning: set_ticklabels() should only be used with a fixed number of ticks,
↪i.e. after set_ticks() or using a FixedLocator.
  ax.set_xticklabels(ax.get_xticklabels(), rotation=45)
```

```
[Text(17548.0, 0, '2018-01-17'),
 Text(17552.0, 0, '2018-01-21'),
 Text(17556.0, 0, '2018-01-25'),
 Text(17560.0, 0, '2018-01-29'),
 Text(17563.0, 0, '2018-02-01'),
 Text(17567.0, 0, '2018-02-05'),
 Text(17571.0, 0, '2018-02-09'),
 Text(17575.0, 0, '2018-02-13')]
```



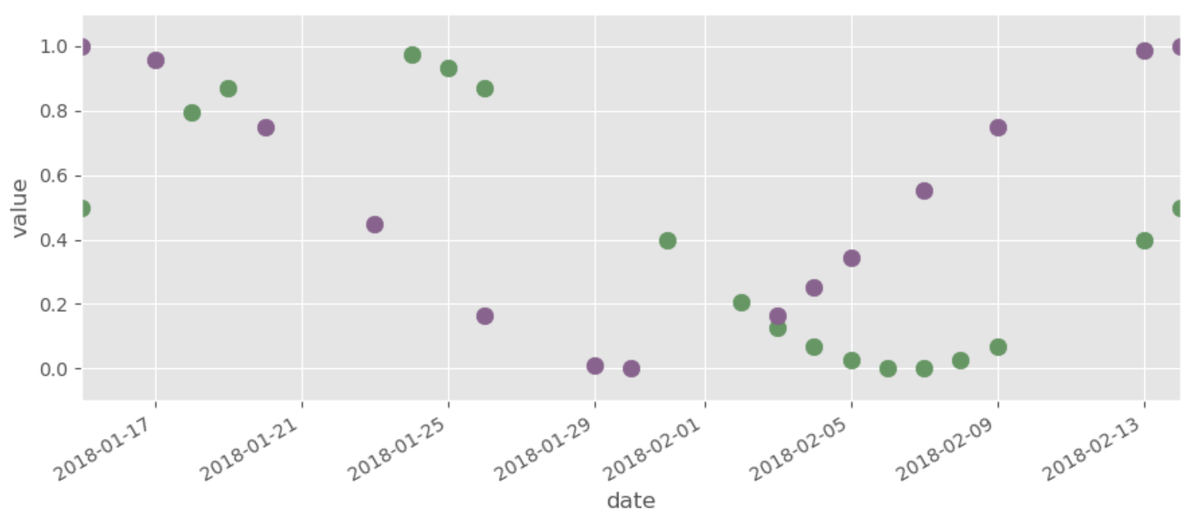
Plot dataset using low-level functions (but with a fine-grained control)

```
params = {'legend.fontsize' : 'large',
          'figure.figsize': (9,4),
          'axes.labelsize': 'large',
          'xtick.labelsize': 'medium',
          'ytick.labelsize': 'medium'}
plt.rcParams.update(params)

colors = {"stock1": sns.color_palette("Set1", n_colors=8, desat=.5)[2],\
```

(continues on next page)

(continued from previous page)

```
          "stock2": sns.color_palette("Set1", n_colors=8, desat=.5)[3]}

fig, ax = plt.subplots()

for stock in ["stock1", "stock2"]:
    ax.scatter(pd.DatetimeIndex(observed_df[observed_df.stock == stock]['datetime']),
               observed_df[observed_df.stock == stock]['value'],
               color=colors[stock],
               s=80)

ax.set_xlabel("date")
ax.set_ylabel("value")

fig.autofmt_xdate()
ax.set_xlim(min(pd.DatetimeIndex(observed_df['datetime'])), max(pd.
↪DatetimeIndex(observed_df['datetime'])))
ax.set_ylim([-0.1, 1.1])

plt.tight_layout()
#plt.savefig('interpolating-timeseries-p1-pandas-fig1.png')
```



Drop `stock2`: we do not want to interpolate data for different stocks.

```
observed_df = observed_df[observed_df["stock"] == "stock1"]
observed_df = observed_df.drop("datetime", axis=1)
observed_df
```

```
              stock     value
datetime
2018-01-15   stock1  0.500000
2018-01-18   stock1  0.793893
2018-01-19   stock1  0.871572
2018-01-24   stock1  0.975528
2018-01-25   stock1  0.933013
2018-01-26   stock1  0.871572
2018-01-31   stock1  0.396044
2018-02-02   stock1  0.206107
2018-02-03   stock1  0.128428
```

(continues on next page)

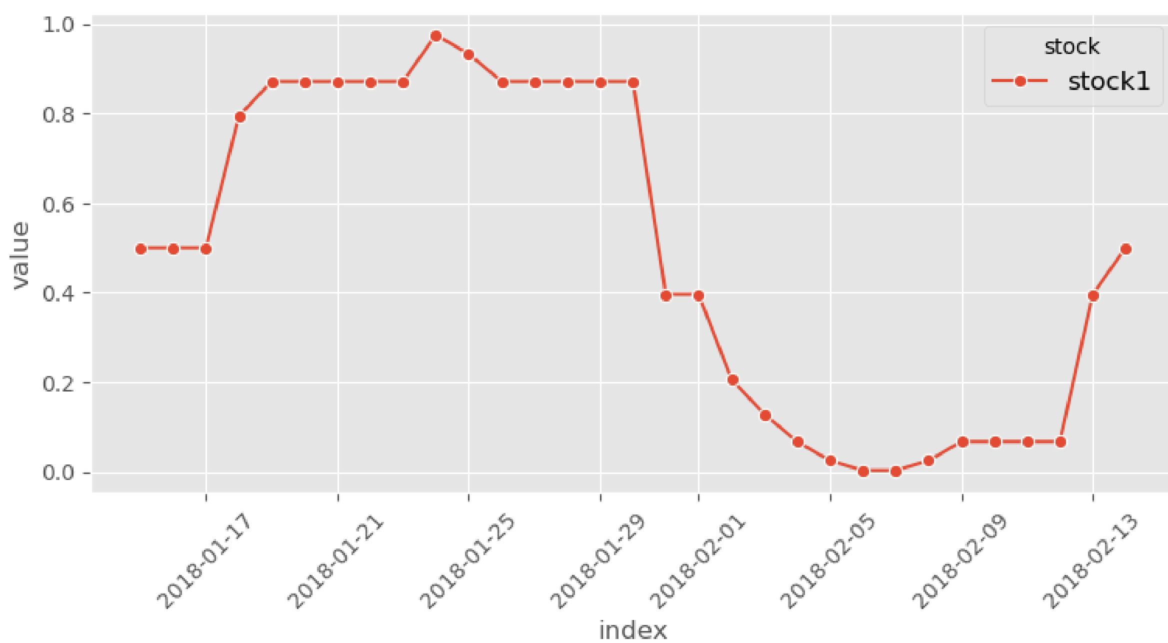
(continued from previous page)

```
2018-02-04  stock1  0.066987
2018-02-05  stock1  0.024472
2018-02-06  stock1  0.002739
2018-02-07  stock1  0.002739
2018-02-08  stock1  0.024472
2018-02-09  stock1  0.066987
2018-02-13  stock1  0.396044
2018-02-14  stock1  0.500000
```

```
ffill_df = observed_df.reindex(pd.DatetimeIndex(pd.date_range(start='1/15/2018', end=
↪'02/14/2018', freq='D')), fill_value=float("NaN"))
ffill_df["value"] = ffill_df["value"].ffill()

ffill_df = ffill_df.reset_index()
ffill_df["stock"] = "stock1"
ax = sns.lineplot(data=ffill_df, x="index", y="value", hue="stock", marker="o")
ax.set_xticklabels(ax.get_xticklabels(), rotation=45)
```

```
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71728/762916674.py:7:␣
↪UserWarning: set_ticklabels() should only be used with a fixed number of ticks,␣
↪i.e. after set_ticks() or using a FixedLocator.
  ax.set_xticklabels(ax.get_xticklabels(), rotation=45)
```

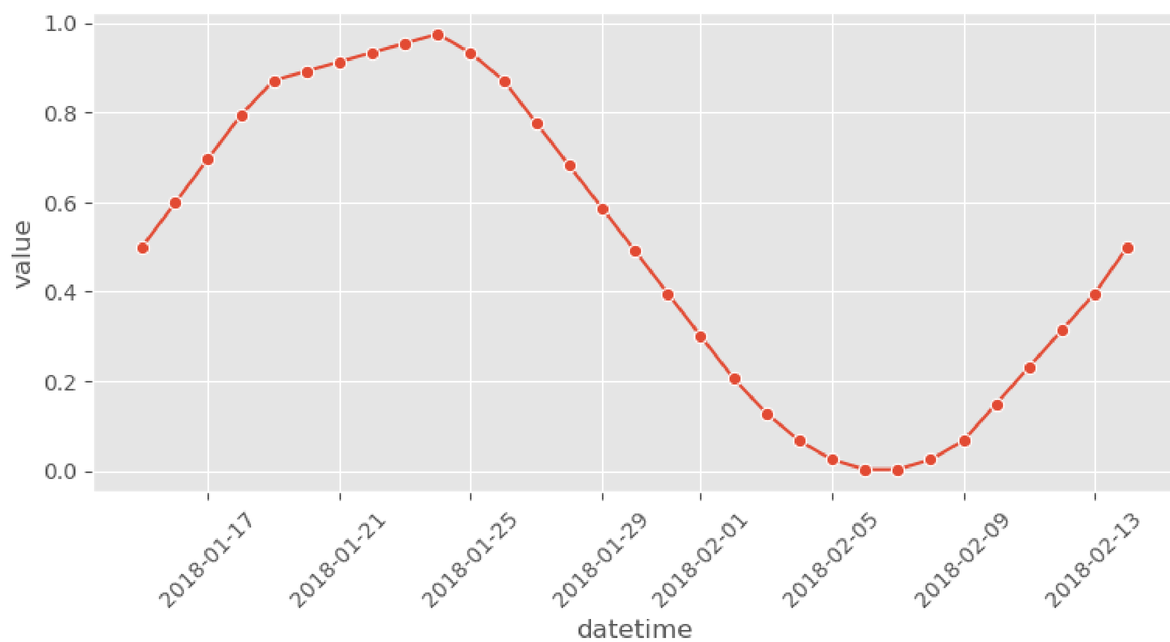
```
[Text(17548.0, 0, '2018-01-17'),
 Text(17552.0, 0, '2018-01-21'),
 Text(17556.0, 0, '2018-01-25'),
 Text(17560.0, 0, '2018-01-29'),
 Text(17563.0, 0, '2018-02-01'),
 Text(17567.0, 0, '2018-02-05'),
 Text(17571.0, 0, '2018-02-09'),
 Text(17575.0, 0, '2018-02-13')]
```



```
df_interpol = observed_df.reindex(pd.DatetimeIndex(pd.date_range(start='1/15/2018',␣
↪end='02/14/2018', freq='D')), fill_value=float("NaN"))

df_interpol['value'] = df_interpol['value'].interpolate()
df_interpol["datetime"] = df_interpol.index
ax = sns.lineplot(data=df_interpol, x="datetime", y="value", marker="o")
ax.set_xticklabels(ax.get_xticklabels(), rotation=45)
```

```
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71728/2855435495.py:6:␣
↪UserWarning: set_ticklabels() should only be used with a fixed number of ticks,␣
↪i.e. after set_ticks() or using a FixedLocator.
  ax.set_xticklabels(ax.get_xticklabels(), rotation=45)
```

```
[Text(17548.0, 0, '2018-01-17'),
 Text(17552.0, 0, '2018-01-21'),
 Text(17556.0, 0, '2018-01-25'),
 Text(17560.0, 0, '2018-01-29'),
 Text(17563.0, 0, '2018-02-01'),
 Text(17567.0, 0, '2018-02-05'),
 Text(17571.0, 0, '2018-02-09'),
 Text(17575.0, 0, '2018-02-13')]
```



4.4 Case Study: Aggregating Book Sales thanks to the `fillna` method

Warning

In this example, we will use:

- merge `DataFrames`

- use methods to deal with `nan` values
- use string manipulation methods

```
from pathlib import Path

import pandas as pd
```

We have two files describing book sales in different countries. The first file `sales1.csv` contains, for each title, the number of books sold and the royalties collected per title. Amounts are expressed in USD.

The second file `sales2.csv` shows the total amount of royalties collected for each title. The amounts are expressed in different currencies (the indication can be found at the end of each group of books).

The aim of the exercise is to find the total amount of royalties collected for each book and each currency.

```
if not Path("sales.zip").is_file():
    !curl -L bit.ly/3qJncJ1 -o sales.zip
    !unzip sales.zip
```

```
sales1 = pd.read_csv("sales1.csv")
sales2 = pd.read_csv("sales2.csv")
```

For the first `DataFrame`, we "just" have to compute our gains and indicate that our gains are in USD.

```
sales1["gain"] = sales1["Royalty paid"] * sales1["Number sold"]
sales1["currency"] = "USD"
sales1
```

```
                          Book title  Number sold  Sales price  Royalty paid  \
0             The Bricklayer's Bible            8         2.99          0.55
1                           Swimrand            2         1.99          0.35
2  Pining For The Fisheries of Yore           28         2.99          0.55
3                  The Duck Goes Here           34         2.99          0.55
4        The Tower Commission Report            4        11.50          4.25

   gain currency
0   4.4      USD
1   0.7      USD
2  15.4      USD
3  18.7      USD
4  17.0      USD
```

For the second `DataFrame`, we have to:

- extract the currency when it is avalaible (currency is indicated between parentheses);
- "propagate" the currency information to the lines it describes;
- remove lines that do not correspond to a book (these are the lines with no title or unit sold information);

```
sales2["currency"] = sales2["Title"].str.extract("\((.*)\)")
sales2["currency"] = sales2["currency"].bfill()
sales2 = sales2.dropna(subset=["Title", "Units sold"])
sales2
```

```
<>:1: SyntaxWarning: invalid escape sequence '\('
<>:1: SyntaxWarning: invalid escape sequence '\('
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71731/172900596.py:1:
↪SyntaxWarning: invalid escape sequence '\('
  sales2["currency"] = sales2["Title"].str.extract("\((.*)\)")
```

```
                                   Title  Units sold  List price  Royalty currency
3   Pining for the Fisheries of Yore        80.0        3.50    14.98      USD
4                               Swimrand         1.0        2.99     0.14      USD
5                 The Bricklayer's Bible        17.0        3.50     5.15      USD
6                     The Duck Goes Here        34.0        2.99     5.78      USD
7            The Tower Commission Report         4.0        9.50     6.20      USD
13  Pining for the Fisheries of Yore        47.0        2.99    11.98      GBP
14                The Bricklayer's Bible        17.0        2.99     3.50      GBP
15           The Tower Commission Report         4.0        6.50     4.80      GBP
21                              Swimrand         8.0        1.99     0.88      EUR
22                    The Duck Goes Here        12.0        1.99     1.50      EUR
```

We can now merge the two `DataFrames` making sure that they use the same column names:

```
df = pd.concat([sales1.rename(columns={
                "Book title": "Title",
                "Number sold": "Units sold",
                "Sales price": "List price",
                "my_money": "Royalty"}),
           sales2], axis=0)
```

Finally, we can gather all information about a given book:

```
df.groupby(["Title", "currency"])["Royalty"].sum().to_frame()
```

```
                                           Royalty
Title                            currency
Pining For The Fisheries of Yore USD          0.00
Pining for the Fisheries of Yore GBP         11.98
                                 USD         14.98
Swimrand                         EUR          0.88
                                 USD          0.14
The Bricklayer's Bible           GBP          3.50
                                 USD          5.15
The Bricklayer’s Bible           USD          0.00
The Duck Goes Here               EUR          1.50
                                 USD          5.78
The Tower Commission Report      GBP          4.80
                                 USD          6.20
```

The result show that we still need to normalize the book titles and the apostrophes:

```
df.assign(Title=lambda x: x["Title"].str.title().str.replace("’", "'"))\
  .groupby(["Title", "currency"])["Royalty"].sum().to_frame()
```

```
                                           Royalty
Title                            currency
Pining For The Fisheries Of Yore GBP         11.98
                                 USD         14.98
Swimrand                         EUR          0.88
```

(continues on next page)

(continued from previous page)

```
                                  USD            0.14
The Bricklayer'S Bible            GBP            3.50
                                  USD            5.15
The Duck Goes Here                EUR            1.50
                                  USD            5.78
The Tower Commission Report       GBP            4.80
                                  USD            6.20
```

CHAPTER

FIVE

MERGING

Data fusion is a key step in many data-analysis workflows, as it allows us to integrate information from multiple sources: datasets collected at regular time intervals, data stored across several files or databases, … A typical case might involve an Excel file containing student information and separate files for each subject listing the students' marks. The main purpose of data fusion is to combine everything into a single, coherent dataframe, ensuring that each student is "matched" with their own marks rather than someone else's.

In `pandas`, there are two broad approaches to achieving this: merging along an axis (adding rows or columns), or merging based on the content of columns. The former is straightforward. The latter relies on assigning a unique identifier to each record in each dataframe, such as a student's first name, surname, and date of birth, and then "aligning" the rows that correspond to the same identifier.

```
import pandas as pd
import numpy as np
```

5.1 Merging DataFrame along axes

The first way of merging dataframes consists of adding rows and columns. For example, let us imagine that we have the two dataframes below, each containing information about different countries:

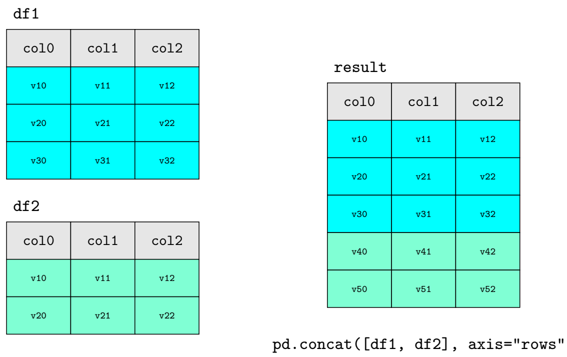
```
countries = pd.DataFrame({
        'country': ['Belgium', 'France', 'Germany', 'Netherlands', 'United Kingdom'],
        'population': [11.3, 64.3, 81.3, 16.9, 64.9],
        'area': [30_510, 67_1308, 35_7050, 41_526, 244_820],
        'capital': ['Brussels', 'Paris', 'Berlin', 'Amsterdam', 'London']
})
countries
```

```
          country  population    area    capital
0         Belgium        11.3   30510   Brussels
1          France        64.3  671308      Paris
2         Germany        81.3  357050     Berlin
3     Netherlands        16.9   41526  Amsterdam
4  United Kingdom        64.9  244820     London
```

```
countries_africa = pd.DataFrame({
    'country': ['Nigeria', 'Rwanda', 'Egypt','Morocco'],
    'capital': ['Abuja', 'Kigali', 'Cairo', 'Rabat'],
    'population': [182.2, 11.3, 94.3, 34.4],
    'area': [923_768, 26_338 , 1_010_408, 710_850],
})
countries_africa
```

```
   country capital  population     area
0  Nigeria   Abuja       182.2   923768
1   Rwanda  Kigali        11.3    26338
2    Egypt   Cairo        94.3  1010408
3  Morocco   Rabat        34.4   710850
```

The `concat` command allows us to place them one after the other in order to obtain a single list of countries, as illustrated in the following figure:




```
pd.concat([countries, countries_africa], axis="rows")
```

```
          country  population     area    capital
0         Belgium        11.3    30510   Brussels
1          France        64.3   671308      Paris
2         Germany        81.3   357050     Berlin
3     Netherlands        16.9    41526  Amsterdam
4  United Kingdom        64.9   244820     London
0         Nigeria       182.2   923768      Abuja
1          Rwanda        11.3    26338     Kigali
2           Egypt        94.3  1010408      Cairo
3         Morocco        34.4   710850      Rabat
```

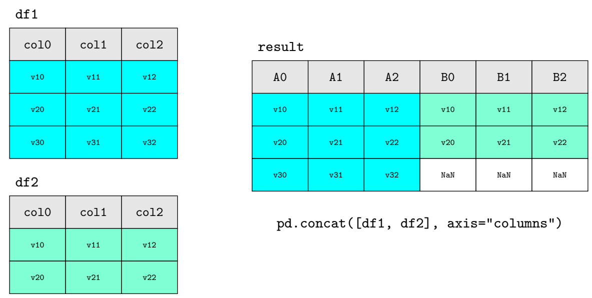
It is worth noting that pandas uses the column name (technically, the index value) to align columns rather than their position. Even if the two dataframes in our example do not have their columns in the same order, pandas ensures that the values remain in the correct column during the merge.

This also becomes evident when a column is missing from one of the dataframes: pandas inserts missing values, as shown in the following example.

```
pd.concat([countries.drop(columns="capital"), countries_africa], axis="rows")
```

```
          country  population     area capital
0         Belgium        11.3    30510     NaN
1          France        64.3   671308     NaN
2         Germany        81.3   357050     NaN
3     Netherlands        16.9    41526     NaN
4  United Kingdom        64.9   244820     NaN
0         Nigeria       182.2   923768   Abuja
1          Rwanda        11.3    26338  Kigali
2           Egypt        94.3  1010408   Cairo
3         Morocco        34.4   710850   Rabat
```

By changing the axis, it is possible to add columns rather than rows, as illustrated in the following figure:



Let's consider a new dataframe:

```
country_economics = pd.DataFrame({
        'country': ['Belgium', 'Germany', 'France'],
        'GDP': [496_477, 2_650_823, 820_726],
        'area': [8.0, 9.9, 6.5]
})

pd.concat([countries.head(3), country_economics], axis="columns")
```

```
   country  population    area   capital  country      GDP  area
0  Belgium        11.3   30510  Brussels  Belgium   496477   8.0
1   France        64.3  671308     Paris  Germany  2650823   9.9
2  Germany        81.3  357050    Berlin   France   820726   6.5
```

The `concat` function (when called with `axis="columns"`) add new columns to an existing `DataFrame` using the index (here the line number) to match the rows. That is why, in the previous example the row describing Germany in the first data frame is "aligned" with the row describing France (they are both the third row of the `DataFrame`).

It is therefore important to take care: contrary to what the illustrative figure might suggest, it is not the position of a row that determines how dataframes are concatenated, but the value associated with their index.

It is possible to change the index of the `DataFrame` to avoid this problem and have the correct semantic:

```
df1 = countries.head(3).set_index("country")
df2 = country_economics.set_index("country")
pd.concat([df1, df2], axis=1)
```

```
         population    area   capital      GDP  area
country
Belgium        11.3   30510  Brussels   496477   8.0
France         64.3  671308     Paris   820726   6.5
Germany        81.3  357050    Berlin  2650823   9.9
```

Note that :

- the `set_index` method does not modify the `DataFrame` and its result must be "saved"
- in this case, the operation is very similar to a `merge` but is not restricted to two `DataFrames`.

5.2 Merging on column values

The second way of merging dataframes generalises the `concat` operation we have just seen, by allowing us to define a criterion for aligning rows from the two dataframes that does not rely solely on the index. This operation, called `merge`, is based on specifying a *key* using the values contained in one or more columns, and aligns (glues) the rows of the two dataframes whose keys match. It is a very powerful operation, reflecting a central concept in databases (particularly SQL): the join between tables.

To illustrate the concept of joins, we will work with two small "toy" dataframes. The first dataframe contains basic information about several countries, including their name and capital city. The second dataframe lists major international events and indicates the country in which each event takes place.

These two datasets allow us to demonstrate different types of matches when merging. In the simplest case, we may have a `1:1` match, where each country appears only once in both dataframes, making the merge straightforward. However, we can also model a many-to-many `(n:m)` relationship, for example if a country hosts several events or if an event is jointly organised by multiple countries. In this situation, merging the dataframes generates multiple rows for each shared match, which provides a clear and intuitive illustration of how joins operate beyond simple one-to-one relationships.

```
countries = pd.DataFrame({
    'country': ['France', 'Germany', 'Japan', 'Brazil', 'Morocco'],
    'capital': ['Paris', 'Berlin', 'Tokyo', 'Brasília', 'Rabat'],
})

events = pd.DataFrame({
    'event': ['Olympics', 'World Cup', 'G7 Summit', 'Expo', 'World Cup', 'Super Bowl
 ↪'],
    'country': ['Japan', 'Brazil', 'Germany', 'France', 'Japan', 'Greece']
})
```

The `merge` function enables us to combine the two dataframes. The `on` parameter is used to specify which column or columns should serve as the key for the merge.

```
pd.merge(countries, events, on="country")
```

```
   country   capital      event
0   France     Paris       Expo
1  Germany    Berlin  G7 Summit
2    Japan     Tokyo   Olympics
3    Japan     Tokyo  World Cup
4   Brazil  Brasília  World Cup
```

The previous command corresponds to an *inner join*: only the keys that appear in both dataframes are retained, and the matching rows from each dataframe are combined. The resulting dataframe contains all the columns from the two merged dataframes, and its number of rows corresponds to the number of rows for which the value in the key column is present in both dataframes.

There are other types of joins that allow us to keep rows whose keys appear in only one of the dataframes. In such cases, the missing values (i.e. those for which the key exists in only one dataframe) are filled with NaN.

With outer joins, all keys are preserved. Starting from the inner join, pandas additionally inserts the keys that appear only in the first dataframe (referred to by convention as the left dataframe) and those that appear only in the second (the right dataframe).

Here is the result of an outer join with our toy example:

```
pd.merge(countries, events, on="country", how="outer")
```

```
   country   capital       event
0   Brazil  Brasília   World Cup
1   France     Paris        Expo
2  Germany    Berlin   G7 Summit
3   Greece       NaN  Super Bowl
4    Japan     Tokyo    Olympics
5    Japan     Tokyo   World Cup
6  Morocco     Rabat         NaN
```

There are two incomplete rows (those with index 3 and 6).

It is possible to keep only the keys from the left dataframe or from the right dataframe by performing a *left join* or a *right join*, respectively as in the following example

```
pd.merge(countries, events, on="country", how="left")
```

```
   country   capital      event
0   France     Paris       Expo
1  Germany    Berlin  G7 Summit
2    Japan     Tokyo   Olympics
3    Japan     Tokyo  World Cup
4   Brazil  Brasília  World Cup
5  Morocco     Rabat        NaN
```

Note that there is only 6 rows and that the last one has no associated event.

5.2.1 "Advanced" merging operation

It is possible to specify different column names in the two dataframe as keys:

```
left = pd.DataFrame({'clé': ['A', 'B', 'C', 'D'],
                     'valeur': np.random.randn(4)})
right = pd.DataFrame({'key': ['B', 'D', 'E', 'F'],
                      'value': np.random.randn(4)})

pd.merge(left, right, how="left", left_on="clé", right_on="key")
```

```
  clé    valeur  key     value
0   A  0.639822  NaN       NaN
1   B  1.659467    B -1.301635
2   C -0.671080  NaN       NaN
3   D -1.113542    D -0.490427
```

The merging criterion can also be defined by the combination of different columns:

```
left = pd.DataFrame({'clé1': ['A', 'A', 'C', 'D'],
                     'clé2': [1, 2, 3, 4],
                     'valeur': np.random.randn(4)})
right = pd.DataFrame({'key1': ['A', 'A', 'E', 'D'],
                      'key2':  [1, 2, 3, 4],
                      'value': np.random.randn(4)})

pd.merge(left, right, how="left", left_on=["clé1", "clé2"], right_on=["key1", "key2"])
```

```
  clé1  clé2    valeur key1  key2     value
0    A     1 -0.994374    A   1.0 -0.509800
1    A     2  0.079090    A   2.0  0.424077
2    C     3 -0.088625  NaN   NaN       NaN
3    D     4 -1.765238    D   4.0 -0.116063
```

It is also possible to perform column transformations "on the fly", by directly specifying operations on the column (here we see once again the consistency of the Pandas API: the different ways of specifying the column are exactly the same as for a `groupby` or any other operation on a column). For example :

```
left = pd.DataFrame({'clé1': ['A', 'A', 'C', 'D'],
                     'valeur': np.random.randn(4)})
right = pd.DataFrame({'key1': ['a', 'b', 'e', 'd'],
                      'value': np.random.randn(4)})

pd.merge(left, right, left_on="clé1", right_on=right["key1"].str.upper())
```

```
  clé1    valeur key1     value
0    A -1.200150    a  0.036707
1    A -0.377007    a  0.036707
2    D -0.781919    d -0.177832
```

5.3 Case Study: Merging DataFrames to Explore Phone Usage by Brand

We will consider three datasets in this exercise:

- `user_usage.csv`, a dataset containing users monthly mobile usage statistics;
- `user_device.csv`, a dataset containing details of an individual "use" of the system, with dates and device information;
- `android_devices.csv`, a dataset with device and manufacturer data, which lists all Android devices and their model code.

We would like to determine if the usage patterns for users differ between different devices. For example, do users using Samsung devices use more call minutes than those using LG devices?

To answer this question, you will draw a box plot of call lengths (in the `outgoing_mins_per_month` column) and of the number of outgoing SMS (in the `outgoing_sms_per_month` column) for the 4 most commons device manufacturers (i.e. the `Retail Branding` column; the most commons manufacturer should be found "automatically"). The graph you obtained should look like: XXX

To do so, you must "connect" the information in:

- `user_usage` and `user_device` to identify the phone that each user is using `use_id`;
- `user_device` and `android_device` to identify the manufacturer of a phone using `device` and `Model` as keys.

You will report the size of the `DataFrames` involved in each joint operation as well as the size of the resulting `DataFrame` and explain the choice of kind of merging you have chosen.

```
from pathlib import Path
```

(continues on next page)

(continued from previous page)

```
import pandas as pd
import seaborn as sns
```

```
if not Path("android_devices.csv").is_file():
    !curl -L https://bit.ly/3EHiPmL -o android.zip
    !unzip android.zip
```

```
user_device = pd.read_csv("user_device.csv")
user_device.head()
```

```
   use_id  user_id platform  platform_version     device  use_type_id
0   22782    26980      ios              10.2  iPhone7,2            2
1   22783    29628  android               6.0    Nexus 5            3
2   22784    28473  android               5.1   SM-G903F            1
3   22785    15200      ios              10.2  iPhone7,2            3
4   22786    28239  android               6.0  ONE E1003            1
```

```
user_usage = pd.read_csv("user_usage.csv")
user_usage.head()
```

```
   outgoing_mins_per_month  outgoing_sms_per_month  monthly_mb  use_id
0                    21.97                    4.82     1557.33   22787
1                  1710.08                  136.88     7267.55   22788
2                  1710.08                  136.88     7267.55   22789
3                    94.46                   35.17      519.12   22790
4                    71.59                   79.26     1557.33   22792
```

```
devices = pd.read_csv("android_devices.csv")
devices.head()
```

```
  Retail Branding Marketing Name    Device                        Model
0             NaN            NaN   AD681H  Smartfren Andromax AD681H
1             NaN            NaN    FJL21                        FJL21
2             NaN            NaN      T31                Panasonic T31
3             NaN            NaN  hws7721g          MediaPad 7 Youth 2
4              3Q        OC1020A   OC1020A                      OC1020A
```

5.3.1 Solution

We will first "connect" each user (in `user_usage`) with their phone information (in `user_device`). To make the output more readable, we will only select those columns that are needed:

```
result = pd.merge(user_usage,
                  user_device[['use_id', 'platform', 'device']],
                  on='use_id',
                  how='inner')
result.head()
```

```
   outgoing_mins_per_month  outgoing_sms_per_month  monthly_mb  use_id  \
0                    21.97                    4.82     1557.33   22787
1                  1710.08                  136.88     7267.55   22788
2                  1710.08                  136.88     7267.55   22789
```

(continues on next page)

(continued from previous page)

```
3                   94.46                    35.17      519.12   22790
4                   71.59                    79.26     1557.33   22792

  platform    device
0  android  GT-I9505
1  android  SM-G930F
2  android  SM-G930F
3  android     D2303
4  android  SM-G361F
```

Note that, by default, `merge` will consider all columns with the same name in the two `DataFrames` as key in the merge and will perform an `inner` join. The previous command could be written:

```
pd.merge(user_usage, user_device[['use_id', 'platform', 'device']]).head()
```

```
   outgoing_mins_per_month  outgoing_sms_per_month  monthly_mb  use_id  \
0                    21.97                    4.82     1557.33   22787
1                  1710.08                  136.88     7267.55   22788
2                  1710.08                  136.88     7267.55   22789
3                    94.46                   35.17      519.12   22790
4                    71.59                   79.26     1557.33   22792

  platform    device
0  android  GT-I9505
1  android  SM-G930F
2  android  SM-G930F
3  android     D2303
4  android  SM-G361F
```

but never forget that “explicit is better than implicit”.

We will then “link” this `DataFrame` with `devices` to be able to link the manufacturer to the phone usage:

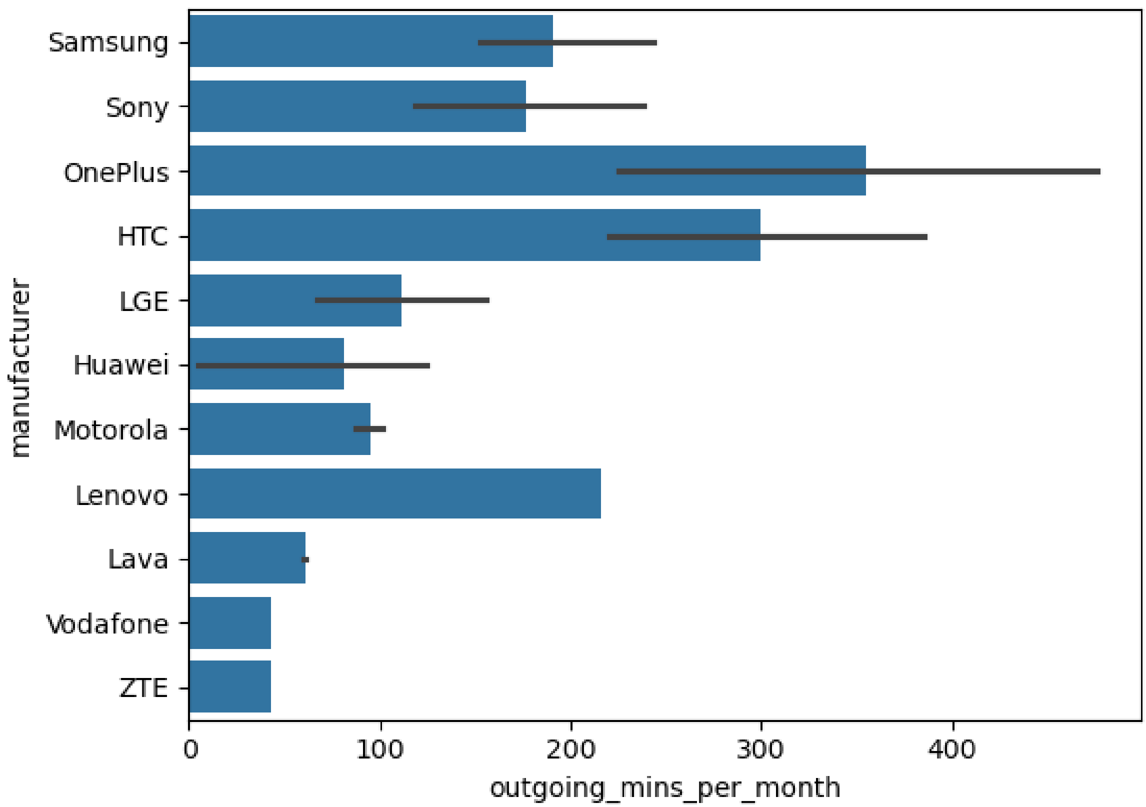
```
devices.rename(columns={"Retail Branding": "manufacturer"}, inplace=True)
result = pd.merge(result,
                  devices[['manufacturer', 'Model']],
                  left_on='device',
                  right_on='Model',
                  how='inner')
result.head()
```

