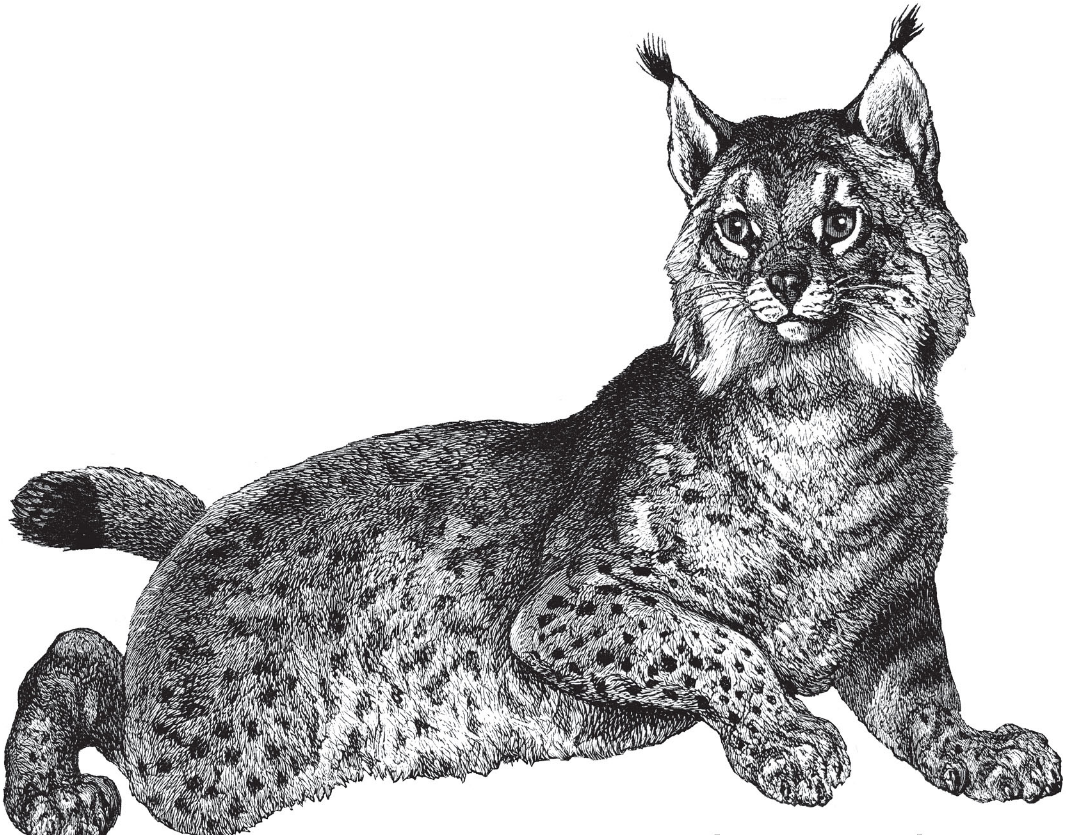
O'REILLY®

Python Polars
The Definitive Guide

Transforming, Analyzing, and Visualizing Data with a Fast and Expressive DataFrame API



Jeroen Janssens
& Thijs Nieuwdorp

Foreword by Ritchie Vink

Python Polars: The Definitive Guide

Blazingly Fast Data Analysis

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Jeroen Janssens and Thijs Nieuwdorp

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Python Polars: The Definitive Guide
by Jeroen Janssens and Thijs Nieuwdorp

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Aaron Black
Development Editor: Sarah Grey
Production Editor: Jonathon Owen
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

November 2024: First Edition

Revision History for the Early Release

2023-11-21: First Release
2024-01-30: Second Release
2024-03-29: Third Release
2024-05-23: Fourth Release
2024-07-15: Fifth Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098156084* for release details.

978-1-098-15608-4

[LSI]

Table of Contents

CHAPTER 1
First Steps

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

Overview

To explore all the exciting features Polars has to offer, you'll need to get it up and running first. In this chapter you're going to set up our working environment. This means you'll install Polars, or build it from source if you want to. After that you will learn how to configure Polars to your liking. You'll also learn how to download the datasets and code examples that are used in this book. Finally, you'll get a crash course in JupyterLab, which is the environment in which you'll be running the code examples in this book. In case you run into problems you can also run the code in a Docker container, which is explained at the end of this chapter.

It's recommended that you follow along with the code examples. Learning new libraries tends to stick much better when you're playing around with what you've learned, as opposed to just reading about the possibilities.

Installing Polars

In order to start working with Polars, you need to install it! The latest information on how to install Polars can always be found on the GitHub page. The following section is based on those instructions at the time of writing.

Polars works with optional dependencies for different use cases. At the time of writing Polars supports the optional dependencies as shown in Table 1-1.

Table 1-1. Polars dependencies

Tag	Description
`all`	Install all optional dependencies (all of the following)
`pandas`	Install with Pandas for converting data to and from Pandas Dataframes/Series
`numpy`	Install with numpy for converting data to and from numpy arrays
`pyarrow`	Reading data formats using PyArrow
`fsspec`	Support for reading from remote file systems
`connectorx`	Support for reading from SQL databases
`xlsx2csv`	Support for reading from Excel files
`deltalake`	Support for reading from Delta Lake Tables
`timezone`	Timezone support, only needed if are on Python<3.9 or you are on Windows

These dependencies can be installed together with Polars by using the following bracket notation. Since in this book we'll explore all the possibilities Polars has to offer we will install all optional dependencies. This can be done by running the following command in a Jupyter cell:

```
$ pip install 'polars[all]'
```

If you want to install a subset of the dependencies you can install it in the following way:

```
$ pip install 'polars[pandas,numpy]'
```

In case you only want to install the base package, the best way to install the latest version of Polars is to use `pip`:

```
$ pip install polars
```

Alternatively, some use `conda` to manage packages:

```
$ conda install -c conda-forge polars
```

However, `pip` is the Polars team's preferred way of installing Polars.

Compiling Polars from Scratch

Compiling Polars code from source has several advantages. Although in the case of Polars it is unlikely, because there are frequent releases, compiling from source allows access to the latest changes right away. Compiling from source allows you to make changes to the source code, re-compile it yourself, and make use of your own custom functionality. (In case it's a useful addition for everyone, be sure to contribute it to the project.) In the case you're working on a non-standard architecture compiling the code yourself is sometimes even required, because a pre-compiled version may not be available. And if you really know what you're doing, it's possible to tweak compiler optimizations when compiling your own code, potentially resulting in more efficient or faster software for your use case.

The steps required to compile Polars from source are as following:

1. Install the Rust compiler by following the instructions on the download page
2. Install maturin, a zero-configuration package that helps build and publish Rust crates with Python bindings.

   ```
   $ python -m pip install maturin
   ```

3. Compile the binary. There's two ways of compiling the binary:
   - In case you're prioritizing runtime performance over build time length (for example building the package once, and running it with maximum performance)

     ```
     $ cd py-polars
     $ maturin develop --release -- -C target-cpu=native
     ```

   - In case you're prioritizing faster build times over fast performance (for example in the case of developing and testing changes):

     ```
     $ cd py-polars
     $ maturin develop --release -- -C codegen-units=16 -C lto=thin \
       -C target-cpu=native
     ```

> Note that the Rust crate implementing the Python bindings is called `py-polars` to distinguish from the wrapped Rust crate Polars itself. However, both the Python package and the Python module are named `polars`, so you can conveniently run `pip install polars` and `import polars`.

Edge Case: Very Large Datasets

In case you'll be working with very large datasets that exceed 4.2 billion rows you will need to install Polars in a different way. Internally Polars uses a 32-bit integer representation to keep track of the data. If the dataset grows larger than that, Polars has to be compiled with a `bigidx` feature flag so the internal representation can reflect that. Additionally it can be installed using `pip install polars-u64-idx`. This might cause a loss of performance in case you don't need it.

Edge Case: Processors Lacking AVX support

Advanced Vector Extensions (AVX) refers to an extension that was made to the x86 instruction set architecture. These extensions allow for more comple and efficient computation operations at the CPU level. This set of features was first implemented on Intel's and AMD's CPUs that were shipped in 2011. These features are unfortunately not available on processors before that time, and are also not available on Apple Silicon, which is based on the ARM architecture. In case you're working with a chipset that doesn't support AVX you will need to install `polars-lts-cpu`. This package can also be found on PyPI, and can be installed with `pip install polars-lts-cpu`.

In case you compile this package yourself, be aware that this can only be compiled with a nightly version of Rust. The stable version doesn't allow building with the avx feature flag and will throw an E0554 error. You can download and set the nightly version as default by running:

```
$ rustup install nightly
$ rustup default nightly
```

If you run other projects that require a stable version of Rust, this command may disrupt them. To switch back to the stable branch of Rust, run `rustup default stable`.

Configuring Polars

Polars provides a number of configuration settings. These options allow you to enable alpha features, change the formatting of printed tables, set logging levels, and set the streaming chunk size. In the `polars.Config` class you can find the following settings, and some additional ones that we won't cover. A complete overview can be found in the Polars config documentation online. The section below is an excerpt from that documentation.

The most important ones are shown in Table 1-2.

Table 1-2. A few of the notable Polars configuration settings

Setting	Description
`activate_decimals(active: bool)`	The `Decimal` datatype is currently in alpha. You have to turn it on manually
`set_fmt_str_lengths(n: int)`	Sets the number of characters used to display string values
`set_tbl_cols(n: int)`	Sets the number of columns that are visible when displaying tables
`set_streaming_chunk_size(size: int)`	Overwrite chunk size used in `streaming` engine
`set_verbose(active: bool)`	Enable additional verbose/debug logging

These config options can be changed, saved, and loaded as a JSON string using the `load(cfg: str | Path)` and `save(file: str | Path)` functions. To see the current state you can call `state()`. To restore all settings back to the defaults you can call `restore_defaults()`.

Temporary Configuration Using a Context Manager

To run a specific scope of code with different a different configuration you can use a context manager. A context manager is a construct in Python that allows for precise creation and removal of resources. The context for which resources are defined is indicated by calling the context manager using the `with` keyword, and indenting the scope of code that should be affected by it. In Polars' case only the code within the scope of the context manager will be executed with the given configuration after which it returns to the previous settings.

```
import polars as pl

with pl.Config() as cfg:
    cfg.set_verbose(True)
    # Polars operation you want to see the verbose logging of

# Code outside of the scope is not affected
```

A more concise approach is to pass the options directly as arguments to the `Config()` constructor. If you use this approach, you can omit the `set_` part of the option.

```
with pl.Config(verbose=True):
    # Polars operation you want to see the verbose logging of
    pass
```

In order to showcase some of the formatting configuration settings you're going to generate your first DataFrame. A DataFrame is a two-dimensional data structure representing data as a table with rows and columns. This is one of the main data structures that is used in Polars. Later on in this book we'll introduce you more deeply to all structures.

In the code below we've made a short function that is able to generate a random string with a length that can be set. After that we create a dictionary that has the keys "column_1" to "column_20" and 5 rows of randomly generated strings with a length of 50 characters.

```
import random
import string

def generate_random_string(length: int) -> str:
    return "".join(random.choice(string.ascii_letters) for i in range(length))

data = {}
for i in range(1, 11):
    data[f"column_{i}"] = [generate_random_string(50) for _ in range(5)]

df = pl.DataFrame(data)
```

Let's see what this DataFrame looks like when you run the code:

```
df
```

```
shape: (5, 10)
┌─────────────┬─────────────┬─────────────┬───┬─────────────┬─────────────┬─────────────┐
│ column_1    ┆ column_2    ┆ column_3    ┆ … ┆ column_8    ┆ column_9    ┆ column_10   │
│ ---         ┆ ---         ┆ ---         ┆   ┆ ---         ┆ ---         ┆ ---         │
│ str         ┆ str         ┆ str         ┆   ┆ str         ┆ str         ┆ str         │
╞═════════════╪═════════════╪═════════════╪═══╪═════════════╪═════════════╪═════════════╡
│ NITxKLUkXv  ┆ vrLgRRjGXL  ┆ ErZIZfRrEq  ┆ … ┆ beymgYVfd   ┆ bIghJrUq0   ┆ HGdFNGSPa   │
│ yOYxtzSnWQ  ┆ QPcPJFsbjj  ┆ jUgWnjTSkj  ┆   ┆ LsnHFrZmS   ┆ JqRwQUErd   ┆ BmfvCdhzj   │
│ …           ┆ …           ┆ …           ┆   ┆ Vg…         ┆ Zq…         ┆ vl…         │
│ MqZmJeOHNK  ┆ ubSBwYOgYk  ┆ HQdUpgsJus  ┆ … ┆ eCkqtkOlh   ┆ WAXfTTOBr   ┆ vMRyUWIKs   │
│ XceAPNdRbO  ┆ fTatOQmkRm  ┆ uscqAuvSfP  ┆   ┆ sGftkqIII   ┆ PsfVWUnPQ   ┆ NxGcuadnN   │
│ …           ┆ …           ┆ …           ┆   ┆ ox…         ┆ Fu…         ┆ Iq…         │
│ HlXQdVFTVL  ┆ ybaZRpdIJh  ┆ VtnYHFRNNA  ┆ … ┆ LPZvTIIwV   ┆ SkjhgiCfk   ┆ WFNCaqjtg   │
│ DbZHFIWPUw  ┆ PzrHJsjSaA  ┆ KCTLizyVyl  ┆   ┆ UqtjLJOoU   ┆ eDxeEcShL   ┆ aEadCeEDR   │
│ …           ┆ …           ┆ …           ┆   ┆ jW…         ┆ Rg…         ┆ Aw…         │
│ dODwyenwQR  ┆ BmfJOHYZkA  ┆ JbXtfyUyNG  ┆ … ┆ NOtdYhuJy   ┆ dbjtoFjvZ   ┆ yQFKPBjQV   │
│ PMqTnmiEzN  ┆ oMfoGlBbBH  ┆ DXbgdKpXjo  ┆   ┆ yTiStIGcI   ┆ NZlgFFPGW   ┆ vjgyHvJrC   │
│ …           ┆ …           ┆ …           ┆   ┆ jZ…         ┆ EW…         ┆ jA…         │
│ IuXzwotCLy  ┆ cZzaPcRNBU  ┆ vTlcINzgBB  ┆ … ┆ kLtrJblaW   ┆ ZxIcqHlie   ┆ WpcBNWzeo   │
│ OyjNrWVpyT  ┆ DfuMHNCJGn  ┆ zoCWOeoaTT  ┆   ┆ xtcSpyOnC   ┆ MgqgEqCXh   ┆ OXJFltrfa   │
│ …           ┆ …           ┆ …           ┆   ┆ ow…         ┆ DU…         ┆ uf…         │
└─────────────┴─────────────┴─────────────┴───┴─────────────┴─────────────┴─────────────┘
```

Unfortunately, the standard DataFrame output doesn't fit in this book. Say you want to make it fit, but you still want to see as many columns as possible by shrinking the text that is displayed. In that case you can set the amount of columns that is showed to minus one (to print all of them), and lower the string length that is displayed to four.

```python
with pl.Config(tbl_cols=-1, fmt_str_lengths=4):
    print(df)
```

```
shape: (5, 10)
┌─────┬───────┬───────┬───────┬───────┬───────┬───────┬───────┬───────┬───────┐
│ col ┆ colu… ┆ colu… ┆ colu… ┆ colu… ┆ colu… ┆ colu… ┆ colu… ┆ colu… ┆ colu… │
│ u…  ┆ ---   ┆ ---   ┆ ---   ┆ ---   ┆ ---   ┆ ---   ┆ ---   ┆ ---   ┆ ---   │
│ --- ┆ str   ┆ str   ┆ str   ┆ str   ┆ str   ┆ str   ┆ str   ┆ str   ┆ str   │
│ str ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       │
╞═════╪═══════╪═══════╪═══════╪═══════╪═══════╪═══════╪═══════╪═══════╪═══════╡
│ NIT ┆ vrLg… ┆ ErZI… ┆ aXoG… ┆ zQXd… ┆ BqAF… ┆ PrRN… ┆ beym… ┆ bIgh… ┆ HGdF… │
│ x…  ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       │
│ MqZ ┆ ubSB… ┆ HQdU… ┆ jvRg… ┆ zcDr… ┆ Pees… ┆ Zqsj… ┆ eCkq… ┆ WAXf… ┆ vMRy… │
│ m…  ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       │
│ HlX ┆ ybaZ… ┆ VtnY… ┆ JFHN… ┆ EXzX… ┆ aBdy… ┆ QkOA… ┆ LPZv… ┆ Skjh… ┆ WFNC… │
│ Q…  ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       │
│ dOD ┆ BmfJ… ┆ JbXt… ┆ rHAq… ┆ pwJO… ┆ oRCW… ┆ OgCG… ┆ NOtd… ┆ dbjt… ┆ yQFK… │
│ w…  ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       │
│ IuX ┆ cZza… ┆ vTlc… ┆ JNjW… ┆ OEuZ… ┆ AXWe… ┆ eQTy… ┆ kLtr… ┆ ZxIc… ┆ WpcB… │
│ z…  ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       ┆       │
└─────┴───────┴───────┴───────┴───────┴───────┴───────┴───────┴───────┴───────┘
```

Compact, yet it shows all of the columns. Perfect.

Context managers contain two key methods under the hood. They consist of a `__enter__` and `__exit__` that are respectively called before and after running the code within the indicated context. A small example would be:

```python
class YourContextManager:
    def __enter__(self):
        print("Entering context")

    def __exit__(self, type , value, traceback):
        print("Exiting context")

with YourContextManager():
    print("Your code")
```

```
Entering context
Your code
Exiting context
```

One of the popular uses of a context manager is to write or read from files, which can be done like this:

```python
with open("filename.txt", "w") as file:
    file.write("Hello, world!")
```

Local Configuration Using a Decorator

If you want to change configuration settings during a specific function call, you can decorate that function with the `pl.Config()` decorator. Just as in the context manager, you can omit the `set_` part of the option.

```python
@pl.Config(ascii_tables=True)
def write_ascii_frame_to_stdout(df: pl.DataFrame) -> None:
    print(str(df))

@pl.Config(verbose=True)
def function_that_im_debugging(df: pl.DataFrame) -> None:
    # Polars operation you want to see the verbose logging of
    pass
```

Downloading Datasets and Code Examples

In order to run the code examples in this book you'll need to download the datasets that are used, using `git`. `git` is a version control system, with which you can download a code repository, and keep track of changes to it. You can install `git` by following the instructions on the Git website.

The datasets in this book are available in the repository that accompanies it. After having downloaded and installed `git`, you can download the repository by running the following command below. It will create a new directory in the current working directory called `python-polars-the-definitive-guide`.

```
$ git clone https://github.com/jeroenjanssens/python-polars-the-definitive-guide.git
```

You can install the dependencies that are needed to run the code examples in this book by running the following command:

```
$ cd python-polars-the-definitive-guide
$ pip install -r requirements.txt
```

This will set up everything on your system to work along with the book.

Crash Course JupyterLab

To run the code examples in this book you'll need to use Jupyter. Jupyter is a web-based interactive development environment of notebooks, code, and data. To get started, you can create a Python 3 Notebook using the button in the top row.

To start Jupyter you can run the following command in the terminal:

```
$ jupyter lab
```

This opens a window in the browser with the Jupyter interface. If this does not pop up, you need to copy the URL that will be printed in the terminal. It will look something like `http://127.0.0.1:8888/lab?token=...`. Click it, or copy and paste it into a browser window to connect to the Jupyter server inside the container. This will open up JupyterLab in your browser, in which you can get to work.

In order to work in a Jupyter notebook you'll need to know some basics. Jupyter content is loaded in cells. These cells can be marked as different programming languages, but also as Markdown. In this book you will mostly work with Python code cells. To navigate and edit these cells Jupyter knows two modes: **command mode** and **edit mode**.

Command mode is the default mode when opening a notebook, or when pressing `Esc` when in a cell. When it's active the selected cell has a blue border, and no cursor inside of it. Command mode is used to edit the notebook as a whole, or add, delete, or edit cell types in the notebook.

Edit mode can be activated by pressing `Enter` when on a selected cell. In this mode the selected cell gets a green border. Edit mode is used to write in cells.

Keyboard Shortcuts

A few important shortcuts you should know are listed below. Table 1-3 lists shortcuts that can be run in any mode, Table 1-4 lists shortcuts that can be run in command mode, and Table 1-5 lists shortcuts that can be run in edit mode.

