Pandas: limitations and alternative ecosystems

Lino Galiana
Version of September 29, 2026

After exploring the `Pandas` ecosystem in depth, this short chapter takes a step back to look at some limitations of its syntax and presents the main alternative paradigms (`Polars`, `DuckDB`, `Spark`) that make it possible to go further, especially with large volumes of data.

⚠️ **Automatically generated PDF.** This document is produced from the course website and may contain formatting issues. For an up-to-date version adapted to your reading, visit the course page: pythonds.linogaliana.fr.

Table of contents

Skills to be acquired by the end of this chapter

- Know the main limitations of `Pandas` syntax;
- Know how to chain operations more readably using pipes;
- Know the main alternative packages to `Pandas` (`Polars`, `DuckDB`, `Spark`).

1 Introduction

The previous two chapters explored in depth the richness of the `Pandas` ecosystem, whether it was associating data through joins or building descriptive statistics and reshaping data. `Pandas` is indispensable in the data scientist's toolbox.

This short chapter takes a step back. Despite all its qualities, `Pandas` is not perfect: its syntax, inherited from more than fifteen years of history, has a few rough edges. We will also see that other, more recent, paradigms make it possible to go beyond `Pandas`, especially when it comes to handling larger volumes of data.

1.1 Data used in this chapter

To illustrate this chapter, we start again from the raw Ademe greenhouse gas emissions dataset, already used in the previous chapters:

```
import numpy as np
import pandas as pd

url   =   "https://data.ademe.fr/data-fair/api/v1/datasets/igt-pouvoir-de-rechauffement-global/
convert"
emissions = pd.read_csv(url)
emissions['dep'] = emissions["INSEE commune"].str[:2]
```

To obtain reproducible results, you can set the seed of the pseudo-random number generator.

```
np.random.seed(123)
```

2 Pandas in a chain of operations

Generally, in a project, data cleaning will consist of a series of methods applied to a `DataFrame` or a `Series` when working exclusively on a single column. In other words, what is usually expected when working with `Pandas` is to have a chain that takes a `DataFrame` as input and outputs the same `DataFrame` enriched or an aggregated version of it.

This way of proceeding is at the heart of the `dplyr` syntax in `R` but is not necessarily native in `Pandas` depending on the operations you want to implement. Indeed, the natural way to update a dataframe in `Pandas` often involves syntax like:

```
import numpy as np
import pandas as pd

data = [[8000, 1000], [9500, np.nan], [5000, 2000]]
df = pd.DataFrame(data, columns=['salaire', 'autre_info'])
df['salaire_net'] = df['salaire']*0.8
```

In `SQL` you could directly update your database with the new column:

```
SELECT *, salaire*0.8 AS salaire_net FROM df
```

The `tidyverse` ecosystem in `R`, the equivalent of `Pandas`, works according to the same logic as SQL table updates. Indeed, you would use the following command with `dplyr`:

```
df %>% mutate(salaire_net = salaire*0.8)
```

Technically, you could do this with an `assign` in `Pandas`:

```
df = df.drop("salaire_net", axis = "columns")
df = df.assign(salaire_net = lambda s: s['salaire']*0.8)
```

Line 1
To delete the variable to start from the initial example

However, this `assign` syntax is not very natural. It requires passing a lambda function that expects a `DataFrame` as input where you would want a column. So, it is not really a readable and practical syntax.

It is nevertheless possible to chain operations on datasets using pipes. These follow the same philosophy as `dplyr`, itself inspired by the Linux pipe. This approach will make the code more readable

by defining functions that perform operations on one or more columns of a DataFrame. The first argument to the function is the `DataFrame`, the others are those controlling its behavior:

```
def calcul_salaire_net(df: pd.DataFrame, col: str, taux: float = 0.8):
  df["salaire_net"] = df[col]*taux
  return df
```

This transforms our production chain into:

```
(
  df
  .pipe(calcul_salaire_net, "salaire")
)
```

	salaire	autre_info	salaire_net
0	8000	1000.0	6400.0
1	9500	NaN	7600.0
2	5000	2000.0	4000.0

2.1 Some limitations regarding `Pandas` syntax

There is a before and after `Pandas` in data analysis with `Python`. Without this incredibly practical package, Python, despite all its strengths, would have struggled to establish itself in the data analysis landscape. However, while `Pandas` offers a coherent syntax in many aspects, it is not perfect either. More recent data analysis paradigms in `Python` sometimes aim to correct these syntactical imperfections.