```
   outgoing_mins_per_month  outgoing_sms_per_month  monthly_mb  use_id  \
0                    21.97                    4.82     1557.33   22787
1                  1710.08                  136.88     7267.55   22788
2                  1710.08                  136.88     7267.55   22789
3                    94.46                   35.17      519.12   22790
4                    71.59                   79.26     1557.33   22792

  platform    device manufacturer     Model
0  android  GT-I9505      Samsung  GT-I9505
1  android  SM-G930F      Samsung  SM-G930F
2  android  SM-G930F      Samsung  SM-G930F
3  android     D2303         Sony     D2303
4  android  SM-G361F      Samsung  SM-G361F
```

We can now aggregate use for each manufacturers using, for instance, `seaborn` to compute the values we are interested in:

```
sns.barplot(data=result, x="outgoing_mins_per_month", y="manufacturer")
```

```
<Axes: xlabel='outgoing_mins_per_month', ylabel='manufacturer'>
```



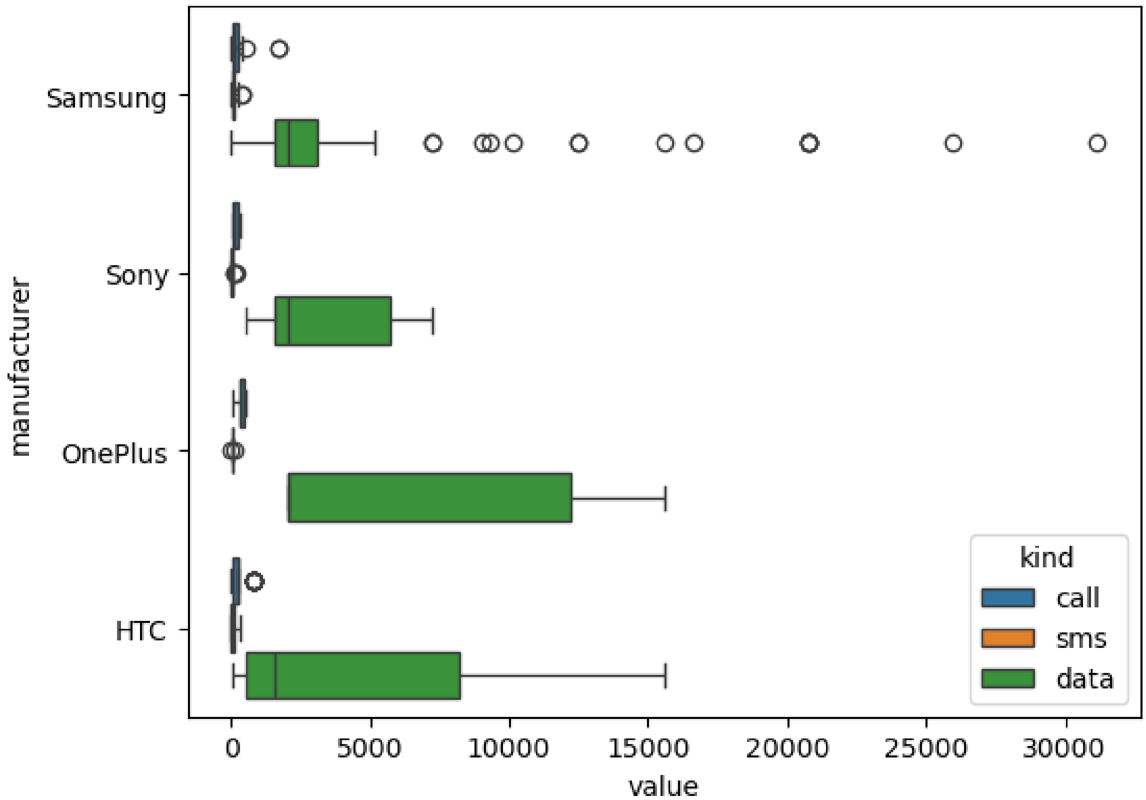
and generate the plot we were ask for:

```
results = result.rename(columns={"outgoing_mins_per_month": "call",
                                 "outgoing_sms_per_month": "sms",
                                 "monthly_mb": "data"})
results.columns
```

```
Index(['call', 'sms', 'data', 'use_id', 'platform', 'device', 'manufacturer',
       'Model'],
      dtype='object')
```

```
results.dropna(inplace=True)
```

```
#results = results.drop("data", axis=1)
results = results[results["manufacturer"].isin(["Samsung", "Sony", "OnePlus", "HTC"])]
ax = sns.boxplot(data=results.melt(id_vars=["manufacturer", "device", "platform",
↪"use_id", "Model"], var_name="kind"),
            y="manufacturer",
            x="value",
           hue="kind")
ax.get_figure().savefig("result.pdf")
```




```
results.melt(id_vars=["manufacturer", "device", "platform", "use_id", "Model"], var_
↪name="kind")
```

```
    manufacturer          device platform  use_id          Model  kind  \
0        Samsung        GT-I9505  android   22787        GT-I9505  call
1        Samsung        SM-G930F  android   22788        SM-G930F  call
2        Samsung        SM-G930F  android   22789        SM-G930F  call
3           Sony           D2303  android   22790           D2303  call
4        Samsung        SM-G361F  android   22792        SM-G361F  call
..           ...             ...      ...     ...             ...   ...
517          HTC  HTC Desire 620  android   23040  HTC Desire 620  data
518      Samsung        SM-G900F  android   23041        SM-G900F  data
519      Samsung        SM-G900F  android   23043        SM-G900F  data
520      Samsung        SM-G900F  android   23044        SM-G900F  data
521      Samsung        SM-G900F  android   23049        SM-G900F  data

       value
0      21.97
1    1710.08
2    1710.08
3      94.46
4      71.59
..       ...
517    74.40
518  5191.12
519  5191.12
520  3114.67
521   519.12
```

(continues on next page)

(continued from previous page)

```
[522 rows x 7 columns]
```

CHAPTER

SIX

RESHAPING DATAFRAME

Reshaping operations allow you to transform rows into columns (and vice versa), for example to create pivot tables. These operations are also used to transform DataFrames to make them easier to work with — for instance, for plotting, as the plot method requires the data to be in columns.

```
import pandas as pd

# reduce the amount of information printed when an exception occurs.
#
# activating this mode is not advised as it makes debugging very difficult
# It is only used here to generate smaller pdf
%xmode minimal
#
# All floats will be displayed with a single digit after the decimal point
pd.options.display.float_format = '{:.1f}'.format
```

```
Exception reporting mode: Minimal
```

```
from pathlib import Path

if not Path("gdp.csv").is_file():
    !curl -L https://short.do/KV2RYs -o gdp.csv
    !curl -L https://short.do/xP2MiT -o supermarket_sales.xlsx
```

6.1 Pivoting

```
df_gdp = pd.read_csv('gdp.csv')[["country", "year", "gdppc"]]
df_gdp.sample(5)
```

```
             country  year   gdppc
672   United Kingdom  2010 36195.9
1330      Mauritania  1900   601.0
1886           Tonga  1930  1042.8
1237        Maldives  1940  1213.9
1207         Moldova  1940  2144.0
```

This DataFrame is in a format known as **long**: it is structured so that each row represents a single observation and each column corresponds to a variable, with repeated measures or categories stacked vertically. Although this format is not particularly readable (for example, it makes it difficult to see trends over time), it is well suited to simple manipulations such as groupby and plot operations, as we will see later.

You can reshape the data into a **wide** `DataFrame`, where each unique value in the `index` column becomes a row and each unique value in the `columns` column becomes a separate column. The corresponding cell values are taken from the `values` column. This wide format is often easier to read and can make patterns and trends clearer.

```
df = df_gdp.pivot(index="country",
                  columns="year",
                  values="gdppc")

df[df.columns[:5]].head(5)
```

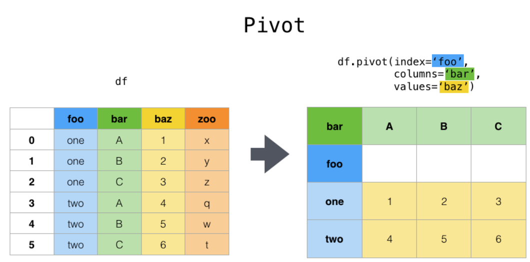
```
year          1880   1890   1900   1910   1920
country
Afghanistan  585.5  635.9  686.4  736.9  731.8
Albania      522.0  598.0  685.0  780.0  861.3
Algeria      819.2  923.4 1027.6 1131.7 1201.2
Angola       533.7  567.3  601.0  628.7  682.6
Argentina   1731.5 2152.0 2756.0 3822.0 3473.0
```

We now have as many columns as there are years in the dataframe:

```
df_gdp["year"].sort_values().unique()
```

```
array([1880, 1890, 1900, 1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980,
       1990, 2000, 2010])
```

Here is a visual explanation of the `pivot` method (reproduced from pandas' official documentation):



The `pivot` function is only **reshaping** the dataframe: it can only "move" data around without modifying values. Here is, for instance, what is happening if we try to `pivot` the data in `supermarket_sales.xlsx` to report the `Total` column with respect to the `Gender` and `City` columns.

```
df_sales = pd.read_excel('supermarket_sales.xlsx')
df_sales.columns
```

```
Index(['Invoice ID', 'Branch', 'City', 'Customer type', 'Gender',
       'Product line', 'Unit price', 'Quantity', 'Tax 5%', 'Total', 'Date',
       'Time', 'Payment', 'cogs', 'gross margin percentage', 'gross income',
       'Rating'],
      dtype='object')
```

```
df_sales.pivot(index="Gender",
               columns="City",
               values="Total")
```

```
ValueError: Index contains duplicate entries, cannot reshape
```

This doesn't work because the operation would produce multiple values for a single cell in the resulting frame; as the error indicates, there are **duplicated** entries in the selected columns.

`pivot` only reshapes data: you must have exactly one value for each index/column pair.

Use `pivot_table` instead and provide an aggregation function to combine the values that map to the same cell (e.g. `"mean"`, `"sum"`, `"max"`):

```
df_wide = df.pivot_table(
    index="index",
    columns="columns",
    values="values",
    aggfunc="mean"    # or "sum", "max", "min", list, etc.
)
```

```
df_sales.pivot_table(index="Gender",
                     columns="City",
                     values="Total",
                     aggfunc='sum')
```

```
City     Mandalay  Naypyitaw  Yangon
Gender
Female    52928.3    61685.5 53269.2
Male      53269.4    48883.2 52931.2
```

You can also manipulate the resulting `DataFrame` by chaining methods, for instance compute the *grand total* for each row of the result of the following operation:

```
df_sales.pivot_table(index="Gender",
                     columns="City",
                     values="Total",
                     aggfunc='sum').sum(axis="rows")
```

```
City
Mandalay     106197.7
Naypyitaw    110568.7
Yangon       106200.4
dtype: float64
```

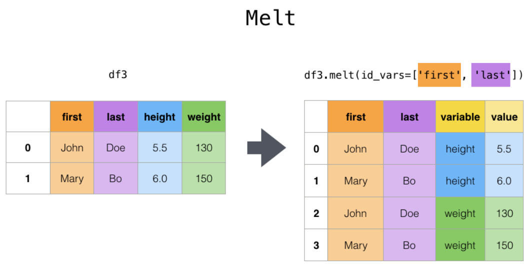
And for the *grand total* for each columns:

```
df_sales.pivot_table(index="Gender",
                     columns="City",
                     values="Total",
                     aggfunc='sum').sum(axis="columns")
```

```
Gender
Female    167882.9
Male      155083.8
dtype: float64
```

6.2 Melting a dataframe

The `melt` method allows you to *unpivot* a `DataFrame`, as illustrated in the figure below (also from the official pandas documentation):



After the DataFrame is melted, one or more columns act as identifier variables — these contain the values that uniquely identify each observation and are not moved during the transformation. All remaining columns are treated as measured variables and are split into two new columns:

- the first holds the original column names (the variable names),
- the second holds the corresponding values.

For example:

```
df = pd.DataFrame(
    {
        "first": ["John", "Mary"],
        "last": ["Doe", "Bo"],
        "height": [5.5, 6.0],
        "weight": [130, 150],
    }
)
df
```

```
  first last  height  weight
0  John  Doe     5.5     130
1  Mary   Bo     6.0     150
```

```
df.melt(id_vars=["first", "last"])
```

```
  first last variable  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0
```

In this example, all columns specified by `id_vars` remain unchanged. The height and weight of each person are reorganised into two separate columns:

- the first contains the name of the variable being described (either height or weight),
- the second contains the corresponding value.

6.3 Exercice

With `titanic` data:

- Make a pivot table with the survival rates for Pclass vs Sex.
- Make a table of the median Fare payed by aged/underaged vs Sex.

CHAPTER

SEVEN

STYLING DATAFRAMES

Create a `DataFrame` with fake values.

```
import pandas as pd
import numpy as np

np.random.seed(24)
df = pd.DataFrame({'A': np.linspace(1, 10, 10)})
df = pd.concat([df, pd.DataFrame(np.random.randn(10, 4), columns=list('BCDE'))],
               axis=1)
df.iloc[3, 3] = np.nan
df.iloc[0, 2] = np.nan

df
```

```
      A         B         C         D         E
0   1.0  1.329212       NaN -0.316280 -0.990810
1   2.0 -1.070816 -1.438713  0.564417  0.295722
2   3.0 -1.626404  0.219565  0.678805  1.889273
3   4.0  0.961538  0.104011       NaN  0.850229
4   5.0  1.453425  1.057737  0.165562  0.515018
5   6.0 -1.336936  0.562861  1.392855 -0.063328
6   7.0  0.121668  1.207603 -0.002040  1.627796
7   8.0  0.354493  1.037528 -0.385684  0.519818
8   9.0  1.686583 -1.325963  1.428984 -2.089354
9  10.0 -0.129820  0.631523 -0.586538  0.290720
```

Chaque dataframe possède un attribut `style` qui permet de contrôler la mise en forme des cellules.

Il existe plusieurs méthodes prédéfinies pour mettre en évidence certains éléments : `highlight_max`, `highlight_min`, `highlight_null`

```
df.style.highlight_min(axis=0)
```

```
<pandas.io.formats.style.Styler at 0x15a01ef90>
```

Il y a également une méthode `background_gradient` :

```
df.style.background_gradient(cmap='Reds', axis=0)
```

```
<pandas.io.formats.style.Styler at 0x15a0cd090>
```

Il est possible de définir ses propres fonctions pour, notamment, mettre en forme les cellules de manière conditionnelle :

- `applymap` permet de mettre en forme des cellules individuellement

- `apply` permet de mettre en forme des colonnes ou des lignes (et donc de définir des conditions sur l'ensemble d'une ligne ou d'une colonne)

```
def color_negative_red(val):
    """
    Takes a scalar and returns a string with
    the css property `'color: red'` for negative
    strings, black otherwise.
    """
    color = 'blue' if val < 0 else 'black'
    return f"color: {color}"

df.style.applymap(color_negative_red)
```

```
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71868/87800094.py:10:
↪FutureWarning: Styler.applymap has been deprecated. Use Styler.map instead.
  df.style.applymap(color_negative_red)
```

```
<pandas.io.formats.style.Styler at 0x12523ad50>
```

```
def color_line_all_positive_green(row):
    color = "green" if all(r > -0.5 for r in row) else "black"
    # attention à bien renvoyer un liste de même taille que l'argument
    return [f"color: {color}"] * len(row)

res = df.style.apply(color_line_all_positive_green, axis=1)
res
```

```
<pandas.io.formats.style.Styler at 0x15a0cd1d0>
```

Il est possible d'exporter le résultat (valeur + mise en forme) soit dans un fichier excel, soit dans un fichier HTML.

Attention : la méthode est bien appliqué à res (df.style) et non à df. Les méthodes `to_*` appliquées à `df` ne permettent d'exporter que les valeurs (il y a donc plus de format d'export possible).

```
res.to_excel("pouet.xlsx")
```

CHAPTER

EIGHT

MANIPULATION DE DONNÉES TEMPORELLE — INTRODUCTION

Warning

To ensure that the output of the various commands is sufficiently compact, only the first lines of the `DataFrame` are generally displayed (this is why I systematically use the `head` method). In "normal" use, it is useful to display both the beginning and the end of the `DataFrame`, as `jupyter` does by default.

The dataset used in this tutorial can be download from this url

```
import pandas as pd
import numpy as np

# limit the amount of information displayed when an exception occurs
# this is bad practice but useful to generate the pdf
%xmode minimal
```

```
Exception reporting mode: Minimal
```

Pandas offre plusieurs fonctions de « haut niveau » pour manipuler les séries temporelles.

Principale contrainte : l'index de la série doit être une date (c.-à-d. de type `DatetimeIndex`). La création d'une `DataFrame` temporelle peut se faire soit lors du chargement de celle-ci soit en sélectionnant explicitement l'index

8.1 Création d'une DataFrame temporelle

8.1.1 Méthode directe

Lors de la lecture :

- deux options nécessaires : `parse_dates` et `index_col`
- possibilité de définir manuellement le format des dates (avec l'options `converters`)
- vérifier systématiquement le type des données lues pour éviter les mauvaises surprises

```
normal_df = pd.read_csv("tr_eikon_eod_data.csv")
normal_df.head(5)
```

```
         Date     AAPL.O  MSFT.O  INTC.O  AMZN.O    GS.N     SPY     .SPX  \
0  2010-01-04  30.572827  30.950   20.88  133.90  173.08  113.33  1132.99
```

(continues on next page)

(continued from previous page)

```
1  2010-01-05  30.625684  30.960   20.87  134.69  176.14  113.63  1136.52
2  2010-01-06  30.138541  30.770   20.80  132.25  174.26  113.71  1137.14
3  2010-01-07  30.082827  30.452   20.60  130.00  177.67  114.19  1141.69
4  2010-01-08  30.282827  30.660   20.83  133.52  174.31  114.57  1144.98

    .VIX    EUR=     XAU=    GDX     GLD
0  20.04  1.4411  1120.00  47.71  109.80
1  19.35  1.4368  1118.65  48.17  109.70
2  19.16  1.4412  1138.50  49.34  111.51
3  19.06  1.4318  1131.90  49.10  110.82
4  18.13  1.4412  1136.10  49.84  111.37
```

En utilisant les options par défaut de `read_csv` (cellule précédente), les dates ne sont par reconnues et l'index d'une observation correspond au numéro de la ligne dans le fichier CSV. Les données sont stockées dans une `dataframe` « normale ».

```
data = pd.read_csv("tr_eikon_eod_data.csv",
                   parse_dates=True,           # demande la reconnaissance des dates
                   index_col=0)                # utilise la colonne 0 comme index
data.head()
```

```
               AAPL.O  MSFT.O  INTC.O  AMZN.O    GS.N     SPY     .SPX   .VIX  \
Date
2010-01-04  30.572827  30.950   20.88  133.90  173.08  113.33  1132.99  20.04
2010-01-05  30.625684  30.960   20.87  134.69  176.14  113.63  1136.52  19.35
2010-01-06  30.138541  30.770   20.80  132.25  174.26  113.71  1137.14  19.16
2010-01-07  30.082827  30.452   20.60  130.00  177.67  114.19  1141.69  19.06
2010-01-08  30.282827  30.660   20.83  133.52  174.31  114.57  1144.98  18.13

              EUR=     XAU=    GDX     GLD
Date
2010-01-04  1.4411  1120.00  47.71  109.80
2010-01-05  1.4368  1118.65  48.17  109.70
2010-01-06  1.4412  1138.50  49.34  111.51
2010-01-07  1.4318  1131.90  49.10  110.82
2010-01-08  1.4412  1136.10  49.84  111.37
```

La date n'est pas plus stockée dans une colonne comme dans l'exemple précédent, mais considéré comme index. Pour vérifier que la lecture s'est bien passée, on peut vérifier le type de l'index :

```
data.index
```

```
DatetimeIndex(['2010-01-04', '2010-01-05', '2010-01-06', '2010-01-07',
               '2010-01-08', '2010-01-11', '2010-01-12', '2010-01-13',
               '2010-01-14', '2010-01-15',
               ...
               '2017-10-18', '2017-10-19', '2017-10-20', '2017-10-23',
               '2017-10-24', '2017-10-25', '2017-10-26', '2017-10-27',
               '2017-10-30', '2017-10-31'],
              dtype='datetime64[ns]', name='Date', length=1972, freq=None)
```

La reconnaissance des dates étant problématiques (les conventions d'écriture des dates sont différentes suivant les pays), il est nécessaire de vérifier que les dates sont correctement reconnues (en particulier, faire attention aux inversion numéro du jour, numéro du mois)

8.1.2 Après la lecture

La méthode `set_index` permet de considérer une colonne comme index d'une `dataframe`. Il est possible de spécifier explicitement le format des dates en utilisant le paramètre `converters`.

```
df = pd.read_csv("tr_eikon_eod_data.csv",
                 converters={"Date": lambda x: pd.to_datetime(x, format="%Y-%m-%d")})
df.head(5)
```

```
        Date     AAPL.O  MSFT.O  INTC.O  AMZN.O    GS.N     SPY     .SPX  \
0 2010-01-04  30.572827  30.950   20.88  133.90  173.08  113.33  1132.99
1 2010-01-05  30.625684  30.960   20.87  134.69  176.14  113.63  1136.52
2 2010-01-06  30.138541  30.770   20.80  132.25  174.26  113.71  1137.14
3 2010-01-07  30.082827  30.452   20.60  130.00  177.67  114.19  1141.69
4 2010-01-08  30.282827  30.660   20.83  133.52  174.31  114.57  1144.98

    .VIX    EUR=     XAU=    GDX     GLD
0  20.04  1.4411  1120.00  47.71  109.80
1  19.35  1.4368  1118.65  48.17  109.70
2  19.16  1.4412  1138.50  49.34  111.51
3  19.06  1.4318  1131.90  49.10  110.82
4  18.13  1.4412  1136.10  49.84  111.37
```

```
df = df.set_index("Date")
```

Attention, comme toutes / beaucoup d'opération, la méthode `set_index` est sans effets de bord et crée une nouvelle `DataFrame` pluôt que de modifier une `DataFrame` existante. C'est pourquoi il est nécessaire d'assigner le résultat de `set_index` à `df` pour « répercuter » la modification.

8.2 Indexation — Accès aux données

Dans le cas d'une `DataFrame` temporelle (c.-à-d. indexée par le temps), il est possible de faire des accès « intelligents » en sélectionnant des plages de temps continue.

Par exemple, pour accéder à toutes les données de 2016 :

```
data.loc["2016"].head()
```

```
            AAPL.O  MSFT.O  INTC.O  AMZN.O    GS.N       SPY     .SPX   .VIX  \
Date
2016-01-04  105.35   54.80   33.99  636.99  177.14  201.0192  2012.66  20.70
2016-01-05  102.71   55.05   33.83  633.79  174.09  201.3600  2016.71  19.34
2016-01-06  100.70   54.05   33.08  632.65  169.84  198.8200  1990.26  20.59
2016-01-07   96.45   52.17   31.84  607.94  164.62  194.0500  1943.09  24.99
2016-01-08   96.96   52.33   31.51  607.05  163.94  191.9230  1922.03  27.01

              EUR=     XAU=    GDX     GLD
Date
2016-01-04  1.0829  1074.30  14.09  102.89
2016-01-05  1.0746  1077.26  14.02  103.18
2016-01-06  1.0778  1094.30  14.25  104.67
2016-01-07  1.0934  1109.10  14.88  106.15
2016-01-08  1.0929  1103.84  14.52  105.68
```

Cette accès n'est possible que dans le cas des `DataFrame` temporelle : dans le cas d'une `DataFrame` normale, `pandas` chercherait à accéder à une colonne appellée `2016` (ce qui résulterait en une erreur) :

```
normal_df.loc["2016"]
```

```
KeyError: '2016'
```

Autre exemple : sélection de toutes les données de mars 2017. Il s'agit à nouveau de la sélection d'une plage continue

```
data.loc["2017-03"].head(5)
```

```
            AAPL.O  MSFT.O  INTC.O  AMZN.O    GS.N     SPY     .SPX   .VIX  \
Date
2017-03-01  139.79   64.94   35.93  853.08  252.71  239.78  2395.96  12.54
2017-03-02  138.96   64.01   35.91  848.91  251.06  238.27  2381.92  11.81
2017-03-03  139.78   64.25   35.90  849.88  252.89  238.42  2383.12  10.96
2017-03-06  139.34   64.27   35.57  846.61  252.01  237.71  2375.31  11.24
2017-03-07  139.52   64.40   35.80  846.02  250.90  237.00  2368.39  11.45

              EUR=       XAU=    GDX     GLD
Date
2017-03-01  1.0546  1248.8600  22.98  119.06
2017-03-02  1.0506  1234.7200  21.93  117.58
2017-03-03  1.0620  1234.2200  22.20  117.51
2017-03-06  1.0578  1225.5600  21.64  116.72
2017-03-07  1.0565  1215.5699  21.51  115.78
```

Comme pour les listes, il est possible d'utiliser des *slices* pour indiquer une date de début et une date de fin. Si un de ces éléments est manquant, le slice commencera à la première date de la `DataFrame` (quand le premier élément n'est pas spécifié) et se terminera à la dernière date de la `DataFrame` (quand le dernier élément n'est pas spécifié).

```
data.loc["2015-03":"2016-03"].head(5)
```

```
            AAPL.O  MSFT.O  INTC.O   AMZN.O    GS.N       SPY     .SPX   .VIX  \
Date
2015-03-02  129.09  43.880  34.060  385.655  191.79  211.9900  2117.39  13.04
2015-03-03  129.36  43.280  34.095  384.610  191.27  211.1200  2107.78  13.86
2015-03-04  128.54  43.055  34.120  382.720  189.67  210.2301  2098.53  14.23
2015-03-05  126.41  43.110  33.730  387.830  190.08  210.4600  2101.04  14.04
2015-03-06  126.60  42.360  33.190  380.090  186.91  207.5000  2071.26  15.20

              EUR=     XAU=    GDX     GLD
Date
2015-03-02  1.1183  1206.70  20.76  115.68
2015-03-03  1.1174  1203.31  20.38  115.47
2015-03-04  1.1077  1199.45  20.04  115.11
2015-03-05  1.1028  1198.20  20.08  115.00
2015-03-06  1.0843  1166.72  18.58  111.86
```

8.3 Pour sélectionner des plages de temps non continues

Il est possible de sélectionner des plages de temps non continues en construisant un masque à partir de l'index. L'index possède des attributs qui permettent d'accéder à une information de la date (`year` pour l'année, `month` pour le mois, `day` pour le jour, `weekday` pour le numéro du jour dans la semaine, ...) et d'utiliser celle-ci, par exemple, dans un test d'égalité

Par exemple : pour sélectionner les données de tous les mardis :

```
data.index.weekday
```

```
Index([0, 1, 2, 3, 4, 0, 1, 2, 3, 4,
       ...
       2, 3, 4, 0, 1, 2, 3, 4, 0, 1],
      dtype='int32', name='Date', length=1972)
```

```
data[data.index.weekday == 2].head(5)
```

```
              AAPL.O  MSFT.O  INTC.O  AMZN.O    GS.N     SPY     .SPX   .VIX  \
Date
2010-01-06  30.138541  30.770   20.80  132.25  174.26  113.71  1137.14  19.16
2010-01-13  30.092827  30.350   20.96  129.11  169.07  114.62  1145.68  17.85
2010-01-20  30.246398  30.585   21.08  125.78  167.79  113.89  1138.04  18.68
2010-01-27  29.697685  29.670   20.24  122.75  151.50  109.83  1097.50  23.14
2010-02-03  28.461400  28.630   19.68  119.10  157.23  109.83  1097.28  21.60

              EUR=     XAU=    GDX      GLD
Date
2010-01-06  1.4412  1138.50  49.34  111.510
2010-01-13  1.4510  1138.40  48.86  111.540
2010-01-20  1.4101  1111.30  45.73  108.940
2010-01-27  1.4017  1086.35  42.77  106.528
2010-02-03  1.3898  1109.25  42.57  108.700
```

Pour sélectionner toutes les données de mars :

```
data[data.index.month == 3].head(3)
```

```
              AAPL.O  MSFT.O   INTC.O  AMZN.O    GS.N     SPY     .SPX  \
Date
2010-03-01  29.855684   29.02  20.8700  124.54  156.54  111.89  1115.71
2010-03-02  29.835684   28.46  20.6975  125.53  158.75  112.20  1118.31
2010-03-03  29.904256   28.46  20.5200  125.89  157.72  112.30  1118.79

             .VIX    EUR=     XAU=     GDX     GLD
Date
2010-03-01  19.26  1.3560  1116.95  44.640  109.43
2010-03-02  19.06  1.3604  1134.25  45.510  111.02
2010-03-03  18.83  1.3698  1139.10  46.364  111.63
```

```
data[(data.index.month == 3) & (data.index.weekday == 2)].head(3)
```

```
              AAPL.O  MSFT.O   INTC.O  AMZN.O    GS.N     SPY     .SPX  \
Date
2010-03-03  29.904256   28.46  20.5200  125.89  157.72  112.30  1118.79
```

(continues on next page)

(continued from previous page)

```
2010-03-10  32.119968   28.97  21.1900  130.51  171.94  114.97  1145.61
2010-03-17  32.017111   29.63  22.2425  131.34  176.64  117.10  1166.21

             .VIX    EUR=     XAU=     GDX     GLD
Date
2010-03-03  18.83  1.3698  1139.10  46.364  111.63
2010-03-10  18.57  1.3659  1107.80  44.960  108.47
2010-03-17  16.91  1.3737  1124.05  46.270  109.59
```

8.4 Opération sur les dates

À partir des structures de données de pandas : on est limité à des opérations sur les jours (addition / soustraction)

```
ts = pd.Timestamp('2016-12-19')
ts
```

```
Timestamp('2016-12-19 00:00:00')
```

```
ts + pd.Timedelta('5 days')
```

```
Timestamp('2016-12-24 00:00:00')
```

La bibliothèque `dateutil` offre des opérations de plus haut niveau. Attention à la sémantique qui dépend de l'application (est-ce que +1 mois veux dire ajouter 30 jours ou se placer à la même date le mois suivant, que doit-on faire si on est le dernier jour du mois ?)

```
from dateutil.relativedelta import relativedelta
from datetime import date

date.today() + relativedelta(months=+6)
```

```
datetime.date(2026, 5, 7)
```

```
date(2010,12,31) + relativedelta(months=+2)
```

```
datetime.date(2011, 2, 28)
```

```
date(2010,12,31) + relativedelta(months=+2)
```

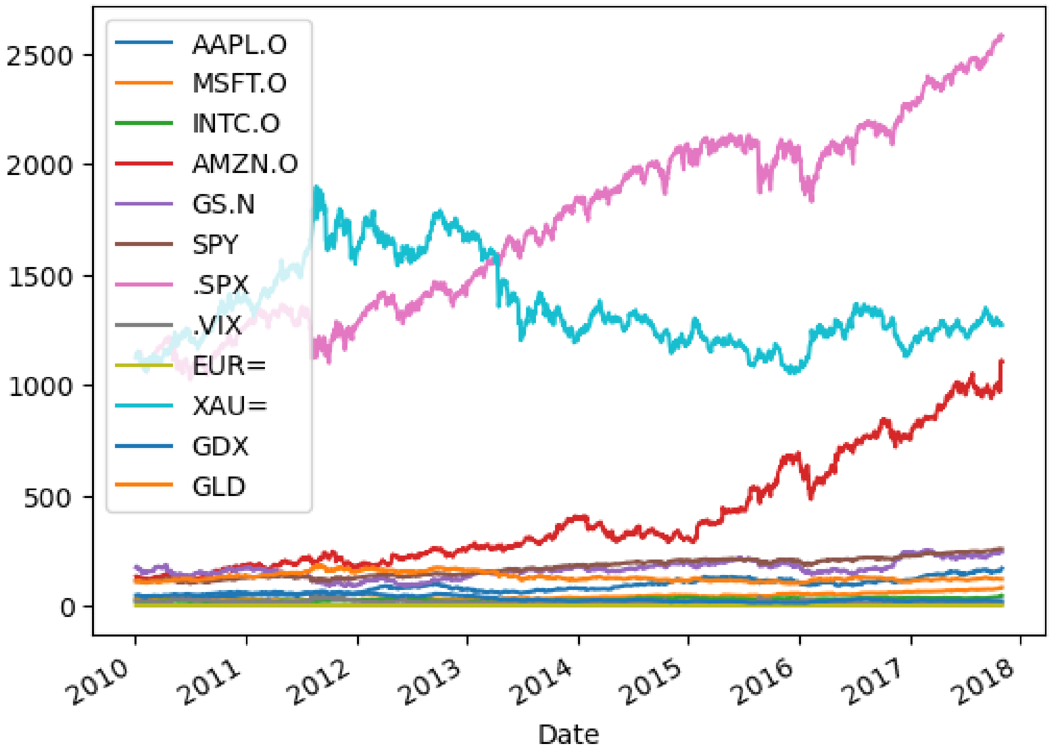
```
datetime.date(2011, 2, 28)
```

8.5 Représentation graphique

La méthode `plot` utilisable sur toutes les `DataFrame` voit son comportement adapté lorsqu'elle est appliquée à une `DataFrame` temporelle (p. ex. les étiquettes de l'axe des abscisses sont adaptées à la nature des données).

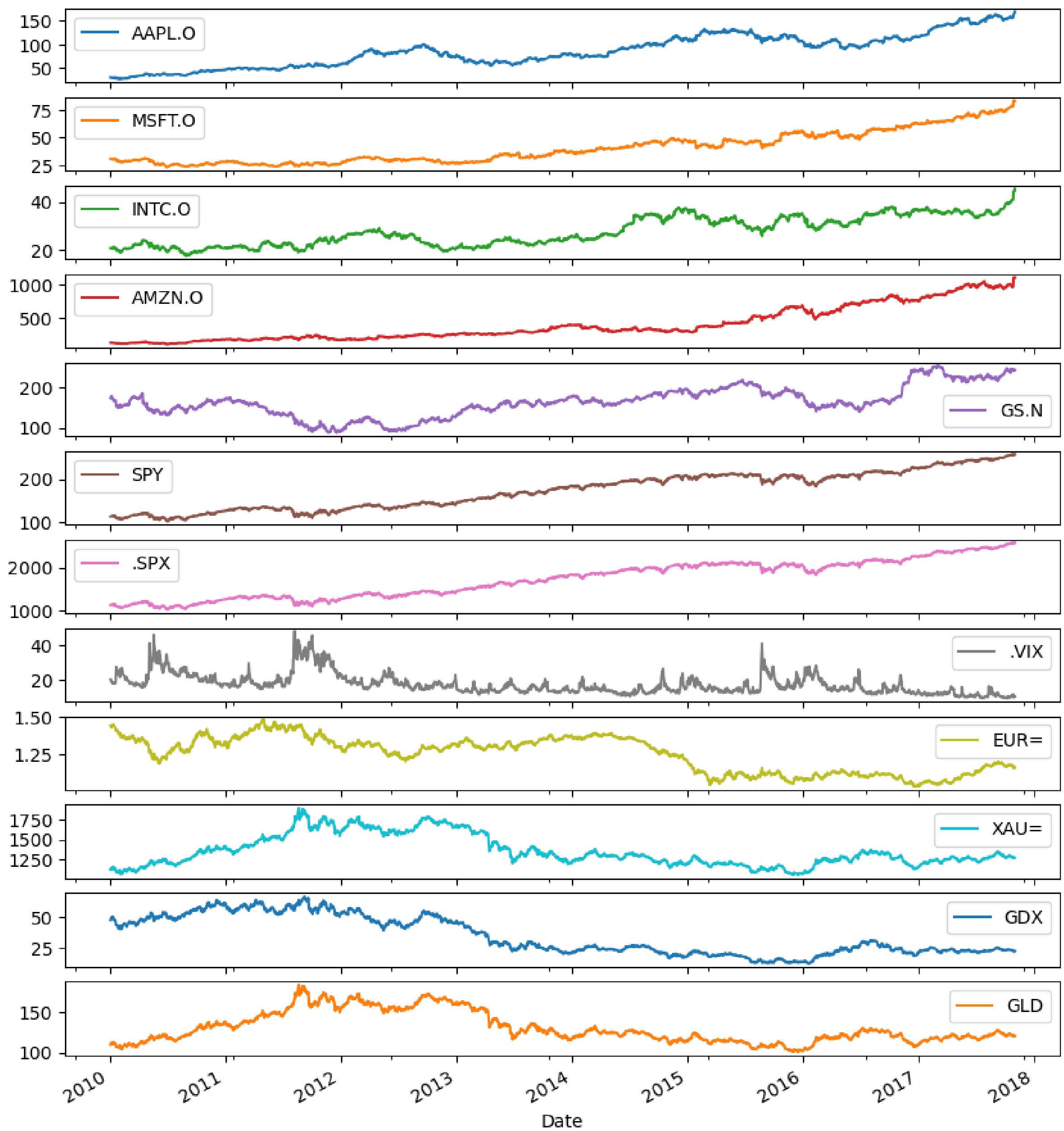
```
data.plot()
```

```
<Axes: xlabel='Date'>
```




```
data.plot(subplots=True, figsize=(10,12))
```

```
array([<Axes: xlabel='Date'>, <Axes: xlabel='Date'>,
       <Axes: xlabel='Date'>, <Axes: xlabel='Date'>,
       <Axes: xlabel='Date'>, <Axes: xlabel='Date'>,
       <Axes: xlabel='Date'>, <Axes: xlabel='Date'>,
       <Axes: xlabel='Date'>, <Axes: xlabel='Date'>,
       <Axes: xlabel='Date'>, <Axes: xlabel='Date'>], dtype=object)
```



8.6 Aggrégation

Mêmes opérations que pour les DataFrame « normales »

```
data.mean()
```

```
AAPL.O       86.530152
MSFT.O       40.586752
INTC.O       27.701411
AMZN.O      401.154006
GS.N        163.614625
```

(continues on next page)

(continued from previous page)

```
SPY         172.835399
.SPX       1727.538342
.VIX         17.209498
EUR=          1.252613
XAU=       1352.471593
GDX          34.499391
GLD         130.601856
dtype: float64
```

```
data.aggregate(["min", "max", "mean", "median"])
```

```
            AAPL.O     MSFT.O     INTC.O       AMZN.O        GS.N         SPY  \
min      27.435687  23.010000  17.665000   108.610000   87.700000  102.200000
max     169.040000  83.890000  45.490000  1110.850000  252.890000  257.710000
mean     86.530152  40.586752  27.701411   401.154006  163.614625  172.835399
median   84.632058  36.540000  26.410000   306.425000  162.090000  178.805000

               .SPX       .VIX      EUR=         XAU=        GDX         GLD
min     1022.580000   9.190000  1.038500  1051.360000  12.470000  100.500000
max     2581.070000  48.000000  1.482600  1897.100000  66.630000  184.590000
mean    1727.538342  17.209498  1.252613  1352.471593  34.499391  130.601856
median  1783.810000  15.650000  1.288400  1288.820000  26.594000  123.895000
```

```
data.loc["2015"].mean()
```

```
AAPL.O      120.039861
MSFT.O       46.713810
INTC.O       32.178730
AMZN.O      478.138194
GS.N        192.965992
SPY         206.187086
.SPX       2061.067738
.VIX         16.674127
EUR=          1.109705
XAU=       1159.066349
GDX          17.156496
GLD         111.146016
dtype: float64
```

```
data.loc["2016"].mean()
```

```
AAPL.O      104.604008
MSFT.O       55.259306
INTC.O       33.329087
AMZN.O      699.523135
GS.N        169.113810
SPY         209.440765
.SPX       2094.651310
.VIX         15.825635
EUR=          1.107160
XAU=       1249.777900
GDX          23.099557
GLD         119.362652
dtype: float64
```

8.7 Changement au cours du temps

Rappel des valeurs contenues dans les premières lignes de `data`

```
data.head(3)
```

```
              AAPL.O  MSFT.O  INTC.O  AMZN.O   GS.N     SPY     .SPX   .VIX  \
Date
2010-01-04  30.572827   30.95   20.88  133.90  173.08  113.33  1132.99  20.04
2010-01-05  30.625684   30.96   20.87  134.69  176.14  113.63  1136.52  19.35
2010-01-06  30.138541   30.77   20.80  132.25  174.26  113.71  1137.14  19.16

             EUR=     XAU=    GDX     GLD
Date
2010-01-04  1.4411  1120.00  47.71  109.80
2010-01-05  1.4368  1118.65  48.17  109.70
2010-01-06  1.4412  1138.50  49.34  111.51
```

La méthode `diff` permet de construire la différence entre deux lignes consécutives :

$x[i] \leftarrow x[i] - x[i-1]$

La différence est bien faite ligne à ligne et c'est à l'utilisateur de garantir la sémantique de l'opération (p. ex. s'il manque des informations sur certains jours ou sur la gestion des week-ends).