Any Mode

Table 1-3. Shortcuts that can be run regardless of the current mode

Shortcut	Effect
Shift + Enter	Run the selected cell, and select the cell below
Ctrl + Enter	Run the selected cell, and don't move the selection
Alt + Enter	Run the selected cell, and insert a new cell below
Ctrl + S	Save the notebook

Command Mode

Table 1-4. Shortcuts that can be run in command mode

Shortcut Key	Action
Enter	Switch to Edit Mode
Up / K	Select the cell above
Down / J	Select the cell below
A	Insert a new cell above the current cell
B	Insert a new cell below the current cell
D, D (press the key twice)	Delete the selected cell

Shortcut Key	Action
Z	Undo cell deletion
M	Change the cell type to Markdown
Y	Change the cell type to Code

Edit Mode

Table 1-5. Shortcuts that can be run in edit mode

Shortcut Key	Action
Esc	Switch to Command Mode
Ctrl + Shift + -	Split the current cell at cursor

Keep these shortcuts handy and within no-time you'll fly across the screen in any Jupyter notebook.

Additionally, Jupyter has a few special marks that can be used in code cells. An exclamation mark (`!`) before a command tells the Jupyter kernel to run the command following it in a bash session, instead of interpreting it as Python. For example, you can install Polars from notebooks with: `!pip install polars`

Another special mark that can be used is the percentage mark (`%`). The percentage mark is a special feature of the IPython kernel called a ``magic``. Magics are built-in commands designed to solve various common problems. These are not part of the Python language but they're features of the IPython shell. Magics come in two kinds: 1. Line magics: These are preceded by a single `%` and work a lot like shell commands.In this case we're using the `%pip` magic with which we can install a package in the virtual environment that the IPython shell is running in. 2. Cell magics: These are preceded by double `%%`. Examples are `%%time`, which times how long the code in that cell takes to run, and `%%bash` which we'll use later to execute multiple bash commands in one go. To see all other commands the IPython shell has to offer you can run `%lsmagic`.

Using Polars in a Docker Container

There are about as many different system configurations in the world as there are systems. In case you run into problems when executing the code, you can alternatively use Docker. Docker allows you to run the code in this book in a container in which we have precise control over what the system configuration looks like. This way we can make sure that the code runs on your system the way it runs on ours.

To get started, you need to pull a Docker image. An image can be thought of as the instructions that describe how to build the container in which you will run our code.

The image you will use in this book is provided by Jupyter, since you will be running our code from notebooks. To pull the Jupyter image you'll first need Docker.

Go to the Docker website and download Docker Desktop for your operating system. Once the download is complete, install it, and launch it.

Now that Docker is running in the background you can open up the terminal (also known as the command prompt on Windows). In that you can run the following command to run the image.

```
$ docker run -p 8888:8888 jupyter/minimal-notebook
```

This command will attempt to run the image `jupyter/minimal-notebook`. In case it is not available on the system, it is pulled from the DockerHub. After that docker starts the container and exposes port 8888 to the system, so that you can open the Jupyter serve running in docker in the browser.

Conclusion

In this chapter you've learned how to:

- install Polars and optional dependencies, and how to compile it from source if necessary.
- tweak the configuration of Polars to make it just right.
- download the datasets and code examples that are used in this book.
- run the code examples in JupyterLab.
- run the code examples in a Docker container in case you run into problems.

This will allow you to run Polars yourself and start exploring the opportunities it brings. In the next chapter you can dive right into that by taking a closer look at the similarities and differences that Polars has compared to the popular DataFrame libraries Spark and Pandas.

CHAPTER 2

Data Types and Data Structures

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

Now that you've gotten a sneak peek at the differences between popular data frame libraries and Polars, it's time to focus on how Polars works.

Data comes in many shapes and sizes, all of which need to be stored in memory order to work with it. To accommodate all the data you'll be working with, Polars implements the Arrow memory specification, which allows an array of data types to work with. In this chapter you'll go through these data types, and we'll elaborate on a few of the ones that aren't so straight forward.

First we'll walk through Apache Arrow, the library Polars uses to manage in-memory data storage. After that we'll go over the different data types that are available. We'll elaborate on some of the data types that aren't so straight forward. Lastly we'll go over the structures Polars uses to work with all these data types.

Arrow Data Types

To store data efficiently, Polars builds on top of the Apache Arrow project.

Arrow describes itself as "a cross-language development platform for in-memory analytics." It defines a "language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware like CPUs and GPUs." Arrow brings a few advantages out of the box.

First, it uses a columnar format. The columnar format enables data adjacency for sequential access or scans, which optimizes the process of reading large quantities of data in a contiguous block. This way you can store the data and read it in large, sequential chunks.

On top of that, this contiguous columnar layout is vectorization-friendly. It also lets you use modern Single Instruction, Multiple Data (SIMD) operations, which perform the same operations on multiple data points simultaneously.

To elaborate on these advantages, we'll introduce the metaphor of a filing cabinet:

This is illustrated in Figure 2-1.

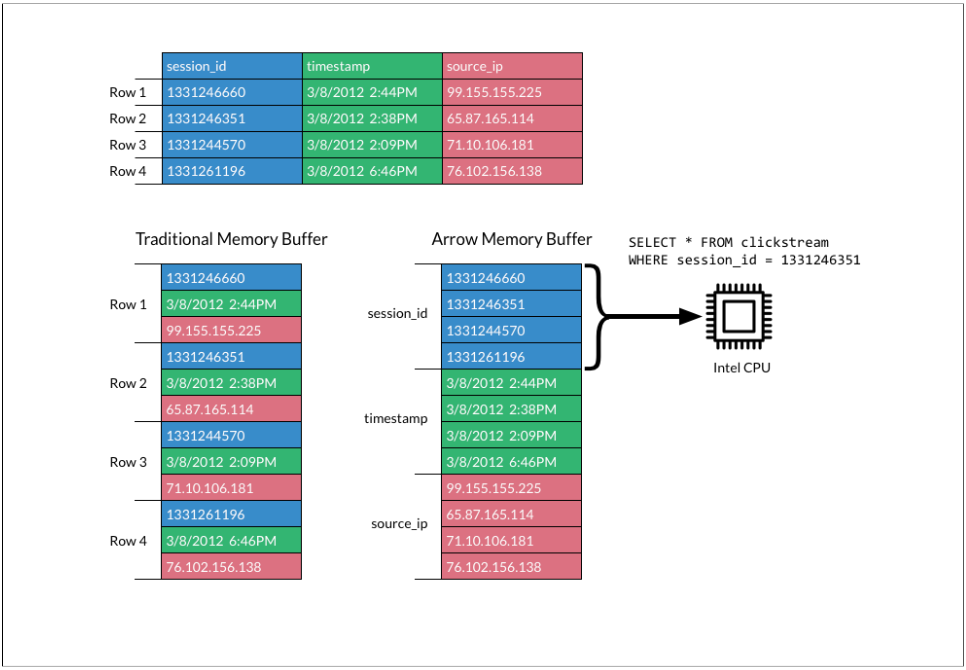


Figure 2-1. Illustration of the Arrow memory buffer and its advantages for computation

The advantages could be well explained using a filing cabinet where you store your sales dossiers. In a row-based format every drawer of the cabinet contains all the data you need on a single sale. A drawer contains the person it was sold to, what items were sold, the price of the sale, and when the sale happened. If you always want to dig up all the information of the sales you made, it's practical to keep all that information bundled together.

In analytical queries, however, it's more common to look for specific parts of the sales dossier, instead of all of it. One example could be the following: you want to make a report that contains your 5 biggest customers. This way you'll know what customers to put some extra effort into and pamper. If you ordered your cabinet in the row-based manner where every drawer contains the file of one customer you'd have to open up every single drawer to look at the total price of sales, and who the dossier belongs to.

When we order our cabinet in a way that is column-based every drawer contains a single data category of all the customers. That means one of the drawers would contain all the sale prices.

The sequential reading means you can just start at the first file in the drawer, and keep going to the next one until you reach the end of the drawer. This speeds things up because you don't have to close the drawer, go to another one, open it, and look for the relevant file. From the price drawer you can then determine the dossier ID's of the biggest customers. In order to know who the customers are you go to name drawer, and go over the files until you've found the 5 names matching the dossier ID's you just found. This means you'll only have to open 2 drawers instead of all of them, saving you a lot of hassle and time.

Because of this columnar format, Arrow provides O(1), or *constant-time*, random access. This means that no matter how large the data set becomes, the time it takes to access any single piece of data remains constant. If we go back to our filing cabinet analogy, this would look like we know exactly where every piece of information is stored. Not only which drawer, but also exactly where in the drawer. This means you don't have to go searching through drawers until you come to the relevant piece of data. For operations that need to access specific data points in a large dataset, this is a significant performance benefit.

Arrow supports implementations in many popular languages. At the time of writing these include: C/GLib, C++, C#, Go, Java, JavaScript, Julia, MATLAB, Python, R, Ruby, and Rust. The degree of implementation might differ between languages: for example the `Float16` data type is not implemented in every language.

A `Float32` data type is a 32-bit floating point number format, also known as single-precision floating point. This is more common than the 16-bit half-precision format and provides a good balance between range and precision.

These 32 bits contain the following information:

- The 1st bit represents the sign bit (0 for positive, 1 for negative)
- The 2-9th bits represent the exponent by which the fraction is multiplied
- The 10-32nd bit represent the fraction with an implicit leading 1 before the binary representation.

The formula for calculating the value of a Float32 is given by:

$(-1)^{sign} * (1 + fraction) * 2^{(exponent - bias)}$

The bias for `Float32` is a constant value of 127. This means that the actual exponent value in decimal form is obtained by subtracting this bias from the exponent's binary representation. The reason a float uses a bias is to ensure it can represent both very large and very tiny numbers efficiently.

As an example consider the following float in bits:

0 10000010 10100000000000000000000

- 0 - means the float is positive.
- 10000010 - The *exponent* in binary, which is 130 in decimal.
- 10100000000000000000000 - This is the *fraction* part in binary. It's calculated by adding an implicit leading 1 (for normalized numbers) to the binary digits, interpreted as follows: 1 (the implicit leading 1) plus $1 * 2^{-1}$ (the first digit, representing 0.5) plus $0 * 2^{-2}$ (the second digit, ignored since it's 0) plus $1 * 2^{-3}$ (the third digit, representing 0.125). Subsequent digits are zeros and do not contribute to the value. Therefore, the fraction equals $1 + 0.5 + 0.125 = 1.625$

Plugging these values into the formula gives:

- Float = $(-1)^0 * (1 + 0.5 + 0.125) * 2^{(130 - 127)}$
- Float = 1 * 1.625 * 8
- Float = 13

Implementations in many languages let you use a shared mutable dataset without serialization/deserialization. Normally different languages have different implementations of the ways data is represented in the bits in memory. This means that in order to match data across languages you first have to deserialize the data from one format, and then serialize it to the format of the other. This translation step takes time. Arrow prevents this by allowing all supported implementations and languages to talk in a unified way to the same dataset. This sharing of a mutable dataset is called Inter Process Communication (IPC).

The core of Polars is written in Rust to benefit from the language's performance. Using the Arrow Rust implementation, Polars has implemented the data types shown in Table 2-1. Some data types occur multiple times with different bit-sizes. This allows you take store data that fits within the range with a smaller memory footprint.

Table 2-1. Data types available in Polars

Group	Type	Details	Range
Base class	`DataType`	Base class for all Polars data types.	
Numeric	`Decimal`	Decimal 128-bit type with an optional precision and non-negative scale.	Can exactly represent 38 significant digits
	`Float32`	32-bit floating point type.	-3.4e+38 to 3.4e+38
	`Float64`	64-bit floating point type.	-1.7e+308 to 1.7e+308
	`Int8`	8-bit signed integer type.	-128 to 128
	`Int16`	16-bit signed integer type.	-32,768 to 32,767
	`Int32`	32-bit signed integer type.	-2,147,483,648 to 2,147,483,647
	`Int64`	64-bit signed integer type.	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
	`UInt8`	8-bit unsigned integer type.	0 to 255
	`UInt16`	16-bit unsigned integer type.	0 to 65,535
	`UInt32`	32-bit unsigned integer type.	0 to 4,294,967,295
	`UInt64`	64-bit unsigned integer type.	0 to 1.8446744e+19
Temporal	`Date`	Calendar date type. Uses the Arrow date32 data type, days since UNIX epoch 1970-01-01 as int32.	-5877641-06-24 to 5879610-09-09
	`Datetime`	Calendar date and time type. Exact timestamp encoded with int64 since UNIX epoch. Default unit microseconds.	
	`Duration`	Time duration/delta type.	
	`Time`	Time of day type.	
Nested	`Array(*args, **kwargs)`	Fixed length list type.	
	`List(*args, **kwargs)`	Variable length list type.	
	`Struct(*args, **kwargs)`	Struct type.	

Group	Type	Details	Range
Other	`Boolean`	Boolean type taking 1 bit of space.	True or False
	`Binary`	Binary type with variable-length bytes.	
	`Categorical`	A categorical encoding of a set of strings. Allows for more efficient memory usage if a column contains few unique strings.	
	`Null`	Type representing Null / None values.	
	`Object`	Type for wrapping arbitrary Python objects.	
	`String`	UTF-8 encoded string type of variable length.	
	`Unknown`	Type representing Datatype values that could not be determined statically.	

Sometimes when creating a DataFrame using Python data, arbitrary Python data may need to be added. One example could be that in a DataFrame you want to store a machine-learning model. In this case the `Object` data type is used. This data type allows for arbitrary Python objects to be put into a DataFrame.

The downside is that this data cannot be processed using the normal functions. None of the optimizations are used, because Polars does not use Python to look at what the data represents. This means that an `Object` column can be seen as a passenger in the DataFrame, which is passed on in joins, but does not take part in optimized calculations.

Generally using an `Object` is discouraged when the data can be represented by another data type, but there might be use cases for it.

In documentation you may find the `Unknown` data type. The `Unknown` data type is only used internally as a placeholder and should not be used in your code.

Nested Data Types

You might've noticed that the nested data types have arguments. This is because nested data types are a special class of data types. A data type is nested when it can contain other data types. The arguments define things like how many elements it can contain, and what data types it contains. Polars has three of these: *Array*, *List*, and *Struct*.

An `Array` is quite similar to a Numpy `ndarray`. It's a collection of elements that are of the same data type. Besides this the length of the array must be the same on all rows. The arguments `Array` takes are the width of the array and the data type in the array.

```
import polars as pl

array_df = pl.DataFrame(
    [
        pl.Series("array_1", [[1, 3], [2, 5]]),
        pl.Series("array_2", [[1, 7, 3], [8, 1, 0]]),
    ],
    schema={
        "array_1": pl.Array(width=2, inner=pl.Int64),
        "array_2": pl.Array(width=3, inner=pl.Int64)
    }
)
array_df
```

```
shape: (2, 2)
┌───────────────┬───────────────┐
│ array_1       ┆ array_2       │
│ ---           ┆ ---           │
│ array[i64, 2] ┆ array[i64, 3] │
╞═══════════════╪═══════════════╡
│ [1, 3]        ┆ [1, 7, 3]     │
│ [2, 5]        ┆ [8, 1, 0]     │
└───────────────┴───────────────┘
```

A `List` is comparable to an `Array` in that it is a collection of elements of the same data type. However in contrast to the `Array`, a `List` does not have to have the same length on every row. Note that it's different from the Python `list` which can contain different data types. It is possible to store Python lists in the column, by making the data type `Object`. The only argument `List` takes is what data type it contains.

```
list_df = pl.DataFrame(
    {
        "integer_lists": [[1, 2], [3, 4]],
        "float_lists": [[1.0, 2.0], [3.0, 4.0]],
    }
)
list_df
```

```
shape: (2, 2)
┌───────────────┬─────────────┐
│ integer_lists ┆ float_lists │
│ ---           ┆ ---         │
│ list[i64]     ┆ list[f64]   │
╞═══════════════╪═════════════╡
│ [1, 2]        ┆ [1.0, 2.0]  │
│ [3, 4]        ┆ [3.0, 4.0]  │
└───────────────┴─────────────┘
```

Lastly, the `Struct` is the idiomatic way of working with multiple columns in Polars. The way Polars transforms data is with the use of expressions. We'll dive deeper into them in Chapter 4, for now all you need to know is that they are functions that map an input `Series`, to an output, also type `Series`: `fn(Series) -> Series` To allow expressions to use multiple columns as input, the `Struct` data type can be used to represent a collection of columns as a single column. This way an expression that requires multiple columns as input can still meet the requirement of only taking a Series as input. This means that a `Struct` can contain different data types, as long as they match over rows. `Struct`s can be constructed using Python dictionaries, like so:

```
rating_series = pl.Series(
    "ratings",
    [
        {"Movie": "Cars", "Theatre": "NE", "Avg_Rating": 4.5},
        {"Movie": "Toy Story", "Theatre": "ME", "Avg_Rating": 4.9},
    ],
)
rating_series
```

```
shape: (2,)
Series: 'ratings' [struct[3]]
[
        {"Cars","NE",4.5}
        {"Toy Story","ME",4.9}
]
```

Missing Values

In Polars, missing data is always represented with `null`. This `null` for a missing value applies to *all data types*, including the numerical ones. Information about missing values is stored in metadata of the Arrow array.

Additionally, whether a value is missing is stored in its *validity bitmap*, which is a bit that is set to 1 if the value is present and 0 if it is missing. This lets you cheaply check how many values are missing in a column, using methods like `null_count()` and `is_null()`.

To demonstrate this, we'll create a DataFrame with some missing values:

```
df = pl.DataFrame(
    {
        "value": [None, 2, 3, 4, None, None, 7, 8, 9, None],
    },
)
print(df)
```

```
shape: (10, 1)
┌───────┐
│ value │
│ ---   │
```

```
│ i64  │
╞══════╡
│ null │
│ 2    │
│ 3    │
│ 4    │
│ null │
│ null │
│ 7    │
│ 8    │
│ 9    │
│ null │
└──────┘
```

You can fill in missing data using the `fill_null()` method, which you can call in multiple ways:

- Using a single value
- Using a fill strategy
- Using an expression
- Using an interpolation

The following example shows how you can fill with a single value `pl.lit(...)` value:

```
print(
    df
    .with_columns(
        pl.col("value")
        .fill_null(-1)
        .alias("filled_with_lit")
    )
)
```

```
shape: (10, 2)
┌───────┬─────────────────┐
│ value ┆ filled_with_lit │
│ ---   ┆ ---             │
│ i64   ┆ i64             │
╞═══════╪═════════════════╡
│ null  ┆ -1              │
│ 2     ┆ 2               │
│ 3     ┆ 3               │
│ 4     ┆ 4               │
│ null  ┆ -1              │
│ null  ┆ -1              │
│ 7     ┆ 7               │
│ 8     ┆ 8               │
│ 9     ┆ 9               │
│ null  ┆ -1              │
└───────┴─────────────────┘
```

The second option is to use a fill strategy. A *fill strategy* allows you to pick an imputation method out of the following list:

- `None`: Do not fill missing values.
- `forward`: Fill with the previous non-null value.
- `backward`: Fill with the next non-null value.
- `min`: Fill with the minimum value of the column.
- `max`: Fill with the maximum value of the column.
- `mean`: Fill with the mean of the column. Note that this `mean` is cast to the data type of the column, which in the case of an int means the part behind the comma is cut off.
- `zero`: Fill with 0.
- `one`: Fill with 1.