Among the most annoying points in everyday use is the need to regularly perform `reset_index` when building descriptive statistics. Indeed, it can be dangerous to keep indices that are not well controlled because, if we are not careful during the merge phases, they can be misused by `Pandas` to join data, leading to surprises.

`Pandas` is extremely well-designed for restructuring data from long to wide format or vice versa. However, this is not the only way to restructure a dataset that we might want to implement. It often happens that we want to compare the value of an observation to that of a group to which it belongs. This is particularly useful in anomaly analysis, outlier detection, or fraud investigation. Natively, in `Pandas`, you need to build an aggregate statistic by group and then merge it back to the initial data using the group variable. This is somewhat tedious:

```
emissions_moyennes = emissions.groupby("dep").agg({"Agriculture": "mean"}).reset_index()
emissions_enrichies = (
  emissions
  .merge(emissions_moyennes, on = "dep", suffixes = ['', '_moyenne_dep'])
)
emissions_enrichies['relatives'] =
emissions_enrichies["Agriculture"]/emissions_enrichies["Agriculture_moyenne_dep"]
emissions_enrichies.head()
```

	INSEE commune	Commune	Agriculture	Autres transports	Autres transports international	CO2 biomasse hors-total	Déchets	Energie	Industrie hors-énergie	Résidentiel	Routier	Tertiaire	dep	Agriculture_moyenne_dep	relatives
0	01001	L'ABERGEMENT-CLEMENCIAT	3711.425991	NaN	NaN	432.751835	101.430476	2.354558	6.911213	309.358195	793.156501	367.036172	01	1974.535382	1.879645
1	01002	L'ABERGEMENT-DE-VAREY	475.330205	NaN	NaN	140.741660	140.675439	2.354558	6.911213	104.866444	348.997893	112.934207	01	1974.535382	0.240730
2	01004	AMBERIEU-EN-BUGEY	499.043526	212.577908	NaN	10313.446515	5314.314445	998.332482	2930.354461	16616.822534	15642.420313	10732.376934	01	1974.535382	0.252740
3	01005	AMBERIEUX-EN-DOMBES	1859.160954	NaN	NaN	1144.429311	216.217508	94.182310	276.448534	663.683146	1756.341319	782.404357	01	1974.535382	0.941569
4	01006	AMBLEON	448.966808	NaN	NaN	77.033834	48.401549	NaN	NaN	43.714019	398.786800	51.681756	01	1974.535382	0.227378

In the `tidyverse`, this two-step operation could be done in a single step, which is more convenient:

```r
emissions %>%
  group_by(dep) %>%
  mutate(relatives = Agriculture/mean(Agriculture))
```

This isn't too bad, but it does make `Pandas` processing chains longer and therefore increases the maintenance burden to keep them running over time.

More generally, `Pandas` processing chains can be quite verbose because it is often necessary to redefine the `DataFrame` rather than just the columns. For example, to filter rows and columns, you have to:

```python
(
  emissions
  .loc[
    (emissions["dep"] == "12") & (emissions["Routier"]>500), ['INSEE commune', 'Commune']
  ]
  .head(5)
)
```

	INSEE commune	Commune
4058	12001	AGEN-D'AVEYRON
4059	12002	AGUESSAC
4062	12006	ALRANCE
4063	12007	AMBEYRAC
4064	12008	ANGLARS-SAINT-FELIX

In SQL, you could simply refer to the columns in the filter:

```sql
SELECT "INSEE commune", 'Commune'
FROM emissions
WHERE dep=="12" AND Routier>500
```

In the `tidyverse` (R), you could also do this simply:

```r
df %>%
  filter(dep=="12", Routier>500) %>%
  select(`INSEE commune`, `Commune`)
```

3 Other paradigms

Despite all the limitations we have just mentioned, and the alternative solutions we will present, `Pandas` remains the central package of the data ecosystem with `Python`. In the following chapters, we will see its native integration with the `Scikit` ecosystem for machine learning or the extension of `Pandas` to spatial data with `GeoPandas`.