```
data.diff().head(10)
```

```
              AAPL.O  MSFT.O  INTC.O  AMZN.O  GS.N   SPY   .SPX  .VIX    EUR=  \
Date
2010-01-04       NaN     NaN     NaN     NaN   NaN   NaN    NaN   NaN     NaN
2010-01-05  0.052857   0.010  -0.010   0.790  3.06  0.30   3.53 -0.69 -0.0043
2010-01-06 -0.487142  -0.190  -0.070  -2.440 -1.88  0.08   0.62 -0.19  0.0044
2010-01-07 -0.055714  -0.318  -0.200  -2.250  3.41  0.48   4.55 -0.10 -0.0094
2010-01-08  0.200000   0.208   0.230   3.520 -3.36  0.38   3.29 -0.93  0.0094
2010-01-11 -0.267143  -0.390   0.120  -3.212 -2.75  0.16   2.00 -0.58  0.0101
2010-01-12 -0.341428  -0.200  -0.342  -2.958 -3.74 -1.07 -10.76  0.70 -0.0019
2010-01-13  0.418571   0.280   0.352   1.760  1.25  0.96   9.46 -0.40  0.0016
2010-01-14 -0.174286   0.610   0.520  -1.760 -0.54  0.31   2.78 -0.22 -0.0008
2010-01-15 -0.500000  -0.100  -0.680  -0.210 -3.32 -1.29 -12.43  0.28 -0.0120

             XAU=   GDX   GLD
Date
2010-01-04    NaN   NaN   NaN
2010-01-05  -1.35  0.46 -0.10
2010-01-06  19.85  1.17  1.81
2010-01-07  -6.60 -0.24 -0.69
2010-01-08   4.20  0.74  0.55
2010-01-11  16.50  0.33  1.48
2010-01-12 -25.30 -1.82 -2.36
2010-01-13  11.10  0.51  1.05
2010-01-14   4.45 -0.26  0.49
2010-01-15 -12.95 -1.18 -1.17
```

À noter : l'introduction du `nan` sur la première ligne pour indiquer qu'aucune valeur ne peut être calculée.

Pour stocker le résultat de `diff` dans un colonne :

```
data["Delta SPY"] = data["SPY"].diff()
data.head(5)
```

```
               AAPL.O  MSFT.O  INTC.O  AMZN.O    GS.N     SPY     .SPX   .VIX  \
Date
2010-01-04  30.572827  30.950   20.88  133.90  173.08  113.33  1132.99  20.04
2010-01-05  30.625684  30.960   20.87  134.69  176.14  113.63  1136.52  19.35
2010-01-06  30.138541  30.770   20.80  132.25  174.26  113.71  1137.14  19.16
2010-01-07  30.082827  30.452   20.60  130.00  177.67  114.19  1141.69  19.06
2010-01-08  30.282827  30.660   20.83  133.52  174.31  114.57  1144.98  18.13

              EUR=     XAU=    GDX     GLD  Delta SPY
Date
2010-01-04  1.4411  1120.00  47.71  109.80        NaN
2010-01-05  1.4368  1118.65  48.17  109.70       0.30
2010-01-06  1.4412  1138.50  49.34  111.51       0.08
2010-01-07  1.4318  1131.90  49.10  110.82       0.48
2010-01-08  1.4412  1136.10  49.84  111.37       0.38
```

La méthode `pct_change` permet de calculer le taux d'évolution

```
data.pct_change().head(5)
```

```
              AAPL.O    MSFT.O    INTC.O    AMZN.O      GS.N       SPY  \
Date
2010-01-04       NaN       NaN       NaN       NaN       NaN       NaN
2010-01-05  0.001729  0.000323 -0.000479  0.005900  0.017680  0.002647
2010-01-06 -0.015906 -0.006137 -0.003354 -0.018116 -0.010673  0.000704
2010-01-07 -0.001849 -0.010335 -0.009615 -0.017013  0.019568  0.004221
2010-01-08  0.006648  0.006830  0.011165  0.027077 -0.018911  0.003328

                .SPX      .VIX      EUR=      XAU=       GDX       GLD  \
Date
2010-01-04       NaN       NaN       NaN       NaN       NaN       NaN
2010-01-05  0.003116 -0.034431 -0.002984 -0.001205  0.009642 -0.000911
2010-01-06  0.000546 -0.009819  0.003062  0.017745  0.024289  0.016500
2010-01-07  0.004001 -0.005219 -0.006522 -0.005797 -0.004864 -0.006188
2010-01-08  0.002882 -0.048793  0.006565  0.003711  0.015071  0.004963

            Delta SPY
Date
2010-01-04        NaN
2010-01-05        NaN
2010-01-06  -0.733333
2010-01-07   5.000000
2010-01-08  -0.208333
```

8.8 Resampling

Permet de regrouper les observations pour avoir une fréquence moins élevée. La « taille » d'un regroupement est spécifiée par un *time offset*

Syntaxe :

```
df.resample(time_offset).aggregation_function()
```

Par exemple pour obtenir la dernière observation de chaque semaine :

```
data.resample("1w").last().head()
```

```
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71864/5763075.py:1:
↪FutureWarning: 'w' is deprecated and will be removed in a future version, please
↪use 'W' instead.
  data.resample("1w").last().head()
```

```
               AAPL.O  MSFT.O  INTC.O  AMZN.O    GS.N     SPY     .SPX   .VIX  \
Date
2010-01-10  30.282827   30.66   20.83  133.52  174.31  114.57  1144.98  18.13
2010-01-17  29.418542   30.86   20.80  127.14  165.21  113.64  1136.03  17.91
2010-01-24  28.249972   28.96   19.91  121.43  154.12  109.21  1091.76  27.31
2010-01-31  27.437544   28.18   19.40  125.41  148.72  107.39  1073.87  24.62
2010-02-07  27.922829   28.02   19.47  117.39  154.16  106.66  1066.19  26.11

              EUR=     XAU=    GDX     GLD  Delta SPY
Date
2010-01-10  1.4412  1136.10  49.84  111.37       0.38
2010-01-17  1.4382  1129.90  47.42  110.86      -1.29
2010-01-24  1.4137  1092.60  43.79  107.17      -2.49
2010-01-31  1.3862  1081.05  40.72  105.96      -1.18
2010-02-07  1.3662  1064.95  42.41  104.68       0.22
```

Pour obtenir la valeur moyenne de l'action chaque semaine :

```
data.resample("1w").mean().head()
```

```
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71864/818262450.py:1:
↪FutureWarning: 'w' is deprecated and will be removed in a future version, please
↪use 'W' instead.
  data.resample("1w").mean().head()
```

```
               AAPL.O    MSFT.O   INTC.O    AMZN.O     GS.N      SPY  \
Date
2010-01-10  30.340541  30.75840  20.7960  132.8720  175.092  113.886
2010-01-17  29.823970  30.50200  20.9596  128.2516  168.438  114.316
2010-01-24  29.735220  30.16375  20.7200  125.3600  162.410  112.465
2010-01-31  28.807171  29.16600  19.9760  122.7960  151.874  108.974
2010-02-07  27.923686  28.27200  19.5380  117.8840  154.428  108.474

                 .SPX    .VIX     EUR=       XAU=     GDX       GLD  Delta SPY
Date
2010-01-10  1138.6640  19.148  1.43842  1129.0300  48.832  110.6400     0.3100
2010-01-17  1142.6740  17.838  1.44802  1138.2100  48.680  111.5540    -0.1860
2010-01-24  1124.1275  21.460  1.41565  1109.0875  45.240  108.7500    -1.1075
```

(continues on next page)

(continued from previous page)

```
2010-01-31  1088.9700  24.290  1.40128  1089.7600  42.396  106.8016    -0.3640
2010-02-07  1083.8180  23.572  1.38364  1091.8000  42.232  107.0460    -0.1460
```

Le résultat du *resampling* étant une `DataFrame` il est possible de combiner cette méthode avec n'importe quelle méthode de manipulation des `DataFrame`. On peut, par exemple, calculer l'augmentation moyenne du cours des actions :

```
data.resample("1y").mean().pct_change()
```

```
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71864/1514818338.py:1:␣
↪FutureWarning: 'y' is deprecated and will be removed in a future version, please␣
↪use 'YE' instead.
  data.resample("1y").mean().pct_change()
```

```
              AAPL.O    MSFT.O    INTC.O    AMZN.O      GS.N       SPY  \
Date
2010-12-31       NaN       NaN       NaN       NaN       NaN       NaN
2011-12-31  0.400849 -0.037197  0.061783  0.413467 -0.157312  0.111443
2012-12-31  0.582536  0.144637  0.132388  0.120158 -0.152920  0.088303
2013-12-31 -0.179524  0.089578 -0.076654  0.352825  0.413835  0.190611
2014-12-31  0.366493  0.306598  0.315195  0.115822  0.107251  0.174754
2015-12-31  0.301040  0.100356  0.061038  0.437789  0.112859  0.067122
2016-12-31 -0.128589  0.182933  0.035749  0.463015 -0.123608  0.015780
2017-12-31  0.398925  0.259409  0.090195  0.332180  0.365660  0.151611

                .SPX      .VIX      EUR=      XAU=       GDX       GLD  \
Date
2010-12-31       NaN       NaN       NaN       NaN       NaN       NaN
2011-12-31  0.111997  0.073338  0.050016  0.281067  0.124598  0.275819
2012-12-31  0.088128 -0.264464 -0.076652  0.061251 -0.159558  0.057331
2013-12-31  0.191717 -0.200640  0.033238 -0.155605 -0.388011 -0.159289
2014-12-31  0.174947 -0.003804 -0.000180 -0.101614 -0.215167 -0.105136
2015-12-31  0.067150  0.176223 -0.164572 -0.084191 -0.264877 -0.087862
2016-12-31  0.016294 -0.050887 -0.002294  0.078263  0.346403  0.073927
2017-12-31  0.152517 -0.290780  0.012434  0.004334 -0.001960  0.000765

            Delta SPY
Date
2010-12-31        NaN
2011-12-31  -1.020049
2012-12-31 -69.181120
2013-12-31   1.480452
2014-12-31  -0.506859
2015-12-31  -1.080096
2016-12-31 -12.772455
2017-12-31   1.052085
```

Ou pour calculer le rapport de la valeur moyenne de l'action `AAPL.O` entre deux années consécutives (cf. ci-dessous pour la définition de `shift`)

```
data.resample("1y").mean()["AAPL.O"] / data.resample("1y").mean()["AAPL.O"].shift(1)
```

```
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71864/828940953.py:1:␣
↪FutureWarning: 'y' is deprecated and will be removed in a future version, please␣
↪use 'YE' instead.
  data.resample("1y").mean()["AAPL.O"] / data.resample("1y").mean()["AAPL.O"].
↪shift(1)
```

```
Date
2010-12-31         NaN
2011-12-31    1.400849
2012-12-31    1.582536
2013-12-31    0.820476
2014-12-31    1.366493
2015-12-31    1.301040
2016-12-31    0.871411
2017-12-31    1.398925
Freq: YE-DEC, Name: AAPL.O, dtype: float64
```

Pou rechercher la date à laquelle l'action `AAPL.O` a atteint sa plus grande valeur chaque année (la méthode `aggregate` permet de spécifier plusieurs fonctions d'aggrégation simultanément)

```
data["AAPL.O"].resample("1Y").aggregate(["idxmax", "max"])
```

```
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71864/359967340.py:1:
↪FutureWarning: 'Y' is deprecated and will be removed in a future version, please
↪use 'YE' instead.
  data["AAPL.O"].resample("1Y").aggregate(["idxmax", "max"])
```

```
               idxmax         max
Date
2010-12-31 2010-12-28   46.495668
2011-12-31 2011-10-18   60.319940
2012-12-31 2012-09-19  100.299900
2013-12-31 2013-12-23   81.441347
2014-12-31 2014-11-26  119.000000
2015-12-31 2015-02-23  133.000000
2016-12-31 2016-10-25  118.250000
2017-12-31 2017-10-31  169.040000
```

8.9 Sur-échantillonage

Il est également possible de « sur-échantillonner » une `DataFrame` pour garantir qu'il existe des entrées pour toutes les dates d'une période en introduisant des `NaN` aux dates manquantes. Cette opération est nécessaire pour garantir la correspondance entre le nombre de lignes et une période : après un sur-échantillonnage on est, par exemple, sûr qu'il y a toujours 5 observations (lignes) dans une `DataFrame` pour une semaine (en ne comptant que les jours ouvrés) même si les données originales ne contenaient pas d'information pour les jours fériés.

Le sur-échantillonage est nécessaire pour empêcher les fuites de données (décalage) lors de la définition d'opération sur les fenêtres glissantes ou lors de l'accès à une date à partir d'une autre. On peut par exemple garantir que la valeur d'une colonne 1 semaine avant la date courante est toujours située 7 observations plus tôt.

```
# select the first 5 columns for a smaller output
data.reindex(pd.date_range(data.index.min(),
                           data.index.max())).head(10)[data.columns[:5]]
```

```
               AAPL.O  MSFT.O  INTC.O   AMZN.O    GS.N
2010-01-04  30.572827  30.950  20.880  133.900  173.08
2010-01-05  30.625684  30.960  20.870  134.690  176.14
2010-01-06  30.138541  30.770  20.800  132.250  174.26
2010-01-07  30.082827  30.452  20.600  130.000  177.67
```

(continues on next page)

(continued from previous page)

```
2010-01-08  30.282827  30.660  20.830  133.520  174.31
2010-01-09        NaN     NaN     NaN      NaN     NaN
2010-01-10        NaN     NaN     NaN      NaN     NaN
2010-01-11  30.015684  30.270  20.950  130.308  171.56
2010-01-12  29.674256  30.070  20.608  127.350  167.82
2010-01-13  30.092827  30.350  20.960  129.110  169.07
```

8.10 Opérations sur des fenêtres

La méthode `shift(n)` permet de « décaler » les observations (lignes) d'une `DataFrame` pour que l'observation de la te observation soit aligné avec la (t - n)e observation :

```
data["AAPL.O"].shift(1).head(9)
```

```
Date
2010-01-04          NaN
2010-01-05    30.572827
2010-01-06    30.625684
2010-01-07    30.138541
2010-01-08    30.082827
2010-01-11    30.282827
2010-01-12    30.015684
2010-01-13    29.674256
2010-01-14    30.092827
Name: AAPL.O, dtype: float64
```

Il est alors possible d'utiliser les méthodes de manipulation classique. L'instruction suivante permet, par exemple, de calculer $\frac{x[i]}{x[i-1]}$

```
(data["AAPL.O"] / data.shift(1)["AAPL.O"]).head(9)
```

```
Date
2010-01-04         NaN
2010-01-05    1.001729
2010-01-06    0.984094
2010-01-07    0.998151
2010-01-08    1.006648
2010-01-11    0.991178
2010-01-12    0.988625
2010-01-13    1.014106
2010-01-14    0.994208
Name: AAPL.O, dtype: float64
```

et pour calculer : $\frac{x[i]-x[i-1]}{x[i-1]}$

```
((data["AAPL.O"] - data.shift(1)["AAPL.O"]) / data.shift(1)["AAPL.O"]).head(9)
```

```
Date
2010-01-04         NaN
2010-01-05    0.001729
2010-01-06   -0.015906
2010-01-07   -0.001849
```

(continues on next page)

(continued from previous page)

```
2010-01-08    0.006648
2010-01-11   -0.008822
2010-01-12   -0.011375
2010-01-13    0.014106
2010-01-14   -0.005792
Name: AAPL.O, dtype: float64
```

et pour calculer, pour chaque jour, le rapport entre la valeur à l'instant t et la valeur à $t - 1an$

```
data["AAPL.O"] / data["AAPL.O"].shift(365)
```

```
Date
2010-01-04         NaN
2010-01-05         NaN
2010-01-06         NaN
2010-01-07         NaN
2010-01-08         NaN
                ...
2017-10-25    1.666063
2017-10-26    1.683709
2017-10-27    1.724302
2017-10-30    1.769851
2017-10-31    1.775257
Name: AAPL.O, Length: 1972, dtype: float64
```

Attention : ici 365 correspond au nombre d'observations (de lignes). C'est à vous de garantir que vous avez bien 365 valeurs par an (à l'aide de la commande `reindex`).

Il est également possible de définir des opérations sur des fenêtres glissantes, c.-à-d. « d'aligner » la ie observation avec les n observations précédentes (par opposition à la méthode `shift` qui permet d'aligner la ie observation avec une seule observation précédente) à l'aide de la méthode `rolling` et d'une fonction d'aggrégation.

Par exemple pour définir une moyenne glissante sur 5 observations :

```
data.rolling(5).mean().head(10)[data.columns[:5]]
```

```
               AAPL.O   MSFT.O   INTC.O    AMZN.O     GS.N
Date
2010-01-04        NaN      NaN      NaN       NaN      NaN
2010-01-05        NaN      NaN      NaN       NaN      NaN
2010-01-06        NaN      NaN      NaN       NaN      NaN
2010-01-07        NaN      NaN      NaN       NaN      NaN
2010-01-08  30.340541  30.7584  20.7960  132.8720  175.092
2010-01-11  30.229113  30.6224  20.8100  132.1536  174.788
2010-01-12  30.038827  30.4444  20.7576  130.6856  173.124
2010-01-13  30.029684  30.3604  20.7896  130.0576  172.086
2010-01-14  29.996827  30.4620  20.9656  129.5276  170.258
2010-01-15  29.823970  30.5020  20.9596  128.2516  168.438
```

Il est possible à l'aide de la méthode `apply` d'utiliser une fonction arbitraire :

```
def div(x):
    # x est une liste de n éléments
    # n = valeur passée en paramètre à rolling

    # x : est une liste
```

(continues on next page)

(continued from previous page)

```
    # x[-1] : dernier élément de la liste
    # x[:-1] : tous les éléments de la liste sauf le dernier
    return x[-1] / np.mean(x[:-1])

data.rolling(5).apply(div).head(10)[data.columns[:5]]
```

```
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71864/2033231817.py:8:
↪FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a
↪future version, integer keys will always be treated as labels (consistent with
↪DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
  return x[-1] / np.mean(x[:-1])
```

```
              AAPL.O    MSFT.O    INTC.O    AMZN.O      GS.N
Date
2010-01-04       NaN       NaN       NaN       NaN       NaN
2010-01-05       NaN       NaN       NaN       NaN       NaN
2010-01-06       NaN       NaN       NaN       NaN       NaN
2010-01-07       NaN       NaN       NaN       NaN       NaN
2010-01-08  0.997623  0.996004  1.002044  1.006104  0.994423
2010-01-11  0.991190  0.985656  1.008424  0.982604  0.977021
2010-01-12  0.984875  0.984675  0.991007  0.968297  0.961995
2010-01-13  1.002630  0.999572  1.010267  0.990909  0.978188
2010-01-14  0.996740  1.020519  1.030859  0.979073  0.987345
2010-01-15  0.983065  1.014714  0.990500  0.989189  0.976159
```

Attention : les fenêtre glissante sont définies sur un nombre d'observations (lignes) fixe et constant. Vous devez vous assurer que la `DataFrame` est bien « organisée » (indexée) pour que la sémantique des opérations soit bonne (par exemple que 5 observations = 1 semaine ouvrée).

La méthode `expanding` permet d'accumuler les observations : $x[i] \leftarrow func(x[0], ..., x[i])$

Par exemple, pour calculer la moyenne d'une colonne entre 0 et la date courante :

```
data.expanding().mean().head()[data.columns[:5]]
```

```
               AAPL.O     MSFT.O   INTC.O      AMZN.O        GS.N
Date
2010-01-04  30.572827  30.950000  20.8800  133.900000  173.080000
2010-01-05  30.599255  30.955000  20.8750  134.295000  174.610000
2010-01-06  30.445684  30.893333  20.8500  133.613333  174.493333
2010-01-07  30.354970  30.783000  20.7875  132.710000  175.287500
2010-01-08  30.340541  30.758400  20.7960  132.872000  175.092000
```

et la somme cumulée :

```
data.expanding().sum().head()[data.columns[:5]]
```

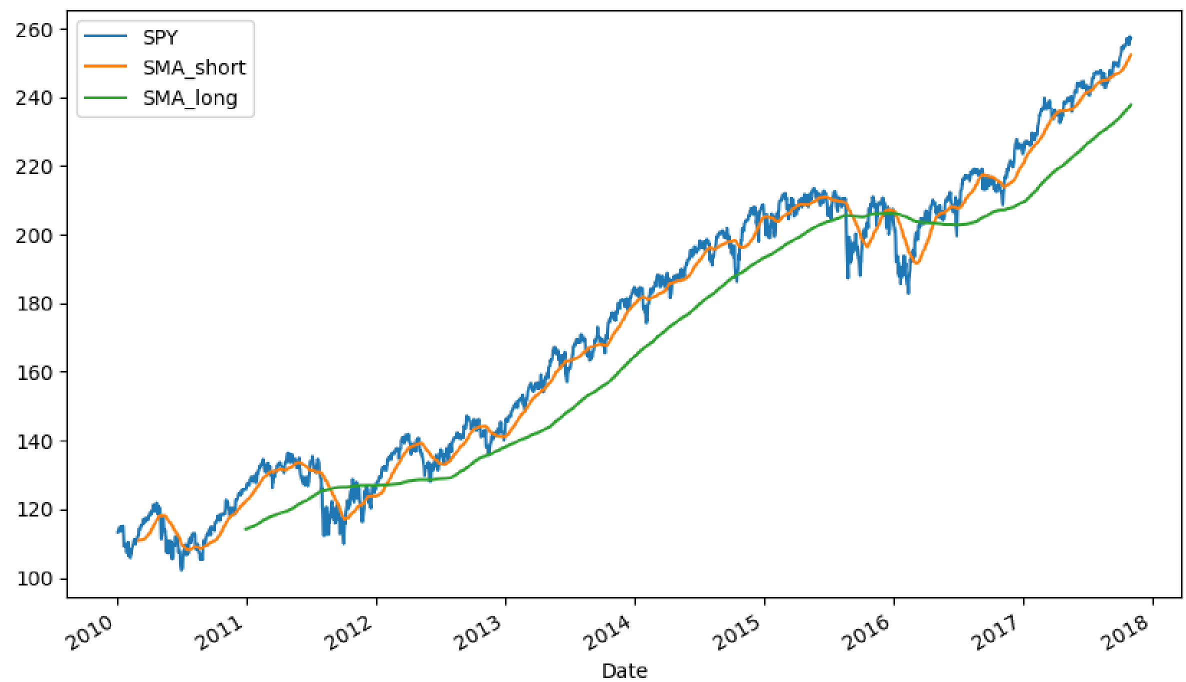
```
                AAPL.O   MSFT.O  INTC.O  AMZN.O    GS.N
Date
2010-01-04   30.572827   30.950   20.88  133.90  173.08
2010-01-05   61.198510   61.910   41.75  268.59  349.22
2010-01-06   91.337052   92.680   62.55  400.84  523.48
2010-01-07  121.419879  123.132   83.15  530.84  701.15
2010-01-08  151.702705  153.792  103.98  664.36  875.46
```

8.11 Exemple : Simple Moving Average

Représentez graphiquement, pour l'action SPY, les dates pour lequelles la moyenne *short-term* (sur 42 jours) est au-dessus de la moyenne *long-term* (sur 252 jours)

```
data["SMA_short"] = data["SPY"].rolling(window=42).mean()
data["SMA_long"] = data["SPY"].rolling(window=252).mean()

# sélection des colonnes que l'on souhaite représenter graphiquement
data[["SPY", "SMA_short", "SMA_long"]].plot(figsize=(10,6))
```

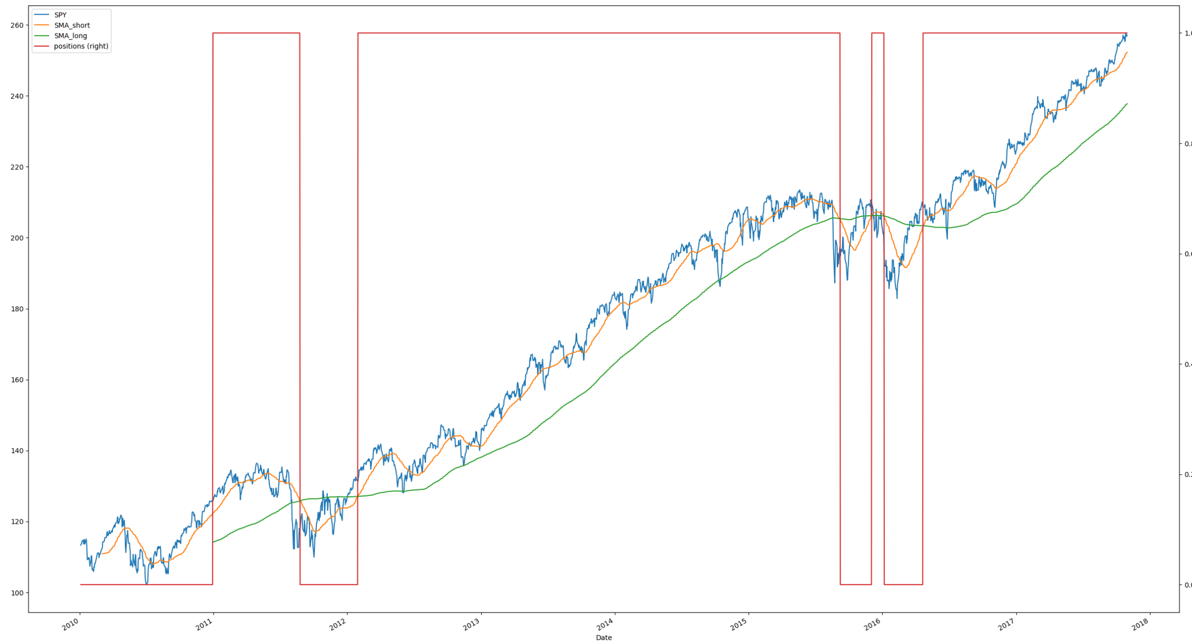
```
<Axes: xlabel='Date'>
```




```
def is_positive(x):
    if x > 0:
        return 1
    else:
        return 0

data["positions"] = (data["SMA_short"] - data["SMA_long"]).apply(is_positive)
```

```
data[["SPY", "SMA_short", "SMA_long", "positions"]].plot(figsize=(30,18),
                                                          secondary_y="positions")
```

```
<Axes: xlabel='Date'>
```


SPY
SMA_short
SMA_long
positions (right)
260
240
220
200
180
160
140
120
100
1.0
0.8
0.6
0.4
0.2
0.0
2010
2011
2012
2013
2014
2015
2016
2017
2018
Date

Part II

Étude de cas

CHAPTER

NINE

ANALYZING PASSWORDS WITH PANDAS

```
from pathlib import Path

import pandas as pd
```

```
if not Path("passwords.csv").is_file():
    !curl -L https://bit.ly/3E3MWnW -o passwords.csv
```

9.1 Questions

1. Which password(s) in the datasets took the shortest time to crack by online guessing? Which took the longest?
2. Plot the online guessing time with respect to the offline guessing time. What can you conclude?
3. How many categories of password are there? What is the distribution of categories in the dataset?
4. How many password begin with the sequence `123`?
5. What is the average time in hours needed to crack these passwords that begin with 123? How does this compare to the average of all other passwords in the dataset?
6. How many passwords do not contain a number?
7. How many passwords contain at least one number?
8. Is there an obvious difference in online cracking time between passwords that don't contain a number vs passwords that contain at least one number?
9. How many passwords contain at least one of the following punctuations: `[.!?\\-]`.

9.2 Solutions

```
df = pd.read_csv("passwords.csv")[["password", "category", "value", "time_unit",
↪"offline_crack_sec"]]
```

```
df["time_unit"].unique()
```

```
array(['years', 'minutes', 'days', 'seconds', 'months', 'weeks', 'hours',
       nan], dtype=object)
```

```
time_mapping = {"years": 365 * 24 * 3_600,
                "minutes": 60,
                "days": 24 * 3_600,
                "seconds": 1,
                "months": 31 * 24 * 3_600,
                "weeks": 7 * 24 * 3_600,
                "hours": 3_600}

df["online_crack_sec"] = df["time_unit"].apply(time_mapping.get) \
                             * df["value"]
```

```
df.apply(lambda row: row["value"] * time_mapping.get(row["time_unit"],0),
         axis=1).head()
```

```
0    2.179138e+08
1    1.111200e+03
2    1.114560e+05
3    1.111000e+01
4    3.214080e+05
dtype: float64
```

```
df.sort_values(by="online_crack_sec").head(1)
```

```
    password              category  value time_unit  offline_crack_sec  \
276     5150  simple-alphanumeric  11.11   seconds       1.110000e-07

     online_crack_sec
276             11.11
```

```
df.sort_values(by="online_crack_sec", ascending=False).head(1)
```

```
     password          category  value time_unit  offline_crack_sec  \
499  passw0rd  password-related  92.27     years              29.02

     online_crack_sec
499      2.909827e+09
```

```
df[df["online_crack_sec"] == df["online_crack_sec"].max()]
```

```
     password             category  value time_unit  offline_crack_sec  \
25   trustno1  simple-alphanumeric  92.27     years              29.02
335  rush2112             nerdy-pop  92.27     years              29.02
405  jordan23                 sport  92.27     years              29.27
499  passw0rd     password-related  92.27     years              29.02

     online_crack_sec
25       2.909827e+09
335      2.909827e+09
405      2.909827e+09
499      2.909827e+09
```

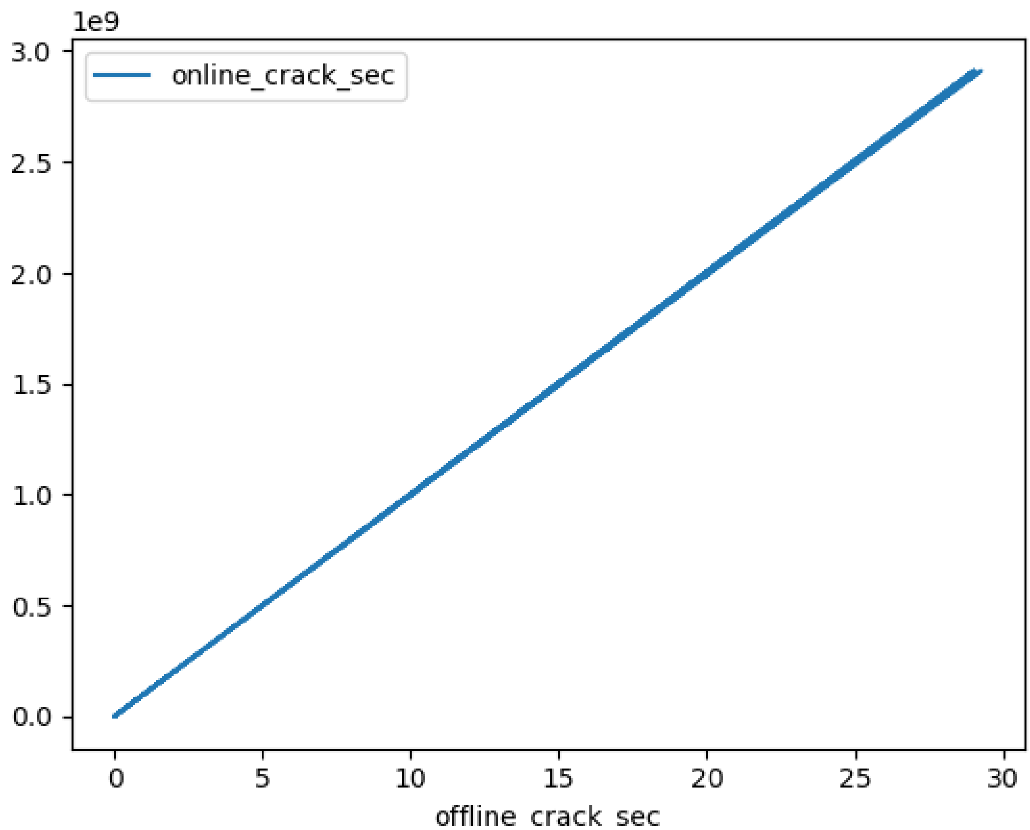
```
df[df["online_crack_sec"] == df["online_crack_sec"].min()]
```