In the example below you'll see all of these strategies next to each other:

```
print(
    df
    .with_columns(
        pl.col("value")
        .fill_null(strategy="forward")
        .alias("forward"),
        pl.col("value")
        .fill_null(strategy="backward")
        .alias("backward"),
        pl.col("value")
        .fill_null(strategy="min")
        .alias("min"),
        pl.col("value")
        .fill_null(strategy="max")
        .alias("max"),
        pl.col("value")
        .fill_null(strategy="mean")
        .alias("mean"),
        pl.col("value")
        .fill_null(strategy="zero")
        .alias("zero"),
        pl.col("value")
        .fill_null(strategy="one")
        .alias("one"),
    )
)
```

```
shape: (10, 8)
┌───────┬─────────┬──────────┬─────┬─────┬──────┬──────┬─────┐
│ value ┆ forward ┆ backward ┆ min ┆ max ┆ mean ┆ zero ┆ one │
│ ---   ┆ ---     ┆ ---      ┆ --- ┆ --- ┆ ---  ┆ ---  ┆ --- │
```

```
│ i64  ┆ i64   ┆ i64     ┆ i64 ┆ i64 ┆ i64 ┆ i64 ┆ i64 │
╞══════╪═══════╪═════════╪═════╪═════╪═════╪═════╪═════╡
│ null ┆ null  ┆ 2       ┆ 2   ┆ 9   ┆ 5   ┆ 0   ┆ 1   │
│ 2    ┆ 2     ┆ 2       ┆ 2   ┆ 2   ┆ 2   ┆ 2   ┆ 2   │
│ 3    ┆ 3     ┆ 3       ┆ 3   ┆ 3   ┆ 3   ┆ 3   ┆ 3   │
│ 4    ┆ 4     ┆ 4       ┆ 4   ┆ 4   ┆ 4   ┆ 4   ┆ 4   │
│ null ┆ 4     ┆ 7       ┆ 2   ┆ 9   ┆ 5   ┆ 0   ┆ 1   │
│ null ┆ 4     ┆ 7       ┆ 2   ┆ 9   ┆ 5   ┆ 0   ┆ 1   │
│ 7    ┆ 7     ┆ 7       ┆ 7   ┆ 7   ┆ 7   ┆ 7   ┆ 7   │
│ 8    ┆ 8     ┆ 8       ┆ 8   ┆ 8   ┆ 8   ┆ 8   ┆ 8   │
│ 9    ┆ 9     ┆ 9       ┆ 9   ┆ 9   ┆ 9   ┆ 9   ┆ 9   │
│ null ┆ 9     ┆ null    ┆ 2   ┆ 9   ┆ 5   ┆ 0   ┆ 1   │
└──────┴───────┴─────────┴─────┴─────┴─────┴─────┴─────┘
```

The third way of filling null values is with an expression like `pl.col("value").mean()`:

```python
print(
    df
    .with_columns(
        pl.col("value")
        .fill_null(pl.col("value").mean())
        .alias("expression_mean")
    )
)
```

```
shape: (10, 2)
┌───────┬─────────────────┐
│ value ┆ expression_mean │
│ ---   ┆ ---             │
│ i64   ┆ f64             │
╞═══════╪═════════════════╡
│ null  ┆ 5.5             │
│ 2     ┆ 2.0             │
│ 3     ┆ 3.0             │
│ 4     ┆ 4.0             │
│ null  ┆ 5.5             │
│ null  ┆ 5.5             │
│ 7     ┆ 7.0             │
│ 8     ┆ 8.0             │
│ 9     ┆ 9.0             │
│ null  ┆ 5.5             │
└───────┴─────────────────┘
```

The fourth and last way of filling nulls is with an interpolation method like `df.interpolate()`

```python
print(
    df.interpolate()
)
```

```
shape: (10, 1)
┌───────┐
│ value │
```

```
│ ---  │
│ f64  │
╞══════╡
│ null │
│ 2.0  │
│ 3.0  │
│ 4.0  │
│ 5.0  │
│ 6.0  │
│ 7.0  │
│ 8.0  │
│ 9.0  │
│ null │
└──────┘
```

`NaN` (meaning "not a number") values are not considered missing data in Polars. These values are used for the `Float` data types to represent the result of an operation that is not a number.

Consequently, `NaN` values are not counted as `null` values in functions like `null_count()` or `fill_null()`. As an alternative, use `is_nan()` and `fill_nan()` to work with these values.

Series, DataFrames, and LazyFrames

All these types of data can be stored in a *Series* or a *DataFrame*. A `Series` is a single column of data of the same data type. A DataFrame is a two-dimensional data structure that represents the data as a table with rows and columns. A DataFrame is internally represented as a collection of `Series` of the same length. Every `Series` (and so every column in a DataFrame) are internally represented as a `ChunkedArray`.

A `ChunkedArray` is a container class for a sequence of arrays of data. Using ChunkedArrays instead of a single array with all the data allows for several optimizations, including optimized memory management. When you add data to a ChunkedArray, the data is added to the existing object. This way Polars doesn't have to copy over data to a new one and also doesn't have to garbage collect the old one, saving time. On top of that, Polars allows for splitting data in chunks that can be operated on individually and in parallel in order to maximize performance. Each chunk can be processed by a different CPU core, speeding up calculations dramatically.

Managing these chunks optimizes the way Polars works with data. *Rechunking* is the process of changing the chunk size of a ChunkedArray. In Polars rechunking generally refers to putting all the data in a single chunk. Within every chunk the data is kept contiguous in memory. In the eager case after reads the data is rechunked. This is done because the assumption is that in eager mode the user wants to perform analysis on the data. Often the same frame will be queried multiple times, which makes the additional time it takes to rechunk worth the effort. When using a lazy evaluation the query optimizer decides when to rechunk.

Generally this is something you won't have to take into account. It's just good to know that when setting the `rechunk` parameter to `True` in an operation, there's actually two operations happening. This is something to be taken into account when benchmarking.

A LazyFrame is a DataFrame that is evaluated lazily. This means that where a DataFrame is an object that contains all the data in memory, a LazyFrame contains no actual data at all. All the read operations and transformations applied to it are not evaluated until they are needed. Until the point where the resulting DataFrame is needed, it is nothing more than a query graph containing the computational steps necessary to get the final result. Working with this graph provides several opportunities for optimization using a query optimizer.

We will dive deeper into the usage of the different API's, among which is the Lazy API, in Chapter 5.

Data Type Conversion

One of the functions of a `Series` is `cast()`. This changes the data type from the current one to the one provided as an argument. Say after parsing a csv file, all the data in it is currently a string. Together with the DataFrame function `.estimated_size()` we can estimate how much memory a DataFrame takes.

```
string_df = pl.DataFrame({"id": ["10000", "20000", "30000"]})
print(string_df)
print(f"Estimated size: {string_df.estimated_size('b')} bytes")
```

```
shape: (3, 1)
┌───────┐
│ id    │
│ ---   │
│ str   │
╞═══════╡
│ 10000 │
│ 20000 │
│ 30000 │
```

```
└───────┘
Estimated size: 15 bytes
```

However you know that one column only contains numeric data types, which can be stored more efficiently. Changing the data type would look like this:

```python
int_df = string_df.select(pl.col("id").cast(pl.UInt16))
print(int_df)
print(f"Estimated size: {int_df.estimated_size('b')} bytes")
```

```
shape: (3, 1)
┌───────┐
│ id    │
│ ---   │
│ u16   │
╞═══════╡
│ 10000 │
│ 20000 │
│ 30000 │
└───────┘
Estimated size: 6 bytes
```

That simple `cast` to a better fitting data type reduced the used memory immensely by over an estimated 60%! Using the optimal data types can provide a lot of performance advantages.

Table Table 2-1 shows the ranges per data type for those which it is relevant. By choosing the smallest size data type that still fits, memory usages can be optimized.

In the example above you used the cast function as an expression. You can also use it on a DataFrame or LazyFrame. In that case you can cast multiple columns at once using a single dtype to which all columns can be mapped, or a *mapping*. This mapping can be a Python dictionary describing which columns should be cast to which data type. The keys can be column names, or column selectors. Here are the ways to use the `cast()` function, starting with casting everything to one dtype:

```python
df = pl.DataFrame(
    {
        "id": [10000, 20000, 30000],
        "value": [1.0, 2.0, 3.0],
        "value2": ["1", "2", "3"],
    }
)
df.cast(pl.UInt16)
```

```
shape: (3, 3)
┌───────┬───────┬────────┐
│ id    ┆ value ┆ value2 │
│ ---   ┆ ---   ┆ ---    │
│ u16   ┆ u16   ┆ u16    │
╞═══════╪═══════╪════════╡
│ 10000 ┆ 1     ┆ 1      │
```

```
│ 20000 ┆ 2     ┆ 2      │
│ 30000 ┆ 3     ┆ 3      │
└───────┴───────┴────────┘
```

Or with a mapping, to specifically cast columns:

```
df.cast({"id": pl.UInt16, "value": pl.Float32, "value2": pl.UInt8})
```

```
shape: (3, 3)
┌───────┬───────┬────────┐
│ id    ┆ value ┆ value2 │
│ ---   ┆ ---   ┆ ---    │
│ u16   ┆ f32   ┆ u8     │
╞═══════╪═══════╪════════╡
│ 10000 ┆ 1.0   ┆ 1      │
│ 20000 ┆ 2.0   ┆ 2      │
│ 30000 ┆ 3.0   ┆ 3      │
└───────┴───────┴────────┘
```

You can also cast specific dtypes to others as follow:

```
df.cast({pl.Float64: pl.Float32, pl.String: pl.UInt8})
```

```
shape: (3, 3)
┌───────┬───────┬────────┐
│ id    ┆ value ┆ value2 │
│ ---   ┆ ---   ┆ ---    │
│ i64   ┆ f32   ┆ u8     │
╞═══════╪═══════╪════════╡
│ 10000 ┆ 1.0   ┆ 1      │
│ 20000 ┆ 2.0   ┆ 2      │
│ 30000 ┆ 3.0   ┆ 3      │
└───────┴───────┴────────┘
```

And lastly, you can use column selectors to cast columns:

```
import polars.selectors as cs
df.cast({cs.numeric(): pl.UInt16})
```

```
shape: (3, 3)
┌───────┬───────┬────────┐
│ id    ┆ value ┆ value2 │
│ ---   ┆ ---   ┆ ---    │
│ u16   ┆ u16   ┆ str    │
╞═══════╪═══════╪════════╡
│ 10000 ┆ 1     ┆ 1      │
│ 20000 ┆ 2     ┆ 2      │
│ 30000 ┆ 3     ┆ 3      │
└───────┴───────┴────────┘
```

Basic casting doesn't always magically work. In some cases special methods need to be used because data cannot be parsed without extra knowledge. One of the examples is when parsing a `DateTime` from a `String`. In Chapter 9 you'll read about methods that allow for this more advanced casting.

Conclusion

In this chapter you went over the following:

- The Arrow memory specification that Polars uses under the hood.
- The different data types Polars offers for data storage.
- Some data types offer their own special operations, such a strings, categoricals, and time-related data types. We'll dive deeper into these specifics in chapter 10.
- How missing data is handled in Polars.
- The structures Polars provides for working with that data: Series, DataFrame, and LazyFrames.
- Changing data types using `cast()`

This knowledge can be used to fill our DataFrames. In the next chapter you'll dive into the different API's Polars offers to work on this data.

CHAPTER 3

Eager and Lazy APIs

> ### A Note for Early Release Readers
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

In this chapter, we look at the two different types of Polars Application Programming Interfaces (APIs): the eager API and the lazy API. Each API addresses specific use cases and has unique performance characteristics. Understanding these APIs is critical to effectively using Polars' data processing and analysis capabilities.

The eager API uses an immediate execution model. Functions are executed sequentially, and data manipulation occurs in real time. This model is ideal for data exploration and iterative tasks, providing immediate feedback after each operation. This immediacy is similar to the user experience in Pandas, providing a smooth transition for those familiar with it.

Conversely, the lazy API defers the execution of data transformations until necessary. This deferred execution allows Polars to comprehensively optimize queries, improving performance, especially in large-scale and performance-sensitive scenarios.

Understanding the nuances of these APIs and their optimization strategies is essential to realizing the full potential of Polars for data analysis and manipulation. By the end of this chapter, you will be equipped with the knowledge to choose the right API for your needs and use it effectively in your data science projects.

Eager API: DataFrame

The eager API in Polars operates on an immediate execution model, where each function is executed sequentially, line by line, on the dataset. This approach is particularly effective for data exploration and iterative analysis, as it allows for direct interaction with the data at every step. Users execute functions on intermediate results, providing immediate feedback and insights, which is invaluable for making informed decisions about subsequent queries. This execution style is very similar to the experience offered by packages like Pandas, making it a familiar and intuitive choice for those transitioning from or accustomed to the Pandas workflow.

In this example, we'll explore the eager API of Polars through a practical application. We have a dataset of taxi trips, and our goal is to analyze the data to derive the top three vendors by revenue per distance traveled. Let's break down the process step by step to understand how the eager API facilitates this analysis. Note that we use the `%%time` cell magic to time and print how long the code execution takes.

```
import polars as pl

%%time
trips = pl.read_parquet("data/taxi/yellow_tripdata_*.parquet")  ❶
sum_per_vendor = trips.group_by("VendorID").sum()  ❷

income_per_distance_per_vendor = sum_per_vendor.select(
    "VendorID",
    income_per_distance=pl.col("total_amount") / pl.col("trip_distance")
)

top_three = (  ❸
    income_per_distance_per_vendor.sort(
        by="income_per_distance",
        descending=True
    )
    .head(3)
)
top_three
```

```
CPU times: user 9.45 s, sys: 8.7 s, total: 18.1 s
Wall time: 8.52 s
shape: (3, 2)
┌──────────┬─────────────────────┐
│ VendorID ┆ income_per_distance │
│ ---      ┆ ---                 │
│ i64      ┆ f64                 │
```

```
├────────────┼────────────────────────┤
│ 1          │ 6.434789               │
│ 6          │ 5.296493               │
│ 5          │ 4.731557               │
└────────────┴────────────────────────┘
```

❶ This reads all the Parquet files that match the glob pattern. A *glob pattern* is a string definition used to specify groups of filenames by matching patterns. We'll dive deeper into this in Chapter 4 on reading and writing data. For now, it is sufficient to know that the dataset consists of several files, which Polars reads into a DataFrame in one go. The function `read_parquet()` returns a DataFrame which is executed using the eager API.

❷ All columns are summed by VendorID, so you can calculate with total amounts.

❸ From these sums you can calculate the average income per distance traveled for all trips per vendor.

After the data is sorted, you can select the top three, answering our earlier question: "Who are the top three vendors by revenue per distance traveled?"

When doing this kind of analysis, it's often better to tackle the main problem in smaller parts. This way, you get to see the data at each step, which helps you make better choices for the next steps.

Lazy API: LazyFrame

The *lazy API* defers executing all selection, filtering, and manipulation until the moment it is actually needed. This gives the query engine more information about what data and transformations are actually needed, and allows for a bunch of optimizations that heavily increase performance. We'll talk about those next. The best uses cases for the lazy API include big and complex datasets and performance-critical applications where speed is of the essence.

Next we'll discuss some of the biggest optimizations the query planner applies to lazy queries. These make the lazy API a great choice for these use cases.

LazyFrame Scan Level Optimizations

The first group of optimizations considers data loading at the scan level. The *scan level* is the layer of execution where Polars reads data from its source. These optimizations are focused on completely avoiding reading data that won't be used.

Projection pushdown means optimizing a query by moving column selection as far upstream as possible. This prevents unused columns from being read into memory.

In this example we'll explore the same dataset as the one we just used for the eager API. We will still try to find out the top three vendors by revenue per distance traveled. However, this time we'll use the lazy API instead:

```
lf = pl.scan_parquet("data/taxi/yellow_tripdata_*.parquet")  ❶
lf.select(pl.col("trip_distance")).show_graph()  ❷
```

❶ `scan_parquet()` does not immediately read the file from disk. Instead it returns a LazyFrame for which only relevant metadata is scanned. This metadata contains information such as the schema and the number of rows and columns. The LazyFrame exposes the lazy API of Polars. The methods available are practically same, with the difference that it's only executed when you call `.collect()`.

❷ This selects only the `trip_distance` column, then prints the query plan with `show_graph()`, so you can see what happens in the query engine. You can see behind π that only 1 in 19 columns will be read into memory in the first place.

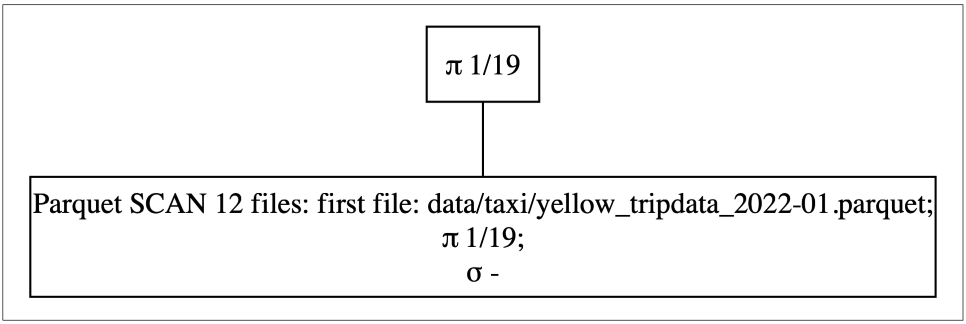


Figure 3-1. The resulting query plan when scanning the taxi dataset and selecting a single column.

The *query plan* requires some explanation:

- The first step executed is the one at the bottom, so read the graph from bottom to top.
- Every box corresponds with a stage in the query plan.
- The σ stands for SELECTION and indicates any row filter conditions.
- The π stands for PROJECTION and indicates choosing a subset of columns.

In Figure 3-1 you can see that π contains a selection of 1 out of the 19 available columns. In Figure 3-2 you can see that the σ contains a filter on the `trip_distance` column.

Moving on to the next optimization, *predicate pushdown* is like projection pushdown, but it focuses on filtering rows instead of selecting columns. This helps avoid reading rows that aren't needed.

```python
lf.filter(pl.col("trip_distance") > 10).show_graph()
```

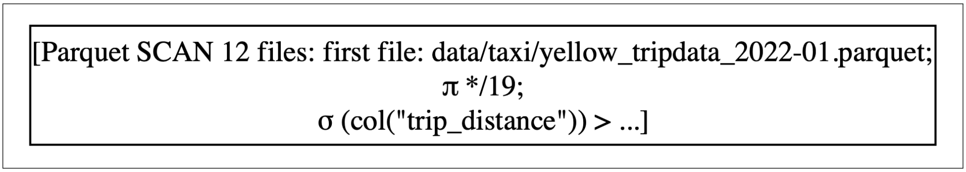


Figure 3-2. The resulting query plan when filtering values in the column trip_distance

The code above filters the `trip_distance` column for values larger than 10. In figure Figure 3-2 you can see the filter behind the σ. This filter will be applied row wise.

The last one is *slice pushdown*, which loads only the required data slice from the scan level (where the data is read into memory). Similarly to predicate pushdown, it prevents reading unused rows, but instead of reading rows based on a filter, it reads rows based on whether they belong to a certain chunk of data, using this command:

```python
lf.fetch(n_rows=2)
```

```
shape: (2, 19)
```

VendorID --- i64	tpep_pickup_datetime --- datetime[ns]	tpep_dropoff_datetim… --- datetime[ns]	…	total_amount --- f64	congestion_surcharge --- f64	airport_fee --- f64
1	2022-01-01 00:35:40	2022-01-01 00:53:29	…	21.95	2.5	0.0
1	2022-01-01 00:33:43	2022-01-01 00:42:07	…	13.3	0.0	0.0

This operation takes only the first two rows of the data at the scan level and collects the frame, which is returned as a DataFrame.

The methods `fetch(10)` and `head(10)` are similar but not the same. The `fetch(nrows: int)` method will load the first `n_rows` rows at the scan level, whereas `head(nrows: int)` is applied at the end. This means that when applying `fetch()`, any aggregations in the query plan will show wildly different results compared to a full run.

On the other hand, using `head()` runs the full calculation and only picks out the results at the end. It's best to use `fetch()` to quickly test if the query plan runs, whereas `head()` can be used to filter out the top results, as calculated with full data.

These pushdowns completely prevent the execution of later applied transformations on data that is not necessary to achieve the end result.