Other technical solutions that we will discuss here may be relevant if you want to handle large volumes of data or if you want to use alternative syntaxes.

The main alternatives to `Pandas` are `Polars`, `DuckDB`, and `Spark`. There is also `Dask`, a library for parallelizing `Pandas` operations.

3.1 Polars

Polars is certainly the paradigm most inspired by Pandas, even in the choice of name. The first fundamental difference lies in the internal layers used. Polars relies on the Rust implementation of Arrow, whereas Pandas relies on Numpy, which results in performance loss. This allows Polars to be more efficient on large volumes of data, especially since many operations are parallelized and rely on lazy evaluation, a programming principle that optimizes queries for logical rather than defined execution order.

Another strength of Polars is its more coherent syntax, benefiting from over fifteen years of Pandas existence and almost a decade of dplyr (the data manipulation package within the R tidyverse paradigm). To take the previous example, there is no longer a need to force the reference to the DataFrame; in an execution chain, all subsequent references will be made with respect to the initial DataFrame.

```
import polars as pl
emissions_polars = pl.from_pandas(emissions)
(
  emissions_polars
  .filter(pl.col("dep") == "12", pl.col("Routier") > 500)
  .select('INSEE commune', 'Commune')
  .head(5)
)
```

INSEE commune	Commune
str	str
"12001"	"AGEN-D'AVEYRON"
"12002"	"AGUESSAC"
"12006"	"ALRANCE"
"12007"	"AMBEYRAC"
"12008"	"ANGLARS-SAINT-FELIX"

To learn about Polars, many online resources are available, including this notebook built for the public statistics data scientists network.

3.2 DuckDB

DuckDB is the newcomer in the data analysis ecosystem, pushing the limits of data processing with Python without resorting to big data tools like Spark. DuckDB epitomizes a new paradigm, the "Big data is dead" paradigm, where large data volumes can be processed without imposing infrastructures.

Besides its great efficiency, as DuckDB can handle data volumes larger than the computer or server's RAM, it offers the advantage of a uniform syntax across languages that call DuckDB (Python, R, C++, or Javascript). DuckDB favors SQL syntax for data processing with many pre-implemented functions to simplify certain data transformations (e.g., for text data, time data, etc.).

Compared to other SQL-based systems like PostGreSQL, DuckDB is very simple to install, as it is just a Python library, whereas many tools like PostGreSQL require an appropriate infrastructure.

To reuse the previous example, we can directly use the SQL code mentioned earlier.

```
import duckdb
duckdb.sql(
  """
```

```
  SELECT "INSEE commune", "Commune"
  FROM emissions
  WHERE dep=='12' AND Routier>500
  LIMIT 5
  """)
```

```
┌──────────────┬──────────────────┐
│ INSEE commune │      Commune     │
│    varchar    │      varchar     │
├──────────────┼──────────────────┤
│ 12001         │ AGEN-D'AVEYRON   │
│ 12002         │ AGUESSAC         │
│ 12006         │ ALRANCE          │
│ 12007         │ AMBEYRAC         │
│ 12008         │ ANGLARS-SAINT-FELIX │
└──────────────┴──────────────────┘
```

Here, the clause `FROM emissions` comes from the fact that we can directly execute SQL from a `Pandas` object via `DuckDB`. If we read directly in the query, it gets slightly more complex, but the logic remains the same.

```
import duckdb
duckdb.sql(
  f"""
  SELECT "INSEE commune", "Commune"
  FROM read_csv_auto("{url}")
  WHERE
    substring("INSEE commune",1,2)=='12'
    AND
    Routier>500
  LIMIT 5
  """)
```

```
┌──────────────┬──────────────────┐
│ INSEE commune │      Commune     │
│    varchar    │      varchar     │
├──────────────┼──────────────────┤
│ 12001         │ AGEN-D'AVEYRON   │
│ 12002         │ AGUESSAC         │
│ 12006         │ ALRANCE          │
│ 12007         │ AMBEYRAC         │
│ 12008         │ ANGLARS-SAINT-FELIX │
└──────────────┴──────────────────┘
```

The rendering of the DataFrame is slightly different from `Pandas` because, like `Polars` and many large data processing systems, `DuckDB` relies on lazy evaluation and thus only displays a sample of data. `DuckDB` and `Polars` are also well integrated with each other. You can run SQL on a `Polars` object via `DuckDB` or apply `Polars` functions to an initially read `DuckDB` object.