```
    password             category  value time_unit  offline_crack_sec  \
3       1234  simple-alphanumeric  11.11   seconds       1.110000e-07
19      2000  simple-alphanumeric  11.11   seconds       1.110000e-07
44      6969  simple-alphanumeric  11.11   seconds       1.110000e-07
76      1111  simple-alphanumeric  11.11   seconds       1.110000e-07
276     5150  simple-alphanumeric  11.11   seconds       1.110000e-07
314     2112  simple-alphanumeric  11.11   seconds       1.110000e-07
315     1212  simple-alphanumeric  11.11   seconds       1.110000e-07
324     7777  simple-alphanumeric  11.11   seconds       1.110000e-07
371     2222  simple-alphanumeric  11.11   seconds       1.110000e-07
373     4444  simple-alphanumeric  11.11   seconds       1.110000e-07
429     1313  simple-alphanumeric  11.11   seconds       1.110000e-07

     online_crack_sec
3               11.11
19              11.11
44              11.11
76              11.11
276             11.11
314             11.11
315             11.11
324             11.11
371             11.11
373             11.11
429             11.11
```

Plot the "online guessing time" with respect to the "offline guessing time". What can you conclude?

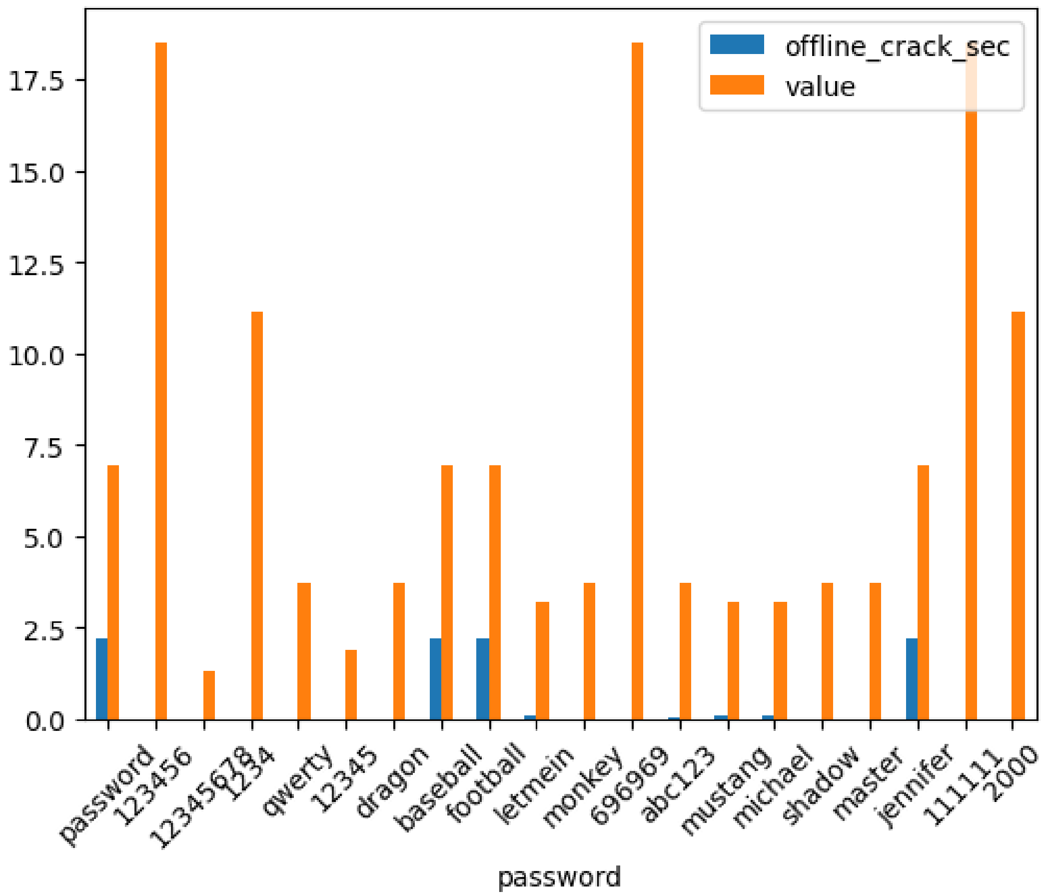
```
df.plot(x="offline_crack_sec", y="online_crack_sec")
```

```
<Axes: xlabel='offline_crack_sec'>
```




```python
ax = df.head(20).plot(x="password",
                      y=["offline_crack_sec", "value"],
                      kind="bar")
ax.set_xticklabels(ax.get_xticklabels(), rotation=45)
```

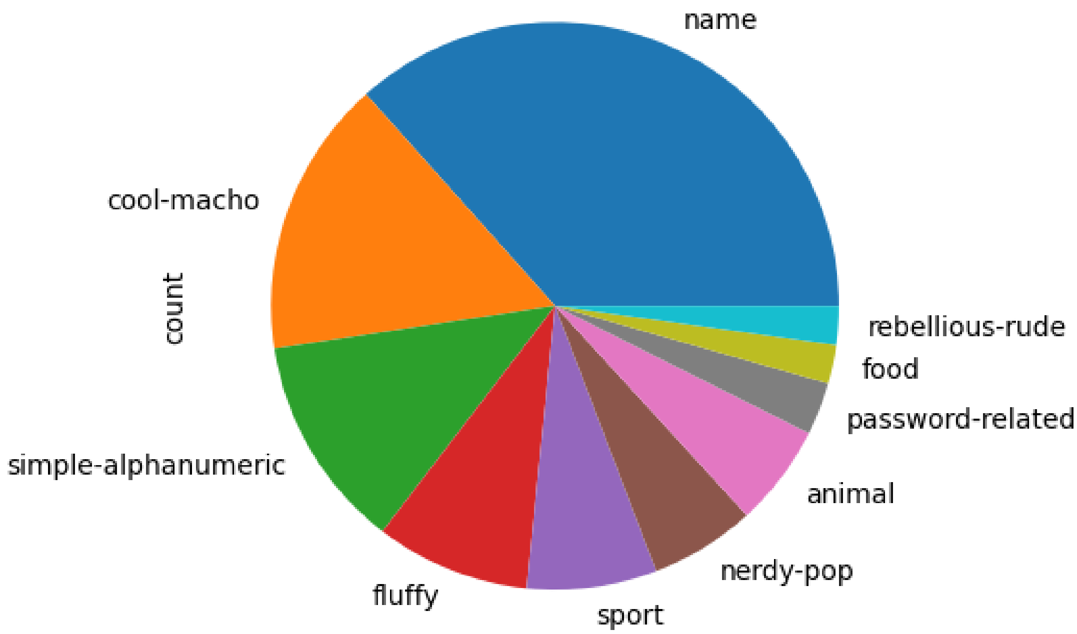
```
[Text(0, 0, 'password'),
 Text(1, 0, '123456'),
 Text(2, 0, '12345678'),
 Text(3, 0, '1234'),
 Text(4, 0, 'qwerty'),
 Text(5, 0, '12345'),
 Text(6, 0, 'dragon'),
 Text(7, 0, 'baseball'),
 Text(8, 0, 'football'),
 Text(9, 0, 'letmein'),
 Text(10, 0, 'monkey'),
 Text(11, 0, '696969'),
 Text(12, 0, 'abc123'),
 Text(13, 0, 'mustang'),
 Text(14, 0, 'michael'),
 Text(15, 0, 'shadow'),
 Text(16, 0, 'master'),
 Text(17, 0, 'jennifer'),
 Text(18, 0, '111111'),
 Text(19, 0, '2000')]
```



How many categories of password are there? What is the distribution of categories in the dataset?

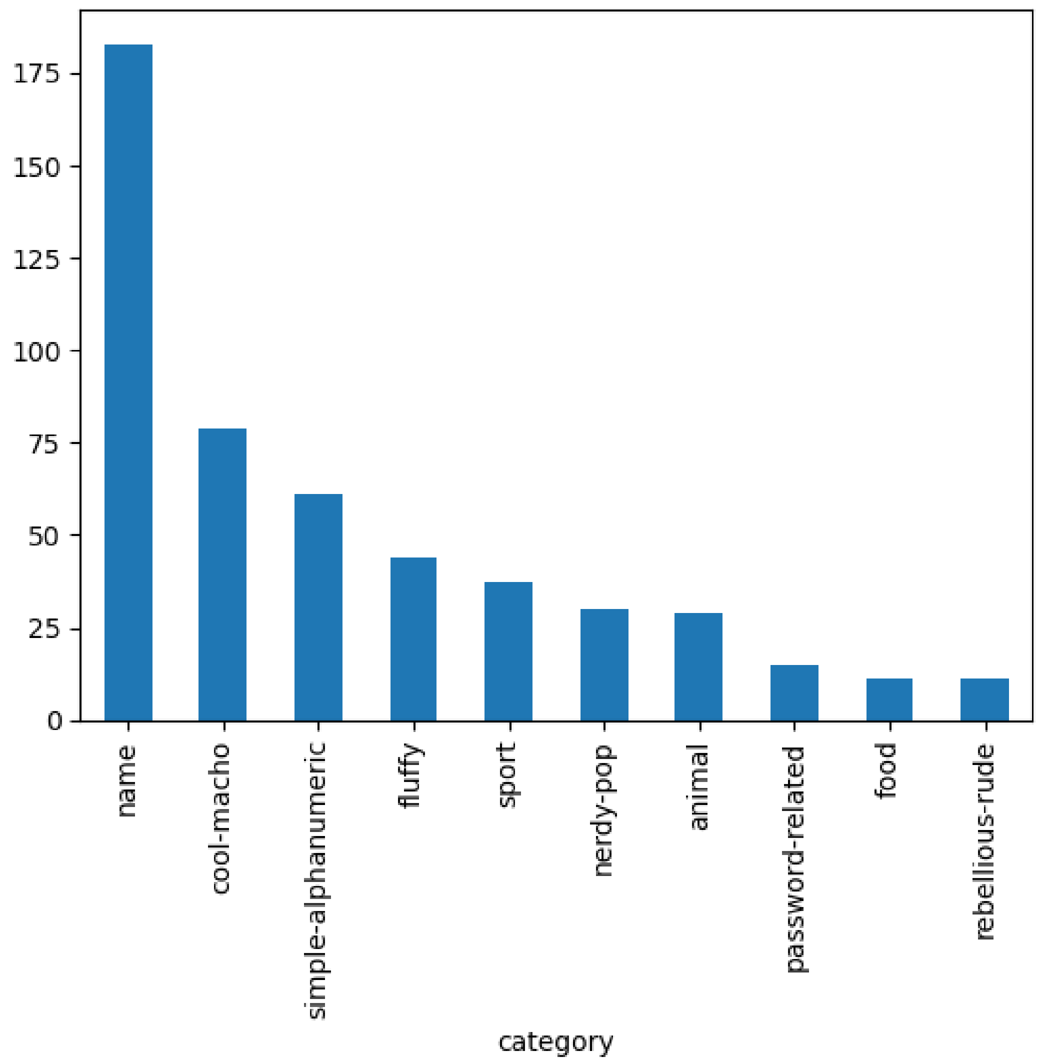
```
df["category"].value_counts().plot(kind="pie")
```

```
<Axes: ylabel='count'>
```




```
df["category"].value_counts().plot(kind="bar")
```

```
<Axes: xlabel='category'>
```



How many password begin with the sequence 123? What is the average time in hours needed to crack these passwords? How does this compare to the average of all other passwords in the dataset?

```
# supprime toutes les lignes qui contiennent au moins un nan
df = df.dropna()
```

```
df[df["password"].str.startswith("123")].shape[0]
```

```
9
```

```
df[df["password"].str.contains("^123")].shape[0]
```

```
9
```

```
df[df["password"].str.startswith("123")]["online_crack_sec"].mean()
```

```
np.float64(386291.96777777775)
```

```
df[~df["password"].str.startswith("123")]["online_crack_sec"].mean()
```

```
np.float64(51063379.65234215)
```

```
df.groupby(df["password"].str.startswith("123"))["online_crack_sec"].mean()
```

```
password
False    5.106338e+07
True     3.862920e+05
Name: online_crack_sec, dtype: float64
```

How many passwords do not contain a number? How many passwords contain at least one number?

```
# with a regular expression: \D = no number \d at least one number
# sample → reduce number of rows
df[df["password"].str.contains("\D")].sample(5)
```

```
<>:3: SyntaxWarning: invalid escape sequence '\D'
<>:3: SyntaxWarning: invalid escape sequence '\D'
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71861/4075640701.py:3:
↪SyntaxWarning: invalid escape sequence '\D'
  df[df["password"].str.contains("\D")].sample(5)
```

```
    password    category  value time_unit  offline_crack_sec  online_crack_sec
155   brandy        name   3.72      days           0.003210          321408.0
330   walter        name   3.72      days           0.003210          321408.0
71    hammer  cool-macho   3.72      days           0.003210          321408.0
208    angel      fluffy   3.43     hours           0.000124           12348.0
51    maggie        name   3.72      days           0.003210          321408.0
```

```
# reduce number of rows to generate the pdf
df[df["password"].apply(lambda x: any(char.isdigit() for char in x))].sample(5)
```

```
     password             category  value time_unit  offline_crack_sec  \
277    222222  simple-alphanumeric  18.52   minutes       1.110000e-05
276      5150  simple-alphanumeric  11.11   seconds       1.110000e-07
402  69696969  simple-alphanumeric   1.29      days       1.110000e-03
49     654321  simple-alphanumeric  18.52   minutes       1.110000e-05
206   7777777  simple-alphanumeric   3.09     hours       1.110000e-04

     online_crack_sec
277           1111.20
276             11.11
402         111456.00
49            1111.20
206          11124.00
```

```
df.groupby(df["password"].str.contains("\D"))["online_crack_sec"].mean()
```

```
<>:1: SyntaxWarning: invalid escape sequence '\D'
<>:1: SyntaxWarning: invalid escape sequence '\D'
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71861/3391280026.py:1:
↪SyntaxWarning: invalid escape sequence '\D'
  df.groupby(df["password"].str.contains("\D"))["online_crack_sec"].mean()
```

```
password
False    4.309575e+04
True     5.450842e+07
Name: online_crack_sec, dtype: float64
```

```
import seaborn as sns
```

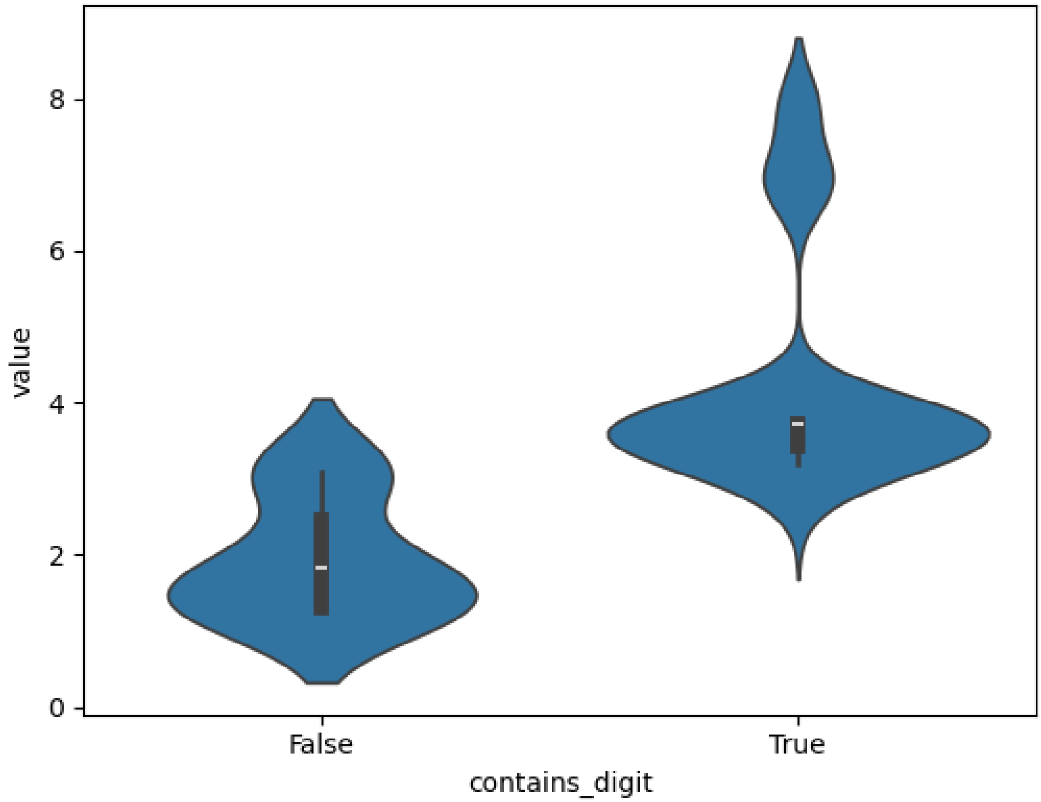
```
df["contains_digit"] = df["password"].str.contains("\D")
df
```

```
<>:1: SyntaxWarning: invalid escape sequence '\D'
<>:1: SyntaxWarning: invalid escape sequence '\D'
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71861/3357837877.py:1:
  SyntaxWarning: invalid escape sequence '\D'
  df["contains_digit"] = df["password"].str.contains("\D")
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71861/3357837877.py:1:
  SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/
  user_guide/indexing.html#returning-a-view-versus-a-copy
  df["contains_digit"] = df["password"].str.contains("\D")
```

```
     password             category  value time_unit  offline_crack_sec  \
0    password     password-related   6.91     years       2.170000e+00
1      123456  simple-alphanumeric  18.52   minutes       1.110000e-05
2    12345678  simple-alphanumeric   1.29      days       1.110000e-03
3        1234  simple-alphanumeric  11.11   seconds       1.110000e-07
4      qwerty  simple-alphanumeric   3.72      days       3.210000e-03
..        ...                  ...    ...       ...                ...
495    reddog           cool-macho   3.72      days       3.210000e-03
496  alexande                 name   6.91     years       2.170000e+00
497   college            nerdy-pop   3.19    months       8.350000e-02
498    jester                 name   3.72      days       3.210000e-03
499  passw0rd     password-related  92.27     years       2.902000e+01

     online_crack_sec  contains_digit
0        2.179138e+08            True
1        1.111200e+03           False
2        1.114560e+05           False
3        1.111000e+01           False
4        3.214080e+05            True
..                ...             ...
495      3.214080e+05            True
496      2.179138e+08            True
497      8.544096e+06            True
498      3.214080e+05            True
499      2.909827e+09            True

[500 rows x 7 columns]
```

```
sns.violinplot(data=df[df["value"] < 10], y="value", x="contains_digit")
```

```
<Axes: xlabel='contains_digit', ylabel='value'>
```



How many passwords contain at least one of the following punctuations: [.!?-].

```
df[df["password"].apply(lambda x: any(char in "[.!?\-]" for char in x))]
```

```
<>:1: SyntaxWarning: invalid escape sequence '\-'
<>:1: SyntaxWarning: invalid escape sequence '\-'
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71861/731872277.py:1:␣
↪SyntaxWarning: invalid escape sequence '\-'
  df[df["password"].apply(lambda x: any(char in "[.!?\-]" for char in x))]
```

```
Empty DataFrame
Columns: [password, category, value, time_unit, offline_crack_sec, online_crack_
↪sec, contains_digit]
Index: []
```

CHAPTER

TEN

CASE STUDY: MERGING BROKER AND FRONT OFFICE DATA

The goal of the exercise is to detect stocks for which the front office *spread* is not in the *ask-bid* range provided by a broker.

The file `DummyFinancialDatas.xlsx` contains the front office data. The information we are interested in is in the `contrib_fo` sheet. Each title is identified by:

- a city name (column `dummy_GroupID`);
- a maturity (column `Term`).

The file `GC_datas_broker_10June2020_16-00_LDN.csv` contains the broker data. The stocks are identified by:

- a city name (column `Description`);
- a maturity (column `Tenor`).

You must produce a `DataFrame` containing for each stock, the *bid*, *ask* and *spread* values and a column containing "Alert" if the spread is not between ask and bid. In this case, the row must have a red background.

```
import pandas as pd
from pathlib import Path
```

```
if not Path("DummyFinancialDatas.xlsx").is_file():
    !curl -L https://lc.cx/rDtkEl -o broker_data.zip
    !unzip broker_data.zip
```

For `pandas` to read excel data (`xlsx` files), the `openpyxl` library must be installed. If this is not the case, the archive also include a CSV version of the file (`DummyFinancialDatas.csv`).

The function `read_excel` allows to read a `DataFrame` from an Excel file. The `sheet_name` parameter can be used to select the sheet, otherwise the function will return the first sheet of the workbook.

```
fo = pd.read_excel("DummyFinancialDatas.xlsx",
              sheet_name="contrib_fo")
fo.head()
```

```
  dummy_GroupID Term  dummy_FOspread
0         Tokyo   1D         -0.4870
1         Tokyo   1M         -0.4964
2         Tokyo   3M         -0.5025
3         Tokyo   6M         -0.5040
4         Tokyo   9M         -0.5045
```

The function `read_csv` can be used to read data from a CSV file

```
broker = pd.read_csv("GC_datas_broker_10June2020_16-00_LDN.csv")
broker.head()
```

```
  Record;Date;Time;Tenor;Description;Start Date;End Date;Bid;Ask
0  BRRGCAT-01M;10/06/2020;16:00;01M;GC_Tokyo;15-J...
1  BRRGCAT-01W;10/06/2020;16:00;01W;GC_Tokyo;12-J...
2  BRRGCAT-02M;10/06/2020;16:00;02M;GC_Tokyo;15-J...
3  BRRGCAT-03M;10/06/2020;16:00;03M;GC_Tokyo;15-J...
4  BRRGCAT-06M;10/06/2020;16:00;06M;GC_Tokyo;15-J...
```

the function returns a "strange" `DataFrame` containing only one column (that can be found out by looking at `broker.shape[1]` or by calling the `info` function) with a weird name. This file does not comply with the standard CSV format and use a semicolon to separate the columns instead of a comma. The `read_csv` function has a lot of parameters to adapt its behavior to the file format. In our case, it is possible to use the `delimiter` parameter to correctly load the data:

```
broker = pd.read_csv("GC_datas_broker_10June2020_16-00_LDN.csv",
                     delimiter=";")
broker.head()
```

```
        Record        Date   Time Tenor Description   Start Date     End Date  \
0  BRRGCAT-01M  10/06/2020  16:00   01M    GC_Tokyo  15-Jun-2020  15-Jul-2020
1  BRRGCAT-01W  10/06/2020  16:00   01W    GC_Tokyo  12-Jun-2020  19-Jun-2020
2  BRRGCAT-02M  10/06/2020  16:00   02M    GC_Tokyo  15-Jun-2020  17-Aug-2020
3  BRRGCAT-03M  10/06/2020  16:00   03M    GC_Tokyo  15-Jun-2020   15-sept-20
4  BRRGCAT-06M  10/06/2020  16:00   06M    GC_Tokyo  15-Jun-2020  15-Dec-2020

    Bid   Ask
0 -0.49 -0.54
1 -0.48 -0.52
2 -0.50 -0.54
3 -0.50 -0.54
4 -0.50 -0.54
```

It is a good habit to make sure that the data have been loaded correctly and, in particular, that their type is correct. In our case, we can make sure that `bid` and `ask` have been recognized as real numbers

```
broker.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 72 entries, 0 to 71
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Record       72 non-null     object
 1   Date         72 non-null     object
 2   Time         72 non-null     object
 3   Tenor        72 non-null     object
 4   Description  72 non-null     object
 5   Start Date   72 non-null     object
 6   End Date     72 non-null     object
 7   Bid          72 non-null     float64
 8   Ask          72 non-null     float64
dtypes: float64(2), object(7)
memory usage: 5.2+ KB
```

Now that the two `DataFrames` have been loaded, we can check that they use the same names to identify stocks. More precisely, we will use the combination of the city name and of the maturity as a primary key that will be used as a merging criterion: row with the same city name *and* maturity in the two dataframes will be matched and "aligned" in the resulting `DataFrame`.

For that, we can look at the different values used it the `dummy_GroupID` and `Description` columns:

```
print(fo["dummy_GroupID"].unique())
print(broker["Description"].unique())
```

```
['Tokyo' 'Berlin' 'Bogota' 'Stockholm' 'Rio' 'Marseille' 'Palerme']
['GC_Tokyo' 'GC_Berlin' 'GC_Rio' 'GC_Helsinki' 'GC_Bogota' 'GC_Stockholm'
 'GC_Marseille' 'GC_Lisbonne']
```

They use almost the same names: we will simply have to add (or) remove the `GC_` prefix.

```
fo["Description"] = "GC_" + fo["dummy_GroupID"]

# alternatives:
# fo["dummy_GroupID"].apply(lambda x: "GC_" + x)
# broker["Description"].str.replace("GC_", "")
```

Normalizing the term/tenor column is a little more complicated as there is not "general" rule to transform the values used by the broker into the values used by the front office:

```
print(fo["Term"].unique())
print(broker["Tenor"].unique())
```

```
['1D' '1M' '3M' '6M' '9M' '1W' '1Y']
['01M' '01W' '02M' '03M' '06M' '09M' '12M' 'OVN' 'TOM']
```

Note that `OVN` and `TOM` both correspond to `1D`.

The first solution is to use a function to explicitely describe the mapping between the values:

```
def convert_term(x):

    if x == "12M":
        return "1Y"

    if x == "OVN" or x == "TOM":
        return "1D"

    return x.replace("0", "")

broker["Tenor"].apply(convert_term)
```

```
0      1M
1      1W
2      2M
3      3M
4      6M
       ..
67     6M
68     9M
69     1Y
70     1D
71     1D
Name: Tenor, Length: 72, dtype: object
```

We can also use a dictionary to describe the mapping. This solution might be better than the previous one as it allows us to clearly separate the code from the data (that can for instance be stored in a file and shared between several programs):

```
term_mapping = {'01M': "1M",
                '01W': "1W",
                '02M': "2M",
                '03M': "3M",
                '06M': "6M",
                '09M': "9M",
                '12M': "1Y",
                'OVN': "1D",
                'TOM': "1D"}
broker["Tenor"] = broker["Tenor"].apply(lambda x: term_mapping[x])
```

When normalizing the values, we have also change the name of the columns to ensure that they are the same between the two dataframes. You can also specify different columns name during the merge or change the column names with:

```
fo = fo.rename(columns={"dummy_FOspread": "spread",
                        "Term": "Tenor"})
```

```
merged_df = pd.merge(fo[["Tenor", "Description", "spread"]],
         broker[["Tenor", "Description", "Bid", "Ask"]],
         on=["Tenor", "Description"])
```

Now that the data from the front office and from the broker are "matched" we can use the `between` function to check if the spread complies with the constraint: ask $\leq$ spread $\leq$ bid.

We will also build a column to indicate when there is an alert.

```
merged_df["Alert"] = ~merged_df["spread"].between(merged_df["Ask"],
                                                   merged_df["Bid"])
merged_df["Alert"] = merged_df["Alert"].apply(lambda x: "Alert" if x else "")
```

The style attribute allows us to control how the DataFrame is rendered in a a notebook or exported to an excel file using CSS attributes. Note that the style attribute only change the way the DataFrame is rendered once. That is why a modification of the style *must be* chained with the the `to_excel` function to export the information to the excel file.

```
def is_alert(row):
    if row["Alert"] == "Alert":
        return ["background-color:red;color:white"] * len(row)
    else:
        return ["background-color:green"] * len(row)

merged_df.sample(10).style.apply(is_alert, axis=1)
```

```
<pandas.io.formats.style.Styler at 0x13c5e67b0>
```

```
merged_df.style.apply(is_alert, axis=1).to_excel("export.xlsx")
```

CHAPTER

ELEVEN

PRACTICAL STUDY: ANALYZING CONNECTIONS TO WIKIPEDIA PAGES

```
import pandas as pd
```

We will start by considering only the data stored in `wikipedia1.csv`

```
from pathlib import Path

# download and uncompress the files if necessary
if not Path("wikipedia1.csv").is_file():
    !curl -L https://short.do/av2RWt -o wiki.zip
    !unzip wiki.zip
```

```
df = pd.read_csv("wikipedia1.csv")
```

11.1 Reducing memory occupation

The `info` method gives an estimation of the memory used to store the `DataFrame` (here ~600MB)

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145063 entries, 0 to 145062
Columns: 551 entries, Page to 2016-12-31
dtypes: float64(550), object(1)
memory usage: 609.8+ MB
```

It is possible to significantly reduce the memory consumption in this case by observing that views are stored into `float` while they are integers. Indeed, missing values are represented with `nan` that can not be represented by integer. After replacing `nan` with 0, it is possible to convert views into integers to save memory.

```
df = df.fillna(0)
for x in df.columns[1:]:
    df[x] = pd.to_numeric(df[x], downcast="integer")
```

Our code is based on the assumption that the first column contains the page name and that all other columns correspond to the number of connections. To make the code more robust, it is possible to select the columns to be converted according to their type rather than their position by doing :

```
for x in df.select_dtypes("float64").columns:
    df[x] = pd.to_numeric(df[x], downcast="integer")
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145063 entries, 0 to 145062
Columns: 551 entries, Page to 2016-12-31
dtypes: int32(550), object(1)
memory usage: 305.5+ MB
```

The size of the dataframe has been reduced by half.

11.2 Analyzing traffic on Wikipedia pages

```
df.head(5)[df.columns[:5]]
```

```
                                                Page  2015-07-01  2015-07-02  \
0            2NE1_zh.wikipedia.org_all-access_spider          18          11
1             2PM_zh.wikipedia.org_all-access_spider          11          14
2              3C_zh.wikipedia.org_all-access_spider           1           0
3         4minute_zh.wikipedia.org_all-access_spider          35          13
4  52_Hz_I_Love_You_zh.wikipedia.org_all-access_s...           0           0

   2015-07-03  2015-07-04
0           5          13
1          15          18
2           1           1
3          10          94
4           0           0
```

Each row correspond to an access to a wikipedia and depends on:

- the page that is view (e.g. the "Tour de France" page on the French wikipedia `fr.wikipedia.org`)
- how the page was accessed (e.g. from a desktop, when the page ends with `desktop_all-agents`

Each column correspond to a date and cells contain the number of time this page has been on a given date with a given agent.

```
print(f"There are {df.shape[0]:,} rows")
```

```
There are 145,063 rows
```

First, we will try to extract the domain on which the page is hosted. This is a good proxy of the language used in page.

The domain follows the pattern: `XX.wikipedia.org` where `XX` corresponds to two letters identifying the langue (ISO-631 convention).

We will first test our regexp on a single page:

```
import re

def extract_lang(page):
    res = re.search("[a-z][a-z].wikipedia.org", page)
```

(continues on next page)

(continued from previous page)

```
    if res != None:
        return res.group(0)

    return None

page = "3C_zh.wikipedia.org_all-access_spider"
extract_lang(page)
```

```
'zh.wikipedia.org'
```

Note that we return a `None` value (a special value used by python to indicate variables the value of which is not set/unknown) when the language can not be identified to make filtering or counting these lines easier (e.g. with `dropna`).

We apply the function to the whole column:

```
df["Page"].apply(extract_lang).head()
```

```
0    zh.wikipedia.org
1    zh.wikipedia.org
2    zh.wikipedia.org
3    zh.wikipedia.org
4    zh.wikipedia.org
Name: Page, dtype: object
```

As the warning suggests (at least in version 1.5.1 of pandas) using apply is generally a bad idea (the code is not vectorized and pandas will internally loop over all the lines).

A better solution is to use the `extract` function to "apply" the regexp and identify the page information:

```
df["lang"] = df["Page"].str.extract("([a-z][a-z]).wikipedia.org")[0]
df[df.columns[:3]]
```

```
/var/folders/nq/c36kxp5x7mg1sjq0lzs5h_jm0000gp/T/ipykernel_71878/2484974235.py:1:
↪PerformanceWarning: DataFrame is highly fragmented.  This is usually the result
↪of calling `frame.insert` many times, which has poor performance.  Consider
↪joining all columns at once using pd.concat(axis=1) instead. To get a de-
↪fragmented frame, use `newframe = frame.copy()`
  df["lang"] = df["Page"].str.extract("([a-z][a-z]).wikipedia.org")[0]
```

```
                                                      Page  2015-07-01  \
0                    2NE1_zh.wikipedia.org_all-access_spider          18
1                     2PM_zh.wikipedia.org_all-access_spider          11
2                      3C_zh.wikipedia.org_all-access_spider           1
3                 4minute_zh.wikipedia.org_all-access_spider          35
4       52_Hz_I_Love_You_zh.wikipedia.org_all-access_s...           0
...                                                    ...         ...
145058  Underworld_(serie_de_películas)_es.wikipedia.o...           0
145059  Resident_Evil:_Capítulo_Final_es.wikipedia.org...           0
145060  Enamorándome_de_Ramón_es.wikipedia.org_all-acc...           0
145061  Hasta_el_último_hombre_es.wikipedia.org_all-ac...           0
145062  Francisco_el_matemático_(serie_de_televisión_d...           0

        2015-07-02
0               11
1               14
```

(continues on next page)

(continued from previous page)

```
2              0
3             13
4              0
...          ...
145058         0
145059         0
145060         0
145061         0
145062         0

[145063 rows x 3 columns]
```

Note that, the `extract` method is returning a dataframe and we have to extract the column we are interested in before inserting it to the dataframe to improve performances.

We can now compute language distribution:

```
df["lang"].value_counts()
```

```
lang
en    24108
ja    20431
de    18547
fr    17802
zh    17229
ru    15022
es    14069
Name: count, dtype: int64
```

As most `pandas` methods, the `value_counts` method ignores `nan` values. They can be included (e.g. to evaluate the quality of our regexp) with the following command:

```
df["lang"].value_counts(dropna=False)
```

```
lang
en     24108
ja     20431
de     18547
NaN    17855
fr     17802
zh     17229
ru     15022
es     14069
Name: count, dtype: int64
```

We are missing quite a lot of pages that essentially to mediawiki page. To see a sample of the page for which are not able to extract the domain:

```
df[df["lang"].isna()]["Page"].sample(5)
```

```
78683    File:Rich_tea.jpg_commons.wikimedia.org_mobile...
46330    File:Torre_de_Hércules,_La_Coruña,_España,_201...
82945    File:Céline_Dion_House,_Laval_06.jpg_commons.w...
20917    Manual:Installation_guide/fr_www.mediawiki.org...
44160    Category:Photographs_by_photographer_commons.w...
Name: Page, dtype: object
```

The page that we can not analyze properly are those pointing to `commons.wikimedia.org` and to `www.mediawiki.org`. We can update the regexp to recognize them properly:

```
df["Page"].str.extract("([a-z][a-z].wikipedia.org|commons.wikimedia.org|www.mediawiki.
↪org)").value_counts(dropna=False)
```

```
0
en.wikipedia.org         24108
ja.wikipedia.org         20431
de.wikipedia.org         18547
fr.wikipedia.org         17802
zh.wikipedia.org         17229
ru.wikipedia.org         15022
es.wikipedia.org         14069
commons.wikimedia.org    10555
www.mediawiki.org         7300
Name: count, dtype: int64
```

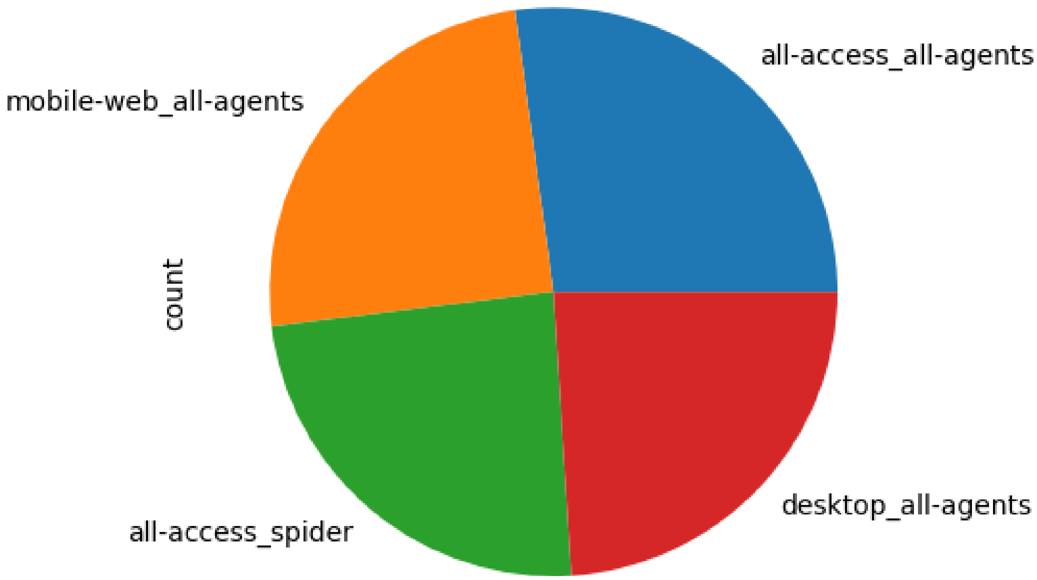
This time, we are able to process all pages correctly: there are no `nan`.