Other Optimizations

Other optimizations are more focused on efficient computing. For this we'll create a small LazyFrame as a running example.

```
lazy_df = pl.LazyFrame({
    "foo": [1, 2, 3, 4, 5],
    "bar": [6, 7, 8, 9, 10]
})
```

One such optimization is *common subplan elimination*. A *subplan*, or *subtree*, is a group of steps in the query plan. When certain operations or file scans are used by multiple subtrees in the query plan, the results are cached for easy reuse. For instance:

```
common_subplan = lazy_df.with_columns(pl.col("foo") * 2)

# Utilizing the common subplan in two separate expressions
expr1 = common_subplan.filter(pl.col("foo") * 2 > 4)
expr2 = common_subplan.filter(pl.col("foo") * 2 < 8)

result = pl.concat([expr1, expr2])

result.show_graph(optimized=False)
result.show_graph()
```

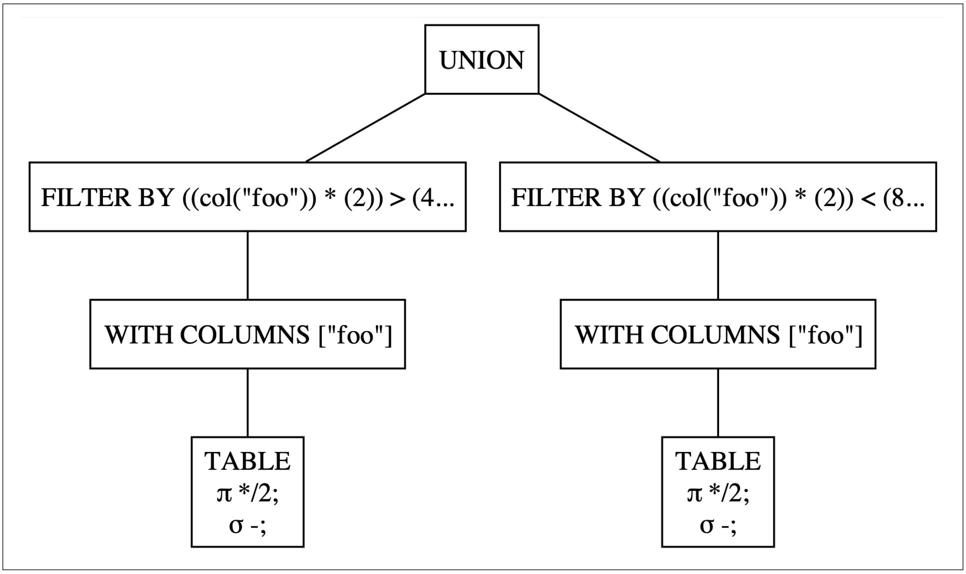


Figure 3-3. Unoptimized query plan

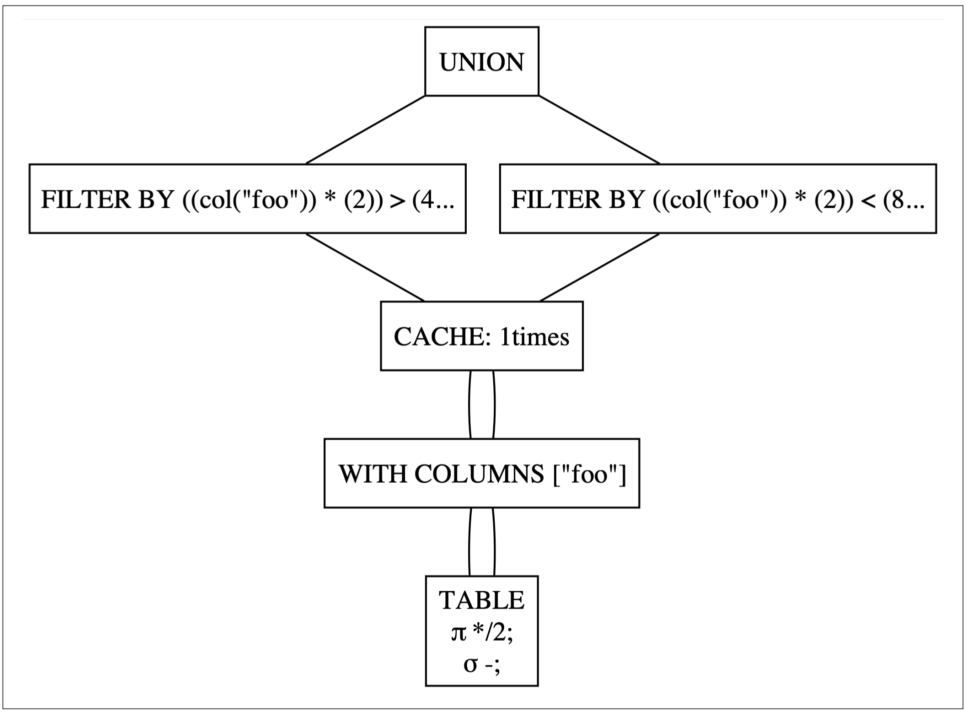


Figure 3-4. Optimized query plan

Here you combine the expressions which share the common subplan via the `pl.concat()` method. Figure 3-3 shows the query if it were not optimized. Figure 3-4 shows the optimized version, in which you can see that the file is read only once, with only the "foo" column being selected, whereas in the unoptimized variant the file is read twice. After that the different filters are applied.

> In many cases, the eager API is actually calling the lazy API under the hood and immediately collecting the result. This has the benefit that the query planner can still make optimizations within the query itself. On top of that, it's easier for the maintainers because the method of the eager API is a thin wrapper around the lazy API, deduplicating the code.

In addition, the lazy API can catch schema errors before processing the data. The query plan contains the knowledge of what needs to happen at each step along the way and what the result should look like.

Take the next example. You'll make a LazyFrame that contains names and ages of three people. If you take the `age` column, which contains the `int` dtype, and treat it as `str`, you'll immediately get a `SchemaError` before any calculation is done.

```
ldf = pl.LazyFrame({
    "name": ["Alice", "Bob", "Charlie"],
    "age": [25, 30, 35]
})

erroneous_query = ldf.with_columns(
    pl.col("age").str.slice(1,3).alias("sliced_age")
)

result_df = erroneous_query.collect()
```

```
SchemaError: invalid series dtype: expected `String`, got `i64`
```

This allows queries to fail fast and provide a short feedback loop that improves programming efficiency. If you were working on large datasets with long-running queries, it would've taken you hours to run into the error. This instant feedback can save your life!

Performance Differences

We recommend you try executing identical queries using both the lazy and eager APIs. It's a good way to see the profound optimization benefits. Let's examine the eager query we ran earlier on a dataset of taxi trip records stored in Parquet format:

```
%%time
trips = pl.scan_parquet("data/taxi/yellow_tripdata_*.parquet")
sum_per_vendor = trips.group_by("VendorID").sum()
```

```python
income_per_distance_per_vendor = sum_per_vendor.select(
    "VendorID",
    income_per_distance=pl.col("total_amount") / pl.col("trip_distance")
)
top_three = income_per_distance_per_vendor.sort(
    by="income_per_distance",
    descending=True
).head(3)
top_three.collect()
```

```
CPU times: user 2.01 s, sys: 301 ms, total: 2.31 s
Wall time: 592 ms
shape: (3, 2)
┌──────────┬─────────────────────┐
│ VendorID ┆ income_per_distance │
│ ---      ┆ ---                 │
│ i64      ┆ f64                 │
╞══════════╪═════════════════════╡
│ 1        ┆ 6.434789            │
│ 6        ┆ 5.296493            │
│ 5        ┆ 4.731557            │
└──────────┴─────────────────────┘
```

This returns the same DataFrame, but the lazy API does it about 10 times faster as the eager API! Now that's what we call blazingly fast.

In Polars, a LazyFrame is evaluated and converted into a DataFrame only when you invoke the `collect()` method. While this lazy evaluation offers efficiency gains, it's crucial to note that subsequent calls to `collect()` will recompute the LazyFrame from scratch. This means the same calculations will be run multiple times, which you want to prevent.

We'll make a small LazyFrame with two columns of three rows and act like it's a very big dataset with long calculation times.

```python
lf = pl.LazyFrame({"col1": [1,2,3], "col2": [4,5,6]})
# Some heavy computation
print(lf.collect())
print(lf.with_columns(pl.col("col1") + 1).collect())  # Recalculates the LazyFrame
```

```
shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
│ ---  ┆ ---  │
│ i64  ┆ i64  │
╞══════╪══════╡
│ 1    ┆ 4    │
│ 2    ┆ 5    │
│ 3    ┆ 6    │
└──────┴──────┘
shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
```

```
│ ---  ┆ ---  │
│ i64  ┆ i64  │
╞══════╪══════╡
│ 2    ┆ 4    │
│ 3    ┆ 5    │
│ 4    ┆ 6    │
└──────┴──────┘
```

Functionality Differences

The big difference between a LazyFrame and a DataFrame is that, in a LazyFrame, the data is not available until it's collected. This means certain functionalities will not be available. We'll go through the different types of operations in the next section and point out the differences.

Aggregations

All the *aggregations* (such as getting the mean, min and max values of a column) that can be applied to a DataFrame can also be applied to a LazyFrame. These operations don't require the query planner to have knowledge about the data up front, and will be added to the query plan to be executed upon data collection. The set of methods available to only the DataFrame are *horizontal aggregations* as shown in Table 3-1. Horizontal aggregations are operations that are applied row-wise across columns, such as `sum_horizontal()`.

Table 3-1. Aggregation methods of DataFrames vs LazyFrames

Method	DataFrame	LazyFrame
`.max()`	✓	✓
`.max_horizontal()`	✓	
`.mean()`	✓	✓
`.mean_horizontal(…)`	✓	
`.median()`	✓	✓
`.min()`	✓	✓
`.min_horizontal()`	✓	
`.null_count()`	✓	✓
`.product()`	✓	
`.quantile(…)`	✓	✓
`.std(…)`	✓	✓
`.sum()`	✓	✓
`.sum_horizontal(…)`	✓	
`.var(…)`	✓	✓

Attributes

Of all the *attributes* that are available to a DataFrame, the LazyFrame lacks `shape`, `height`, and `flags` as shown in Table 3-2. The first two describe the number of columns and rows of the DataFrame has, which can only be given once the data is available. `flags` is a dictionary containing indicators like whether a column is sorted, which is used internally for optimizations.

Table 3-2. Attributes of DataFrames vs LazyFrames

Attribute	DataFrame	LazyFrame
`.columns`	✓	✓
`.dtypes`	✓	✓
`.flags`	✓	
`.height`	✓	
`.schema`	✓	✓
`.shape`	✓	
`.width`	✓	✓

Computation

DataFrames have the *computation* methods `fold()` and `hash_rows()` where a Lazy‐Frame doesn't have computation methods at all. Both of these computations are row-wise reductions. `fold()` allows you to provide a function that reduces two Series to one, where `hash_rows()` just hashes all the information on a row to a `UInt64` value.

Descriptive

The only *descriptive* methods a LazyFrame has are `explain()` and `show_graph()`, to showcase the query plan as shown in Table 3-3. A DataFrame has a lot of methods to showcase specifics about the data, such as `describe()` and `estimated_size()`.

Table 3-3. Descriptive methods of DataFrames vs LazyFrames

Method	DataFrame	LazyFrame
`.approx_n_unique()`	✓	
`.describe(…)`	✓	
`.estimated_size(…)`	✓	
`.explain(…)`		✓
`.glimpse()`	✓	
`.is_duplicated()`	✓	

Method	DataFrame	LazyFrame
`.is_empty()`	✓	
`.is_unique()`	✓	
`.n_chunks()`	✓	
`.n_unique(…)`	✓	
`.show_graph(…)`		✓

GroupBy

All the methods you can apply to a *group* in the GroupBy context are the same in both, except that a DataFrame lets you iterate over the groups as shown in Table 3-4.

Table 3-4. GroupBy methods of DataFrames vs LazyFrames

Method	DataFrame	LazyFrame
`.__iter__()`	✓	
`.agg(…)`	✓	✓
`.all()`	✓	✓
`.apply(…)`	✓	✓
`.count()`	✓	✓
`.first()`	✓	✓
`.head(…)`	✓	✓
`.last()`	✓	✓
`.map_groups(…)`	✓	✓
`.max()`	✓	✓
`.mean()`	✓	✓
`.median()`	✓	✓
`.min()`	✓	✓
`.n_unique()`	✓	✓
`.quantile(…)`	✓	✓
`.sum()`	✓	✓
`.tail(…)`	✓	✓

Exporting

A DataFrame has several options of exporting the data to different formats. Formats include Arrow, Numpy, Pandas, dictionaries, a Series containing structs, and even a string containing the Python code required to initialize the DataFrame! Since a LazyFrame doesn't have any data, there's no possibility for exports.

Manipulation and Selection

The *manipulation* and *selection* methods are the most important ones. They contain the core functionality of data manipulation. Table 3-5 shows the many differences between the two APIs.

Table 3-5. Manipulation methods of DataFrames vs LazyFrames

Method	DataFrame	LazyFrame
`.approx_n_unique()`		✓
`.bottom_k(…)`	✓	✓
`.cast(…)`	✓	✓
`.clear(…)`	✓	✓
`.clone()`	✓	✓
`.drop(…)`	✓	✓
`.drop_in_place(…)`	✓	
`.drop_nulls(…)`	✓	✓
`.explode(…)`	✓	✓
`.extend(…)`	✓	
`.fill_nan(…)`	✓	✓
`.fill_null(…)`	✓	✓
`.filter(…)`	✓	✓
`.find_idx_by_name(…)`	✓	
`.first()`		✓
`.gather_every(…)`	✓	✓
`.get_column(…)`	✓	
`.get_column_index(…)`	✓	
`.get_columns()`	✓	
`.group_by(…)`	✓	✓
`.group_by_dynamic(…)`	✓	✓
`.group_by_rolling(…)`	✓	✓
`.head(…)`	✓	✓
`.hstack(…)`	✓	
`.insert_at_idx(…)`	✓	
`.insert_column(…)`	✓	
`.inspect(…)`		✓
`.interpolate()`	✓	✓
`.item(…)`	✓	
`.iter_columns()`	✓	

Method	DataFrame	LazyFrame
.iter_rows()	✓	
.iter_slices(…)	✓	
.join(…)	✓	✓
.join_asof(…)	✓	✓
.last()		✓
.limit(…)	✓	✓
.melt(…)	✓	✓
.merge_sorted(…)	✓	✓
.partition_by()	✓	
.pipe(…)	✓	
.pivot(…)	✓	
.rechunk()	✓	
.rename(…)	✓	✓
.replace(…)	✓	
.replace_at_idx(…)	✓	
.replace_column(…)	✓	
.reverse()	✓	✓
.rolling(…)	✓	✓
.row()	✓	
.rows()	✓	
.rows_by_key(…)	✓	
.sample(…)	✓	
.select(…)	✓	✓
.select_seq(…)	✓	✓
.set_sorted(…)	✓	✓
.shift(…)	✓	✓
.shift_and_fill(…)	✓	✓
.shrink_to_fit(…)	✓	
.slice(…)	✓	✓
.sort(…)	✓	✓
.tail(…)	✓	✓
.take_every(…)	✓	✓
.top_k(…)	✓	✓
.to_dummies(…)	✓	
.to_series(…)	✓	
.transpose(…)	✓	

Method	DataFrame	LazyFrame
`.unique(…)`	✓	✓
`.unnest(…)`	✓	✓
`.unstack(…)`	✓	
`.update(…)`	✓	✓
`.upsample(…)`	✓	
`.vstack(…)`	✓	
`.with_columns(…)`	✓	✓
`.with_columns_seq(…)`	✓	✓
`.with_context(…)`		✓
`.with_row_count(…)`	✓	✓

Miscellaneous

The *miscellaneous* methods are the ones that don't fit in any of the other categories, shown in Table 3-6.

Table 3-6. Miscellaneous methods of DataFrames vs LazyFrames

Method	DataFrame	LazyFrame
`.cache()`		✓
`.collect(…)`		✓
`.collect_async()`		✓
`.corr(…)`	✓	
`.equals(…)`	✓	
`.fetch(…)`		✓
`.frame_equal(…)`	✓	
`.lazy()`	✓	✓
`.map(…)`		✓
`.map_batches(…)`		✓
`.map_rows(…)`	✓	
`.pipe(…)`		✓
`.profile(…)`		✓

Out-of-Core Computation with Lazy API's Streaming Mode

The lazy API offers a special mode to do computations *out-of-core*: that is processing data that would be too large to fit into RAM by doing the calculations on *chunks* of data instead. The amazing thing about supporting out-of-core computation is that it moves the barrier for processing data from the size of your RAM to the size of your hard disk, which can be a difference of orders of magnitude! You can trigger this mode by passing `streaming=True` to the `collect()` function to collect the end result to RAM, or you can write the results to disk using `.sink_csv(…)`, `.sink_ipc(…)` or `.sink_parquet(…)`. If you use `.collect(streaming=True)`, the end result must fit in RAM.

In streaming mode, the API reads the data in chunks of rows. This chunk size is determined based on the number of threads available to perform the work in and the number of columns in the dataset.

How many threads are available on your system? To find out, you need the number of logical CPU cores available on your machine by default (or the container you are working in). By running the following code you can find this number:

```
pl.thread_pool_size()
```

```
12
```

Although this number generally works out of the box, there is an option to add or reduce the number of threads through the environment variables. This could be useful if other CPU-intensive tasks are running at the same time and the system needs some breathing room to prevent time-outs in other processes. You must set this environment variable before importing Polars, using the following code. For example:

```
import os
os.environ["POLARS_MAX_THREADS"] = "2"
import polars as pl
```

Understanding Python Environment Variables

Example 3-1.

Environment variables in Python are key-value pairs that can be set for and read from the runtime environment. They can be particularly useful for several reasons:

1. **Security**: They provide a secure way to store sensitive information like database credentials and API keys, keeping them out of your source code.
2. **Configuration**: Environment variables allow you to change the behavior of your Python application without altering the code. For example, you can set variables to differentiate between development and production environments.

3. **Portability**: By using environment variables, you can easily migrate your application across different environments (local, staging, production) without code changes.

In Python, you can access environment variables using the `os` module, specifically `os.environ`. This acts like a dictionary, where you can retrieve values using their keys. For example, `os.environ['POLARS_MAX_THREADS']` would give you the number of threads Polars is allowed to use.

To determine chunk size works use the following formula:

$$\texttt{thread\_factor} = \max\left\{\frac{12}{\texttt{n\_threads}}, 1\right\}$$

$$\texttt{chunk\_size} = \max\left\{\frac{50000}{\texttt{n\_cols} * \texttt{thread\_factor}}, 1000\right\}$$

Let's split that up:

The `thread_factor` will be 1 if you have 12 or more threads, and will be greater than 1 if you have fewer threads. This means the `thread_factor` goes down the more threads are available.

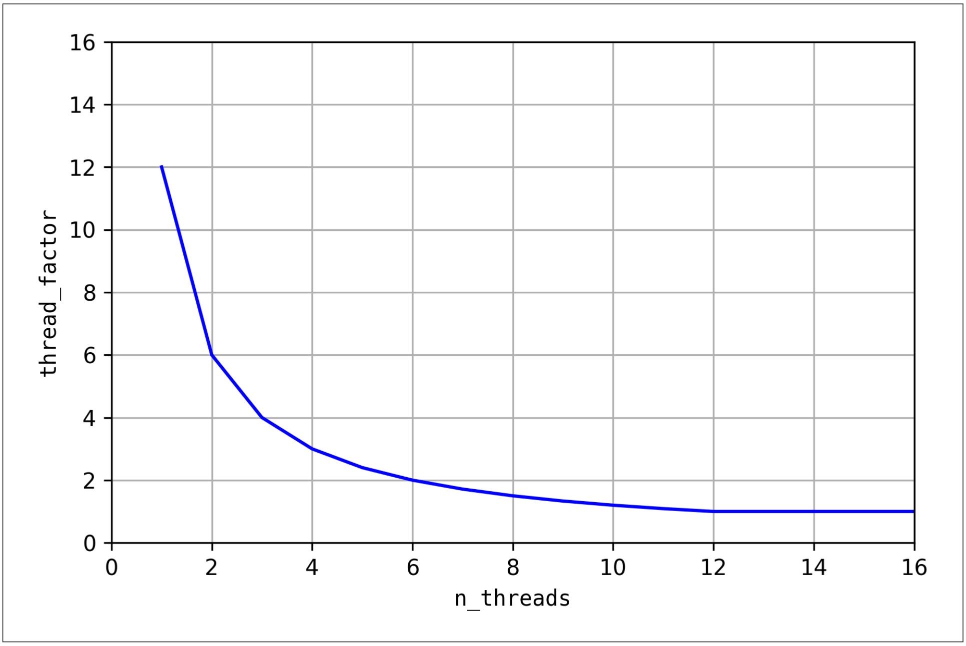


Figure 3-5. Thread factor

This code sets the chunk size to the maximum value, 1000, or `(50000 / n_cols * thread_factor)`.

The chunk size goes up if the thread factor goes up, which it does when fewer threads are available. This means that if there are more threads available, the chunk size will shrink. The idea is to process more chunks of data at the same time, using more RAM.

If there are more columns in the dataset, the chunk size also goes down, because each row contains more data (and thus uses more RAM).