One of the interests of `DuckDB` is its excellent integration with the `Parquet` ecosystem, the already mentioned data format that is becoming a standard in data sharing (for example, it is the cornerstone of data sharing on the HuggingFace platform). To learn more about `DuckDB` and discover its usefulness for reading data from the French population census, you can check out this blog post.

3.3 `Spark`

`DuckDB` has pushed the boundaries of big data, which can be defined as the volume of data that can no longer be processed on a single machine without implementing a parallelization strategy.

Nevertheless, for very large data volumes, `Python` is well-equipped with the `PySpark` library. This is a Python API for the Spark language, a big data language based on Scala. This paradigm is built on the idea that `Python` users access it via clusters with many nodes to process data in parallel. The data will be read in blocks, processed in parallel depending on the number of parallel nodes. The `Spark` DataFrame API has a syntax close to previous paradigms with more complex engineering in the background related to native parallelization.

Informations additionnelles

i Python environment

This site was built automatically through a `Github` action using the `Quarto` reproducible publishing software (version 1.10.18).

The environment used to obtain the results is reproducible via uv. The `pyproject.toml` file used to build this environment is available on the `linogaliana/python-datascientist` repository

```
[project]
name = "python-datascientist"
version = "0.1.0"
description = "Source code for Lino Galiana's Python for data science course"
readme = "README.md"
requires-python = "≥3.13,<3.14"
dependencies = [
    "altair≥6.0.0",
    "cartiflette",
    "contextily==1.6.2",
    "duckdb≥0.10.1",
    "folium≥0.19.6",
    "gdal==3.11.4",
    "graphviz==0.20.3",
    "great-tables≥0.12.0",
    "gt-extras≥0.0.8",
    "ipykernel≥6.29.5",
    "jupyter≥1.1.1",
    "jupyter-cache≥1.0.0",
    "kaleido≥0.2.1",
    "langchain-community≥0.3.27",
    "loguru==0.7.3",
    "markdown≥3.8",
    "nbclient≥0.10.0",
    "nbformat≥5.10.4",
    "nltk≥3.9.1",
    "pandas≥3.0",
    "pip≥25.1.1",
    "plotly≥6.1.2",
    "plotnine≥0.15",
    "polars≥1.8.2",
    "pyarrow≥17.0.0",
    "pynsee≥0.1.8",
    "python-dotenv≥1.0.1",
```

```
    "python-frontmatter>=1.1.0",
    "pywaffle>=1.1.1",
    "requests>=2.32.3",
    "scikit-image>=0.24.0",
    "scikit-learn>=1.8.0",
    "scipy>=1.13.0",
    "seaborn>=0.13.2",
    "selenium<4.39.0",
    "spacy>=3.8.4",
    "webdriver-manager>=4.0.2",
    "wordcloud==1.9.3",
]

[tool.uv.sources]
cartiflette = { git = "https://github.com/inseefrlab/cartiflette" }
gdal = [
  { index = "gdal-wheels", marker = "sys_platform == 'linux'" },
  { index = "geospatial_wheels", marker = "sys_platform == 'win32'" },
]

[[tool.uv.index]]
name = "geospatial_wheels"
url = "https://nathanjmcdougall.github.io/geospatial-wheels-index/"
explicit = true

[[tool.uv.index]]
name = "gdal-wheels"
url = "https://gitlab.com/api/v4/projects/61637378/packages/pypi/simple"
explicit = true

[dependency-groups]
dev = [
    "nb-clean>=4.0.1",
]
```

To use exactly the same environment (version of `Python` and *packages*), please refer to the documentation for `uv`.

Bibliography