11.3 Extracting User-Agent Information

The previous regular expression can be generalized to :

- be more readable by using named group in the regexp;
- extract also user-agent

```
agent_df = df["Page"].str.\
    extract("(?P<site>(?P<lang>[a-z][a-z]).wikipedia.org|commons.wikimedia.org|www.
↪mediawiki.org)_(?P<agent>.*)")
ax = agent_df["agent"].value_counts().plot(kind="pie")
```



Note that pie chart are usually a bad idea (see here for instance).

11.4 Identify pages that are accessed from a mobile phone only

```
pages = df["Page"].str.extract("(?P<title>.*)_(?P<lang>[a-z][a-z]).wikipedia.org_(?P
↪<agent>.*)")

one_access = pages.groupby(["title", "lang"]).size()\
                    .reset_index()\
                    .rename(columns={0: "count"})\
                    .query("count == 1")
```

```
what = 'mobile-web_all-agents'

pd.merge(one_access, pages, how="left", on=["title", "lang"])\
        .query("agent == @what")\
        .sample(5)     # to reduce output
```

```
                              title lang  count                  agent
7826                           Opio   es      1  mobile-web_all-agents
7941                    Pain_&_Gain   es      1  mobile-web_all-agents
5199                      Jan_Oblak   es      1  mobile-web_all-agents
13238                     ⍰⍰⍰⍰⍰⍰⍰⍰⍰   zh      1  mobile-web_all-agents
2175   Captain_Marvel_(DC_Comics)   en      1  mobile-web_all-agents
```

11.5 Graphical representation of the data

For each language/domain plots the number of page viewed each day

For the moment, each date corresponds to a different column. To plot the number of views according to their date, we have to transform the `DataFrame` so that all dates are stored in the same column, with the corresponding number of views in the same row, as in the following figure:

	Page	Date1	Date2	...
0	P_0	A	B	...
1	P_1	C	D	...
2	P_2	E	F	...
⋮				

$\longrightarrow$

	Page	Date	Views
0	P_0	Date1	A
1	P_1	Date1	C
2	P_2	Date1	E
⋮	⋮	⋮	⋮
n	P_0	Date2	B
n+1	P_1	Date2	D
n+2	P_2	Date2	F
⋮	⋮	⋮	⋮

This *unpivoting* transformation corresponds to the `melt` operation.

```
df = df.melt(id_vars=['Page', "lang"],
             value_vars=df.columns[1:-1],
             var_name='Date',
             value_name='Views')
df.head(5)
```

```
                                                 Page lang        Date  Views
0             2NE1_zh.wikipedia.org_all-access_spider   zh  2015-07-01     18
1              2PM_zh.wikipedia.org_all-access_spider   zh  2015-07-01     11
2               3C_zh.wikipedia.org_all-access_spider   zh  2015-07-01      1
3          4minute_zh.wikipedia.org_all-access_spider   zh  2015-07-01     35
4   52_Hz_I_Love_You_zh.wikipedia.org_all-access_s...   zh  2015-07-01      0
```

Now we can aggregate views for each language and each date:

```
agg_df = df.groupby(["lang", "Date"])["Views"].sum()
agg_df
```

```
lang  Date
de    2015-07-01    13260519
      2015-07-02    13079896
      2015-07-03    12554042
      2015-07-04    11520379
      2015-07-05    13392347
                      ...
zh    2016-12-27     6478442
      2016-12-28     6513400
      2016-12-29     6042545
      2016-12-30     6111203
      2016-12-31     6298565
Name: Views, Length: 3850, dtype: int32
```

The `groupby` operation result in a **hierarchical index** (here, for instance, the `de` is not repeated, but all corresponding rows are gathered on a single "entry".

We have to "reset" the index to repeat this information on every row so that it can be used by `plot` function.

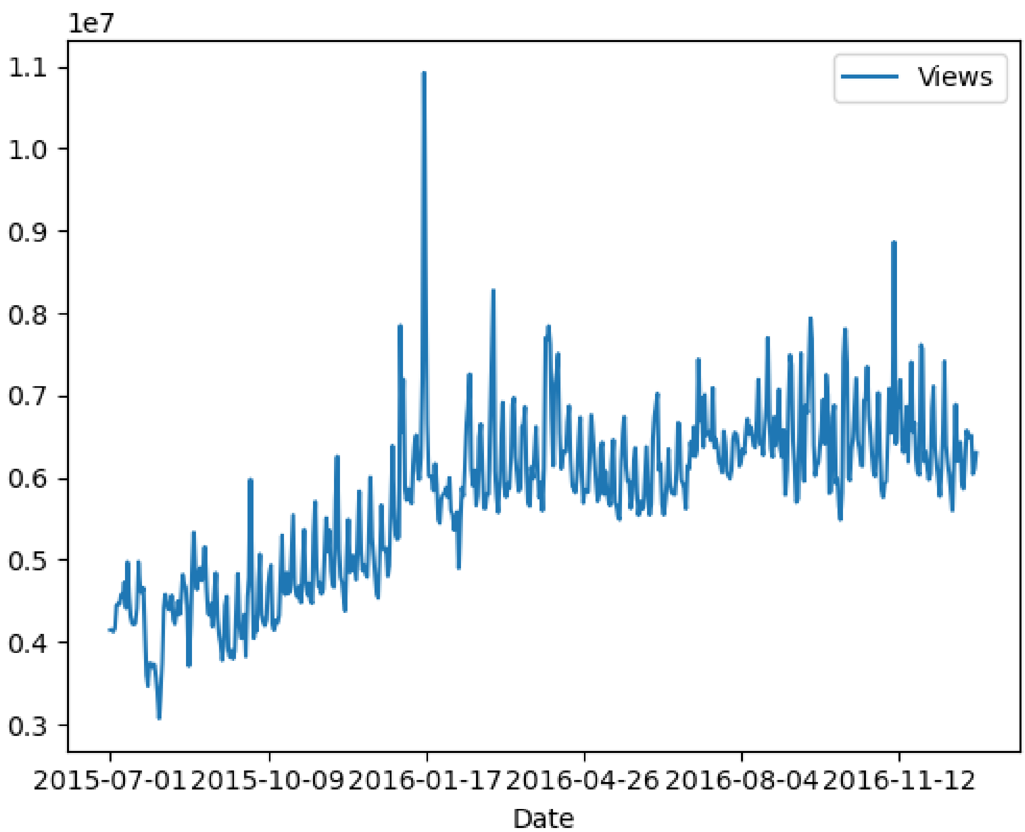
```
agg_df = agg_df.reset_index()
agg_df
```

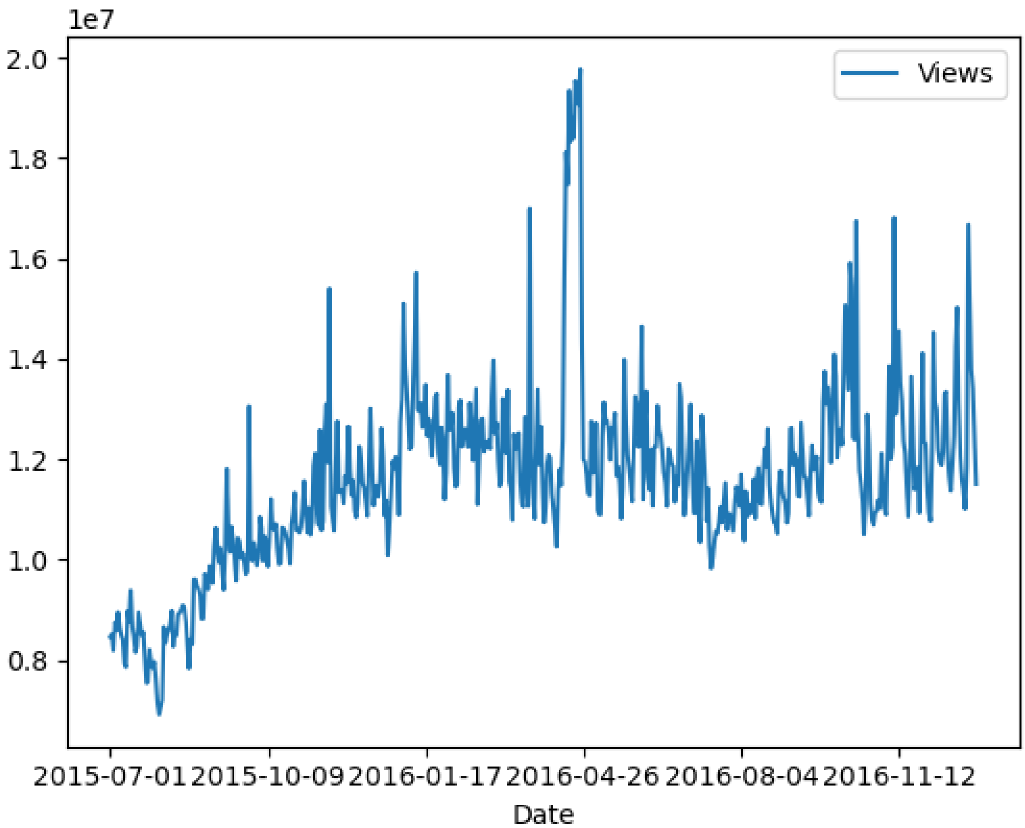
```
     lang        Date     Views
0      de  2015-07-01  13260519
1      de  2015-07-02  13079896
2      de  2015-07-03  12554042
3      de  2015-07-04  11520379
4      de  2015-07-05  13392347
...   ...         ...       ...
3845   zh  2016-12-27   6478442
3846   zh  2016-12-28   6513400
3847   zh  2016-12-29   6042545
3848   zh  2016-12-30   6111203
3849   zh  2016-12-31   6298565

[3850 rows x 3 columns]
```

We can now plot the number of view on the French and Chinese wikipedia.

```
agg_df[agg_df["lang"] == "zh"].plot(x="Date", y="Views")
agg_df[agg_df["lang"] == "fr"].plot(x="Date", y="Views")
```

```
<Axes: xlabel='Date'>
```

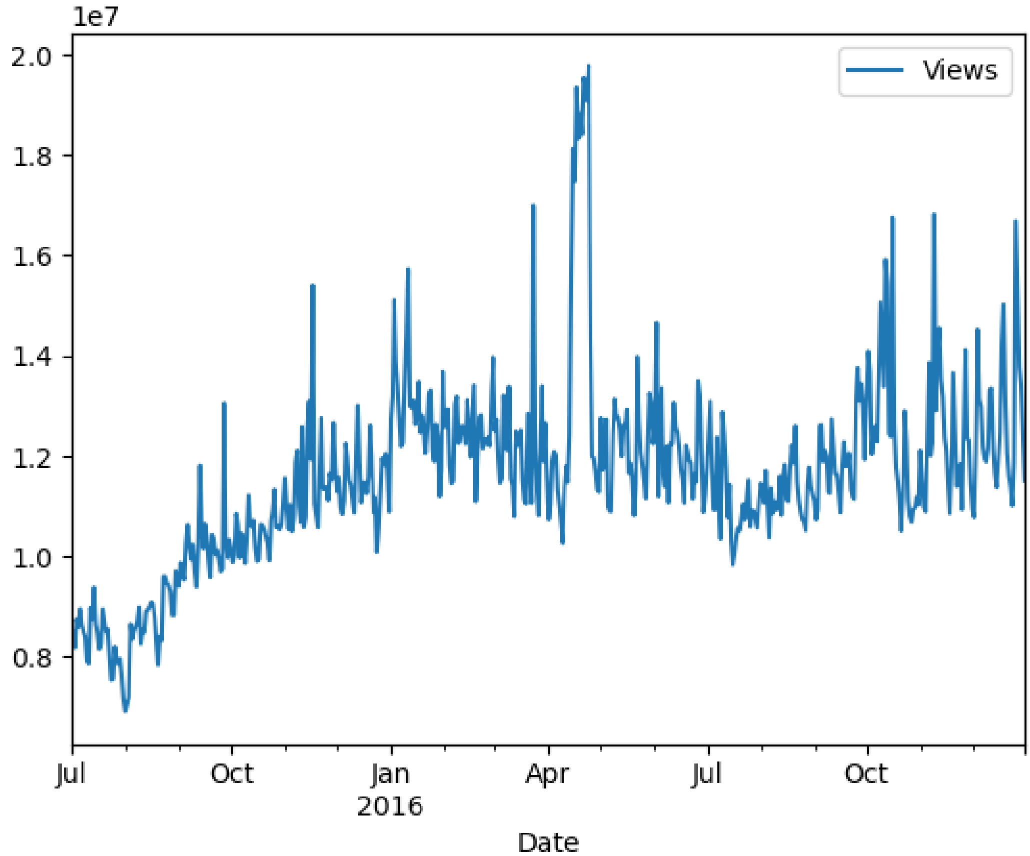




Note that the plot is clearer (the labels on the x-axis are displayed in a more readable way) if the `Date` has a correct type: for the moment, dates are represented as a `str` (its dtypes is `object`) while it should be a `datetime` object.

```
agg_df["Date"] = pd.to_datetime(agg_df["Date"])
agg_df[agg_df["lang"] == "fr"].plot(x="Date", y="Views")
```

```
<Axes: xlabel='Date'>
```



The previous processing can be (greatly) simplified by using the pivot_table method directly:

```
# Select the language to minimise processing time + display only the first five␣
↪columns.
df.query("lang == 'de' or lang == 'ru'")\
  .pivot_table(index="lang", columns="Date", values="Views", aggfunc="sum").iloc[:,␣
↪:5]
```

```
Date  2015-07-01  2015-07-02  2015-07-03  2015-07-04  2015-07-05
lang
de      13260519    13079896    12554042    11520379    13392347
ru       9463854     9627643     8923463     8393214     8938528
```

11.6 Finding the date with the highest number of view

For each language, find the day with the highest number of view:

```
agg_df.set_index("Date").groupby("lang")["Views"].agg(["idxmax", "max"])
```

```
          idxmax        max
lang
de    2016-12-25   23760349
en    2016-07-26  202062970
```

(continues on next page)

(continued from previous page)

```
es   2016-11-09   29871935
fr   2016-04-24   19773082
ja   2016-01-11   29422004
ru   2016-07-28   44537181
zh   2016-01-16   10922626
```

Finding the largest number of views for each language can be easily done with a simple `groupby`.

The trick, here, is to transform the `DataFrame` first to use the `Date` column as an index in order to take advantage of the `idxmax` function to find the date corresponding to the largest value.

Similarly, it is possible to find the page with the largest number of views for each language, using the same trick:

```
df.set_index("Page").groupby("lang")["Views"].agg(["idxmax", "max"])
```

```
                                                  idxmax       max
lang
de    Wikipedia:Hauptseite_de.wikipedia.org_all-acce...   3907598
en     Main_Page_en.wikipedia.org_all-access_all-agents  67264258
es    Wikipedia:Portada_es.wikipedia.org_all-access_...   3471430
fr    Organisme_de_placement_collectif_en_valeurs_mo...   3414474
ja     ⍰⍰⍰⍰⍰⍰⍰⍰⍰_ja.wikipedia.org_all-access_all-agents   2195285
ru    Заглавная_страница_ru.wikipedia.org_all-access...  17846030
zh    ⍰⍰⍰⍰⍰⍰⍰⍰⍰Q_zh.wikipedia.org_all-access_all-agents    827706
```

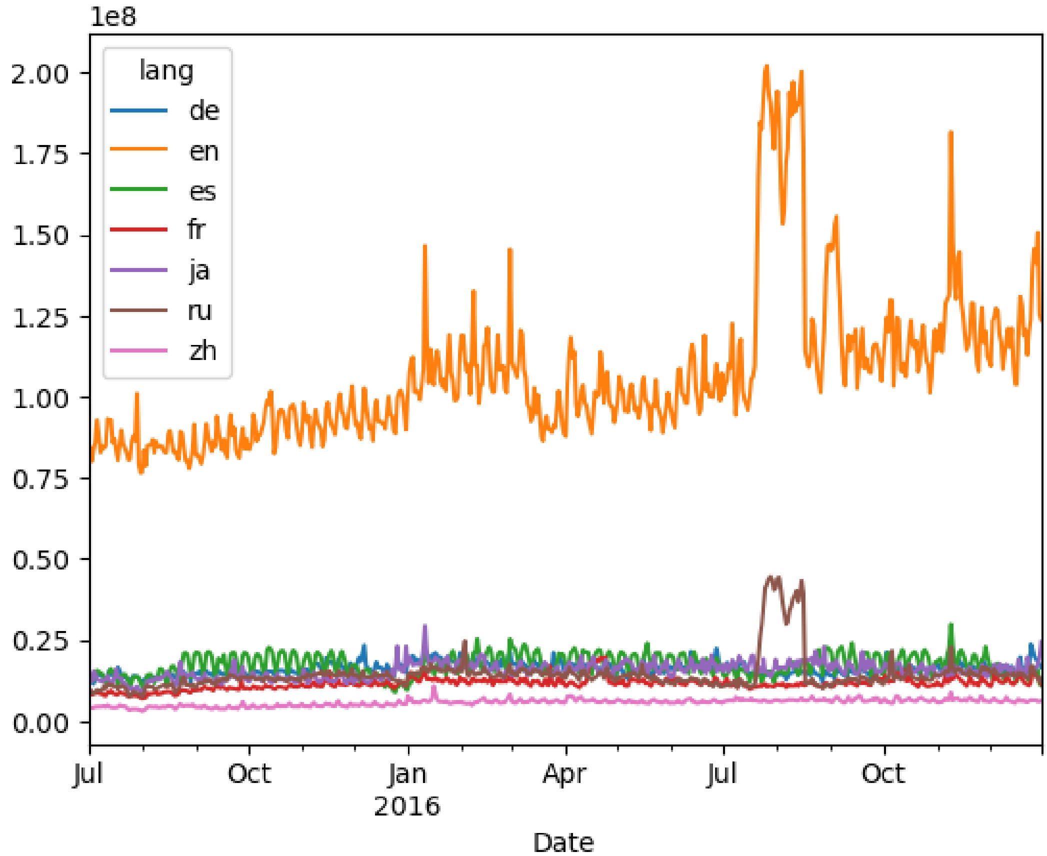
11.7 Represent the number of views for all languages

This can be easily done by taking advantage of possibility of passing a list of columns to the `plot` function; each column will be displayed with a different color.

That is why, we first have to use the `pivot`function to create a column with the number of views for each language and than plot the resulting `DataFrame`

```
p = agg_df.pivot(index="Date",
                 columns="lang",
                 values="Views")\
          .reset_index()
# plot all columns but the first one
p.plot(x="Date", y=p.columns[1:])
```

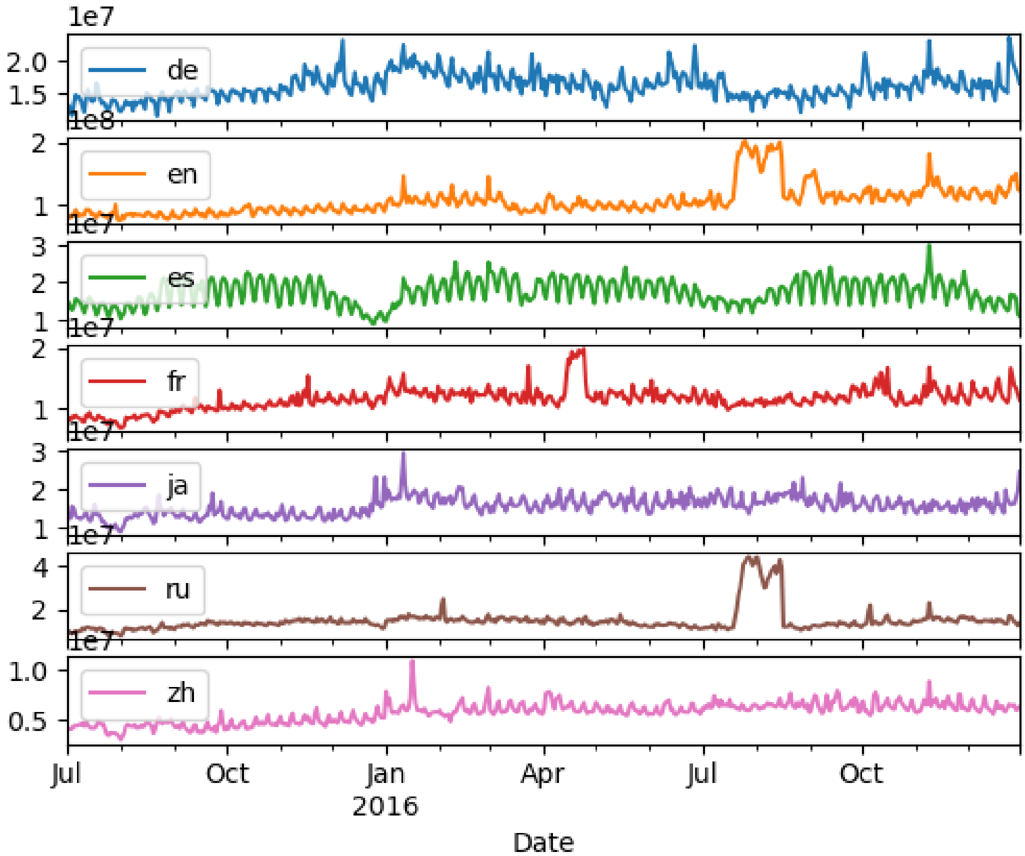
```
<Axes: xlabel='Date'>
```



It is also possible to plot each curve on different axis by passing `subplots=True` to the `plot` function

```
p.plot(x="Date", y=p.columns[1:], subplots=True)
```

```
array([<Axes: xlabel='Date'>, <Axes: xlabel='Date'>,
       <Axes: xlabel='Date'>, <Axes: xlabel='Date'>,
       <Axes: xlabel='Date'>, <Axes: xlabel='Date'>,
       <Axes: xlabel='Date'>], dtype=object)
```



11.8 Merging `DataFrames`

The file `wikipedia2.csv` contains views of wikipedia pages for another set of dates.

We will :

- merge the two dataframes
- plot the number of views of the "Tour de France" page on the whole period.

We start by reloading the two dataframes so they will have the "structure" (rows/columns)

```
wiki1 = pd.read_csv("wikipedia1.csv")
wiki2 = pd.read_csv("wikipedia2.csv")
```

```
print(wiki1.shape)
print(wiki2.shape)
```

```
(145063, 551)
(145063, 254)
```

The two dataframes have the same number of rows but different number of columns. We can assume that the i-th row in `wiki1` describes the same page as the i-th row in `wiki2` and that the two dataframes have different columns that correspond to different dates.

Let us check these two hypotheses.

We first check that the `Page` column is the same in the two dataframes by:

- comparing the cell row by row (test of equality between the two columns that will create a new column)
- aggregating the results with the `all` method

```
(wiki1["Page"] == wiki2["Page"]).all()
```

```
np.True_
```

```
wiki1_dates = pd.to_datetime(wiki1.columns[1:])
wiki2_dates = pd.to_datetime(wiki2.columns[1:])

print(f"Period in first dataframe: {min(wiki1_dates):%Y-%m-%d} → {max(wiki1_dates):%Y-
↪%m-%d}")
print(f"Period in second dataframe: {min(wiki2_dates):%Y-%m-%d} → {max(wiki2_dates):
↪%Y-%m-%d}")
```

```
Period in first dataframe: 2015-07-01 → 2016-12-31
Period in second dataframe: 2017-01-01 → 2017-09-10
```

the two dataframes describe consecutive periods. Merging consists in copying the columns of the second dataframe after the columns of the first dataframe.

This can be done simply with either the `concat` function:

```
df = pd.concat([wiki1, wiki2.drop("Page", axis=1)],
               axis=1)
```

We remove the `Page` column before merging to avoid having two columns with the same name. Note that the `Page` column **is not** removed from `wiki2`: the `dropna` function like almost all `pandas` function does not modifiy the dataframe it operates on; it rather creates a new dataframe (that is not assigned to any variable) that get concatenated to `wiki1` and is then discarded.

The `merge` function can also be used:

```
tmp_df = pd.merge(wiki1, wiki2, on="Page", how="inner")
tmp_df.head(5)[tmp_df.columns[:5]]
```

```
                                                Page  2015-07-01  2015-07-02  \
0            2NE1_zh.wikipedia.org_all-access_spider        18.0        11.0
1             2PM_zh.wikipedia.org_all-access_spider        11.0        14.0
2              3C_zh.wikipedia.org_all-access_spider         1.0         0.0
3         4minute_zh.wikipedia.org_all-access_spider        35.0        13.0
4  52_Hz_I_Love_You_zh.wikipedia.org_all-access_s...         NaN         NaN

   2015-07-03  2015-07-04
0         5.0        13.0
1        15.0        18.0
2         1.0         1.0
3        10.0        94.0
4         NaN         NaN
```

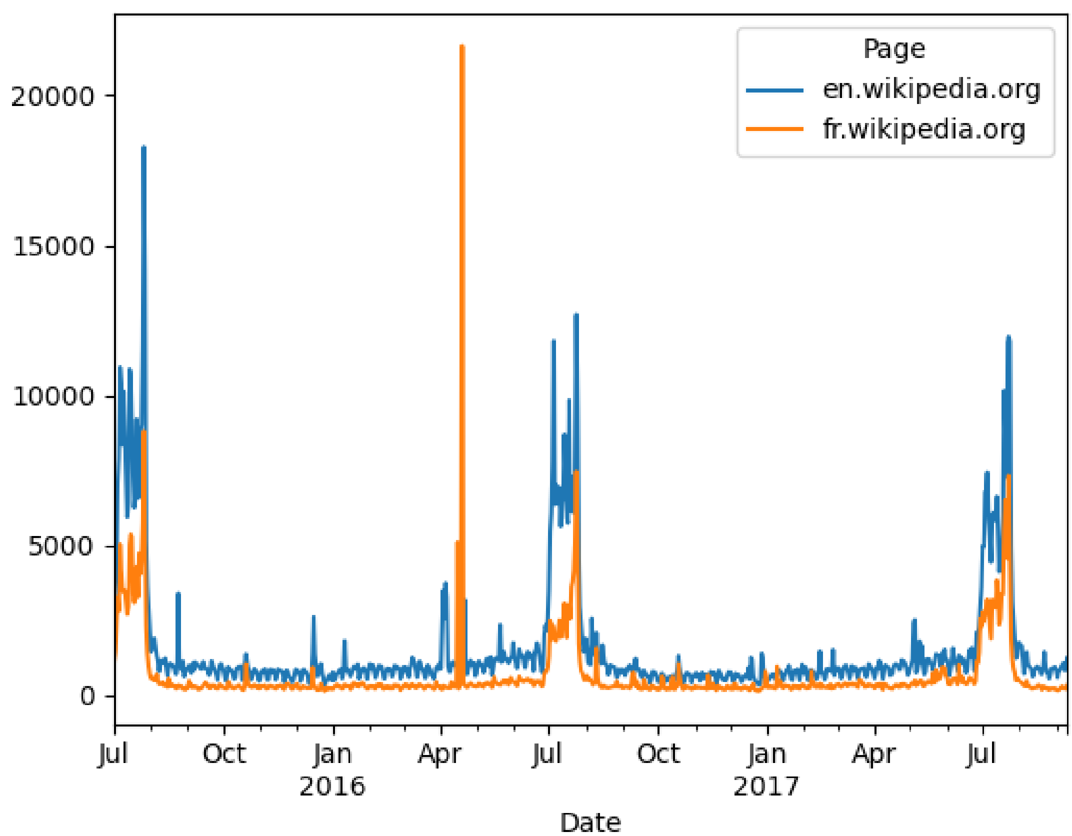
```
# there are other pages related to the Tour de France
# we will only consider this one for the sake of clarity
fr_tour = "Tour_de_France_en.wikipedia.org_desktop_all-ag"
en_tour = "Tour_de_France_fr.wikipedia.org_desktop_all-ag"
```

(continues on next page)

(continued from previous page)

```
tf = df[(df["Page"].str.startswith(fr_tour))|
        (df["Page"].str.startswith(en_tour))
        ].copy()

tf["Page"] = tf["Page"].apply(extract_lang)

tf = tf.melt(id_vars="Page",
             value_vars=tf.columns[1:],
             value_name="Views",
             var_name="Date")

tf["Date"] = pd.to_datetime(tf["Date"])

tf = tf.pivot(index="Date",
              columns="Page",
              values="Views").reset_index()
tf.plot(x="Date", y=tf.columns[1:])
```

```
<Axes: xlabel='Date'>
```



11.9 Correlating views

We now want to determine how similar the behavior (i.e. the number of views) is between two countries (language). To do this we start with the dataframe we built to visualize the number of views per language and per date.

```
df["lang"] = df["Page"].str.extract("([a-z][a-z]).wikipedia.org")

df = df.melt(id_vars=['Page', "lang"],
             value_vars=df.columns[1:-1],
             var_name='Date',
             value_name='Views')\
        .groupby(["lang", "Date"])["Views"].sum()
```

We want to measure the correlation between the number of views by date for each language. For that, we can use the `corr` methods that computes the correlations between all columns of a `DataFrame`. We just have to *pivot* the table first to ensure that data for each language is organized into columns:

```
correlations = agg_df.reset_index().pivot(index="Date",
                        columns="lang",
                        values="Views")\
                 .corr()
correlations
```

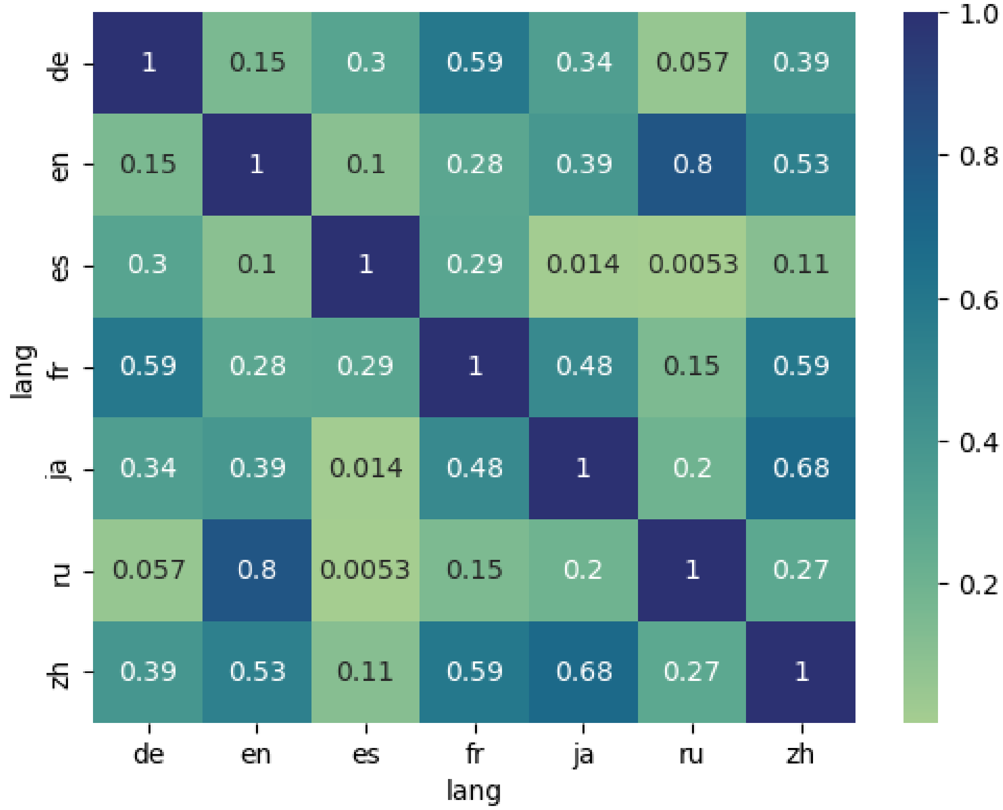
```
lang        de        en        es        fr        ja        ru        zh
lang
de    1.000000  0.154222  0.300227  0.586279  0.337375  0.057187  0.392841
en    0.154222  1.000000  0.101900  0.284672  0.386656  0.799605  0.534174
es    0.300227  0.101900  1.000000  0.290677  0.013924  0.005300  0.113134
fr    0.586279  0.284672  0.290677  1.000000  0.476908  0.147278  0.588744
ja    0.337375  0.386656  0.013924  0.476908  1.000000  0.196398  0.684289
ru    0.057187  0.799605  0.005300  0.147278  0.196398  1.000000  0.273804
zh    0.392841  0.534174  0.113134  0.588744  0.684289  0.273804  1.000000
```

Note that we have to call `reset_index` as the `groupby` operation as used the `Date` and `lang` column to create an index.

Correlations can be represented graphically by the `heatmap` method of the seaborn library:

```
import seaborn as sns
sns.heatmap(correlations, annot=True, cmap="crest")
```

```
<Axes: xlabel='lang', ylabel='lang'>
```

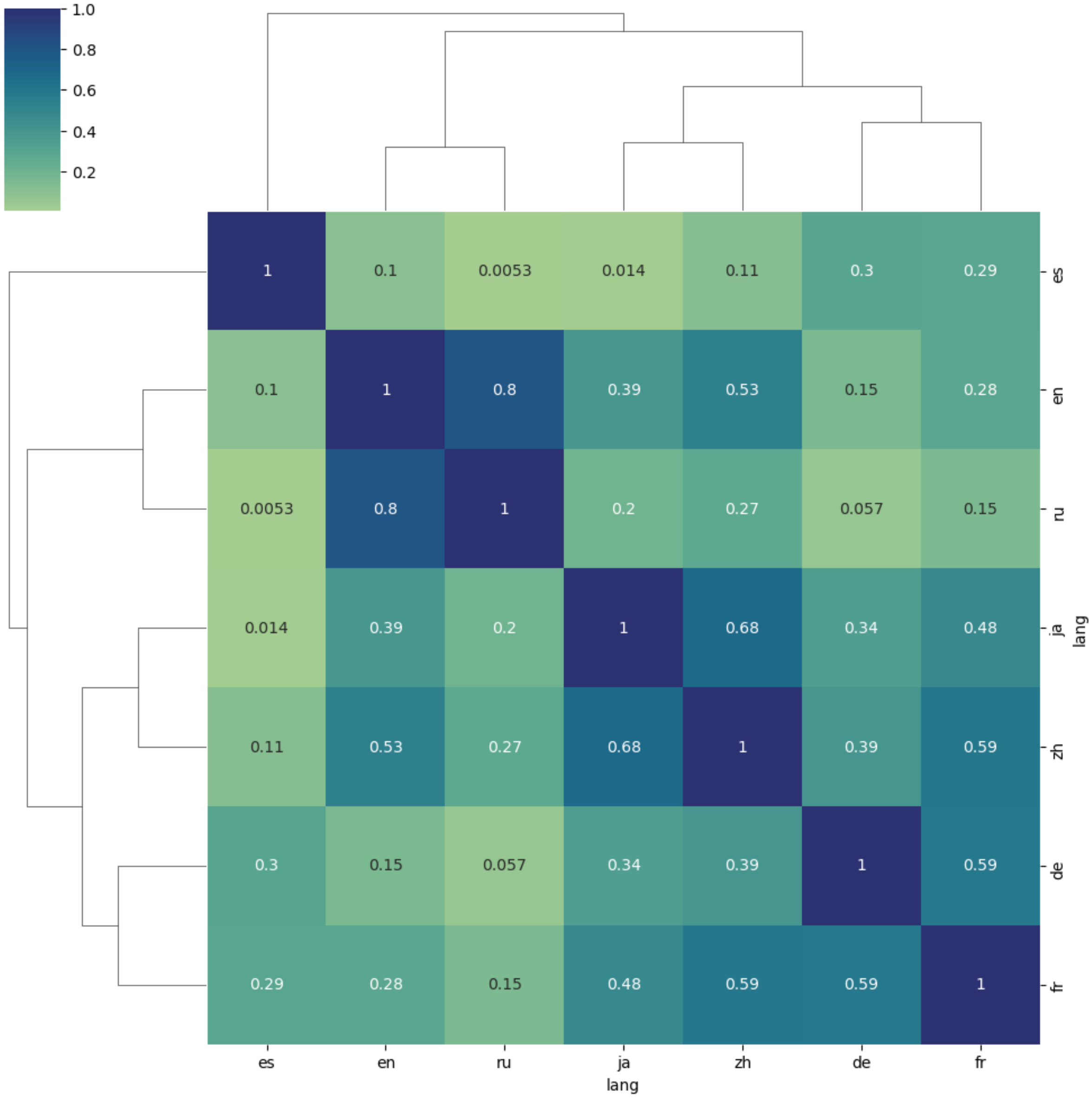


This example shows how the different libraries of the python ecosystem are perfectly designed to be used together: the `heatmap` method takes directly as parameter the result of the `corr` method without any additional transformation.