However, it is possible to overwrite the streaming chunk size. This can be necessary if the chunk size Polars determines by default still causes memory issues. You can do this with the following config setting:

```
pl.Config.set_streaming_chunk_size(1000)
```

Tips and Tricks

In the next section we'll cover some tips and tricks. Most of these will be very practical in your day-to-day usage of Polars. This is typically the kind of information that you won't find in the documentation, but that can make your life a lot easier.

Going from LazyFrame to DataFrame and Vice Versa

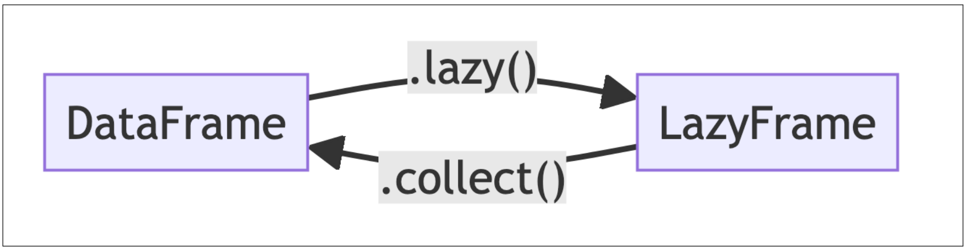


Figure 3-6. The operations to swap to the other API

You can swap from one API to the other with a single command, as shown in Figure 3-6.

You can go from the eager API to the lazy API by adding `.lazy()` behind a DataFrame, or methods returning a DataFrame. This results in no computation, but tells the query planner to use the data in memory as a starting point for a new query plan.

You can go from the lazy to eager API by calling `.collect()` on a LazyFrame, or a function returning a LazyFrame. This executes the query plan built for that LazyFrame, triggering computation. Afterwards, the result will be stored in RAM.

If you're using streaming mode and not calling `.collect()`, but calling `.sink_parquet()` instead, the result is written to disk.

Joining a DataFrame and a LazyFrame

When you perform joins in Polars, the data structures involved *must be of the same type*. Specifically, you cannot directly join a DataFrame with a LazyFrame. You might want to do this if for example you've got a small DataFrame with metadata that you want to join to a large dataset that you've got in a LazyFrame.

Here's a snippet that would result in an error:

```
lf = pl.LazyFrame({"id": [1,2,3], "value1": [4,5,6]})
df = pl.DataFrame({"id": [1,2,3], "value2": [7,8,9]})

lf.join(df, on="id")
```

```
TypeError: expected `other` join table to be a LazyFrame, not a 'DataFrame'
```

Fortunately, resolving this is straightforward. You can either make the DataFrame lazy by appending `.lazy()`, or materialize the LazyFrame using `.collect()`. We advise sticking with the lazy API for better performance and efficiency.

Here's how to successfully perform the join by making the DataFrame lazy:

```
lf = pl.LazyFrame({"id": [1,2,3], "value1": [4,5,6]})
df = pl.DataFrame({"id": [1,2,3], "value2": [7,8,9]})

lf.join(df.lazy(), on="id")
```

```
<LazyFrame [3 cols, {"id": Int64 … "value2": Int64}] at 0x284228A90>
```

Where in the first output we got a `TypeError`, we now get a valid LazyFrame!

Caching Intermittent Stages

To avoid unnecessarily recomputing the frame, you can cache the LazyFrame in memory by chaining `.collect().lazy()` after the heavy computation. This will evaluate the LazyFrame, keep it in memory, and return a new LazyFrame pointing to the materialized data stored in RAM.

Here's how you can optimize the above example:

```
lf = pl.LazyFrame({"col1": [1,2,3], "col2": [4,5,6]})
# Some heavy computation
lf = lf.collect().lazy()
print(lf.collect())
print(lf.with_columns(pl.col("col1") + 1).collect())  # Utilizes the cached LazyFrame
```

```
shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
```

```
│ ---  ┆ ---  │
│ i64  ┆ i64  │
╞══════╪══════╡
│ 1    ┆ 4    │
│ 2    ┆ 5    │
│ 3    ┆ 6    │
└──────┴──────┘
shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
│ ---  ┆ ---  │
│ i64  ┆ i64  │
╞══════╪══════╡
│ 2    ┆ 4    │
│ 3    ┆ 5    │
│ 4    ┆ 6    │
└──────┴──────┘
```

This pattern can be a lifesaver when dealing with resource-intensive computations, as it enables you to leverage the benefits of lazy evaluation while mitigating its computational drawbacks.

Conclusion

In this chapter we've covered eager and lazy APIs in Polars. Among other things, you learned about:

- The eager API and its representation in Polars as DataFrames.
- The lazy API and its representation in Polars as LazyFrames.
- The best use cases for each API.
- The optimizations possible in the lazy API.
- The functionality differences between the eager and lazy APIs.
- The lazy API streaming mode which lets you calculate out-of-core with larger-than-RAM datasets.
- Some practical tips, like how use caching to avoid calculating the same LazyFrame multiple times, and how to join DataFrames and LazyFrames.

With this knowledge, you can determine which is the perfect API for your use case. Now it's time to learn to load data from files into the structures we've talked about in this chapter. The next chapter is about reading and writing data to and from different file formats.

CHAPTER 4

Reading and Writing Data

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

Now that you've seen some essential concepts such as data types and the different APIs, you're ready to learn about working with external data sources. That includes reading data from files and databases into Polars. We'll also cover how to write your results to files and databases. By the end of this chapter, you'll be able to start working with your own data. We encourage you to start using your own data as soon as possible, because it will make learning about Polars not only more enjoyable but also more effective.

Because external data can come in all sorts of ways from all sorts of places, Polars has over 30 functions related to reading data, and those functions accept many arguments. It would be challenging and, more importantly, extremely boring to cover every function and every argument in this chapter. That's what the official API documentation is for. Instead, we will focus on the formats and situations that you're most likely to encounter.

In this chapter, you'll learn how to:

- Read and write data in many formats, including CSV, Excel, and Parquet
- Handle multiple files efficiently using globbing
- Correctly read missing values
- Deal with character encodings
- Read data eagerly and lazily

We're using a couple of additional packages:

- `xlsx2csv` to read Excel spreadsheets
- `chardet` to determine the character encoding of a file
- `connectorx` to connect to databases
- `pyarrow` to read PyArrow datasets

Chapter 2 has instructions for how to install these packages.

In order to demonstrate working with various data formats, this chapter uses a lot of datasets. The instructions to get the corresponding files are in Chapter 2. We assume that you have the files in the *data* subdirectory.

As usual, we start by importing Polars:

```
import polars as pl
```

Reading CSV Files

We'll start with comma-separated values (CSV), the file format that is perhaps most prevalent in programming, data analysis, and scientific research. Despite its prevalence, it's not without its flaws. When you're handed a file with the extension *.csv*, there's no knowing what's inside:

- Is the delimiter a comma, a tab, a semicolon, or something else?
- Is the character encoding UTF-8, ASCII, or something else?
- Is there a header with column names? How many lines is it?
- How are missing values represented?
- Are values properly quoted?

Polars can handle all these situations, but there might be some trial and error involved.

Imagine, for a moment, that we have a straightforward CSV file such as *data/penguins.csv*. Before we immediately start loading this data into Polars, let's have a look at the raw contents of the file using the command-line tool `cat` (Note that the output is truncated):

```
$ cat data/penguins.csv
```

```
"rowid","species","island","bill_length_mm","bill_depth_mm","flipper_length_mm"…
"1","Adelie","Torgersen",39.1,18.7,181,3750,"male",2007
"2","Adelie","Torgersen",39.5,17.4,186,3800,"female",2007
"3","Adelie","Torgersen",40.3,18,195,3250,"female",2007
"4","Adelie","Torgersen",NA,NA,NA,NA,NA,2007
… with 340 more lines
```

At first glance, this CSV file appears to be straightforward indeed. The first line is a header and the delimiter is a comma, which matches Polars' defaults. Moreover, the character encoding is compatible with UTF-8. (More on this later.) This makes us feel confident enough to read the dataset into a Polars DataFrame:

```python
penguins = pl.read_csv("data/penguins.csv")
penguins
```

```
shape: (344, 9)
┌───────┬───────────┬───────────┬───┬─────────────┬────────┬──────┐
│ rowid ┆ species   ┆ island    ┆ … ┆ body_mass_g ┆ sex    ┆ year │
│ ---   ┆ ---       ┆ ---       ┆   ┆ ---         ┆ ---    ┆ ---  │
│ i64   ┆ str       ┆ str       ┆   ┆ str         ┆ str    ┆ i64  │
╞═══════╪═══════════╪═══════════╪═══╪═════════════╪════════╪══════╡
│ 1     ┆ Adelie    ┆ Torgersen ┆ … ┆ 3750        ┆ male   ┆ 2007 │
│ 2     ┆ Adelie    ┆ Torgersen ┆ … ┆ 3800        ┆ female ┆ 2007 │
│ 3     ┆ Adelie    ┆ Torgersen ┆ … ┆ 3250        ┆ female ┆ 2007 │
│ 4     ┆ Adelie    ┆ Torgersen ┆ … ┆ NA          ┆ NA     ┆ 2007 │
│ 5     ┆ Adelie    ┆ Torgersen ┆ … ┆ 3450        ┆ female ┆ 2007 │
│ …     ┆ …         ┆ …         ┆ … ┆ …           ┆ …      ┆ …    │
│ 340   ┆ Chinstrap ┆ Dream     ┆ … ┆ 4000        ┆ male   ┆ 2009 │
│ 341   ┆ Chinstrap ┆ Dream     ┆ … ┆ 3400        ┆ female ┆ 2009 │
│ 342   ┆ Chinstrap ┆ Dream     ┆ … ┆ 3775        ┆ male   ┆ 2009 │
│ 343   ┆ Chinstrap ┆ Dream     ┆ … ┆ 4100        ┆ male   ┆ 2009 │
│ 344   ┆ Chinstrap ┆ Dream     ┆ … ┆ 3775        ┆ female ┆ 2009 │
└───────┴───────────┴───────────┴───┴─────────────┴────────┴──────┘
```

It looks like this CSV file has been read correctly, except for one thing: "NA" values are not interpreted as missing values. We'll fix that in the next section.

If your CSV file is different then perhaps the arguments listed in Table 4-1 can help.

Table 4-1. Common arguments for the function `pl.read_csv()`

Argument	Description
`source`	Path to a file or a file-like object.
`has_header`	Indicate if the first row of dataset is a header or not.
`columns`	Columns to select. Accepts a list of column indices (starting at zero) or a list of column names.
`separator`	Single byte character to use as delimiter in the file.

Argument	Description
`skip_rows`	Start reading after a certain number of lines.
`null_values`	Values to interpret as null values.
`encoding`	Default: `utf8`. `utf8-lossy` means that invalid UTF-8 values are replaced with � characters. When using other encodings than `utf8` or `utf8-lossy`, the input is first decoded in memory with Python.

Parsing Missing Values Correctly

It's quite common for a dataset to have missing values. Unfortunately for plain-text formats such as CSV, there's no standard way to represent these. Representations that we've seen in the wild include NULL, Nil, None, NA, N/A, NaN, 999999, and the empty string.

By default, Polars only interprets empty strings as missing values. Any other representations need to be passed explicitly as a string (or a list of strings) to the `null_values` argument. So let's fix those missing values in *data/penguins.csv*:

```
penguins = pl.read_csv("data/penguins.csv", null_values="NA")
penguins
```

```
shape: (344, 9)
┌───────┬───────────┬───────────┬───┬─────────────┬────────┬──────┐
│ rowid ┆ species   ┆ island    ┆ … ┆ body_mass_g ┆ sex    ┆ year │
│ ---   ┆ ---       ┆ ---       ┆   ┆ ---         ┆ ---    ┆ ---  │
│ i64   ┆ str       ┆ str       ┆   ┆ i64         ┆ str    ┆ i64  │
╞═══════╪═══════════╪═══════════╪═══╪═════════════╪════════╪══════╡
│ 1     ┆ Adelie    ┆ Torgersen ┆ … ┆ 3750        ┆ male   ┆ 2007 │
│ 2     ┆ Adelie    ┆ Torgersen ┆ … ┆ 3800        ┆ female ┆ 2007 │
│ 3     ┆ Adelie    ┆ Torgersen ┆ … ┆ 3250        ┆ female ┆ 2007 │
│ 4     ┆ Adelie    ┆ Torgersen ┆ … ┆ null        ┆ null   ┆ 2007 │
│ 5     ┆ Adelie    ┆ Torgersen ┆ … ┆ 3450        ┆ female ┆ 2007 │
│ …     ┆ …         ┆ …         ┆ … ┆ …           ┆ …      ┆ …    │
│ 340   ┆ Chinstrap ┆ Dream     ┆ … ┆ 4000        ┆ male   ┆ 2009 │
│ 341   ┆ Chinstrap ┆ Dream     ┆ … ┆ 3400        ┆ female ┆ 2009 │
│ 342   ┆ Chinstrap ┆ Dream     ┆ … ┆ 3775        ┆ male   ┆ 2009 │
│ 343   ┆ Chinstrap ┆ Dream     ┆ … ┆ 4100        ┆ male   ┆ 2009 │
│ 344   ┆ Chinstrap ┆ Dream     ┆ … ┆ 3775        ┆ female ┆ 2009 │
└───────┴───────────┴───────────┴───┴─────────────┴────────┴──────┘
```

When DataFrames are rendered in ASCII, such as in this book, all strings are displayed without quotes. That means you won't be able to check visually whether missing values are interpreted correctly.

When you're using Jupyter Notebook, you'll get an HTML rendering of a DataFrame. Here, missing values are displayed as "null" without quotes, whereas regular strings are displayed with quotes.

If you're not sure whether all missing values have been parsed correctly, you can count them programmatically using the `null_count()` method:

```
(
    penguins
    .null_count()
    .transpose(include_header=True, column_names=["null_count"]) ❶
)
```

```
shape: (9, 2)
┌───────────────────┬────────────┐
│ column            ┆ null_count │
│ ---               ┆ ---        │
│ str               ┆ u32        │
╞═══════════════════╪════════════╡
│ rowid             ┆ 0          │
│ species           ┆ 0          │
│ island            ┆ 0          │
│ bill_length_mm    ┆ 2          │
│ bill_depth_mm     ┆ 2          │
│ flipper_length_mm ┆ 2          │
│ body_mass_g       ┆ 2          │
│ sex               ┆ 11         │
│ year              ┆ 0          │
└───────────────────┴────────────┘
```

❶ We transpose the output to get a better overview of all the counts.

Reading Files with Encodings Other than UTF-8

Every text file has a certain *character encoding*. A character encoding is a system that assigns unique codes to individual characters in a set, allowing them to be represented and processed by computers.

Polars assumes that the CSV file is encoded in UTF-8, which is a widely used encoding. UTF-8 can represent any character in the Unicode standard, which includes a vast range of characters from a multitude of languages, both modern and historic, as well as a wide array of symbols.

If you try to read a CSV file with a different encoding than UTF-8, you'll ideally[1] get an error, just like we get here with *data/directors.csv*:

```
pl.read_csv("data/directors.csv")
```

```
ComputeError: could not parse `����` as dtype `str` at column 'name' (column num
ber 1)

The current offset in the file is 19 bytes.
```

1 We say "ideally", because then it's clear that you've not specified the correct encoding.

```
You might want to try:
- increasing `infer_schema_length` (e.g. `infer_schema_length=10000`),
- specifying correct dtype with the `dtypes` argument
- setting `ignore_errors` to `True`,
- adding `����` to the `null_values` list.

Original error: ```invalid utf-8 sequence```
```

Apparently *data/directors.csv* is not encoded in UTF-8.

If you start guessing the encoding, you could end up using one that doesn't upset Polars, but the bytes in your file could still get interpreted incorrectly. If you're not familiar with the language, then it's difficult to spot something's off.

Now let's imagine you're told that your file contains the names of directors, including some Asian names. Your best guess is to try an encoding common for Chinese characters:

```python
pl.read_csv("data/directors.csv", encoding="EUC-CN")
```

```
shape: (4, 3)
┌───────────┬──────┬──────────┐
│ name      ┆ born ┆ country  │
│ ---       ┆ ---  ┆ ---      │
│ str       ┆ i64  ┆ str      │
╞═══════════╪══════╪══════════╡
│ 考侯      ┆ 1930 ┆ 泣塑     │
│ Verhoeven ┆ 1938 ┆ オランダ │
│ 弟宏      ┆ 1942 ┆ 泣塑     │
│ Tarantino ┆ 1963 ┆ 势柜     │
└───────────┴──────┴──────────┘
```

That worked. Or did it? When you verify this by translating (using, for example, your favorite search engine) the first country from Chinese to English, it says "Weeping plastic." What? That's no country we've ever heard of!

Instead of guessing the encoding, it's better to let the `chardet` package detect it. The function below returns the encoding for a given filename. Let's apply this function to our CSV file:

```python
import chardet

def detect_encoding(filename: str) -> str:
    """Return the most probable character encoding for a file."""

    with open(filename, "rb") as f:
        raw_data = f.read()
        result = chardet.detect(raw_data)
        return result["encoding"]


detect_encoding("data/directors.csv")
```

```
'EUC-JP'
```

So `chardet` detected a different encoding—one that's often used for Japanese characters. Let's try the "EUC-JP" encoding with Polars:

```
pl.read_csv("data/directors.csv", encoding="EUC-JP")
```

```
shape: (4, 3)
┌────────────┬──────┬──────────┐
│ name       ┆ born ┆ country  │
│ ---        ┆ ---  ┆ ---      │
│ str        ┆ i64  ┆ str      │
╞════════════╪══════╪══════════╡
│ 深作       ┆ 1930 ┆ 日本     │
│ Verhoeven  ┆ 1938 ┆ オランダ │
│ 宮崎       ┆ 1942 ┆ 日本     │
│ Tarantino  ┆ 1963 ┆ 米国     │
└────────────┴──────┴──────────┘
```

Now this is correct. Trust us, we checked it.

Conclusion: you'd better not guess the encoding of a file. This holds not just for CSV files, but for all text-based files, including JSON, XML, and HTML.

Reading Excel Spreadsheets

While CSV is common in data-heavy, programmatic, and analytical contexts, Excel spreadsheets are common in business contexts, which often involve manual data inspection, data entry, and basic analyses.

They can contain complex data, markup, formulas, and charts. Although useful for business applications, these features can hamper reading the spreadsheet into Polars. Ideally, the spreadsheet would only contain data in a rectangular shape, just like a CSV file.

To read Excel spreadsheets into a DataFrame, Polars uses the `xlsx2csv` package. (Instructions on how to install this package can be found in Chapter 2.) Let's read *data/top-2000-2023.xlsx*, which is a spreadsheet from Top2000, an annual Dutch radio program. It contains the 2,000 most popular songs as voted by the station's listeners in 2023.

```
songs = pl.read_excel("data/top2000-2023.xlsx") ❶
songs
```

```
shape: (2_001, 4)
| positie | titel                   | artiest         | jaar |
| ---     | ---                     | ---             | ---  |
| i64     | str                     | str             | i64  |
|---------|-------------------------|-----------------|------|
| null    | null                    | null            | null |
| 1       | Bohemian Rhapsody       | Queen           | 1975 |
```

```
| 2        | Roller Coaster       | Danny Vera     | 2019 |
| 3        | Hotel California     | Eagles         | 1977 |
| 4        | Piano Man            | Billy Joel     | 1974 |
| …        | …                    | …              | …    |
| 1996     | Charlie Brown        | Coldplay       | 2011 |
| 1997     | Beast Of Burden      | Bette Midler   | 1984 |
| 1998     | It Was A Very Good Y… | Frank Sinatra | 1968 |
| 1999     | Hou Van Mij          | 3JS            | 2008 |
| 2000     | Drivers License      | Olivia Rodrigo | 2021 |
```

❶ The Dutch column names translate to position, title, artist, and year. (Fun fact: Dutch is, after Frysian, the closest relative of English.)

Our spreadsheet has only one flaw: the header spans two rows. (Note that the first row only contains missing values.) This can be fixed as follows:

```
songs_fixed = pl.read_excel(
    "data/top2000-2023.xlsx", read_options={"skip_rows_after_header": 1}
)
songs_fixed
```

```
shape: (2_000, 4)
┌─────────┬──────────────────────┬────────────────┬──────┐
│ positie ┆ titel                ┆ artiest        ┆ jaar │
│ ---     ┆ ---                  ┆ ---            ┆ ---  │
│ i64     ┆ str                  ┆ str            ┆ i64  │
╞═════════╪══════════════════════╪════════════════╪══════╡
│ 1       ┆ Bohemian Rhapsody    ┆ Queen          ┆ 1975 │
│ 2       ┆ Roller Coaster       ┆ Danny Vera     ┆ 2019 │
│ 3       ┆ Hotel California     ┆ Eagles         ┆ 1977 │
│ 4       ┆ Piano Man            ┆ Billy Joel     ┆ 1974 │
│ 5       ┆ Fix You              ┆ Coldplay       ┆ 2005 │
│ …       ┆ …                    ┆ …              ┆ …    │
│ 1996    ┆ Charlie Brown        ┆ Coldplay       ┆ 2011 │
│ 1997    ┆ Beast Of Burden      ┆ Bette Midler   ┆ 1984 │
│ 1998    ┆ It Was A Very Good Y… ┆ Frank Sinatra ┆ 1968 │
│ 1999    ┆ Hou Van Mij          ┆ 3JS            ┆ 2008 │
│ 2000    ┆ Drivers License      ┆ Olivia Rodrigo ┆ 2021 │
└─────────┴──────────────────────┴────────────────┴──────┘
```

The additional argument that we pass to `pl.read_excel()` is a dictionary of arguments that will be passed on to the `pl.read_csv()`. That's because, under the hood, the Excel spreadsheet is first converted to a CSV file. Table 4-2 lists some other commonly used arguments.

Table 4-2. Common arguments for the function `pl.read_excel()`

Argument	Description
`source`	Path to a file or a file-like object.
`sheet_id`	Sheet number to convert (0 for all sheets). Defaults to 1 if neither this nor sheet_name are specified.

Argument	Description
`sheet_name`	Sheet name to convert. Cannot be used in conjunction with `sheet_id`.
`xlsx2csv_options`	Extra options passed to `xlsx2csv.Xlsx2csv()`. e.g.: `{"skip_empty_lines": True}`
`read_csv_options`	Extra options passed to `pl.read_csv()` for parsing the CSV file returned by `xlsx2csv.Xlsx2csv().convert()`

Polars only supports Excel spreadsheets with the *.xlsx* extension. If you find that `pl.read_excel()` doesn't work with your spreadsheet files, we recommend you try the Pandas function `pd.read_excel()`. Besides *.xlsx*, this function supports *.xls*, *.xlsm*, *.xlsb*, *.odf*, *.ods*, and *.odt*. Later in this chapter, in "Other File Formats" on page 73, we'll explain how to convert a Pandas DataFrame into a Polars DataFrame.

Working with Multiple Files

If your data is spread across multiple files and those files all have the same format and schema, you might be able to read them all at once.

For instance, let's consider daily stock information for three companies: ASML Holding N.V. (ASML), NVIDIA Corporation (NVDA), and Taiwan Semiconductor Manufacturing Company Limited (TSM). The data is split across multiple CSV files, such that we have one file per company per year. The files are named according to the pattern `data/stock/`\`_<symbol>_/_<year>_.csv\`. For example: *data/stock/nvda/2010.csv* and *data/stock/asml/2022.csv*.

Because these files have the same format and schema, we can use a *globbing pattern*. Globbing patterns can contain special characters, such as asterisks (`*`), question marks (`?`), or square brackets (`[]`), which act as wildcards. An asterisk matches zero or more characters in a string, while a question mark matches exactly one character. For example, the pattern `*.csv` will match any filename that ends in *.csv*, and the pattern `file?.csv` will match files like *file1.csv* or *fileA.csv* but not *file12.csv*. To match one character of a certain set or a range, you can use square brackets. For example, `file-[ab].csv` matches *file-a.csv* and *file-b.csv*. The pattern `file-[0-9].csv` matches *file-0.csv*, *file-1.csv*, *file-2.csv* up to *file-9.csv*.

To read NVIDIA stock data for years 2010 through 2019, use the following pattern:

```
pl.read_csv("data/stock/nvda/201[0-9].csv")
```

```
shape: (2_516, 8)
┌────────┬────────────┬─────────┬───┬─────────┬───────────┬──────────┐
│ symbol ┆ date       ┆ open    ┆ … ┆ close   ┆ adj close ┆ volume   │
│ ---    ┆ ---        ┆ ---     ┆   ┆ ---     ┆ ---       ┆ ---      │
│ str    ┆ str        ┆ f64     ┆   ┆ f64     ┆ f64       ┆ i64      │
╞════════╪════════════╪═════════╪═══╪═════════╪═══════════╪══════════╡
│ NVDA   ┆ 2010-01-04 ┆ 4.6275  ┆ … ┆ 4.6225  ┆ 4.24115   ┆ 80020400 │
```

NVDA	2010-01-05	4.605	…	4.69	4.303082	72864800
NVDA	2010-01-06	4.6875	…	4.72	4.330608	64916800
NVDA	2010-01-07	4.695	…	4.6275	4.245738	54779200
NVDA	2010-01-08	4.59	…	4.6375	4.254913	47816800
…	…	…	…	…	…	…
NVDA	2019-12-24	59.549999	…	59.654999	59.432919	13886400
NVDA	2019-12-26	59.689999	…	59.797501	59.574883	18285200
NVDA	2019-12-27	59.950001	…	59.217499	58.997044	25464400
NVDA	2019-12-30	58.997501	…	58.080002	57.863789	25805600
NVDA	2019-12-31	57.724998	…	58.825001	58.606007	23100400

To read all CSV files in *data/stock* directory, use two asterisks, because they're located in different subdirectories:

```
all_stocks = pl.read_csv("data/stock/*/*.csv")
all_stocks
```

shape: (18_476, 8)

symbol --- str	date --- str	open --- f64	…	close --- f64	adj close --- f64	volume --- i64
ASML	1999-01-04	11.765625	…	12.140625	7.5722	1801867
ASML	1999-01-05	11.859375	…	13.96875	8.712416	8241600
ASML	1999-01-06	14.25	…	16.875	10.525064	16400267
ASML	1999-01-07	14.742188	…	16.851563	10.510445	17722133
ASML	1999-01-08	16.078125	…	15.796875	9.852628	10696000
…	…	…	…	…	…	…
TSM	2023-06-26	102.019997	…	100.110001	100.110001	8560000
TSM	2023-06-27	101.150002	…	102.080002	102.080002	9732000
TSM	2023-06-28	100.5	…	100.919998	100.919998	8160900
TSM	2023-06-29	101.339996	…	100.639999	100.639999	7383900
TSM	2023-06-30	101.400002	…	100.919998	100.919998	11701700

If you cannot express the files you wish to read through a globbing pattern, then you can use a manual approach:

1. Construct a list of filenames to read.
2. Read those files using the appropriate Polars function (e.g., `pl.read_csv()`).
3. Combine the Polars DataFrames using the `pl.concat()` function.

Here's an example where we read all ASML stock data from leap years:

```
import calendar

filenames = [
    f"data/stock/asml/{year}.csv"
    for year in range(1999, 2024)
    if calendar.isleap(year)
```

```
]
```

```python
filenames
```

```
['data/stock/asml/2000.csv',
 'data/stock/asml/2004.csv',
 'data/stock/asml/2008.csv',
 'data/stock/asml/2012.csv',
 'data/stock/asml/2016.csv',
 'data/stock/asml/2020.csv']
```

```python
pl.concat(pl.read_csv(f) for f in filenames)
```

```
shape: (1_512, 8)
┌────────┬────────────┬────────────┬───┬────────────┬────────────┬─────────┐
│ symbol ┆ date       ┆ open       ┆ … ┆ close      ┆ adj close  ┆ volume  │
│ ---    ┆ ---        ┆ ---        ┆   ┆ ---        ┆ ---        ┆ ---     │
│ str    ┆ str        ┆ f64        ┆   ┆ f64        ┆ f64        ┆ i64     │
╞════════╪════════════╪════════════╪═══╪════════════╪════════════╪═════════╡
│ ASML   ┆ 2000-01-03 ┆ 43.875     ┆ … ┆ 43.640625  ┆ 27.218985  ┆ 1121600 │
│ ASML   ┆ 2000-01-04 ┆ 41.953125  ┆ … ┆ 40.734375  ┆ 25.406338  ┆ 968800  │
│ ASML   ┆ 2000-01-05 ┆ 39.28125   ┆ … ┆ 39.609375  ┆ 24.704666  ┆ 1458133 │
│ ASML   ┆ 2000-01-06 ┆ 36.75      ┆ … ┆ 37.171875  ┆ 23.184378  ┆ 3517867 │
│ ASML   ┆ 2000-01-07 ┆ 36.867188  ┆ … ┆ 38.015625  ┆ 23.710632  ┆ 1631200 │
│ …      ┆ …          ┆ …          ┆ … ┆ …          ┆ …          ┆ …       │
│ ASML   ┆ 2020-12-24 ┆ 478.950012 ┆ … ┆ 483.089996 ┆ 471.932404 ┆ 271900  │
│ ASML   ┆ 2020-12-28 ┆ 487.140015 ┆ … ┆ 480.23999  ┆ 469.148193 ┆ 449300  │
│ ASML   ┆ 2020-12-29 ┆ 489.450012 ┆ … ┆ 484.01001  ┆ 472.831177 ┆ 377200  │
│ ASML   ┆ 2020-12-30 ┆ 488.130005 ┆ … ┆ 489.910004 ┆ 478.594879 ┆ 381900  │
│ ASML   ┆ 2020-12-31 ┆ 490.0      ┆ … ┆ 487.720001 ┆ 476.455444 ┆ 312700  │
└────────┴────────────┴────────────┴───┴────────────┴────────────┴─────────┘
```

Reading Parquet

The Parquet format is a columnar storage file format optimized for use in big-data processing frameworks like Apache Spark, Apache Hive, and of course, Polars. It offers efficient compression and encoding schemes, improving performance and reducing storage space.

Compared to row-based formats like CSV and Excel, Parquet is more efficient at reading and writing large datasets, especially when querying specific columns. Additionally, Parquet supports complex nested data structures, while CSV and Excel are generally flat, making Parquet a more versatile choice for complex datasets.

Parquet files also include the schema of the data, eliminating the kind of errors that we saw when reading CSV files.

Here's an example using trip data from yellow cabs in New York City:

```python
trips = pl.read_parquet("data/taxi/yellow_tripdata_*.parquet")
trips
```

```
shape: (39_656_098, 19)
┌──────────┬──────────────────────┬─────┬──────────────────────┬─────────────┐
│ VendorID ┆ tpep_pickup_datetime ┆ …   ┆ congestion_surcharge ┆ airport_fee │
│ ---      ┆ ---                  ┆     ┆ ---                  ┆ ---         │
│ i64      ┆ datetime[ns]         ┆     ┆ f64                  ┆ f64         │
╞══════════╪══════════════════════╪═════╪══════════════════════╪═════════════╡
│ 1        ┆ 2022-01-01 00:35:40  ┆ …   ┆ 2.5                  ┆ 0.0         │
│ 1        ┆ 2022-01-01 00:33:43  ┆ …   ┆ 0.0                  ┆ 0.0         │
│ 2        ┆ 2022-01-01 00:53:21  ┆ …   ┆ 0.0                  ┆ 0.0         │
│ 2        ┆ 2022-01-01 00:25:21  ┆ …   ┆ 2.5                  ┆ 0.0         │
│ 2        ┆ 2022-01-01 00:36:48  ┆ …   ┆ 2.5                  ┆ 0.0         │
│ …        ┆ …                    ┆ …   ┆ …                    ┆ …           │
│ 2        ┆ 2022-12-31 23:46:00  ┆ …   ┆ null                 ┆ null        │
│ 2        ┆ 2022-12-31 23:13:24  ┆ …   ┆ null                 ┆ null        │
│ 2        ┆ 2022-12-31 23:00:49  ┆ …   ┆ null                 ┆ null        │
│ 1        ┆ 2022-12-31 23:02:50  ┆ …   ┆ null                 ┆ null        │
│ 2        ┆ 2022-12-31 23:00:15  ┆ …   ┆ null                 ┆ null        │
└──────────┴──────────────────────┴─────┴──────────────────────┴─────────────┘
```

On our modest laptops, reading nearly 40 million rows with `pl.read_parquet()` takes only about 5 seconds.

Table 4-3 lists some commonly used arguments for reading Parquet files.

Table 4-3. Common arguments for the function `pl.read_parquet()`

Argument	Description
`source`	Path to a file, or a file-like object. If the path is a directory, files in that directory will all be read. If `fsspec` is installed, it will be used to open remote files.
`columns`	Columns to select. Accepts a list of column indices (starting at zero) or a list of column names.
`n_rows`	Stop reading from parquet file after reading `n_rows`. Only valid when `use_pyarrow=False`.
`use_pyarrow`	Use pyarrow instead of the Rust native parquet reader. The pyarrow reader is more stable (default: `False`).

Parquet's speed and robustness make it, in our humble opinion, the best file format when working with DataFrames. You'll be seeing a lot more of it in the rest of this book.

Reading JSON and NDJSON

In this section we discuss how to read JavaScript Object Notation (JSON), and its cousin Newline Delimited JSON (NDJSON).

JSON

JSON is a text format that is easy for humans to read and write, and easy for machines to parse and generate. Unlike CSV and Excel, JSON can contain nested data structures. This flexibility makes it a popular choice for APIs, NoSQL databases, and configuration files.

Let's look at the raw contents of *data/pokedex.json* using the command-line tool `cat`:

```
$ cat data/pokedex.json
{
  "pokemon": [{
    "id": 1,
    "num": "001",
    "name": "Bulbasaur",
    "img": "http://www.serebii.net/pokemongo/pokemon/001.png",
    "type": [
      "Grass",
      "Poison"
    ],
    "height": "0.71 m",
    "weight": "6.9 kg",
    "candy": "Bulbasaur Candy",
    "candy_count": 25,
    "egg": "2 km",
    "spawn_chance": 0.69,
    "avg_spawns": 69,
    "spawn_time": "20:00",
    "multipliers": [1.58],
    "weaknesses": [
      "Fire",
      "Ice",
      "Flying",
      "Psychic"
    ],
    "next_evolution": [{
      "num": "002",
      "name": "Ivysaur"
    }, {
      "num": "003",
      "name": "Venusaur"
    }]
  }, {
… with 4053 more lines
```

This JSON file starts and ends with a curly brace, meaning that the entire file is one JSON object. Those curly braces are precisely what allows JSON to be highly nested.

The object has one key `pokemon`, which contains a list of objects. The first 33 lines show also the first Pokemon object, namely `Bulbasaur`. This object, in turn, has some keys that contain other objects. Again, this flexibility has many advantages, but as we'll see next, also poses some challenges when reading it with Polars.

So let's see what happens when we read this JSON file into a Polars DataFrame:

```
pokedex = pl.read_json("data/pokedex.json")
pokedex
```

```
shape: (1, 1)
┌─────────────────────────────────────────────────────────────────────────────────┐
│ pokemon                                                                         │
│ ---                                                                             │
│ list[struct[17]]                                                                │
╞═════════════════════════════════════════════════════════════════════════════════╡
│ [{1,"001","Bulbasaur","http://www.serebii.net/pokemongo/pokemon/001.png",["G │
│ rass", "Poison"],"0.71 m","6.9 kg","Bulbasaur Candy","2 km",0.69,69.0,"20:0… │
└─────────────────────────────────────────────────────────────────────────────────┘
```

Notice how everything is read as a single value? That's because the JSON object has only one key called `pokemon` whose value is a list of objects. Polars doesn't make any assumptions as how to flatten a nested structure into a rectangular shape.

Luckily, Polars offers two methods to flatten the data manually: `df.explode()`, which is used to turn every item in a list into a new row and `df.unnest()`, which is used to turn every key of an object into a new column. For now, let's flatten the Pokedex to some extent:

```
(
    pokedex.explode("pokemon")
    .unnest("pokemon")
    .select("id", "name", "type", "height", "weight")
)
```

```
shape: (151, 5)
┌─────┬────────────┬───────────────────────┬────────┬──────────┐
│ id  ┆ name       ┆ type                  ┆ height ┆ weight   │
│ --- ┆ ---        ┆ ---                   ┆ ---    ┆ ---      │
│ i64 ┆ str        ┆ list[str]             ┆ str    ┆ str      │
╞═════╪════════════╪═══════════════════════╪════════╪══════════╡
│ 1   ┆ Bulbasaur  ┆ ["Grass", "Poison"]   ┆ 0.71 m ┆ 6.9 kg   │
│ 2   ┆ Ivysaur    ┆ ["Grass", "Poison"]   ┆ 0.99 m ┆ 13.0 kg  │
│ 3   ┆ Venusaur   ┆ ["Grass", "Poison"]   ┆ 2.01 m ┆ 100.0 kg │
│ 4   ┆ Charmander ┆ ["Fire"]              ┆ 0.61 m ┆ 8.5 kg   │
│ 5   ┆ Charmeleon ┆ ["Fire"]              ┆ 1.09 m ┆ 19.0 kg  │
│ …   ┆ …          ┆ …                     ┆ …      ┆ …        │
│ 147 ┆ Dratini    ┆ ["Dragon"]            ┆ 1.80 m ┆ 3.3 kg   │
│ 148 ┆ Dragonair  ┆ ["Dragon"]            ┆ 3.99 m ┆ 16.5 kg  │
│ 149 ┆ Dragonite  ┆ ["Dragon", "Flying"]  ┆ 2.21 m ┆ 210.0 kg │
│ 150 ┆ Mewtwo     ┆ ["Psychic"]           ┆ 2.01 m ┆ 122.0 kg │
│ 151 ┆ Mew        ┆ ["Psychic"]           ┆ 0.41 m ┆ 4.0 kg   │
└─────┴────────────┴───────────────────────┴────────┴──────────┘
```

Table 4-4 lists some commonly used arguments for reading JSON and NDJSON, which we cover next.

Table 4-4. Common arguments for the functions `pl.read_json()` and `pl.read_ndjson()`

Argument	Description
`source`	Path to a file or a file-like object.
`schema`	The DataFrame schema may be declared in several ways: (1) As a dictionairy of `{name: type}` pairs; if type is None, it will be auto-inferred. (2) As a list of column names; in this case types are automatically inferred. (3) As a list of `(name, type)` pairs; this is equivalent to the dictionary form.
`schema_overrides`	Support type specification or override of one or more columns; note that any types inferred from the schema param will be overridden. underlying data, the names given here will overwrite them.

NDJSON

NDJSON is a convenient format for storing or streaming structured data to be processed one record at a time. It's essentially a collection of JSON objects, separated by newline characters.

Each line in an NDJSON dataset is a valid JSON object, but the file as a whole is not a valid JSON array because the newline characters are not part of the JSON syntax. This format is beneficial because it allows you to add to the dataset easily and, read the data efficiently, line by line, which can be particularly useful in streaming scenarios or when dealing with large datasets that cannot fit into memory all at once. NDJSON is used in settings from log files to RESTful APIs.