It is also possible to use more “advanced” visualization methods, such as `clustermap` that will try to highlight the similarity between the columns (here by reorganizing the columns to make “similar” columns appear close).

```
sns.clustermap(correlations, annot=True, cmap="crest")
```

```
<seaborn.matrix.ClusterGrid at 0x310c3aba0>
```


1.0
0.8
0.6
0.4
0.2
1 0.1 0.0053 0.014 0.11 0.3 0.29 es
0.1 1 0.8 0.39 0.53 0.15 0.28 en
0.0053 0.8 1 0.2 0.27 0.057 0.15 ru
0.014 0.39 0.2 1 0.68 0.34 0.48 ja
0.11 0.53 0.27 0.68 1 0.39 0.59 zh
0.3 0.15 0.057 0.34 0.39 1 0.59 de
0.29 0.28 0.15 0.48 0.59 0.59 1 fr
lang
es en ru ja zh de fr
lang

CHAPTER

TWELVE

ANALYZING TRAIN DELAY IN FRANCE

```
# data can be downloaded from http://bit.ly/3hOnpGr
import pandas as pd
```

```
df = pd.read_csv("regularite-mensuelle-tgv-aqst.csv", sep=";")
df["Mois"] = df["Mois"].astype("int8")
df.head(3)
```

```
   Année  Mois   Service Gare de départ Gare d'arrivée  \
0   2017     9  National      PARIS EST           METZ
1   2017     9  National          REIMS      PARIS EST
2   2017     9  National      PARIS EST     STRASBOURG

   Durée moyenne du trajet (min)  Nombre de circulations prévues  \
0                      85.133779                           299.0
1                      47.064516                           218.0
2                     116.234940                           333.0

   Nombre de trains annulés  Commentaire (facultatif) annulations  \
0                       0.0                                   NaN
1                       1.0                                   NaN
2                       1.0                                   NaN

   Nombre de trains en retard au départ  ...  \
0                                  15.0  ...
1                                  10.0  ...
2                                  20.0  ...

   Retard moyen trains en retard > 15min  Nombre trains en retard > 30min  \
0                              24.033333                              1.0
1                              21.498148                              1.0
2                              24.694048                              3.0

   Nombre trains en retard > 60min  Période  Retard pour causes externes  \
0                              0.0  2017-09                    25.000000
1                              0.0  2017-09                    25.000000
2                              0.0  2017-09                    21.428571

   Retard à cause infrastructure ferroviaire Retard à cause gestion trafic  \
0                                   0.000000                     16.666667
1                                  37.500000                     12.500000
2                                  21.428571                      7.142857

   Retard à cause matériel roulant  \
```

(continues on next page)

(continued from previous page)

```
0                        41.666667
1                        12.500000
2                        28.571429

   Retard à cause gestion en gare et réutilisation de matériel  \
0                                          16.666667
1                                           6.250000
2                                          21.428571

   Retard à cause prise en compte voyageurs
0                                     0.00
1                                     6.25
2                                     0.00

[3 rows x 34 columns]
```

This DataFrame contains information about train delays in France. Here are the full list of avalaible information:

```
df.columns
```

```
Index(['Année', 'Mois', 'Service', 'Gare de départ', 'Gare d'arrivée',
       'Durée moyenne du trajet (min)', 'Nombre de circulations prévues',
       'Nombre de trains annulés', 'Commentaire (facultatif) annulations',
       'Nombre de trains en retard au départ',
       'Retard moyen des trains en retard au départ (min)',
       'Retard moyen de tous les trains au départ (min)',
       'Commentaire (facultatif) retards au départ',
       'Nombre de trains en retard à l'arrivée',
       'Retard moyen des trains en retard à l'arrivée (min)',
       'Retard moyen de tous les trains à l'arrivée (min)',
       'Commentaire (facultatif) retards à l'arrivée',
       '% trains en retard pour causes externes (météo, obstacles, colis suspects,
 malveillance, mouvements sociaux, etc.)',
       '% trains en retard à cause infrastructure ferroviaire (maintenance,
 travaux)',
       '% trains en retard à cause gestion trafic (circulation sur ligne
 ferroviaire, interactions réseaux)',
       '% trains en retard à cause matériel roulant',
       '% trains en retard à cause gestion en gare et réutilisation de matériel',
       '% trains en retard à cause prise en compte voyageurs (affluence, gestions
 PSH, correspondances)',
       'Nombre trains en retard > 15min',
       'Retard moyen trains en retard > 15min',
       'Nombre trains en retard > 30min', 'Nombre trains en retard > 60min',
       'Période', 'Retard pour causes externes',
       'Retard à cause infrastructure ferroviaire',
       'Retard à cause gestion trafic', 'Retard à cause matériel roulant',
       'Retard à cause gestion en gare et réutilisation de matériel',
       'Retard à cause prise en compte voyageurs'],
      dtype='object')
```

12.1 Question n°1: Distribution of delays

For each departure station and each date (i.e. pair (`year`, `month`)) compute the percentage of trains that was late and represent it graphically.

We start by extracting the variable we will use:

```
df = df[["Année",
    "Mois",
    "Gare de départ",
    "Nombre de circulations prévues",
    "Nombre de trains annulés",
    "Nombre de trains en retard à l'arrivée"]].rename(columns={"Nombre de trains en␣
↪retard à l'arrivée": "n_retard",
                                                            "Nombre de trains␣
↪annulés": "n_annulés",
                                                            "Nombre de␣
↪circulations prévues": "n_prévues"})
```

As we are interested in the information for each departure station we have to first aggregate all rows corresponding the same key `("Année", "Mois", "Gare de départ")`.

Then (and only then) we can compute the percentage of trains that is late (without forgetting that some trains are also cancelled)

```
retards = df.groupby(["Année", "Mois", "Gare de départ"])[["n_prévues", "n_annulés",
↪"n_retard"]].sum().reset_index()

retards["%"] = retards["n_retard"] / (retards["n_prévues"] - retards["n_annulés"])
```

It is possible to compute the expected `DataFrame` with a pivot operation:

```
with pd.option_context('display.float_format', '{:0.1%}'.format):
    # we should use `retard`, consider only a subsample to reduce output
    df = retards[retards["Année"] == 2015]
    display(df.pivot(index="Gare de départ", columns=["Année", "Mois"], values="%").
↪sample(3))
```

```
Année              2015                                                    \
Mois                  1    2    3    4    5     6     7    8     9   10   11
Gare de départ
DIJON VILLE        6.8% 7.9% 2.9% 6.2% 4.7%  5.4%  9.0% 5.7% 10.3% 7.0% 6.8%
REIMS              4.4% 3.7% 4.7% 3.9% 3.5%  4.9%  4.7% 2.2%  7.1% 5.3% 3.9%
LA ROCHELLE VILLE 4.3% 5.9% 5.9% 3.7% 3.2% 11.5% 11.4% 6.5%  4.1% 4.9% 4.6%

Année
Mois                 12
Gare de départ
DIJON VILLE        5.4%
REIMS              3.3%
LA ROCHELLE VILLE 0.9%
```

Note that:

- we change the way number are formatted in the first line of this cell by setting the corresponding `pandas` option. As the option is set with the `with` instruction, its value will be automatically restored at the end of the block
- as the pivot is computed within a block it is not displayed automatically by `jupyter` and we have to explicetely

ask `jupyter` to render it. To do so, we use the `display` method (a jupyter-specific method) rather than the standard `print` to render the `DataFrame` (with colors and bold characters) rather than just printing its values.

It is also possible to use the `style` option of the `DataFrame` to visualize the evolution of delays across time.

Note that, in this case, we have to use the `method` of the `style` attribute to control how float numbers are rendered rather than fixing the corresponding `pandas` option. We also replace all `nan` with 0 so that there color is "better".

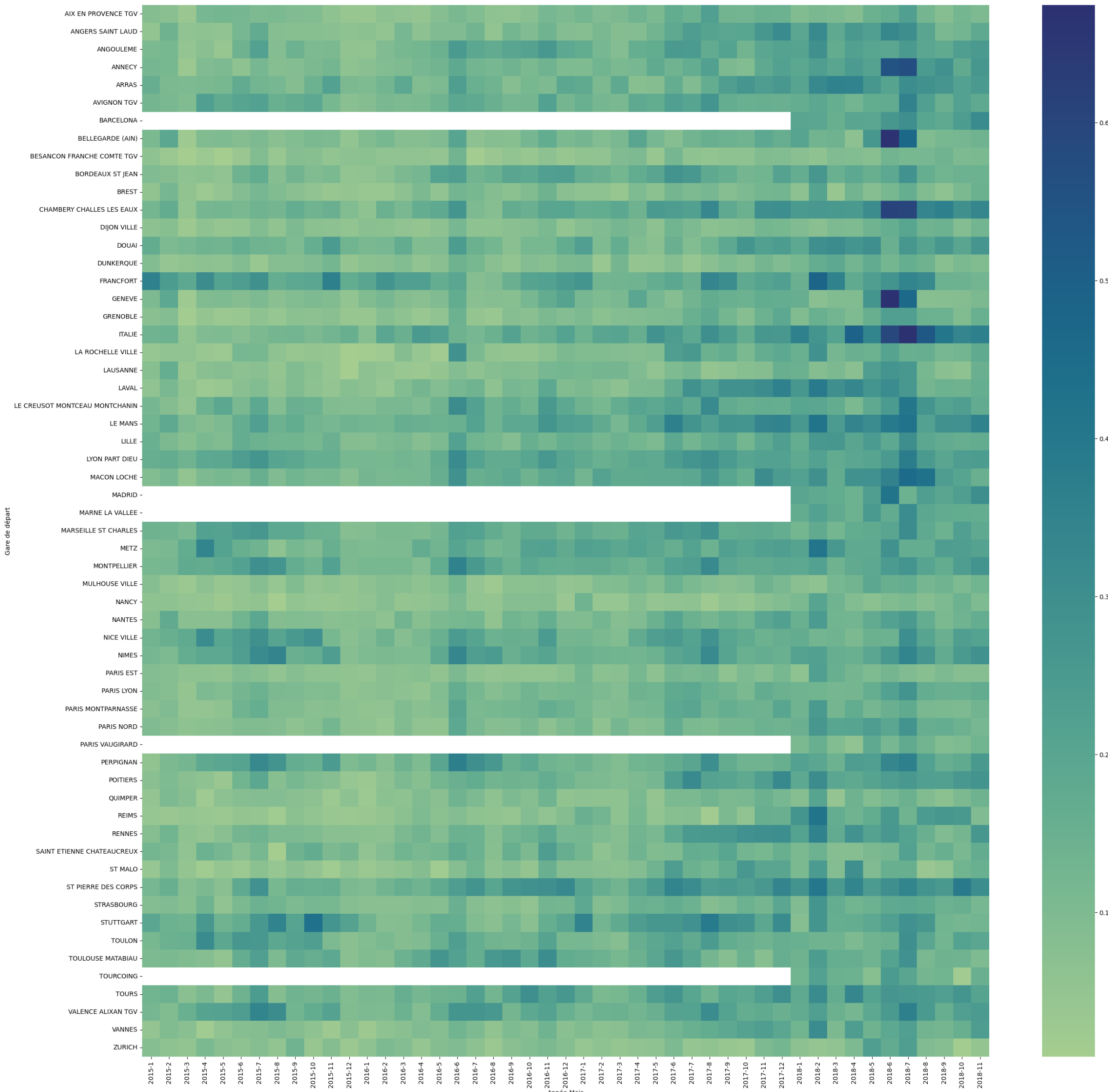
```
retards.pivot(index="Gare de départ", columns=["Année", "Mois"], values="%")\
            .fillna(0)\
            .transpose()\
            .style.background_gradient()\
            .format("{:.1%}")
```

```
<pandas.io.formats.style.Styler at 0x1322741a0>
```

A better/other option is to build a heatmap with seaborn

```
import seaborn as sns
import matplotlib.pyplot as plt

# increase the size of the heatmap
fig, ax = plt.subplots(figsize=(30, 30))
sns.heatmap(retards.pivot(index="Gare de départ", columns=["Année", "Mois"], values="%
↪"), cmap="crest", ax=ax)
```

```
<Axes: xlabel='Année-Mois', ylabel='Gare de départ'>
```



12.2 For each line, finds out the main cause for delays

```
df = pd.read_csv("regularite-mensuelle-tgv-aqst.csv", sep=";")
```

```
df = df[['Gare de départ',
    "Gare d'arrivée",
    '% trains en retard pour causes externes (météo, obstacles, colis suspects,␣
↪malveillance, mouvements sociaux, etc.)',
       '% trains en retard à cause infrastructure ferroviaire (maintenance, travaux)',
       '% trains en retard à cause gestion trafic (circulation sur ligne ferroviaire,␣
↪interactions réseaux)',
       '% trains en retard à cause matériel roulant',
       '% trains en retard à cause gestion en gare et réutilisation de matériel',
       '% trains en retard à cause prise en compte voyageurs (affluence, gestions PSH,
```

(continues on next page)

(continued from previous page)

```
↪ correspondances)',
    ]].rename(columns={
    '% trains en retard pour causes externes (météo, obstacles, colis suspects,␣
↪malveillance, mouvements sociaux, etc.)':
          "externes",
    '% trains en retard à cause infrastructure ferroviaire (maintenance, travaux)':
          "infrastructure",
       '% trains en retard à cause gestion trafic (circulation sur ligne ferroviaire,␣
↪interactions réseaux)':
          "gestion",
       '% trains en retard à cause matériel roulant':
          "matériel",
       '% trains en retard à cause gestion en gare et réutilisation de matériel':
          "gestion",
       '% trains en retard à cause prise en compte voyageurs (affluence, gestions PSH,
↪ correspondances)':
          "voyageurs"}
         )
```

```
group = df.groupby(["Gare de départ", "Gare d'arrivée"]).mean()\
    .reset_index()\
    .melt(
         id_vars=["Gare de départ", "Gare d'arrivée"],
         value_vars=["externes", "infrastructure", "gestion", "matériel", "gestion",
↪"voyageurs"],
         value_name="%",
         var_name="kind")\


group[group.groupby(["Gare de départ", "Gare d'arrivée"])["%"].transform("max") ==␣
↪group["%"]]
```

```
          Gare de départ      Gare d'arrivée      kind         %
0    AIX EN PROVENCE TGV          PARIS LYON  externes  0.349947
3                 ANNECY          PARIS LYON  externes  0.265488
5            AVIGNON TGV          PARIS LYON  externes  0.316737
6              BARCELONA          PARIS LYON  externes  0.418050
7       BELLEGARDE (AIN)          PARIS LYON  externes  0.292547
..                   ...                 ...       ...       ...
595           PARIS LYON         MACON LOCHE  matériel  0.353270
604           PARIS LYON  VALENCE ALIXAN TGV  matériel  0.326314
605           PARIS LYON              ZURICH  matériel  0.333420
626      PARIS VAUGIRARD    BORDEAUX ST JEAN  matériel  0.446814
628      PARIS VAUGIRARD              RENNES  matériel  0.278788

[130 rows x 4 columns]
```

12.3 Understanding `transform`

The `transform` methods can be applied to the result of a `groupby` to "propagate" the result of the groupby to all the elements of a group.

We will illustrate its working on the following toy `DataFrame`:

```
toy = pd.DataFrame({"group": ["A", "B", "A"],
                    "count": [1, 2, 3],
                    "kind": ["tulipe", "rose", "muguet"]})
```

By grouping the `DataFrame` according to the `group` column, it is easy to extract the maximum value of `count` for each `group`:

```
max_values = toy.groupby("group")["count"].max().to_frame()
max_values
```

```
       count
group
A          3
B          2
```

Extracting the corresponding `kind` value (i.e. `muguet` for `A` and `rose` for `B`) is, however, more difficult: the aggregation function (the `max`) can keep track of the values in the columns other than the one it is applied to. It is therefore necessary to manually "match" the value aggregated in each group to all the rows (in the original `DataFrame`) using a `merge`:

```
df = pd.merge(toy, max_values.reset_index(), on="group", validate="m:1")
df
```

```
  group  count_x    kind  count_y
0     A        1  tulipe        3
1     B        2    rose        2
2     A        3  muguet        3
```

The `kind` value can then be easily extracted with a simple filter:

```
df[df["count_x"] == df["count_y"]]
```

```
  group  count_x    kind  count_y
1     B        2    rose        2
2     A        3  muguet        3
```

The `transform` method simplifies this workflow by creating a column (when it is applied to a column!) that has as many as the original `DataFrame`, the value in each row corresponds to the result of the aggregation function for the group the row belong to

```
toy.groupby("group")["count"].transform("max")
```

```
0    3
1    2
2    3
Name: count, dtype: int64
```

In this example, the first and last rows of the `DataFrame` (that are in the same group) contains 3 (the max value for this group). Using this column, the original `DataFrame` can be easily filtered:

```
toy[toy["count"] == toy.groupby("group")["count"].transform("max")]
```

```
  group  count    kind
1     B      2    rose
2     A      3  muguet
```

which gives the expected answer.

CHAPTER

THIRTEEN

ANALYZING US INAUGURAL ADDRESSES

```
from pathlib import Path

import pandas as pd
```

```
if not Path("US_Inaugural_Addresses").is_dir():
    !curl -L https://bit.ly/3AY9nZB -o US_Inaugural_Addresses.tar.bz2
    !tar xvfj US_Inaugural_Addresses.tar.bz2
```

13.1 Data Loading

The corpus is made of a directory containing 58 files, one for each inaugural address.

The name of the file follows the pattern: `{president_number}_{president_name}_{year}.txt`.

```
directory_path = Path("./US_Inaugural_Addresses/")
df = pd.DataFrame({"path": list(directory_path.glob("*.txt"))})

df["content"] = df["path"].apply(lambda x: open(x).read())
df["titles"] = df["path"].apply(lambda x: x.stem)

df = df.set_index("titles")

df = df.drop("path", axis=1)

# keep track of the original dataframe for the last part
speeches = df.copy()
```

Example of a speech (we are only displaying the 500 first characters):

```
print(df.loc["53_clinton_1997"]["content"][:500] + "...")
```

```
Bill Clinton        1/20/1997        My fellow citizens, at this last Presidential␣
↪Inauguration of the 20th century, let us lift our eyes toward the challenges␣
↪that await us in the next century. It is our great good fortune that time and␣
↪chance have put us not only at the edge of a new century, in a new millennium,␣
↪but on the edge of a bright new prospect in human affairs, a moment that will␣
↪define our course and our character for decades to comes. We must keep our old␣
↪democracy forever young. Guided by the a...
```

13.2 tf-idf representations

Our goal is to "characterize" presidents by the speech they gave at their inauguration. The most natural idea is to try to identify the most frequent words they use.

The first step is to be able to identify the words: the computer only sees the sentence `le lion et la hyène` as a sequence of 19 characters (even if the notion of character is ambiguous and, depending on the way this string is encoded, it can have 19 of 20 characters) and has no additional information about the way it should interpret it. Tokenization is the process of segmenting a sentence into a list of words:

```
from sacremoses import MosesTokenizer
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[5], line 1
----> 1 from sacremoses import MosesTokenizer

ModuleNotFoundError: No module named 'sacremoses'
```

```
mt = MosesTokenizer(lang='en')

mt.tokenize("This ain't funny. It's actually hillarious, yet double Ls.",␣
↪escape=False)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 mt = MosesTokenizer(lang='en')
      3 mt.tokenize("This ain't funny. It's actually hillarious, yet double Ls.",␣
↪escape=False)

NameError: name 'MosesTokenizer' is not defined
```

We will use this sacremoses to tokenize our data:

```
# pass return_str=True to keep content a string rather than a list of strings -->␣
↪this is the format expected
# by TfidfVectorizer (see below)
df["content"] = df["content"].apply(lambda x: mt.tokenize(x, escape=False, return_
↪str=True))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[7], line 3
      1 # pass return_str=True to keep content a string rather than a list of␣
↪strings --> this is the format expected
      2 # by TfidfVectorizer (see below)
----> 3 df["content"] = df["content"].apply(lambda x: mt.tokenize(x, escape=False,␣
↪return_str=True))

File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/core/series.py:4935, in Series.apply(self, func, convert_
↪dtype, args, by_row, **kwargs)
   4800 def apply(
   4801     self,
```

(continues on next page)

(continued from previous page)

```
   4802     func: AggFuncType,
   (...)   4807     **kwargs,
   4808 ) -> DataFrame | Series:
   4809     """
   4810     Invoke function on values of Series.
   4811 
   (...)   4926     dtype: float64
   4927     """
   4928     return SeriesApply(
   4929         self,
   4930         func,
   4931         convert_dtype=convert_dtype,
   4932         by_row=by_row,
   4933         args=args,
   4934         kwargs=kwargs,
-> 4935     ).apply()

File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/core/apply.py:1422, in SeriesApply.apply(self)
   1419     return self.apply_compat()
   1421 # self.func is Callable
-> 1422 return self.apply_standard()

File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/core/apply.py:1502, in SeriesApply.apply_standard(self)
   1496 # row-wise access
   1497 # apply doesn't have a `na_action` keyword and for backward compat reasons
   1498 # we need to give `na_action="ignore"` for categorical data.
   1499 # TODO: remove the `na_action="ignore"` when that default has been changed␣
↪in
   1500 #  Categorical (GH51645).
   1501 action = "ignore" if isinstance(obj.dtype, CategoricalDtype) else None
-> 1502 mapped = obj._map_values(
   1503     mapper=curried, na_action=action, convert=self.convert_dtype
   1504 )
   1506 if len(mapped) and isinstance(mapped[0], ABCSeries):
   1507     # GH#43986 Need to do list(mapped) in order to get treated as nested
   1508     #  See also GH#25959 regarding EA support
   1509     return obj._constructor_expanddim(list(mapped), index=obj.index)

File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/core/base.py:925, in IndexOpsMixin._map_values(self,␣
↪mapper, na_action, convert)
    922 if isinstance(arr, ExtensionArray):
    923     return arr.map(mapper, na_action=na_action)
--> 925 return algorithms.map_array(arr, mapper, na_action=na_action,␣
↪convert=convert)

File ~/Dropbox/Mac/Desktop/cours/python4datascience/.pixi/envs/default/lib/python3.
↪13/site-packages/pandas/core/algorithms.py:1743, in map_array(arr, mapper, na_
↪action, convert)
   1741 values = arr.astype(object, copy=False)
   1742 if na_action is None:
-> 1743     return lib.map_infer(values, mapper, convert=convert)
   1744 else:
   1745     return lib.map_infer_mask(
   1746         values, mapper, mask=isna(values).view(np.uint8), convert=convert
```

(continues on next page)

(continued from previous page)

```
   1747     )

File pandas/_libs/lib.pyx:2999, in pandas._libs.lib.map_infer()

Cell In[7], line 3, in <lambda>(x)
      1 # pass return_str=True to keep content a string rather than a list of␣
 ↪strings --> this is the format expected
      2 # by TfidfVectorizer (see below)
----> 3 df["content"] = df["content"].apply(lambda x: mt.tokenize(x, escape=False,␣
 ↪return_str=True))

NameError: name 'mt' is not defined
```

Python provides all the methods to easily find the `n` most frequent words in one speech:

```
from collections import Counter

Counter(df.loc["53_clinton_1997"]["content"].split()).most_common(5)
```

```
[('the', 124), ('of', 96), ('and', 80), ('to', 62), ('a', 59)]
```

```
Counter(df.loc["13_van_buren_1837"]["content"].split()).most_common(5)
```

```
[('the', 240), ('of', 198), ('and', 148), ('to', 134), ('in', 68)]
```

For these two speeches, as for any texts, the most frequent words are **stop words** (in French *mots outils*) that do not bring any information ! We need to find a better representation of the speeches if we hope to extract some information.

There are two possible solutions:

- use a list of stop words and delete all “un-interested” words from your input;
- weights the words according to their importance.

The most used weighting scheme is `tf.idf` that relies on two intuitions to decide the “importance” of a word:

- the more a word appears in a document the more important this word is to determine the meaning of this document;
- a word that appears in all the documents of a corpus provides no information

The `td.idf` scores is the combination of two components to compute the score of a word `w`:

- `tf(w)` (term frequency): number of occurrences of `w` in the document
- `df(w)` (document frequency): number of document in which `w` appears.

Final score is a fonction of `td / df` (`i` stands for “inverse”).

The `tf.idf` score can be easily computed using the `sklearn` library:

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(stop_words='english')
# we have to convert the dataframe into a numpy array (more suited for “heavy”␣
 ↪computations)
data = vectorizer.fit_transform(df["content"].values)
data
```

```
<Compressed Sparse Row sparse matrix of dtype 'float64'
	with 36821 stored elements and shape (58, 8999)>
```

```
# transform the data back into a dataframe
tfidf_df = pd.DataFrame(data.toarray(),
             index=df.index,
             columns=vectorizer.get_feature_names_out())
# only prints the 5 last columns and the 7 last lines
# words are sorted alphabetically
tfidf_df[tfidf_df.columns[-5:]].tail(7)
```

```
                            zachary      zeal   zealous  zealously      zone
titles
42_eisenhower_1953              0.0  0.000000  0.000000   0.000000  0.000000
40_roosevelt_franklin_1945      0.0  0.000000  0.000000   0.000000  0.000000
43_eisenhower_1957              0.0  0.000000  0.000000   0.000000  0.000000
08_monroe_1817                  0.0  0.020943  0.023891   0.000000  0.031879
06_madison_1809                 0.0  0.000000  0.000000   0.048467  0.000000
58_trump_2017                   0.0  0.000000  0.000000   0.000000  0.000000
31_taft_1909                    0.0  0.000000  0.000000   0.000000  0.000000
```

We can also select the word we are interested in to see how relevant they are for a specific speech:

```
tfidf_slice = tfidf_df[['government', 'borders', 'people', 'obama']]
# sample to reduce the size of the output
tfidf_slice.sort_index().round(decimals=2).sample(5)
```

```
                            government  borders  people  obama
titles
39_roosevelt_franklin_1941        0.05      0.0    0.08    0.0
06_madison_1809                   0.00      0.0    0.02    0.0
44_kennedy_1961                   0.00      0.0    0.01    0.0
42_eisenhower_1953                0.01      0.0    0.10    0.0
17_pierce_1853                    0.08      0.0    0.05    0.0
```

```
tfidf_slice.sort_index().sample(10).style.background_gradient().format("{:.2f}")
```

```
<pandas.io.formats.style.Styler at 0x14d4330e0>
```

Now we will look for the 10 most "significant" word for each president.

For one president:

```
# in more recent pandas version we can simply use the `nlargest` method
tfidf_df.loc["57_obama_2013"]\
        .sort_values(ascending=False)\
        .head(10)
```

```
journey       0.167591
creed         0.139659
generation    0.127260
america       0.125044
complete      0.114891
requires      0.114891
people        0.110351
```

(continues on next page)

(continued from previous page)

```
time           0.105563
today          0.103668
evident        0.100896
Name: 57_obama_2013, dtype: float64
```

```
tfidf_df.index.name = "discours_id"
tfidf_df.reset_index()\
         .melt(id_vars="discours_id",
               var_name="word",
               value_name="tf.idf")\
         .sort_values(by=["discours_id", "tf.idf"])\
         .groupby("discours_id")\
         .tail(5)
```

```
               discours_id          word    tf.idf
326622  01_washington_1789         ought  0.103728
238288  01_washington_1789     immutable  0.103883
242174  01_washington_1789   impressions  0.103883
367570  01_washington_1789  providential  0.103883
215030  01_washington_1789    government  0.113681
...                    ...           ...       ...
366558       58_trump_2017     protected  0.132439
268770       58_trump_2017          jobs  0.142766
26852        58_trump_2017      american  0.149226
153292       58_trump_2017        dreams  0.156436
26794        58_trump_2017       america  0.350162

[290 rows x 3 columns]
```

```
df = tfidf_df.reset_index()\
         .melt(id_vars="discours_id",
               var_name="word",
               value_name="tf.idf")\
         .sort_values(by=["discours_id", "tf.idf"], ascending=False)\
         .groupby("discours_id")\
         .head(10)
df["word_rank"] = df.groupby("discours_id")\
         .cumcount() + 1

words = df.pivot(index="discours_id", columns="word_rank", values="word")
values = df.pivot(index="discours_id", columns="word_rank", values="tf.idf")
```

```
words.style.apply(lambda x: values.applymap(color_cells), axis=None)
values.sample(5).style.background_gradient()
```

```
<pandas.io.formats.style.Styler at 0x14cd7cf50>
```

13.3 Finding similarities between speeches

```
from sklearn.metrics import pairwise_distances
from scipy.spatial.distance import squareform

X = pairwise_distances(tfidf_df, metric="cosine")
X.shape
```

```
(58, 58)
```

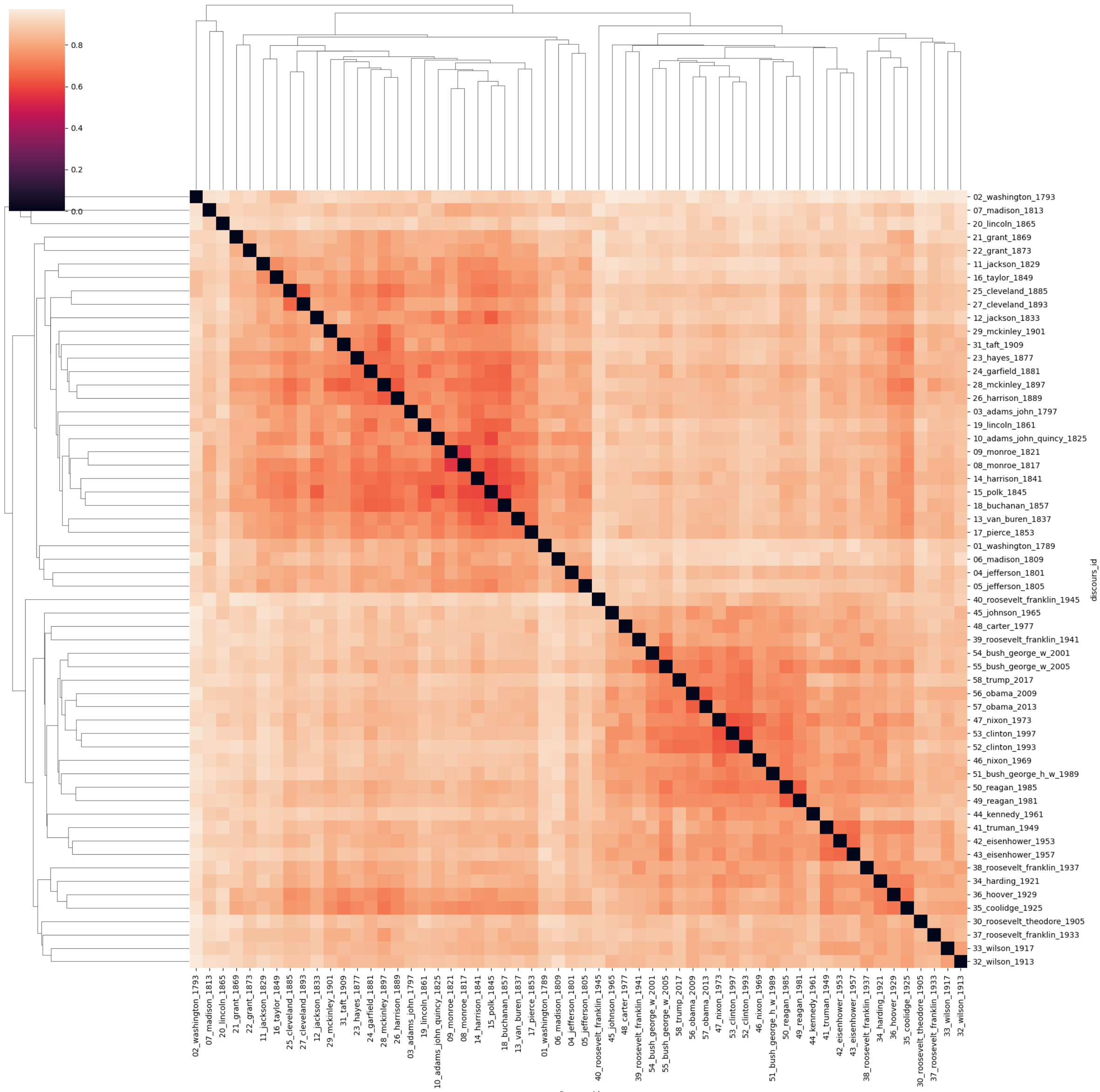
```
sim = pd.DataFrame(X,
            index=tfidf_df.index,
            columns=tfidf_df.index)
```

```
sim.loc["57_obama_2013"].sort_values().head(5)
```

```
discours_id
57_obama_2013            0.000000
56_obama_2009            0.644398
53_clinton_1997          0.685165
52_clinton_1993          0.686302
55_bush_george_w_2005    0.689727
Name: 57_obama_2013, dtype: float64
```

```
import seaborn as sns
from scipy.cluster.hierarchy import ClusterWarning
from warnings import simplefilter
simplefilter("ignore", ClusterWarning)

sns.clustermap(sim, figsize=(20,20))
```

```
<seaborn.matrix.ClusterGrid at 0x14e035160>
```



13.4 Word Clouds

!pip install wordcloud

```
from wordcloud import WordCloud
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[22], line 1
----> 1 from wordcloud import WordCloud

ModuleNotFoundError: No module named 'wordcloud'
```

```
# Generate a word cloud image
wordcloud = WordCloud().generate(speeches.loc["56_obama_2009", "content"])

# Display the generated image:
# the matplotlib way:
import matplotlib.pyplot as plt
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")

# lower max_font_size
wordcloud = WordCloud(max_font_size=40).generate(speeches.loc["56_obama_2009",
↪"content"])
plt.figure()
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.show()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[23], line 2
      1 # Generate a word cloud image
----> 2 wordcloud = WordCloud().generate(speeches.loc["56_obama_2009", "content"])
      4 # Display the generated image:
      5 # the matplotlib way:
      6 import matplotlib.pyplot as plt

NameError: name 'WordCloud' is not defined
```

CHAPTER FOURTEEN

BILLBOARD TOP 100 CHRISTMAS CAROL

This dataset was created by merging together two sources of data: Billboard Top 100 data from 1958 to 2017 and Wikipedia's list of most popular Christmas carols.

```
import pandas as pd
from pathlib import Path
```

```
# download dataset if it does not exist
if not Path("christmas_billboard.csv").is_file():
    !curl -L bit.ly/3vgPSeY -o christmas_billboard.csv
```

```
songs = pd.read_csv("christmas_billboard.csv")
songs.head()
```

```
                                                 url      weekid  \
0  http://www.billboard.com/charts/hot-100/1958-1...  12/13/1958
1  http://www.billboard.com/charts/hot-100/1958-1...  12/20/1958
2  http://www.billboard.com/charts/hot-100/1958-1...  12/20/1958
3  http://www.billboard.com/charts/hot-100/1958-1...  12/20/1958
4  http://www.billboard.com/charts/hot-100/1958-1...  12/27/1958

   week_position              song     performer
0             83   RUN RUDOLPH RUN  Chuck Berry
1             57  JINGLE BELL ROCK  Bobby Helms
2             73   RUN RUDOLPH RUN  Chuck Berry
3             86   WHITE CHRISTMAS  Bing Crosby
4             44   GREEN CHRI$TMA$ Stan Freberg
```

14.1 Questions

1. How many different songs are there?
2. What is the best Christmas song ever? We will consider three different ways to define "best":
    - highest position in the ranking
    - best average position in the ranking
    - largest number of weeks the song has been in the TOP100 ranking
3. What is the duration (in days) of the period covered by the corpus?
4. What is the highest number of songs in the top 100 in the same week?

5. How many songs have more than one performer?
6. Which song has the most different performers?
7. Find the best Christmas song every year (the one the with the best position in chart)
8. Plot the number of songs in the TOP 100 each year
9. Find out the songs that have been in the TOP 100 more than N weeks apart
10. Find out the maximum number of consecutive years a song has been "best Christmas song"

14.2 Solutions

14.2.1 How many different songs are there?

```
# first two possibilities:
songs.groupby("song").size().shape[0]
songs["song"].value_counts().shape[0]
```

```
70
```

```
# direct answer:
songs["song"].nunique()
```

```
70
```

14.2.2 Most Popular Song

```
# 1st criteria: best position in chart

# we use head() only to generate a good-looking pdf
songs.sort_values(by=["week_position", "weekid"], ascending=[True, False]).head(5)
```

```
                                                  url     weekid  \
149  http://www.billboard.com/charts/hot-100/1965-0...   1/9/1965
272  http://www.billboard.com/charts/hot-100/2000-0...   1/8/2000
244  http://www.billboard.com/charts/hot-100/1990-0...   1/6/1990
245  http://www.billboard.com/charts/hot-100/1990-0...  1/13/1990
148  http://www.billboard.com/charts/hot-100/1965-0...   1/2/1965

     week_position                         song            performer
149              7                         AMEN      The Impressions
272              7               AULD LANG SYNE              Kenny G
244              7  THIS ONE'S FOR THE CHILDREN  New Kids On The Block
245              7  THIS ONE'S FOR THE CHILDREN  New Kids On The Block
148              8                         AMEN      The Impressions
```

```
# to “extract” the best songs
songs[songs["week_position"] == songs["week_position"].min()]
```

```
                                                 url      weekid  \
149  http://www.billboard.com/charts/hot-100/1965-0...   1/9/1965
244  http://www.billboard.com/charts/hot-100/1990-0...   1/6/1990
245  http://www.billboard.com/charts/hot-100/1990-0...  1/13/1990
272  http://www.billboard.com/charts/hot-100/2000-0...   1/8/2000

     week_position                         song            performer
149              7                         AMEN      The Impressions
244              7  THIS ONE'S FOR THE CHILDREN  New Kids On The Block
245              7  THIS ONE'S FOR THE CHILDREN  New Kids On The Block
272              7               AULD LANG SYNE              Kenny G
```

```
# 2 criteria: best average position in chart
songs.groupby(["song", "performer"])["week_position"]\
     .mean()\
     .reset_index()\
     .sort_values(by="week_position")\
     .head(5)
```

```
                           song              performer  week_position
67      THIS ONE'S FOR THE CHILDREN  New Kids On The Block      26.250000
2   ALL I WANT FOR CHRISTMAS IS YOU          Mariah Carey      28.947368
48                     PRETTY PAPER           Roy Orbison      30.285714
4                              AMEN       The Impressions      31.818182
17      DO THEY KNOW IT'S CHRISTMAS?              Band-Aid      32.500000
```

```
# 3rd criteria: number of weeks in the chart
songs.groupby(["song", "performer"])["week_position"].size()\
    .reset_index()\
    .rename(columns={"week_position": "n_entries"})\
    .sort_values(by="n_entries", ascending=False)\
    .head(5)
```

```
                                         song  \
32                            JINGLE BELL ROCK
2              ALL I WANT FOR CHRISTMAS IS YOU
50             ROCKIN' AROUND THE CHRISTMAS TREE
58  THE CHIPMUNK SONG (CHRISTMAS DON'T BE LATE)
75                             WHITE CHRISTMAS

                       performer  n_entries
32                  Bobby Helms         20
2                  Mariah Carey         19
50                   Brenda Lee         19
58  David Seville And The Chipmunks         16
75                  Bing Crosby         14
```

14.2.3 Period covered by the DataFrame

We first need to parse the date in the weekid column. This can be done manually:

```
dates = songs["weekid"].str.split("/", expand=True).rename(columns={0: "month",
                                                                     1: "day",
                                                                     2: "year"})

# split returns strings --> convert them to number to make sorting/comparing more␣
↪meaningful
dates["month"] = pd.to_numeric(dates["month"])
dates["day"] = pd.to_numeric(dates["day"])
dates["year"] = pd.to_numeric(dates["year"])

# reduce the output
pd.concat([songs, dates], axis=1).sort_values(by=["year", "month", "day"])[["song",
↪"performer", "month", "year"]].head(3)
```

```
               song    performer  month  year
0    RUN RUDOLPH RUN  Chuck Berry     12  1958
1   JINGLE BELL ROCK  Bobby Helms     12  1958
2    RUN RUDOLPH RUN  Chuck Berry     12  1958
```

A better way is to used pandas function to parse dates:

```
print(pd.to_datetime(songs["weekid"], format="%m/%d/%Y").max() - pd.to_datetime(songs[
↪"weekid"], format="%m/%d/%Y").min())
```

```
21217 days 00:00:00
```

14.2.4 What is the highest number of songs in the top 100 in the same week?

```
songs.groupby("weekid").size().agg(["max", "idxmax"])
```

```
max               10
idxmax    12/17/1960
dtype: object
```

14.2.5 How many songs have more than one performer?

```
s = songs.groupby("song")["performer"].nunique()\
    .reset_index()\
    .rename(columns={"performer": "n_performers"})

s[s["n_performers"] != 1]
```

```
                              song  n_performers
2    ALL I WANT FOR CHRISTMAS IS YOU             2
16      DO THEY KNOW IT'S CHRISTMAS?             2
30                  JINGLE BELL ROCK             2
31                    LAST CHRISTMAS             3
36                         MISTLETOE             2
```

(continues on next page)

(continued from previous page)

```
40   PLEASE COME HOME FOR CHRISTMAS             2
68                  WHITE CHRISTMAS             2
```

Using the `query` method allows us to avoid creating a new `DataFrame`:

```
songs.groupby("song")["performer"].nunique()\
    .reset_index()\
    .rename(columns={"performer": "n_performers"})\
    .query("n_performers > 1")
```

```
                               song  n_performers
2   ALL I WANT FOR CHRISTMAS IS YOU             2
16      DO THEY KNOW IT'S CHRISTMAS?            2
30                  JINGLE BELL ROCK            2
31                    LAST CHRISTMAS            3
36                         MISTLETOE            2
40   PLEASE COME HOME FOR CHRISTMAS             2
68                  WHITE CHRISTMAS             2
```

14.2.6 Which song has the most different performers?

```
songs.groupby("song")["performer"].nunique()\
    .reset_index()\
    .rename(columns={"performer": "n_performers"})\
    .set_index("song")\
    .agg(["max", "idxmax"])
```

```
          n_performers
max                  3
idxmax  LAST CHRISTMAS
```

14.2.7 Best song each year

We will start with a two-step "manual" approach: the first step consist in finding the best position reached each year, and the second is to use this information to find the song title.

```
songs["weekid"] = pd.to_datetime(songs["weekid"])
songs.set_index("weekid")\
     .resample("YE")["week_position"]\
     .min()\
     .head()  # to avoid displaying the whole Serie
```

```
weekid
1958-12-31    35.0
1959-12-31    34.0
1960-12-31    14.0
1961-12-31    12.0
1962-12-31    12.0
Freq: YE-DEC, Name: week_position, dtype: float64
```

Using `resample` makes it very easy to obtain the information we are looking for. But the result is put into a dataframe whose index is the last day of the year, which complicates the merging necessary for the second step: on the one hand, we will have dates aligned with the last day of each week, and on the other, dates aligned with the last day of each year. To facilitate the second step, we can manually group the dates so as to retain only the year and not a complete timestamp.

```
min_pos = songs.groupby(songs["weekid"].dt.year)["week_position"].min()\
    .reset_index()\
    .rename(columns={"week_position": "best_position_this_year",
                     "weekid": "year"})
min_pos.head(5)  # to avoid displaying the whole Serie
```

```
   year  best_position_this_year
0  1958                       35
1  1959                       34
2  1960                       14
3  1961                       12
4  1962                       12
```

Now, we simply have to merge the two dataframes:

```
df = pd.merge(songs,
              min_pos,
              right_on="year",
              left_on=songs["weekid"].dt.year,
              validate="m:1")
# head to avoid listing the whole dataframe
df[df["week_position"] == df["best_position_this_year"]][["year", "song", "performer
↪"]].head()
```

```
    year                             song                      performer
7   1958                 JINGLE BELL ROCK                    Bobby Helms
19  1959               THE HAPPY REINDEER  Dancer, Prancer And Nervous
44  1960  ROCKIN' AROUND THE CHRISTMAS TREE                   Brenda Lee
83  1961                  WHITE CHRISTMAS                   Bing Crosby
90  1962                  WHITE CHRISTMAS                   Bing Crosby
```

These two steps can be carried out in one instruction using the `transform` method:

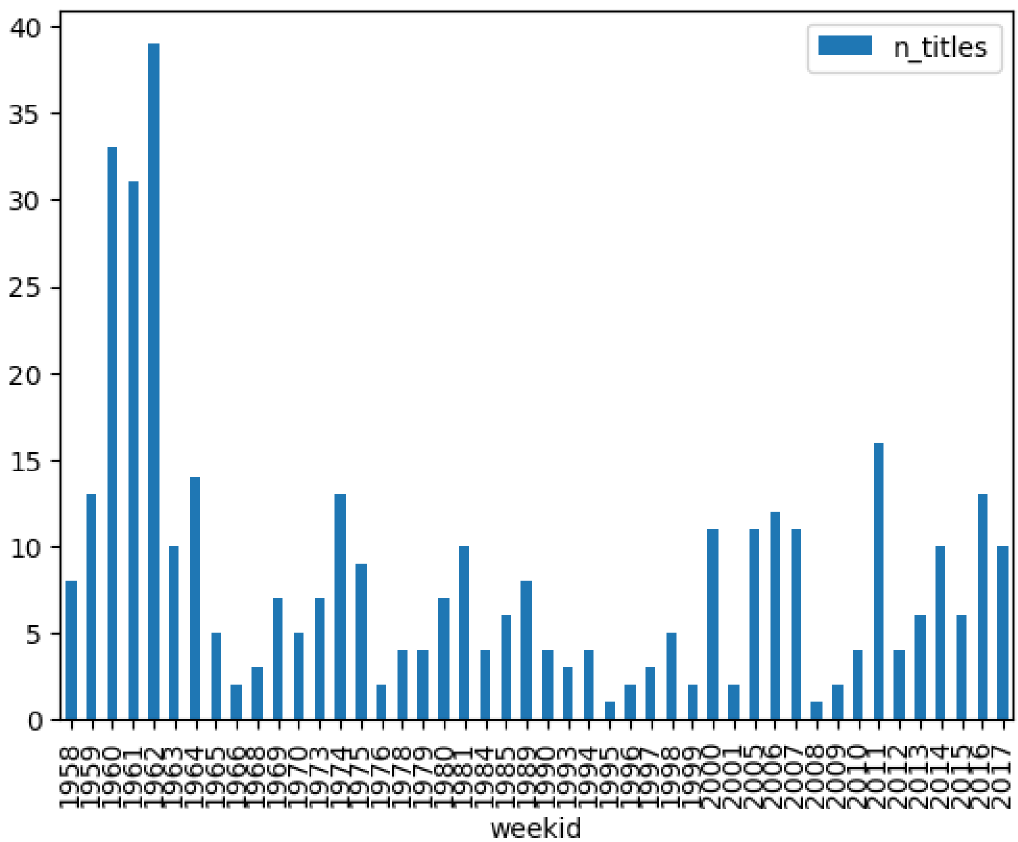
```
songs[songs["week_position"] == songs.groupby(songs["weekid"].dt.year)["week_position
↪"]\
                                     .transform("min")][["weekid", "song", "performer
↪"]].head(5)
```

```
       weekid                               song                      performer
7  1958-12-27                   JINGLE BELL ROCK                    Bobby Helms
19 1959-12-26                 THE HAPPY REINDEER  Dancer, Prancer And Nervous
44 1960-12-24  ROCKIN' AROUND THE CHRISTMAS TREE                   Brenda Lee
83 1961-12-30                    WHITE CHRISTMAS                   Bing Crosby
90 1962-01-06                    WHITE CHRISTMAS                   Bing Crosby
```

14.2.8 Plot of the number of songs in the TOP 100 each year

```
songs.groupby(songs["weekid"].dt.year).size()\
    .reset_index()\
    .rename(columns={0: "n_titles"})\
    .plot(x="weekid", y="n_titles", kind="bar")
```

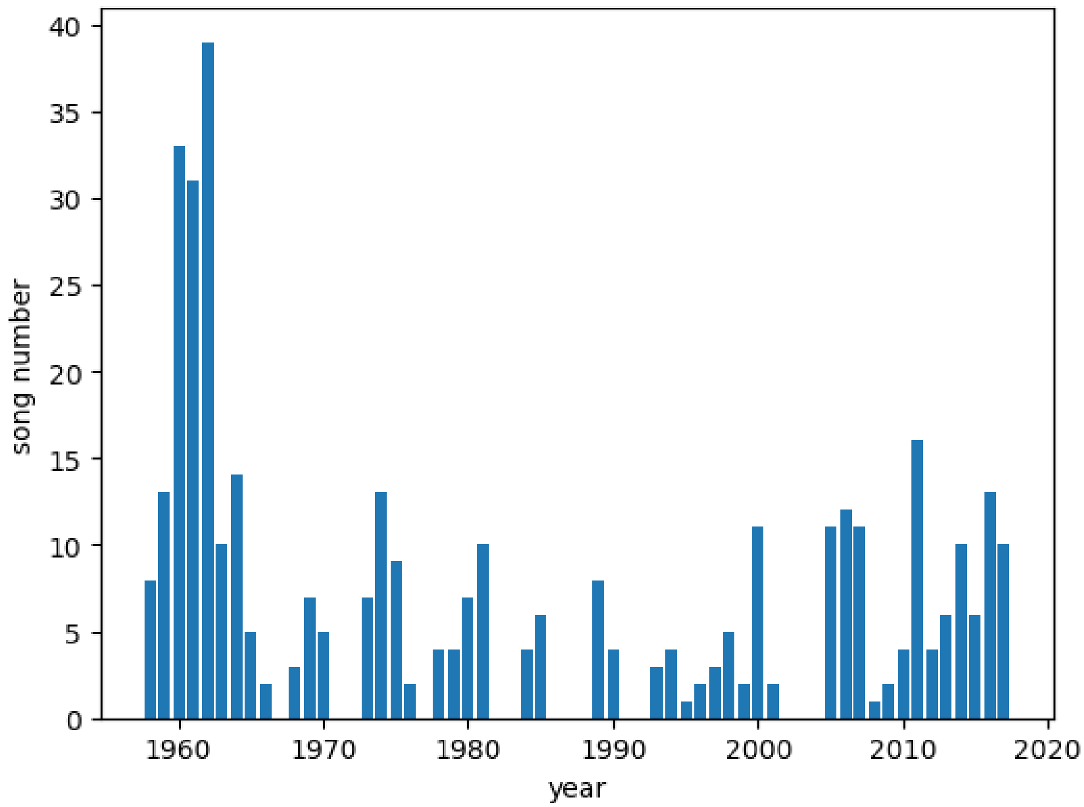
```
<Axes: xlabel='weekid'>
```



Using `matplotlib` directly results in a better rendering of the dates (see here for an explaination). More importantly: it also displays dates for which there is no entries (the previous date is misleading as it does not clearly show that the `weekid` are not continuous).

```
import matplotlib.dates as mdates
import matplotlib.pyplot as plt

plot_df = songs.groupby(songs["weekid"].dt.year).size()\
    .reset_index()\
    .rename(columns={0: "n_titles"})

fig, ax = plt.subplots()
ax.bar(plot_df["weekid"], plot_df["n_titles"])
ax.set_xlabel("year")
ax.set_ylabel("song number")
```

```
Text(0, 0.5, 'song number')
```



14.2.9 Songs that have been in the TOP 100 more than `N` weeks apart

```
df = songs.groupby("song")["weekid"].agg(["min", "max"]).reset_index()
df = df[df["max"] - df["min"] > pd.Timedelta(5 * 365, unit="days")]
df
```

```
                                          song        min        max
2                ALL I WANT FOR CHRISTMAS IS YOU 2000-01-08 2017-01-14
16                  DO THEY KNOW IT'S CHRISTMAS? 1984-12-22 2011-12-31
30                              JINGLE BELL ROCK 1958-12-20 2017-01-07
31                                LAST CHRISTMAS 2009-12-19 2017-01-14
40                PLEASE COME HOME FOR CHRISTMAS 1961-12-23 1979-01-27
43                 ROCKIN' AROUND THE CHRISTMAS TREE 1960-12-10 2017-01-07
53   THE CHRISTMAS SONG (MERRY CHRISTMAS TO YOU) 1960-12-10 2017-01-07
```

Note that we have to use a `Timedelta` object to compare the dates (so that `python` can ensure that the two dates difference are expressed in the same unit).

We can also display the performer of these songs:

```
songs[songs["song"].isin(df["song"])].groupby("song")["performer"].unique()
```

```
song
ALL I WANT FOR CHRISTMAS IS YOU                           [Mariah Carey, Michael␣
↪Buble]
DO THEY KNOW IT'S CHRISTMAS?                                      [Band-Aid,␣
```

(continues on next page)

(continued from previous page)

```
↪Glee Cast]
JINGLE BELL ROCK                                [Bobby Helms, Bobby Rydell/Chubby␣
↪Checker]
LAST CHRISTMAS                                        [Glee Cast, Ariana Grande,␣
↪Wham!]
PLEASE COME HOME FOR CHRISTMAS                              [Charles Brown,␣
↪Eagles]
ROCKIN' AROUND THE CHRISTMAS TREE                                  ␣
↪[Brenda Lee]
THE CHRISTMAS SONG (MERRY CHRISTMAS TO YOU)                      [Nat␣
↪King Cole]
Name: performer, dtype: object
```

14.2.10 Maximum number of consecutive years a song has been "best Christmas song"

```
df = pd.merge(songs,
              min_pos,
              right_on="year",
              left_on=songs["weekid"].dt.year,
              validate="m:1")

df = df[df["week_position"] == df["best_position_this_year"]]\
    [["year", "song", "performer"]]
```

```
df["song_id"] = df["song"] + "_" + df["performer"]
df["prev_best_id"] = df["song_id"].shift(1)
df["cons"] = (df["prev_best_id"] != df["song_id"]).cumsum()
df[["song_id", "cons"]].groupby("cons").size().idxmax()
```

```
np.int64(28)
```

```
df[df["cons"] == 28]
```

```
     year                              song     performer  \
342  2013  ALL I WANT FOR CHRISTMAS IS YOU  Mariah Carey
348  2014  ALL I WANT FOR CHRISTMAS IS YOU  Mariah Carey
362  2015  ALL I WANT FOR CHRISTMAS IS YOU  Mariah Carey
366  2016  ALL I WANT FOR CHRISTMAS IS YOU  Mariah Carey
382  2017  ALL I WANT FOR CHRISTMAS IS YOU  Mariah Carey

                                          song_id  \
342  ALL I WANT FOR CHRISTMAS IS YOU_Mariah Carey
348  ALL I WANT FOR CHRISTMAS IS YOU_Mariah Carey
362  ALL I WANT FOR CHRISTMAS IS YOU_Mariah Carey
366  ALL I WANT FOR CHRISTMAS IS YOU_Mariah Carey
382  ALL I WANT FOR CHRISTMAS IS YOU_Mariah Carey

                                     prev_best_id  cons
342                       MISTLETOE_Justin Bieber    28
348  ALL I WANT FOR CHRISTMAS IS YOU_Mariah Carey    28
362  ALL I WANT FOR CHRISTMAS IS YOU_Mariah Carey    28
```

(continues on next page)

(continued from previous page)

```
366  ALL I WANT FOR CHRISTMAS IS YOU_Mariah Carey    28
382  ALL I WANT FOR CHRISTMAS IS YOU_Mariah Carey    28
```

CHAPTER

FIFTEEN

EXERCICE : VARIATIONS IN TEMPERATURES

```
import pandas as pd

from pathlib import Path
```

```
if not Path("temperatures.csv").is_file():
    !curl -L https://bit.ly/3VklZVG -o temperatures.csv
```

15.1 Questions

1. Compute the mean temperature in °C.
2. Find out the day with the highest average daily temperature.
3. Plot the temperature with respect to time and, for each month the highest and lowest temperature.
4. Calculate the average temperature, distinguishing between :

- hours worked (from 7 a.m. to 7 p.m.) and hours not worked;
- weekdays and weekends;

(so there are 4 values to calculate)

5. Plot of temperature distributions distinguishing these 4 conditions.
6. Detect days when the average daily temperature is more than 10% of the average weekly temperature

15.2 Solutions

15.2.1 Mean temperature

We will start by reading the csv file, taking care to store the dates in the correct format:

```
df = pd.read_csv("temperatures.csv", parse_dates=True, index_col=0)
```

The resulting dataframe can be manipulated like any other dateframe: we can create a column to store the temperature in °C and calculate the average for this column.

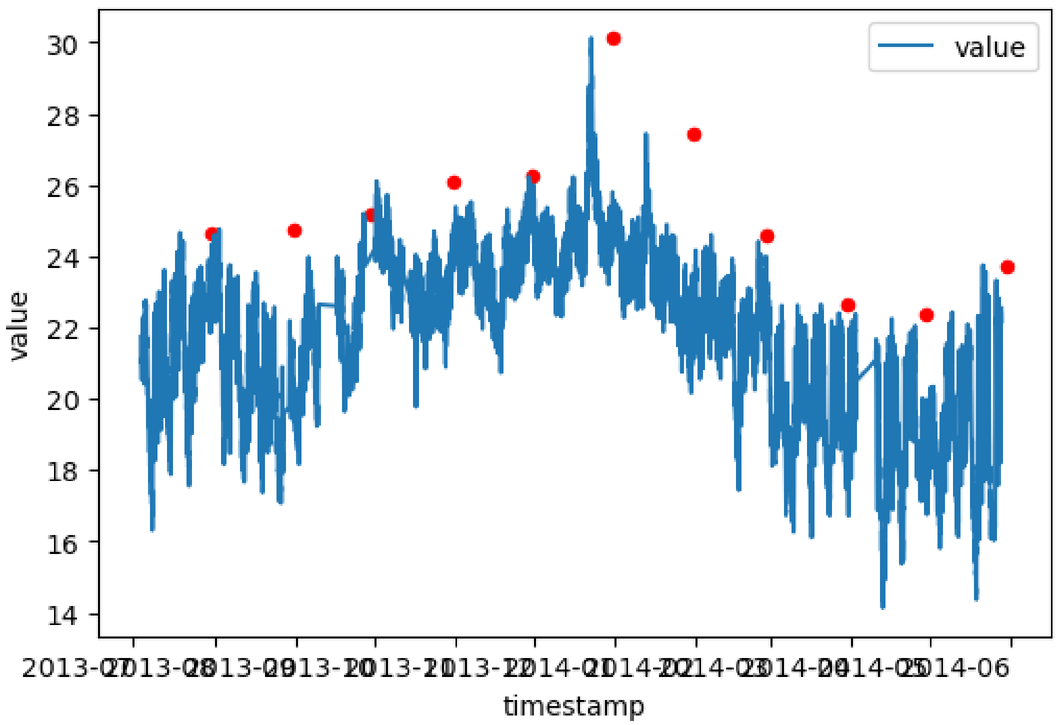
```
df["value"] = (df["value"] - 32) * 5  / 9   # use °C
df["value"].mean()
```

```
np.float64(21.801351504604533)
```

15.2.2 Plotting temperature variations with min/max values

```
ax = df.plot()
df.resample("ME")\
  .max()\
  .reset_index()\
  .plot(kind="scatter", x="timestamp", y="value", color="red", ax=ax)
```

```
<Axes: xlabel='timestamp', ylabel='value'>
```



All `plot` methods return an `Axis` object (defined in the `matplotlib` library) corresponding (as a first approximation) to the area on which you can draw. By passing this object in another call to the `plot` function (using the `ax` parameter), it is possible to plot two graphs on the same axis (i.e. two "overlapping" graphs).

The `resample` method does not keep track of the date on which the maximum temperature was reached: the maximum value is systematically associated with the last day of the resampling period (here: the last day of each month). You need to explicitly ask `pandas` to keep track of the date corresponding to the maximum value:

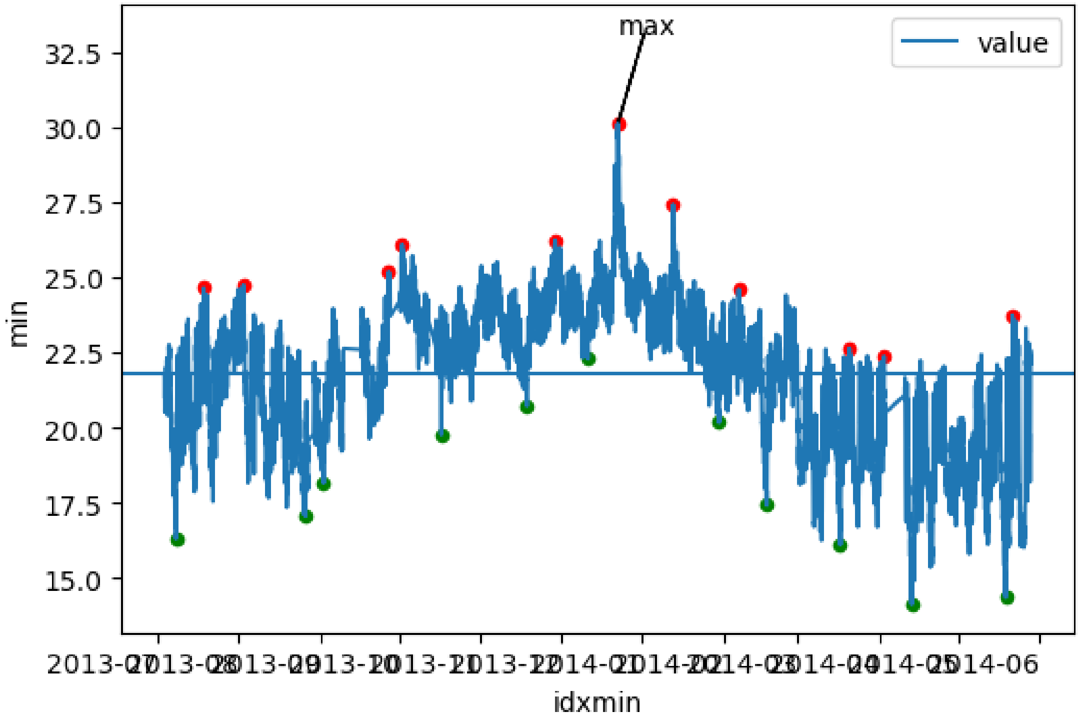
```
ax = df.plot()
df.resample("1ME")["value"]\
  .agg(["max", "idxmax"])\
  .plot(kind="scatter", x='idxmax', y='max', color="red", ax=ax)
df.resample("1ME")["value"]\
  .agg(["min", "idxmin"])\
  .plot(kind="scatter", x='idxmin', y='min', color="green", ax=ax)

ax.arrow(df["value"].idxmax(), df["value"].max(), 10, 3)
```

(continues on next page)

(continued from previous page)

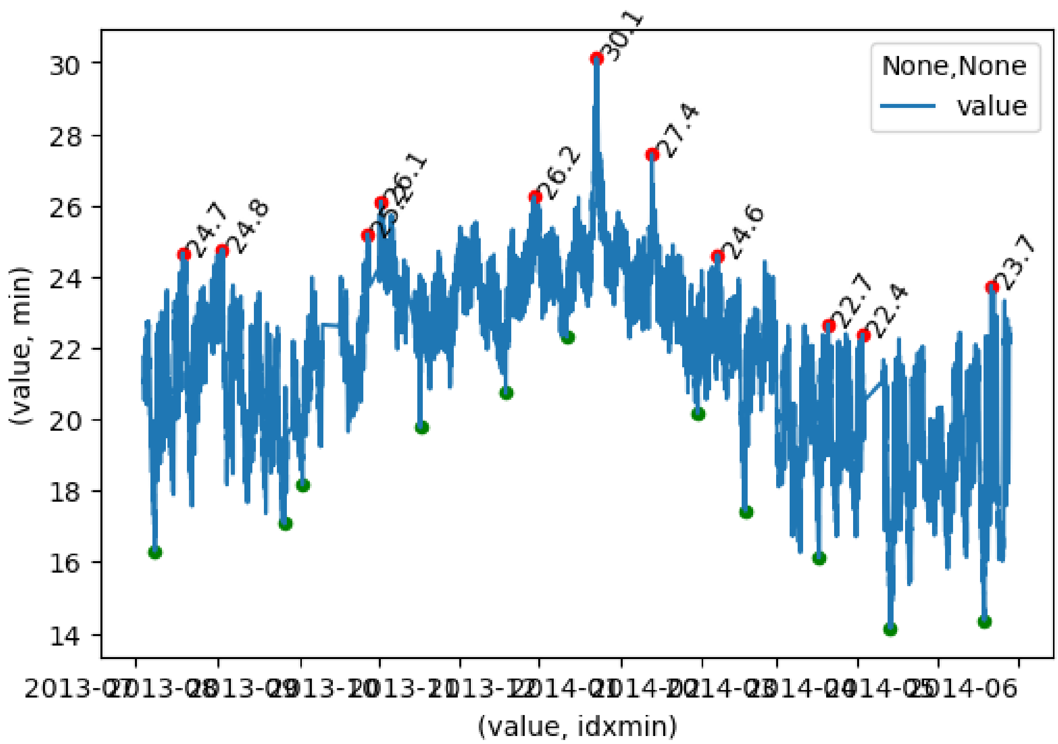
```
ax.annotate("max", (df["value"].idxmax(), df["value"].max() + 3))
ax.axhline(y=df["value"].mean())
```

```
<matplotlib.lines.Line2D at 0x13f959950>
```



As illustrated in the previous cell, there are plenty of methods attached to the axes for displaying various elements (text, arrows, lines, etc.) to generate the desired graph. For instance, it's easy to add the temperature value reached next to each point:

```
ax = df.plot()
df.resample("1ME")\
  .agg(["max", "idxmax"])\
  .reset_index()\
  .plot(kind="scatter", x=('value', 'idxmax'), y=('value', 'max'), color="red", ax=ax)
df.resample("1ME")\
  .agg(["min", "idxmin"])\
  .reset_index()\
  .plot(kind="scatter", x=('value', 'idxmin'), y=('value', 'min'), color="green",␣
↪ax=ax)

import numpy as np
for row_id, row in df.resample("1ME").agg(["max", "idxmax"]).iterrows():
    max_value = row[("value", "max")]
    max_date = row[("value", "idxmax")]
    ax.text(max_date, max_value, f"{max_value:.1f}", rotation=np.degrees(45))
```



15.3 Mean temperature over work days/week-ends

```
df["Day"] = df.index.day_name()
df["Kind"] = df["Day"].isin(["Saturday", "Sunday"]).apply(lambda x: "weekend" if x␣
↪else "workday")
```

```
df["Kind"].value_counts()
```

```
Kind
workday    5243
weekend    2024
Name: count, dtype: int64
```

```
#df["Time"] =
df["Hour"] = df.index.hour
df["Time"] = df["Hour"].between(7, 22).apply(lambda x: "day" if x else "night")
```

```
res = df[["Kind", "Time", "value"]].groupby(["Kind", "Time"])["value"].agg(["mean",
↪"min", "max", "median", "std"])
res
```

```
                   mean        min        max     median       std
Kind    Time
weekend day    20.880284  14.143559  30.124007  20.974415  2.795973
        night  21.443838  14.358096  30.041505  21.665714  2.630054
```

(continues on next page)

(continued from previous page)

```
workday day     22.278417  15.297943  28.126271  22.497091  1.845065
        night   21.692415  14.367725  29.836661  22.043937  2.548452
```

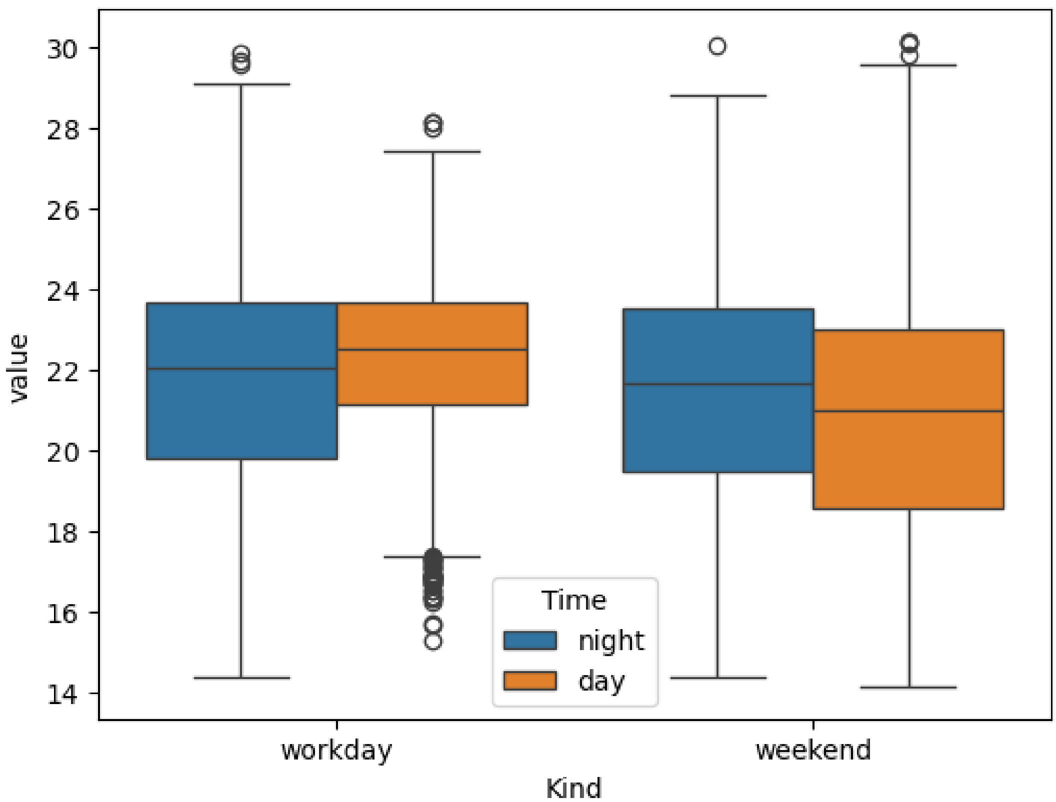
```
res["max"] - res["min"]
```

```
Kind     Time
weekend  day      15.980448
         night    15.683409
workday  day      12.828328
         night    15.468936
dtype: float64
```

```
import seaborn as sns
```

```
sns.boxplot(data=df, y="value", x="Kind", hue="Time")
```

```
<Axes: xlabel='Kind', ylabel='value'>
```



It is also possible to use the `pivot` method:

```
df["is_weekend"] = df.index.day_name().isin(["Saturday", "Sunday"])
df["is_day"] = df.index.hour.isin(range(8, 21))
```

```
df.pivot_table(index="is_weekend", columns="is_day", values="value",
               aggfunc="mean")
```

```
is_day          False      True
is_weekend
False        21.824531  22.30205
True         21.343148  20.83617
```

15.3.1 Detecting outliers

Find all dates for which the daily mean (i.e. temperatures averaged over one day) is, at least, 10% above the weekly mean (i.e. temperatures averaged over a week).