We've prepared *data/wikimedia.ndjson* by listening to the stream of the Wikimedia API for a while and slightly cleaning it up. Here are the first 5 lines of that file:

```
$ cat data/wikimedia.ndjson
```

```
{"$schema":"/mediawiki/recentchange/1.0.0","meta":{"uri":"https://en.wikipedia.…
{"$schema":"/mediawiki/recentchange/1.0.0","meta":{"uri":"https://en.wikipedia.…
{"$schema":"/mediawiki/recentchange/1.0.0","meta":{"uri":"https://en.wikipedia.…
{"$schema":"/mediawiki/recentchange/1.0.0","meta":{"uri":"https://en.wikipedia.…
{"$schema":"/mediawiki/recentchange/1.0.0","meta":{"uri":"https://en.wikipedia.…
… with 95 more lines
```

Again, every line is a single JSON object. Let's have a closer look at the first one:

```
from json import loads
from pprint import pprint

with open("data/wikimedia.ndjson") as f:
    pprint(loads(f.readline()))
```

```
{'$schema': '/mediawiki/recentchange/1.0.0',
 'bot': False,
 'comment': '/* League champions, runners-up and play-off finalists */',
 'id': 1659529639,
```

```
 'length': {'new': 91166, 'old': 91108},
 'meta': {'domain': 'en.wikipedia.org',
          'dt': '2023-07-29T07:51:39Z',
          'id': '0416300b-980c-45bb-b0a2-c9d7a9e2b7eb',
          'offset': 4820784717,
          'partition': 0,
          'request_id': 'ea0541fb-4e72-4fc3-82f0-6c26651b2043',
          'stream': 'mediawiki.recentchange',
          'topic': 'eqiad.mediawiki.recentchange',
          'uri': 'https://en.wikipedia.org/wiki/EFL_Championship'},
 'minor': False,
 'namespace': 0,
 'notify_url': 'https://en.wikipedia.org/w/index.php?diff=1167689309&oldid=1166…
 'parsedcomment': '<span dir="auto"><span class="autocomment"><a '
                  'href="/wiki/EFL_Championship#League_champions,_runners-up_an…
                  'title="EFL Championship">→\u200eLeague champions, '
                  'runners-up and play-off finalists</a></span></span>',
 'revision': {'new': 1167689309, 'old': 1166824248},
 'server_name': 'en.wikipedia.org',
 'server_script_path': '/w',
 'server_url': 'https://en.wikipedia.org',
 'timestamp': 1690617099,
 'title': 'EFL Championship',
 'title_url': 'https://en.wikipedia.org/wiki/EFL_Championship',
 'type': 'edit',
 'user': '87.12.215.232',
 'wiki': 'enwiki'}
```

Notice that this JSON object is slightly nested. Three keys, namely `length`, `meta`, and `revision`, have multiple keys and values. Let's see how Polars loads this data using the `pl.read_ndjson()` function:

```python
wikimedia = pl.read_ndjson("data/wikimedia.ndjson")
wikimedia
```

```
shape: (100, 20)
┌──────────────────────┬──────────────────────┬───┬────────┬──────────────────────┐
│ $schema              ┆ meta                 ┆ … ┆ wiki   ┆ parsedcomment        │
│ ---                  ┆ ---                  ┆   ┆ ---    ┆ ---                  │
│ str                  ┆ struct[9]            ┆   ┆ str    ┆ str                  │
╞══════════════════════╪══════════════════════╪═══╪════════╪══════════════════════╡
│ /mediawiki/recentc…  ┆ {"https://en.wikip…  ┆ … ┆ enwiki ┆ <span dir="auto"><…  │
│ /mediawiki/recentc…  ┆ {"https://en.wikip…  ┆ … ┆ enwiki ┆                      │
│ /mediawiki/recentc…  ┆ {"https://en.wikip…  ┆ … ┆ enwiki ┆ <span dir="auto"><…  │
│ /mediawiki/recentc…  ┆ {"https://en.wikip…  ┆ … ┆ enwiki ┆ Nominated for dele…  │
│ /mediawiki/recentc…  ┆ {"https://en.wikip…  ┆ … ┆ enwiki ┆ Rescuing 1 sources…  │
│ …                    ┆ …                    ┆ … ┆ …      ┆ …                    │
│ /mediawiki/recentc…  ┆ {"https://en.wikip…  ┆ … ┆ enwiki ┆ <span dir="auto"><…  │
│ /mediawiki/recentc…  ┆ {"https://en.wikip…  ┆ … ┆ enwiki ┆ Ce                   │
│ /mediawiki/recentc…  ┆ {"https://en.wikip…  ┆ … ┆ enwiki ┆                      │
│ /mediawiki/recentc…  ┆ {"https://en.wikip…  ┆ … ┆ enwiki ┆                      │
│ /mediawiki/recentc…  ┆ {"https://en.wikip…  ┆ … ┆ enwiki ┆ <span dir="auto"><…  │
└──────────────────────┴──────────────────────┴───┴────────┴──────────────────────┘
```

Just as with the Pokedex, we can `unnest()` columns to turn the keys into new columns.

```
(
    wikimedia.rename({"id": "edit_id"})
    .unnest("meta")
    .select("timestamp", "title", "user", "comment")
)
```

```
shape: (100, 4)
┌────────────┬────────────────────┬────────────────────┬────────────────────┐
│ timestamp  ┆ title              ┆ user               ┆ comment            │
│ ---        ┆ ---                ┆ ---                ┆ ---                │
│ i64        ┆ str                ┆ str                ┆ str                │
╞════════════╪════════════════════╪════════════════════╪════════════════════╡
│ 1690617099 ┆ EFL Championship   ┆ 87.12.215.232      ┆ /* League champio… │
│ 1690617102 ┆ Lim Sang-choon     ┆ Preferwiki         ┆                    │
│ 1690617104 ┆ Higher             ┆ Ss112              ┆ /* Albums */ add   │
│ 1690617104 ┆ International Pok… ┆ Piotrus            ┆ Nominated for del… │
│ 1690617105 ┆ Abdul Hamid Khan … ┆ InternetArchiveBo… ┆ Rescuing 1 source… │
│ …          ┆ …                  ┆ …                  ┆ …                  │
│ 1690617238 ┆ Havering Resident… ┆ MRSC               ┆ /* 2018 election … │
│ 1690617235 ┆ Olha Kharlan       ┆ 2603:7000:2101:AA… ┆ Ce                 │
│ 1690617238 ┆ Mukim Kota Batu    ┆ Pangalau           ┆                    │
│ 1690617239 ┆ User:IDK1213safas… ┆ 94.101.29.27       ┆                    │
│ 1690617234 ┆ List of bus route… ┆ Pedroperezhumbert… ┆ /* Non-TfL bus ro… │
└────────────┴────────────────────┴────────────────────┴────────────────────┘
```

Note that we need to rename the `id` column to `edit_id` because otherwise `df.unnest()` fails, complaining about duplicate column names.

Other File Formats

Polars also supports the formats Arrow IPC (Feather version 2), Apache Avro, Delta lake tables, and PyArrow datasets. For these formats, use the `pl.read_ipc()`, `pl.read_avro()`, `pl.read_delta`, and `pl.scan_pyarrow_dataset()` functions, respectively.

If you have a file that's not supported by Polars, then perhaps Pandas can lend a hand. Pandas has been around for over 13 years, so it's not surprising that it supports more formats. You can convert a Pandas DataFrame to a Polars DataFrame using `pl.from_pandas()`. Here's an example of reading a table from an HTML page:

```
import pandas as pd

url = "https://en.wikipedia.org/wiki/List_of_Latin_abbreviations"
pl.from_pandas(pd.read_html(url)[0])
```

```
shape: (62, 4)
┌──────────────┬────────────────────┬────────────────────┬────────────────────┐
│ abbreviation ┆ Latin              ┆ translation        ┆ usage and notes    │
```

```
│ ---          │ ---                │ ---                │ ---                │
│ str          │ str                │ str                │ str                │
╞══════════════╪════════════════════╪════════════════════╪════════════════════╡
│ A.D.         │ anno Domini        │ "in the year of t… │ Used to label or … │
│ A.I.         │ ad interim         │ "temporarily"      │ Used in business … │
│ a.m.         │ ante meridiem      │ "before midday"[1… │ Used on the twelv… │
│ ca./c.       │ circa              │ "around", "about"… │ Used with dates t… │
│ Cap.         │ capitulus          │ "chapter"          │ Used before a cha… │
│ …            │ …                  │ …                  │ …                  │
│ S.O.S.       │ si opus sit        │ "if there is need… │ A prescription in… │
│ sic          │ sic erat scriptum  │ "thus it was writ… │ Often used when c… │
│ stat.        │ statim             │ "immediately"      │ Often used in med… │
│ viz.         │ videlicet          │ "namely", "to wit… │ In contradistinct… │
│ vs. v.       │ versus             │ "against"          │ Sometimes is not … │
└──────────────┴────────────────────┴────────────────────┴────────────────────┘
```

Besides HTML, Pandas (not Polars) offers support for reading Feather, Fixed-Width Text Files, HDF5, ORC, SAS, SPSS, Stata, XLS, XML, the local clipboard, and various spreadsheet formats. Some of these formats require an additional package to be installed. For instance, the HTML example above requires the `lxml` package. See the IO Tools section in the Pandas User Guide for more information.

Querying Databases

Polars provides a convenient way to interface with relational databases using the `pl.read_database()` function. This function allows you to execute SQL queries directly and retrieve the results as a DataFrame. Polars supports retrieving data from various relational databases, including Postgres, MsSQL, MySQL, Oracle, SQLite, and BigQuery.

The `pl.read_database()` function needs an SQL query and a connection string. The connection string allows you to specify the database's type, its location, and, if needed, your credentials. For example, the connection string to a Postgres database follows the pattern: `postgres://username:password@server:port/database`.

A database usually runs somewhere else (or at least in a separate process) and usually requires credentials. A SQlite database, however, is just a single local file. So, to keep things easy for ourselves, we're going to use a SQLite database to demonstrate how Polars can query databases. The process is the same for the other types of databases, except that you need to specify a different connection string and perhaps use a different SQL dialect.

We're using the Sakila database, a sample database originally developed by the MySQL development team and ported to SQLite by Bradley Grant. The following query selects 10 imaginary film titles, along with a category, rating, and length for each:

```
pl.read_database_uri(
    query="""
    SELECT
        f.film_id,
        f.title,
        c.name AS category,
        f.rating,
        f.length / 60.0 AS length
    FROM
        film AS f,
        film_category AS fc,
        category AS c
    WHERE
        fc.film_id = f.film_id
        AND fc.category_id = c.category_id
    LIMIT 10
    """,
    uri="sqlite:::data/sakila.db",
)
```

```
shape: (10, 5)
┌─────────┬──────────────────┬─────────────┬────────┬──────────┐
│ film_id ┆ title            ┆ category    ┆ rating ┆ length   │
│ ---     ┆ ---              ┆ ---         ┆ ---    ┆ ---      │
│ i64     ┆ str              ┆ str         ┆ str    ┆ f64      │
╞═════════╪══════════════════╪═════════════╪════════╪══════════╡
│ 1       ┆ ACADEMY DINOSAUR ┆ Documentary ┆ PG     ┆ 1.433333 │
│ 2       ┆ ACE GOLDFINGER   ┆ Horror      ┆ G      ┆ 0.8      │
│ 3       ┆ ADAPTATION HOLES ┆ Documentary ┆ NC-17  ┆ 0.833333 │
│ 4       ┆ AFFAIR PREJUDICE ┆ Horror      ┆ G      ┆ 1.95     │
│ 5       ┆ AFRICAN EGG      ┆ Family      ┆ G      ┆ 2.166667 │
│ 6       ┆ AGENT TRUMAN     ┆ Foreign     ┆ PG     ┆ 2.816667 │
│ 7       ┆ AIRPLANE SIERRA  ┆ Comedy      ┆ PG-13  ┆ 1.033333 │
│ 8       ┆ AIRPORT POLLOCK  ┆ Horror      ┆ R      ┆ 0.9      │
│ 9       ┆ ALABAMA DEVIL    ┆ Horror      ┆ PG-13  ┆ 1.9      │
│ 10      ┆ ALADDIN CALENDAR ┆ Sports      ┆ NC-17  ┆ 1.05     │
└─────────┴──────────────────┴─────────────┴────────┴──────────┘
```

If SQL is not your cup of tea but you still need to read from a database, you can use one or more `SELECT * FROM table` queries to select everything and continue in Polars. The following three SQL queries and Polars code produce the same result as the single SQL query above:

```
db = "sqlite:::data/sakila.db"
films = pl.read_database_uri("SELECT * FROM film", db)
film_categories = pl.read_database_uri("SELECT * FROM film_category", db)
categories = pl.read_database_uri("SELECT * FROM category", db)

(
    films.join(film_categories, on="film_id", suffix="_fc")
    .join(categories, on="category_id", suffix="_c")
    .select(
        "film_id",
```

```
            "title",
            pl.col("name").alias("category"),
            "rating",
            pl.col("length") / 60,
        )
        .limit(10)
)
```

```
shape: (10, 5)
┌─────────┬───────────────────┬─────────────┬────────┬──────────┐
│ film_id ┆ title             ┆ category    ┆ rating ┆ length   │
│ ---     ┆ ---               ┆ ---         ┆ ---    ┆ ---      │
│ i64     ┆ str               ┆ str         ┆ str    ┆ f64      │
╞═════════╪═══════════════════╪═════════════╪════════╪══════════╡
│ 1       ┆ ACADEMY DINOSAUR  ┆ Documentary ┆ PG     ┆ 1.433333 │
│ 2       ┆ ACE GOLDFINGER    ┆ Horror      ┆ G      ┆ 0.8      │
│ 3       ┆ ADAPTATION HOLES  ┆ Documentary ┆ NC-17  ┆ 0.833333 │
│ 4       ┆ AFFAIR PREJUDICE  ┆ Horror      ┆ G      ┆ 1.95     │
│ 5       ┆ AFRICAN EGG       ┆ Family      ┆ G      ┆ 2.166667 │
│ 6       ┆ AGENT TRUMAN      ┆ Foreign     ┆ PG     ┆ 2.816667 │
│ 7       ┆ AIRPLANE SIERRA   ┆ Comedy      ┆ PG-13  ┆ 1.033333 │
│ 8       ┆ AIRPORT POLLOCK   ┆ Horror      ┆ R      ┆ 0.9      │
│ 9       ┆ ALABAMA DEVIL     ┆ Horror      ┆ PG-13  ┆ 1.9      │
│ 10      ┆ ALADDIN CALENDAR  ┆ Sports      ┆ NC-17  ┆ 1.05     │
└─────────┴───────────────────┴─────────────┴────────┴──────────┘
```

When you take this approach, consider how much data will be transferred. For a better performance, it's usually a good idea to let the database do as much work as possible and select only the columns you need.

Writing Data

Python Polars offers a wide range of methods when it comes to writing data to a file. Understanding the nuances of each format helps you to make an informed decision tailored to your specific data needs.

CSV Format

One of the most popular choices for writing is the CSV format. CSV stands out for its universal recognition and compatibility with a vast array of software and tools. To save a DataFrame in this format, you can use the `df.write_csv()` method:

```
all_stocks.write_csv("data/all_stocks.csv")
```

Table 4-5 lists some frequently used arguments for writing CSV files.

Table 4-5. Common arguments for the `df.write_csv()` method

Argument	Description
`file`	File path to write the DataFrame to. If set to None (default), the output is returned as a string instead.

Argument	Description
`has_header`	Whether to include a header (default: `True`).
`separator`	Character to separate CSV fields (default: `,`).
`quote`	Character to use for quoting values (default: "`).
`null_value`	String to represent missing values (default: empty string).

Since CSV is a text-based format, it's easily readable by humans. However, as we've seen, it does come with some challenges related to encoding, missing data, and schema inference.

Excel Format

If you're looking to write data in a format familiar to many business users, the Excel format is an optimal choice. The method `df.write_excel("filename.xlsx")` accomplishes this:

```
all_stocks.write_excel("data/all_stocks.xlsx")
```

Table 4-6 lists some frequently used arguments for writing Excel files.

Table 4-6. Common arguments for the `df.write_excel()` method

Argument	Description
`worksheet`	Name of target worksheet (default: `Sheet1`).
`position`	Table position in Excel notation (eg: "A1"), or a (row,col) integer tuple.
`table_style`	A named Excel table style, such as "Table Style Medium 4", or a dictionary of `{"key":value}` options containing one or more of the following keys: "style", "first_column", "last_column", "banded_columns, "banded_rows".
`column_widths`	A `{colname:int}` dict or single integer that sets (or overrides if auto fitting) table column widths in integer pixel units. If given as an integer the same value is used for all table columns.

Excel's primary advantage lies in its support for multisheet workbooks and its capability to incorporate styling and formulas directly into the data. Nevertheless, it is a binary format, which means direct human readability is compromised. Moreover, it's not the best choice for very large datasets, as performance can be an issue.

Parquet Format

If your DataFrame is large and you need an efficient read/write mechanism, the Parquet format is ideal. Using the `df.write_parquet("filename.parquet")` method, you can save data in this columnar storage format:

```
all_stocks.write_parquet("data/all_stocks.parquet")
```

Table 4-7 lists some frequently used arguments for writing Parquet files.

Table 4-7. Common arguments for the `df.write_parquet()` method

Argument	Description
`file`	File path to which the DataFrame should be written.
`compression`	Choose `zstd` for good compression performance. Choose `lz4` for fast compression and decompression. Choose `snappy` for more backwards compatibility guarantees when you deal with older parquet readers.
`compression_level`	The level of compression to use. Higher compression means smaller files on disk.

Parquet is designed for efficiency; it compresses data for optimal storage and supports intricate nested data structures. Furthermore, it retains the schema information, allowing for consistent data retrieval. However, Parquet isn't as universally recognized as CSV or Excel, so you might need specific tools or libraries to read the data.

Other Considerations

Polars also supports writing to other formats like Avro and JSON. When determining the appropriate format, it's essential to weigh factors like the data's intended use, compatibility with other software, the size of the dataset, and the intricacy of the required data structures.

Conclusion

Throughout this chapter, we've explored Polars' capabilities for reading and writing data. We've detailed how to interact efficiently with various file formats, from CSV and Excel to Parquet. Incorporating globbing techniques has enabled you to effectively handle multiple files. We've addressed the nuances of correctly reading missing values and the intricacies involved in managing different character encodings. With these functions under your belt, you should have no problem applying the upcoming topics and code samples to your own data.

CHAPTER 5

Beginning Expressions

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

The goal of this chapter is to introduce expressions, which are what makes the Polars API so powerful and elegant. This chapter forms the basis for the remaining chapters of Part II, where we go into more detail regarding specific expressions and how to use them.

Polars Expressions versus Regular Expressions

Polars expressions should not be confused with regular expressions. A *regular expression*, or *regex*, is a sequence of characters that is used to match text. For example, the regex `[Pp](ol|and)ar?s` matches both `pandas` and `Polars`, but it doesn't match `panda` or `polaris`. A few Polars methods do accept regexes, such as `pl.col()` for selecting columns and `Expr.str.replace()` for replacing values. The interactive website RegExr by Grant Skinner and the book *Introducing Regular Expressions* by Michael FitzGerald are useful resources for learning more about regexes.

Expressions, in Polars, are reusable building blocks that enable you to perform many data-wrangling tasks, including selecting existing columns, creating new columns, filtering rows on a condition, and calculating aggregations. In short, they pop up everywhere.

Expressions have so much to offer that we've split it into three chapters as pictured in Figure 5-1. In Chapter 13 we cover various methods that are accessible through so-called namespaces (explained in the next section).

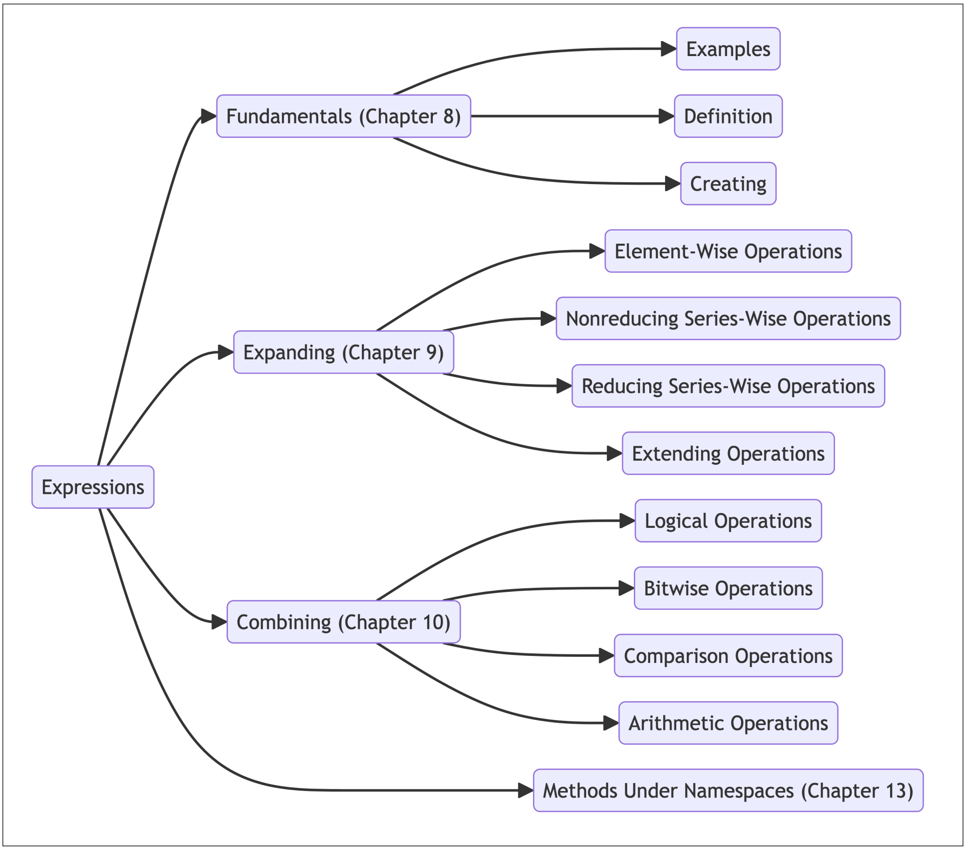


Figure 5-1. The many expression methods are organized into three chapters

In this chapter you'll learn:

- What expressions are
- Where expressions can be used
- How to create expressions from existing columns
- How to create expressions from literal values

- How to create expressions from ranges
- How to rename expressions
- Why expressions are the recommended way of working with Polars

Afterwards, in Chapter 6 and Chapter 7 you'll learn how to expand expressions and how to combine them, respectively.