```
weekly = df.resample("1W")["value"].mean()\
                            .to_frame()\
                            .rename(columns={"value": "weekly_mean"})
weekly.head(5)
```

```
            weekly_mean
timestamp
2013-07-07    20.451477
2013-07-14    20.445381
2013-07-21    21.577086
2013-07-28    21.634587
2013-08-04    22.517238
```

```
daily = df.resample("1D")["value"].mean()\
                            .to_frame()\
                            .rename(columns={"value": "daily_mean"})
daily.head(3)
```

```
            daily_mean
timestamp
2013-07-04   21.372692
2013-07-05   21.862560
2013-07-06   20.400209
```

```
pd.concat([weekly.reindex(daily.index).bfill(),
                 daily], axis=1).query("daily_mean > 1.1 * weekly_mean")
```

```
            weekly_mean  daily_mean
timestamp
2013-12-22    25.336017   28.373459
2014-04-10    17.789209   20.889947
```

CHAPTER

SIXTEEN

ANALYZING INTERNET TRAFFIC IN MILAN

The Milan, Italy mobile phone activity dataset is a part of the Telecom Italia Big Data Challenge 2014, which is a rich and open multi-source aggregation of telecommunications, weather, news, social networks and electricity data from the city of Milan and the Province of Trentino (Italy).

A complete version of the dataset along with the original paper presenting the data can be found here

```
import pandas as pd
import numpy as np

from pathlib import Path
```

```
# download and extract dataset if this has not been done yet
if not Path("sms-call-internet-mi-2013-11-01.csv").is_file():
    !curl -L bit.ly/3EgFapf -o milan.tar.bz2
    !tar xvfj milan.tar.bz2
```

16.1 Data loading

First way of loading data: we use the fact that the file name is “regular” to automatically build the name of all the files we are interested in using a `for` loop.

The goal is to build a list of `DataFrame` that we can then merge with the `concat` command.

```
all_df = []
for n in range(1, 8):
    df = pd.read_csv(f"sms-call-internet-mi-2013-11-0{n}.csv")
    all_df.append(df)
```

This code can be written more compactly using a list comprehension:

```
all_df = [pd.read_csv(f"sms-call-internet-mi-2013-11-0{n}.csv") for n in range(1, 8)]
```

It is also possible to use Python standard library to list all the files in a directory matching a given criterion:

```
from pathlib import Path

data_dir = Path(".")
all_df = []
for filename in data_dir.glob("sms*.csv"):
    df = pd.read_csv(filename)
    all_df.append(df)
```

Merge all data into a final `DataFrame`

```
all_df = pd.concat(all_df, axis=0)
all_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 15089165 entries, 0 to 1828062
Data columns (total 8 columns):
 #   Column       Dtype
---  ------       -----
 0   datetime     object
 1   CellID       int64
 2   countrycode  int64
 3   smsin        float64
 4   smsout       float64
 5   callin       float64
 6   callout      float64
 7   internet     float64
dtypes: float64(5), int64(2), object(1)
memory usage: 1.0+ GB
```

As shown by the `info` method, dates and times of the events are stored as simple strings. We have to explicitely convert them into `datetime` object in order to be able to use the *high-level* methods provided by `pandas`.

```
all_df["datetime"] = pd.to_datetime(all_df["datetime"])
```

16.2 Representation of the *internet* traffic with respect to time

In the following we will consider three representative neighborhod:

- Duomo (historical center): cell n°5060
- Bocconi (university): cell n°4259
- Navigli (night life): cell n°4456

We aim at representing how the *internet* traffic in each of these neighborhod changes over time.

We fill first extract the three cells we are interested in, and rename them to make their manipulation clearer.

```
def set_name(what):
    mapping = {5060: "Duomo",
               4259: "Bocconi",
               4456: "Navigli"}
    return mapping[what]

selected_df = all_df[all_df["CellID"].isin([5060, 4259, 4456])].copy()
selected_df["CellID"] = selected_df["CellID"].apply(set_name)
```

Note the use of the `copy` method to be sure we are not using a *view* of the `DataFrame` so that we can modify the `DataFrame`.

We now have to aggregate the different events that occur at the same time. As we are interested in analyzing the traffic in each cell, we also have to partition data according to their `CellID` value.

```
to_plot = selected_df.groupby(["datetime", "CellID"])["internet"]\
                                .sum()\
                                .to_frame("internet")\
                                .reset_index()
to_plot.head()
```

```
             datetime   CellID   internet
0 2013-11-01 00:00:00  Bocconi  1369.8585
1 2013-11-01 00:00:00    Duomo  2583.3265
2 2013-11-01 00:00:00  Navigli  5595.5020
3 2013-11-01 01:00:00  Bocconi  1269.1230
4 2013-11-01 01:00:00    Duomo  4393.2422
```

As usual, we have to explictely convert the result of the `groupby` method into a `DataFrame` by calling the `to_frame` method and to "reset" the index to be ensure that the information used to partition the data is stored in columns rather than in the index (accessing indexes is not as convinient as accessing columns).

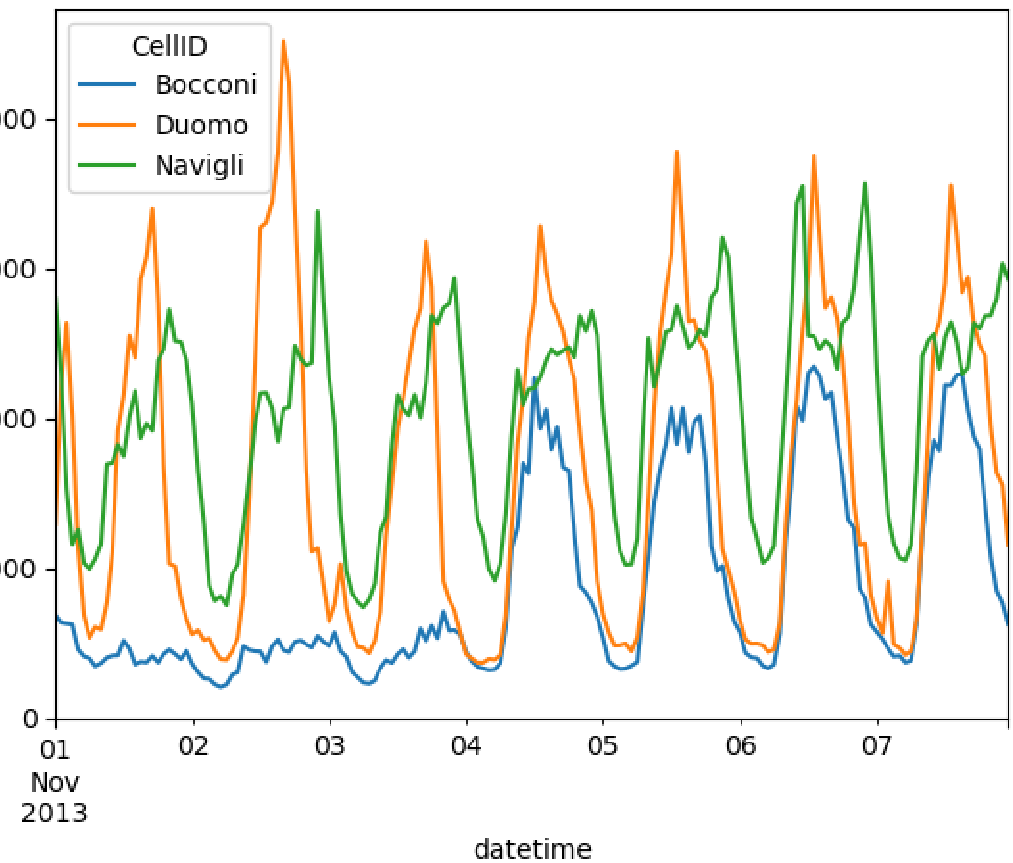
We now transform the `DataFrame` to ensure that the values related to each cell are stored in different columns, as this will make plotting easier (the plot function can take values from a *list* of columns displaying values from different columns in different colors).

```
to_plot = to_plot.pivot(index="datetime",
                        columns="CellID",
                        values="internet")\
                 .reset_index()
to_plot.head()
```

```
CellID            datetime    Bocconi      Duomo    Navigli
0      2013-11-01 00:00:00  1369.8585  2583.3265  5595.5020
1      2013-11-01 01:00:00  1269.1230  4393.2422  4596.8881
2      2013-11-01 02:00:00  1257.7803  5271.0795  3067.0964
3      2013-11-01 03:00:00  1244.2495  4038.8490  2311.3179
4      2013-11-01 04:00:00   908.3717  2241.5051  2508.5471
```

```
# select all columns but the first one
to_plot.plot(x="datetime", y=to_plot.columns[1:])
```

```
<Axes: xlabel='datetime'>
```



16.3 Distribution of traffic kind in each cell

In each cell, find out which the distribution of traffic for the different type of traffic (SMS, voice or internet)

```
# we do not want to distinguish in and out traffic
all_df["sms"] = all_df["smsin"] + all_df["smsout"]
all_df["call"] = all_df["callin"] + all_df["callout"]

# select only the columns we are interested in
sel_df = all_df[["CellID", "sms", "call", "internet"]]
sel_df.head()
```

```
   CellID     sms    call  internet
0       1     NaN     NaN       NaN
1       1  3.1237  0.4643   55.8591
2       2     NaN     NaN       NaN
3       2  3.1504  0.4681   56.0243
4       3     NaN     NaN       NaN
```

Surprisingly enougth there are cells in with no data (only `nan`)!

Indeed, when summing any number with a `nan` the result is `nan` (to comply with the IEEE 754 standard):

```
all_df.head()
```

```
     datetime  CellID  countrycode   smsin  smsout  callin  callout  internet  \
0 2013-11-07       1            0  0.2463     NaN     NaN      NaN       NaN
1 2013-11-07       1           39  1.4627  1.6610  0.2999   0.1644   55.8591
2 2013-11-07       2            0  0.2467     NaN     NaN      NaN       NaN
3 2013-11-07       2           39  1.4768  1.6736  0.3033   0.1648   56.0243
4 2013-11-07       3            0  0.2471     NaN     NaN      NaN       NaN

      sms    call
0     NaN     NaN
1  3.1237  0.4643
2     NaN     NaN
3  3.1504  0.4681
4     NaN     NaN
```

The values in `sel_df` are therefore incorrect (see, for example, the first line). We have to ignore `nan` values in the summation, which can be done by:

- using the `fillna` method to replace all `nan` values by 0;
- using the `sum` rather than the operator + as the former (as all pandas methods) will ignore nan (see the `skipna` parameter).

We will use, here, the second method that consists in:

- selecting the columns we want to sum (`["smsin", "smsout]`);
- summing them using row by row (i.e. along the axis n°1).

```
all_df["sms"] = all_df[["smsin", "smsout"]].sum(axis=1)
all_df["call"] = all_df[["callin", "callout"]].sum(axis=1)

sel_df = all_df[["CellID", "sms", "call", "internet"]]
sel_df.head()
```

```
   CellID     sms    call  internet
0       1  0.2463  0.0000       NaN
1       1  3.1237  0.4643   55.8591
2       2  0.2467  0.0000       NaN
3       2  3.1504  0.4681   56.0243
4       3  0.2471  0.0000       NaN
```

Find the total traffic in each cell

```
# in each cell, sum the traffic of each kind
# (i.e.: for each cell compute how much SMS/internet/voice traffic) there was
#
# all_traffic : on row for each cell x one column for each kind of traffic
all_traffic = sel_df.groupby("CellID").sum()
# in each cell, sum the different kinds of traffic and store the total in a new column
all_traffic["total"] = all_traffic.sum(axis=1)
```

Compute the % for each kind of traffic in each cell

This can be expressed as a simple operation between two columns: total amount for a given kind of traffic / total amount of traffic. We have to create a column to store the sum, because we are going to modify the columns over which the sum is computed.

```
all_traffic["sms"] = all_traffic["sms"] / all_traffic["total"] * 100
all_traffic["internet"] = all_traffic["internet"] / all_traffic["total"] * 100
```

(continues on next page)

(continued from previous page)

```
all_traffic["call"] = all_traffic["call"] / all_traffic["total"] * 100
all_traffic.round(1)
```

```
        sms  call  internet    total
CellID
1       8.1   6.9      85.0  11956.9
2       8.2   6.9      84.8  12029.2
3       8.3   7.0      84.7  12106.2
4       7.9   6.7      85.4  11747.4
5       8.1   6.8      85.1  10759.4
...     ...   ...       ...      ...
9996    7.3   6.9      85.8  43790.4
9997    7.2   6.9      85.9  47900.9
9998    7.2   6.9      86.0  47097.9
9999    7.9   7.6      84.5  30173.7
10000   8.9   8.7      82.4  24856.5

[10000 rows x 4 columns]
```

16.4 Finding cells with the most "unbalanced" distribution

For each kind of communication, find the 5 cells with the largest and smallest percentage of this communication

```
# change head with tail for the 5 smallest
all_traffic.reset_index()\
              .melt(var_name="kind",
                    value_name="%",
                    id_vars="CellID")\
              .sort_values(by="%", ascending=False)\
              .groupby("kind").head(5)\
              .sort_values(by="kind")
```

```
       CellID      kind             %
14573    4574      call  4.965841e+01
15237    5238      call  7.676842e+01
15337    5338      call  8.096834e+01
15238    5239      call  9.768525e+01
15338    5339      call  9.418676e+01
27214    7215  internet  9.332384e+01
27314    7315  internet  9.332384e+01
20146     147  internet  9.333797e+01
27114    7115  internet  9.332918e+01
20047      48  internet  9.335489e+01
3059     3060       sms  6.093316e+01
3160     3161       sms  6.095847e+01
3159     3160       sms  6.095847e+01
3259     3260       sms  6.095847e+01
3260     3261       sms  6.095847e+01
36063    6064     total  1.280324e+06
35060    5061     total  1.381252e+06
35258    5259     total  1.429056e+06
35058    5059     total  1.653950e+06
35160    5161     total  1.789843e+06
```

16.5 Anomaly detection: find when the daily mean internet traffic is over the weakly mean

- compute mean internet traffic every day for each cell (d)
- compute mean internet traffic over the week for each cell (w)
- for each cell, find when $d \geq \alpha \times w$

```
sel_df = all_df[["datetime", "CellID", "internet"]]
sel_df.head()
```

```
    datetime  CellID  internet
0 2013-11-07       1       NaN
1 2013-11-07       1   55.8591
2 2013-11-07       2       NaN
3 2013-11-07       2   56.0243
4 2013-11-07       3       NaN
```

```
# mean traffic over the week for every cell
weekly_mean = sel_df.groupby("CellID").mean()
weekly_mean.reset_index(inplace=True)
weekly_mean = weekly_mean.rename(columns={"internet": "weekly_mean"})
```

```
# mean traffic over a day for every cell
daily_mean = sel_df.groupby(["CellID", sel_df["datetime"].dt.day])["internet"].mean()
daily_mean = daily_mean.to_frame(name="daily_mean").reset_index()
```

```
result = pd.merge(daily_mean,
                  weekly_mean,
                  on="CellID",
                  how="outer")
```

```
alpha = 2
result[result['daily_mean'] > alpha * result['weekly_mean']]
```

```
       CellID  datetime_x  daily_mean                    datetime_y  \
5691      814           1   34.549095 2013-11-04 17:33:33.720316672
13620    1946           6  391.941014 2013-11-04 20:14:19.961315072
30646    4379           1  294.552084 2013-11-04 19:34:41.988743168
30653    4380           1  230.101890 2013-11-04 19:39:34.974567680
31353    4480           1  415.909862 2013-11-04 22:38:29.687034368
...       ...         ...         ...                           ...
67879    9698           1  218.391867 2013-11-04 22:32:01.311475456
67886    9699           1  255.167829 2013-11-04 22:52:50.322580736
67893    9700           1  165.456413 2013-11-04 22:49:37.238239744
68579    9798           1  111.903298 2013-11-04 22:25:01.214574848
68586    9799           1  144.582802 2013-11-04 22:34:20.301507584

       weekly_mean
5691     16.874087
13620   191.046583
30646   108.679119
30653    96.646144
31353   121.710280
```

(continues on next page)

(continued from previous page)

```
...              ...
67879    86.511171
67886   100.714089
67893    75.305560
68579    51.824152
68586    62.477990

[99 rows x 5 columns]
```

Highlight the lines for which the `daily_mean` is larger than the `weekly_mean`

```
def f(row):
    if row["daily_mean"] > row["weekly_mean"]:
        color = "color: white; background-color:red"
    else:
        color = "color: black"

    return [color] * len(row)

result.head(10).style.apply(f, axis=1)
```

```
<pandas.io.formats.style.Styler at 0x172796910>
```

Find the entry (cell & date), for each the observed value is the most different from the mean value.

The difference is estimated as: $|y_{\text{observed}} - \bar{y}|$

```
result["diff"] = np.abs(result["daily_mean"] - result["weekly_mean"])
```

```
# value of the larges difference
result["diff"].max()
# index of the largest difference (argmax)
result["diff"].idxmax()
# corresponding entry:
result.loc[result["diff"].idxmax()]
```

```
CellID                                7052
datetime_x                               2
daily_mean                      193.721918
datetime_y     2013-11-04 16:23:01.034482944
weekly_mean                    1197.137083
diff                           1003.415166
Name: 49358, dtype: object
```

16.6 Modeling Traffic Variations

Goal: fit a function (sinusoidal) to the internet traffic observed in cell n°5060 (Duomo)

```
df = all_df[all_df["CellID"] == 5060].groupby("datetime", as_index=False)["internet"].
↪sum()
```

We have to convert the data to the suitable format:

- x has to be continuous: for the moment, it is made of a day and an hour –> convert it to a number of hours

- all data have to be stored in a numpy array –> de facto standard for scientific computation in python

```
df["x"] = df["datetime"].dt.hour + (df["datetime"].dt.day - 1) * 24

y = df["internet"].to_numpy()
x = df["x"].to_numpy()
```

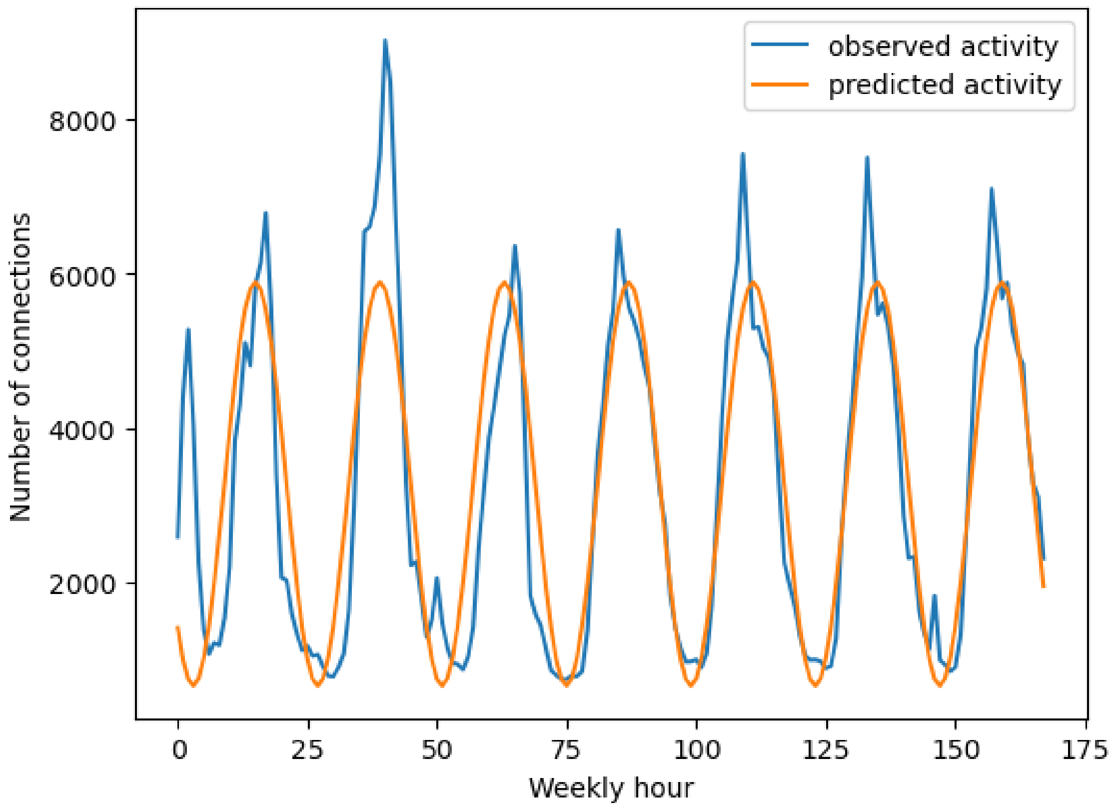
The `curve_fit` function of the `scipy.optimize` library allows us to estimate the parameter of a function (non-linear last square)

```
from scipy.optimize import curve_fit

def func(xdata, a, b, c):
    # 1st argument: observations (a numpy array)
    # other arguments: function parameters (to be estimated from data)
    return a * np.sin(2 * np.pi * (1 / 24) * xdata + b) + c

popt, pcov = curve_fit(func, x, y)
```

Plot the predicted function and the observed values

```
import matplotlib.pyplot as plt

# value predicted by the model
yfit = func(x, *popt)

plt.plot(x, y, label="observed activity")
plt.plot(x, yfit, label="predicted activity")

plt.xlabel("Weekly hour")
plt.ylabel("Number of connections")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1727a4310>
```


observed activity
predicted activity
8000
6000
4000
2000
Number of connections
0
25
50
75
100
125
150
175
Weekly hour

CHAPTER

SEVENTEEN

ANALYZING NEURIPS PAPERS

In this chapter, we'll be looking at a corpus containing all the publications from the Neurips conference (the leading statistical learning conference). In particular, our study will show how pandas can be interfaced with "external" libraries for both textual content analysis and "dynamic" graph representation.

```
from pathlib import Path

import pandas as pd
import seaborn as sns
```

```
if not Path("neurips_papers.zip").is_file():
    !curl -L https://bit.ly/3Y7ZCCS -o neurips_papers.zip
    !unzip neurips_papers.zip
```

17.1 Data Loading

```
authors = pd.read_csv("authors.csv")
mapping = pd.read_csv("paper_authors.csv")
papers = pd.read_csv("papers.csv")
```

The `mapping` data frame can be used to identify the authors of an article. This data frame contains only one association between article identifiers and author identifiers. Associations between identifiers (numbers) and full names (article title or author name) are defined in the other two data frames. To make it easier to interpret the results, and to enable certain analysis (e.g. grouping authors by year), we'll start by merging the different data frames so as to have "explicit" information.

```
tmp = pd.merge(mapping, authors, right_on="id", left_on="author_id")
df = pd.merge(tmp, papers, left_on="paper_id", right_on="id")[["name", "title", "year
↪"]]
```

To check that the merge is as expected, you can look at the size of the dataframes:

```
print(f"before merging: {mapping.shape[0]}")
print(f"after merging: {df.shape[0]}")
```

```
before merging: 20838
after merging: 20838
```

Since both data frames have the same size, we know that `pandas` has found the corresponding author names and titles in the other data frames for each line of the original data frame.

17.1.1 Questions

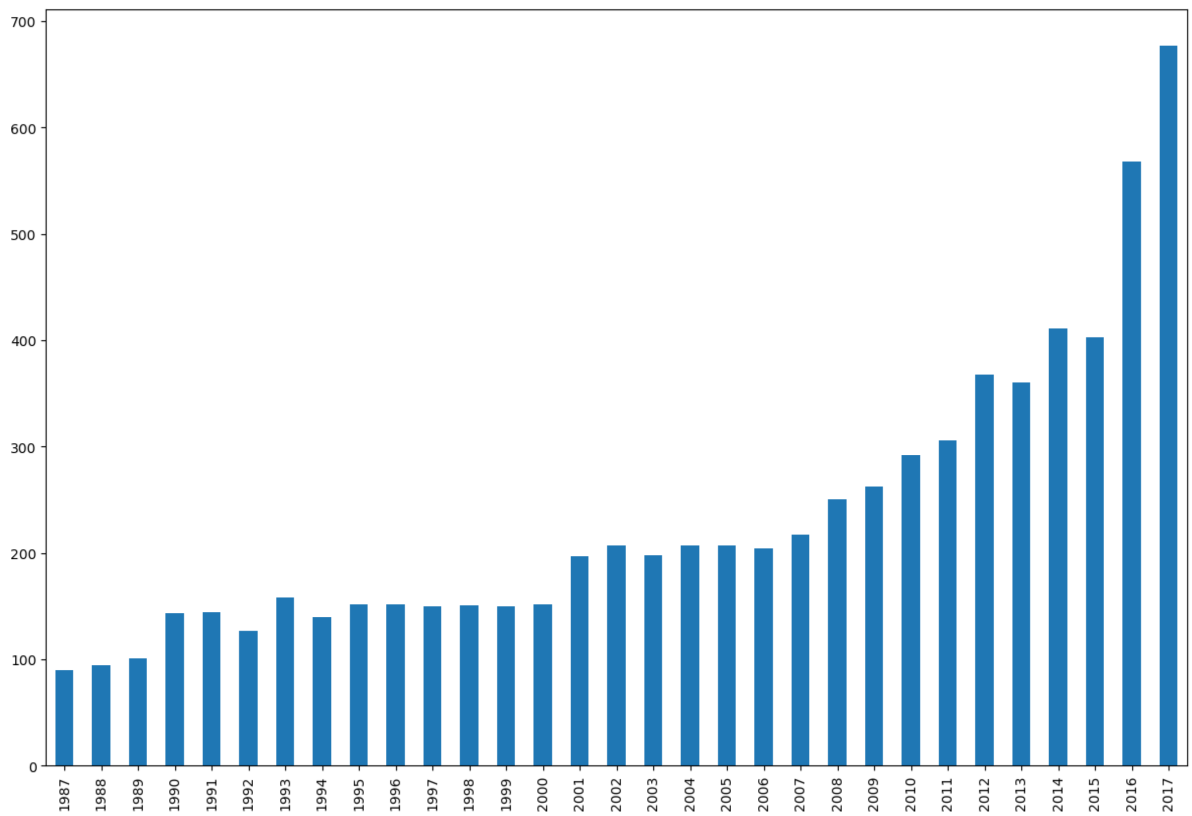
- What is the time period covered by the data?
- Plot the number of papers published each year?
- What is the average number of authors per paper per year?
- Which author has published the most papers?
- How many authors have published at least 5 papers?
- which two authors have published the most together?
- Who is the author with the largest number of different co-authors?

```
# Question 1
df["year"].agg(["min", "max"])
```

```
min    1987
max    2017
Name: year, dtype: int64
```

```
# Question 2
df.drop_duplicates("title")["year"]\
        .value_counts()\
        .sort_index()\
        .plot(kind="bar", figsize=(15,10))
```

```
<Axes: xlabel='year'>
```



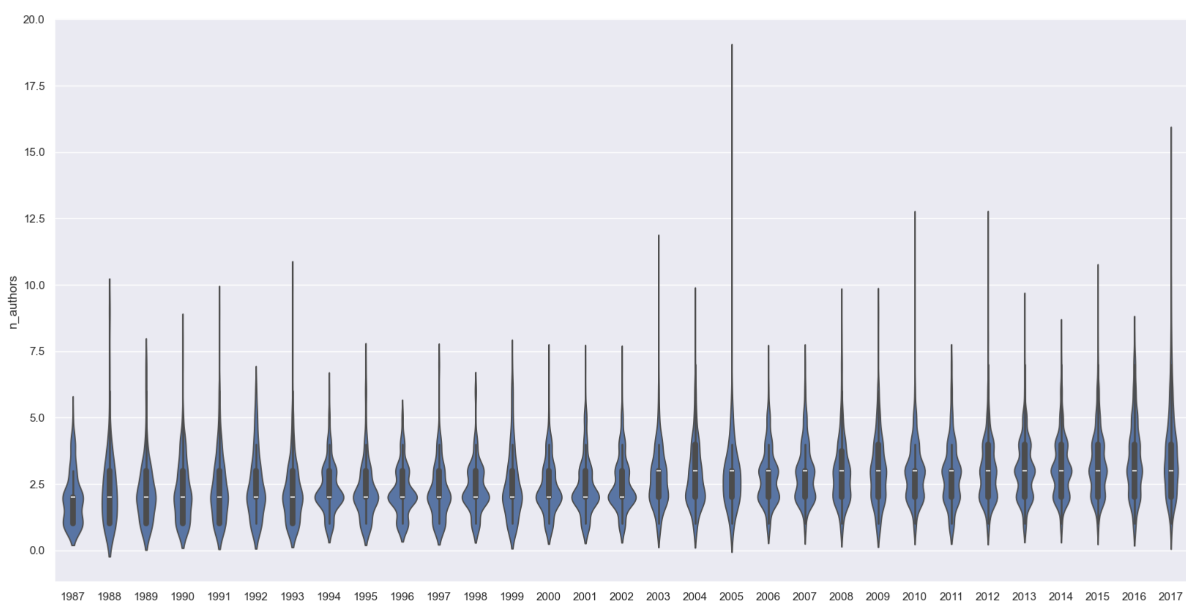
```
# Question 3
df.groupby(["title", "year"]).size()\
        .reset_index()\
        .rename(columns={0: "n_authors"})\
        .groupby("year")["n_authors"].mean()\
        .head()     # only to have a samller output
```

```
year
1987    1.966667
1988    2.372340
1989    2.257426
1990    2.272727
1991    2.361111
Name: n_authors, dtype: float64
```

To be able to conclude anything (for example, on the evolution of the number of authors per article over time), it is generally preferable to compare distributions rather than averages:

```
plot_df = df.groupby(["title", "year"]).size()\
        .reset_index()\
        .rename(columns={0: "n_authors"})

sns.set_theme(rc={'figure.figsize':(20,10)})
sns.violinplot(data=plot_df, x="year", y="n_authors")
```

```
<Axes: xlabel='year', ylabel='n_authors'>
```




```
# Question 4
df["name"].value_counts().head(1)
```

```
name
Michael I. Jordan    101
Name: count, dtype: int64
```

```
# Question 5
df["name"].value_counts().reset_index().query("count > 5")
```

```
                      name  count
0          Michael I. Jordan    101
1        Bernhard Sch?lkopf     62
2              Yoshua Bengio     60
3        Geoffrey E. Hinton     58
4         Zoubin Ghahramani     51
..                     ...    ...
612     Karsten M. Borgwardt      6
613       Andreas G. Andreou      6
614              Yihong Gong      6
615       Michael E. Tipping      6
616   Ralph Etienne-Cummings      6

[617 rows x 2 columns]
```

```
# Question 6
pairs = pd.merge(df, df, on=["title", "year"], how="inner")
# there is one author with no name → will make the deduplication in the next step fail
pairs = pairs.fillna("missing")

# we “normalize” the name (A, B) & (B, A) will both become (A, B)
pairs.query("name_x != name_y")[["name_x", "name_y"]]\
  .apply(lambda x: sorted(x.values), axis=1)\
  .value_counts() / 2
```

```
[Inderjit S. Dhillon, Pradeep K. Ravikumar]    18.0
[Alex J. Smola, Bernhard Sch?lkopf]            12.0
[Corinna Cortes, Mehryar Mohri]                11.0
[Han Liu, Zhaoran Wang]                        11.0
[Martin J. Wainwright, Michael I. Jordan]      11.0
                                               ...
[Magnus Rattray, Michalis K. Titsias]           1.0
[Mrinal Kalakrishnan, Stefan Schaal]            1.0
[Mrinal Kalakrishnan, Sethu Vijayakumar]        1.0
[Jo-anne Ting, Stefan Schaal]                   1.0
[Csaba Szepesvari, Dale Schuurmans]             1.0
Name: count, Length: 22145, dtype: float64
```

```
#7
pairs.groupby("name_x")["name_y"].nunique().sort_values(ascending=False).head(5)
```

```
name_x
Michael I. Jordan     136
Bernhard Sch?lkopf    117
Yoshua Bengio         103
Lawrence Carin         80
Andrew Y. Ng           77
Name: name_y, dtype: int64
```

17.2 Analyzing the content of the papers

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
title_vect = TfidfVectorizer(max_features=3_000, max_df=.8, min_df=3, stop_words=
↪"english")
title = title_vect.fit_transform(papers["title"].values)
```

```
paper_vect = TfidfVectorizer(max_features=3_000, max_df=.8, min_df=3, stop_words=
↪"english")
papers_content = paper_vect.fit_transform(papers["paper_text"])
```

```
from sklearn.decomposition import LatentDirichletAllocation
lda_model_title = LatentDirichletAllocation(n_components=10, learning_method='online',
↪ random_state=42)
lda_model_papers = LatentDirichletAllocation(n_components=10, learning_method='online
↪', random_state=42)
```

```
lda_top=lda_model_title.fit_transform(title)
```

```
lda_papers = lda_model_papers.fit_transform(papers_content)
```

```
df = pd.DataFrame(lda_model_title.components_, columns=[f"'{v}'" for v in title_vect.
↪get_feature_names_out()])

df.reset_index()\
  .rename(columns={"index": "topic"})\
  .melt(id_vars="topic", var_name="word", value_name="importance")\
  .groupby("topic").apply(lambda x: x.nlargest(10, "importance")\
                                                      .assign(rank=lambda xx: xx[
↪'importance'].rank(ascending=False)
                                                            ), include_
↪groups=False)\
  .reset_index()\
  .pivot(index="topic", values="word", columns="rank")
```

```
rank             1.0            2.0              3.0             4.0   \
topic
0          'feature'   'clustering'   'minimization'       'neurons'
1         'gaussian'    'inference'      'efficient'     'processes'
2         'learning'   'supervised'           'fast'         'graph'
3       'stochastic'     'gradient'   'optimization'        'neural'
4          'optimal'     'learning'         'random'       'dynamic'
5           'online'       'bounds'    'convergence'   'dimensional'
6             'deep'  'variational'  'probabilistic'        'models'
7         'networks'         'data'       'learning'        'neural'
8      'information'        'spike'         'hidden'         'brain'
9       'prediction'       'convex'       'analysis'  'unsupervised'

rank               5.0            6.0                 7.0               8.0   \
topic
0      'convolutional'        'scale'            'images'         'natural'
1            'process'   'regression'          'learning'          'models'
2             'method'      'kernels'              'semi'          'vector'
```

(continues on next page)

(continued from previous page)

```
3           'learning'     'decision'          'spectral'          'object'
4             'neural'      'control'       'recognition'   'reinforcement'
5            'bandits'         'high'        'submodular'           'large'
6          'graphical'   'structured'   'representations'        'learning'
7          'structure'       'latent'       'statistical'       'recurrent'
8          'filtering'       'markov'            'neural'  'regularization'
9           'temporal'  'distributed'               'low'      'processing'

rank              9.0           10.0
topic
0           'learning'       'metric'
1             'graphs'       'sparse'
2        'propagation'     'machines'
3           'networks'      'descent'
4             'speech'        'using'
5        'adversarial'       'margin'
6              'multi'         'task'
7               'time'   'generative'
8           'distance'       'domain'
9          'embedding'     'sequence'
```

17.2.1 Question

1. For each document, find the topic with the highest score
2. Count the number of documents assigned to each topic

```
mat = pairs.query("name_x != name_y")[["name_x", "name_y"]]\
      .value_counts()\
      .reset_index()\
      .query("count > 3")
```

```
from d3graph import d3graph, vec2adjmat
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[22], line 1
----> 1 from d3graph import d3graph, vec2adjmat

ModuleNotFoundError: No module named 'd3graph'
```

```
adjmat = vec2adjmat(list(mat["name_x"].values),
           list(mat["name_y"].values),
           list(mat["count"].values))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[23], line 1
----> 1 adjmat = vec2adjmat(list(mat["name_x"].values),
      2            list(mat["name_y"].values),
      3            list(mat["count"].values))

NameError: name 'vec2adjmat' is not defined
```

```
d3 = d3graph()
d3.graph(adjmat)
d3.show(filepath="./graph.html")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[24], line 1
----> 1 d3 = d3graph()
      2 d3.graph(adjmat)
      3 d3.show(filepath="./graph.html")

NameError: name 'd3graph' is not defined
```