Methods and Namespaces

The `pl.Expr` class, which represents a Polars expression, has about 350 methods(!) at the time of writing. More than a hundred expression methods are accessible through *namespaces*: groups of methods that each deal with a particular data type.

For example, the `Expr.str` namespace has methods for working with strings, the `Expr.dt` namespace has methods for working with temporal values, and the `Expr.cat` namespace has methods for working with categories. These types and their associated methods are covered in Chapter 10. In this chapter we'll focus on the fundamentals and more general methods of expressions.

Expressions by Example

Expressions really shine when they're being applied. That may sound obvious, but expressions by themselves don't do anything. They're lazy, just like LazyFrames. In practice, expressions are applied by passing them as arguments to some DataFrame or LazyFrame method.

Before we dive into the details of expressions, we're going to demonstrate expressions through some examples:

- Selecting columns with the method `df.select()`
- Creating new columns with the method `df.with_columns()`
- Filtering rows with the method `df.filter()`
- Aggregating with the method `df.group_by()`
- Sorting rows with the method `df.sort()`

As you're going through these examples, keep in mind that it's not the methods but the expressions that matter most. Each method will be covered in more details in its own chapter.

We'll use the following DataFrame about 10 delicious fruits[1] from around the world:

1 Yes, avocado is actually a fruit—a large single-seeded berry, to be precise.

```python
import polars as pl

fruit = pl.read_csv("data/fruit.csv")
fruit
```

```
shape: (10, 5)
┌────────────┬────────┬────────┬──────────┬───────────────┐
│ name       ┆ weight ┆ color  ┆ is_round ┆ origin        │
│ ---        ┆ ---    ┆ ---    ┆ ---      ┆ ---           │
│ str        ┆ i64    ┆ str    ┆ bool     ┆ str           │
╞════════════╪════════╪════════╪══════════╪═══════════════╡
│ Avocado    ┆ 200    ┆ green  ┆ false    ┆ South America │
│ Banana     ┆ 120    ┆ yellow ┆ false    ┆ Asia          │
│ Blueberry  ┆ 1      ┆ blue   ┆ false    ┆ North America │
│ Cantaloupe ┆ 2500   ┆ orange ┆ true     ┆ Africa        │
│ Cranberry  ┆ 2      ┆ red    ┆ false    ┆ North America │
│ Elderberry ┆ 1      ┆ black  ┆ false    ┆ Europe        │
│ Orange     ┆ 130    ┆ orange ┆ true     ┆ Asia          │
│ Papaya     ┆ 1000   ┆ orange ┆ false    ┆ South America │
│ Peach      ┆ 150    ┆ orange ┆ true     ┆ Asia          │
│ Watermelon ┆ 5000   ┆ green  ┆ true     ┆ Africa        │
└────────────┴────────┴────────┴──────────┴───────────────┘
```

The methods we demonstrate below are also available for LazyFrames, but since we're only dealing with 10 rows, a DataFrame will do just fine. And since we don't have to materialize the result with the method `lf.collect()`, it keeps the examples shorter.

Selecting Columns with Expressions

You can select one or more existing columns from a DataFrame using the method `df.select()`. Any columns not mentioned in the expressions are dropped from the output. The following code snippet selects the fruit's name, origin, and weight (in kilograms):

```python
fruit.select(
    pl.col("name"),  ❶
    pl.col("^.*or.*$"),  ❷
    pl.col("weight") / 1000,  ❸
    "is_round"  ❹
)
```

```
shape: (10, 5)
┌────────────┬────────┬───────────────┬────────┬──────────┐
│ name       ┆ color  ┆ origin        ┆ weight ┆ is_round │
│ ---        ┆ ---    ┆ ---           ┆ ---    ┆ ---      │
│ str        ┆ str    ┆ str           ┆ f64    ┆ bool     │
╞════════════╪════════╪═══════════════╪════════╪══════════╡
│ Avocado    ┆ green  ┆ South America ┆ 0.2    ┆ false    │
│ Banana     ┆ yellow ┆ Asia          ┆ 0.12   ┆ false    │
│ Blueberry  ┆ blue   ┆ North America ┆ 0.001  ┆ false    │
│ Cantaloupe ┆ orange ┆ Africa        ┆ 2.5    ┆ true     │
│ Cranberry  ┆ red    ┆ North America ┆ 0.002  ┆ false    │
```

```
│ Elderberry │ black  │ Europe        │ 0.001  │ false    │
│ Orange     │ orange │ Asia          │ 0.13   │ true     │
│ Papaya     │ orange │ South America │ 1.0    │ false    │
│ Peach      │ orange │ Asia          │ 0.15   │ true     │
│ Watermelon │ green  │ Africa        │ 5.0    │ true     │
└────────────┴────────┴───────────────┴────────┴──────────┘
```

❶ The function `pl.col()` is the most common way to start an expression. The argument is a string that refers to an existing column—in this case `name`.

❷ `pl.col()` also accepts regular expressions as arguments. This regular expression matches the two columns `color` and `origin`, because their names both contain the string "or".

❸ You can perform arithmetic (addition, subtraction, multiplication, and division) on expressions using the operators you're already familiar with. (We'll discuss performing arithmetic further in Chapter 7.) Notice how Polars automatically casts the `weight` column from an integer (`i64`) to a float (`f64`) to allow for fractional weights.

❹ The method `df.select()` also accepts strings to refer to existing columns. This might be convenient because you have to type less. However, since a string is not an expression, you won't be able to apply any arithmetic or other operations to it.

Creating New Columns with Expressions

With the method `df.with_columns()` you can create one or more columns, either based on existing columns or from scratch. In this example we add two columns to our `fruit` DataFrame: one that indicates whether a fruit is a fruit (which is obviously always `True`) and one that indicates whether a fruit is a berry (based on its name):

```
fruit.with_columns(
    pl.lit(True).alias("is_fruit"),  ❶
    pl.col("name").str.ends_with("berry").alias("is_berry")  ❷
)
```

```
shape: (10, 7)
┌────────────┬────────┬────────┬──────────┬──────────────┬──────────┬──────────┐
│ name       ┆ weight ┆ color  ┆ is_round ┆ origin       ┆ is_fruit ┆ is_berry │
│ ---        ┆ ---    ┆ ---    ┆ ---      ┆ ---          ┆ ---      ┆ ---      │
│ str        ┆ i64    ┆ str    ┆ bool     ┆ str          ┆ bool     ┆ bool     │
╞════════════╪════════╪════════╪══════════╪══════════════╪══════════╪══════════╡
│ Avocado    ┆ 200    ┆ green  ┆ false    ┆ South Amer…  ┆ true     ┆ false    │
│ Banana     ┆ 120    ┆ yellow ┆ false    ┆ Asia         ┆ true     ┆ false    │
│ Blueberry  ┆ 1      ┆ blue   ┆ false    ┆ North Amer…  ┆ true     ┆ true     │
│ Cantaloupe ┆ 2500   ┆ orange ┆ true     ┆ Africa       ┆ true     ┆ false    │
│ Cranberry  ┆ 2      ┆ red    ┆ false    ┆ North Amer…  ┆ true     ┆ true     │
│ Elderberry ┆ 1      ┆ black  ┆ false    ┆ Europe       ┆ true     ┆ true     │
```

```
│ Orange     │ 130    │ orange │ true     │ Asia        │ true     │ false    │
│ Papaya     │ 1000   │ orange │ false    │ South Amer… │ true     │ false    │
│ Peach      │ 150    │ orange │ true     │ Asia        │ true     │ false    │
│ Watermelon │ 5000   │ green  │ true     │ Africa      │ true     │ false    │
└────────────┴────────┴────────┴──────────┴─────────────┴──────────┴──────────┘
```

❶ With the function `pl.lit()`, you start an expression based on a literal value, such as `True`. The method `Expr.alias()` allows you to name new columns and rename existing columns.

❷ The `Expr.str.ends_with()` method is one the many string methods in the `str` namespace. As mentioned, these will be covered in Chapter 9.

Filtering Rows with Expressions

To filter rows based on an expression, use the method `df.filter()`. Only rows for which the expression evaluates to `True` are kept. This example only keeps fruits that are round *and* weigh more than 1,000 grams:

```
fruit.filter(
    pl.col("is_round") &  ❶
    (pl.col("weight") > 1000)  ❷
)
```

```
shape: (2, 5)
┌────────────┬────────┬────────┬──────────┬────────┐
│ name       │ weight │ color  │ is_round │ origin │
│ ---        │ ---    │ ---    │ ---      │ ---    │
│ str        │ i64    │ str    │ bool     │ str    │
╞════════════╪════════╪════════╪══════════╪════════╡
│ Cantaloupe │ 2500   │ orange │ true     │ Africa │
│ Watermelon │ 5000   │ green  │ true     │ Africa │
└────────────┴────────┴────────┴──────────┴────────┘
```

❶ Here we combine two expressions using the logical AND (`&`) operator. The output is `True` if and only if both expressions are `True`. (We discuss logical operators in Chapter 7.)

❷ Existing columns can be turned into Boolean ones using comparison operators, such as the greater-than (`>`) operator.

Aggregating with Expressions

Aggregation typically involves creating groups of rows, then summarizing each group into one row. This example creates groups based on the last part of the `origin` column, then calculates the number of fruits per group and their average weight. Note that it uses expressions in two different places: in determining the groups, and then in summarizing the groups:

```
fruit.group_by(
    pl.col("origin").str.split(" ").list.last()  ❶
).agg(
    pl.count(),  ❷
    pl.col("weight").mean().alias("average_weight")  ❸
)
```

```
shape: (4, 3)
┌─────────┬───────┬────────────────┐
│ origin  ┆ count ┆ average_weight │
│ ---     ┆ ---   ┆ ---            │
│ str     ┆ u32   ┆ f64            │
╞═════════╪═══════╪════════════════╡
│ America ┆ 4     ┆ 300.75         │
│ Europe  ┆ 1     ┆ 1.0            │
│ Asia    ┆ 3     ┆ 133.333333     │
│ Africa  ┆ 2     ┆ 3750.0         │
└─────────┴───────┴────────────────┘
```

❶ Each unique value of this expression (the last part of the `origin` column) leads to one group.

❷ The expression created by the function `pl.count()` returns the number of rows in the group.

❸ The method `Expr.mean()` is one of many that summarize data—turning multiple values into one.

> We don't want to get ahead of ourselves too much, but we're pretty excited to let you that multiple expressions are executed in parallel—as is the case with both the aggregation and selection examples. This is one of the reasons why Polars is so blazingly fast.

Sorting Rows with Expressions

To Rearrange a DataFrame based on one or more columns, use the method `df.sort()`. This (arguably contrived) example sorts the fruits based on the length of their names:

```
fruit.sort(
    pl.col("name").str.len_bytes(),  ❶
    descending=True  ❷
)
```

```
shape: (10, 5)
┌────────────┬────────┬───────┬──────────┬──────────────┐
│ name       ┆ weight ┆ color ┆ is_round ┆ origin       │
│ ---        ┆ ---    ┆ ---   ┆ ---      ┆ ---          │
│ str        ┆ i64    ┆ str   ┆ bool     ┆ str          │
```

```
┌────────────┬──────┬────────┬───────┬───────────────┐
│ Cantaloupe │ 2500 │ orange │ true  │ Africa        │
│ Elderberry │ 1    │ black  │ false │ Europe        │
│ Watermelon │ 5000 │ green  │ true  │ Africa        │
│ Blueberry  │ 1    │ blue   │ false │ North America │
│ Cranberry  │ 2    │ red    │ false │ North America │
│ Avocado    │ 200  │ green  │ false │ South America │
│ Banana     │ 120  │ yellow │ false │ Asia          │
│ Orange     │ 130  │ orange │ true  │ Asia          │
│ Papaya     │ 1000 │ orange │ false │ South America │
│ Peach      │ 150  │ orange │ true  │ Asia          │
└────────────┴──────┴────────┴───────┴───────────────┘
```

❶ You can sort on an expression that's not actually present in the `fruits` DataFrame. (While the names are present, their *lengths* are not.) It's not necessary to explicitly add a new column if you only want to use it for sorting.

❷ For ascending order (the default), remove this argument or set it to `False`.

What Exactly Is an Expression?

Now that you've seen some concrete examples of expressions and how they can be applied, it's time to define what exactly an expression is.

> ### Expression Definition
>
> An expression is a tree of operations that describe how to construct one or more Series.

Let's break this definition down into five parts:

Series
: Recall from Chapter 2 that a Series is an array of values with the same data type, such as `pl.Float64` for 64-bit floats or `pl.String` for strings. You can think of a Series as a column in a DataFrame, but keep in mind that a Series can exist on its own and is therefore not *always* part of a DataFrame.

Tree of operations
: An expression can consist of: a single operation, multiple operations in a linear sequence, and multiple operations organized in a tree-like structure.

: Figure 5-2 shows three example expressions. If these expressions were to be executed they would produce three columns with the values 7, 12, and 6, respectively. Note that the third diagram in Figure 5-2 is indeed tree-like.

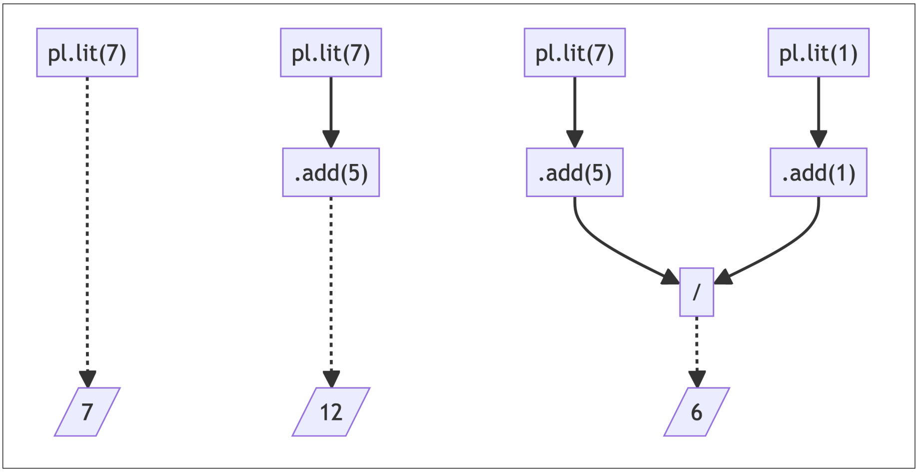


Figure 5-2. An expression is a tree of operations

Generally speaking, all expressions are tree-like, but they don't necessarily have branches or parents.

Describe
An expression is just a description; it doesn't construct any Series by itself nor does it have a method to execute itself. Expressions are only executed when passed as arguments to functions such as `pl.select()` and methods such as `df.group_by()`. Then one or more Series is constructed.

If you think of an expression as a recipe, then the operations would the steps, and the functions and methods such the cooks.

Construct
You don't always see the Series that's being constructed. Whether the constructed Series becomes (or become) part of the DataFrame will depend on the function or method executing the expression. A example of where this is not the case is the function `pl.filter()`. The constructed Series is only used to determine which rows of the DataFrame should be kept; it doesn't become a new column. The word *construct* should also be taken with a grain of salt: if the expression only consists of a single operation that references an existing column in a DataFrame, then no Series is actually being constructed.

One or more
A single expression can describe the construction of more than one Series. For example, the function `pl.all()` refers to all Series in a DataFrame. The expression `pl.all().mul(10).name.suffix("_times_10")` multiplies the values in all existing Series by 10 and adds "_times_10" to their names:

```
(
    pl.DataFrame({"a": [1, 2, 3], "b": [0.4, 0.5, 0.6]})
    .with_columns(pl.all().mul(10).name.suffix("_times_10"))
)
```

```
shape: (3, 4)
┌─────┬─────┬────────────┬────────────┐
│ a   ┆ b   ┆ a_times_10 ┆ b_times_10 │
│ --- ┆ --- ┆ ---        ┆ ---        │
│ i64 ┆ f64 ┆ i64        ┆ f64        │
╞═════╪═════╪════════════╪════════════╡
│ 1   ┆ 0.4 ┆ 10         ┆ 4.0        │
│ 2   ┆ 0.5 ┆ 20         ┆ 5.0        │
│ 3   ┆ 0.6 ┆ 30         ┆ 6.0        │
└─────┴─────┴────────────┴────────────┘
```

With the method `Expr.meta.has_multiple_outputs()` you can check whether an expression describes the potential construction of multiple Series:

```
pl.all().mul(10).name.suffix("_times_10").meta.has_multiple_outputs()
```

```
True
```

Whether multiple Series are actually constructed depends on the DataFrame to which it's applied. If the DataFrame only has one Series to begin with, `pl.all()` will only construct one Series.

Properties of Expressions

It's one thing to know the definition of expressions; it's another thing to understand how they work in practice. Here are a couple of properties of expressions worth mentioning:

Lazy
: Expressions are lazy: By themselves, they don't do anything. Perhaps being lazy is their most important property, because without it, they wouldn't have the other five properties we're about to mention.

Function and data dependent
: Expressions depend on both the function that executes them and the DataFrame (or LazyFrame) onto which they are applied. The function determines what happens to the Series being constructed; the DataFrame determines the type and the length of the Series.

To demonstrate, let's pass the same expression (``is_orange``) to three different functions (methods), shown here alongside their output:

```
is_orange = (pl.col("color") == "orange").alias("is_orange")
```

```
fruit.with_columns(is_orange)
```

```
shape: (10, 6)
┌────────────┬────────┬────────┬──────────┬───────────────┬───────────┐
│ name       ┆ weight ┆ color  ┆ is_round ┆ origin        ┆ is_orange │
│ ---        ┆ ---    ┆ ---    ┆ ---      ┆ ---           ┆ ---       │
│ str        ┆ i64    ┆ str    ┆ bool     ┆ str           ┆ bool      │
╞════════════╪════════╪════════╪══════════╪═══════════════╪═══════════╡
│ Avocado    ┆ 200    ┆ green  ┆ false    ┆ South America ┆ false     │
│ Banana     ┆ 120    ┆ yellow ┆ false    ┆ Asia          ┆ false     │
│ Blueberry  ┆ 1      ┆ blue   ┆ false    ┆ North America ┆ false     │
│ Cantaloupe ┆ 2500   ┆ orange ┆ true     ┆ Africa        ┆ true      │
│ Cranberry  ┆ 2      ┆ red    ┆ false    ┆ North America ┆ false     │
│ Elderberry ┆ 1      ┆ black  ┆ false    ┆ Europe        ┆ false     │
│ Orange     ┆ 130    ┆ orange ┆ true     ┆ Asia          ┆ true      │
│ Papaya     ┆ 1000   ┆ orange ┆ false    ┆ South America ┆ true      │
│ Peach      ┆ 150    ┆ orange ┆ true     ┆ Asia          ┆ true      │
│ Watermelon ┆ 5000   ┆ green  ┆ true     ┆ Africa        ┆ false     │
└────────────┴────────┴────────┴──────────┴───────────────┴───────────┘
```

```python
fruit.filter(is_orange)
```

```
shape: (4, 5)
┌────────────┬────────┬────────┬──────────┬───────────────┐
│ name       ┆ weight ┆ color  ┆ is_round ┆ origin        │
│ ---        ┆ ---    ┆ ---    ┆ ---      ┆ ---           │
│ str        ┆ i64    ┆ str    ┆ bool     ┆ str           │
╞════════════╪════════╪════════╪══════════╪═══════════════╡
│ Cantaloupe ┆ 2500   ┆ orange ┆ true     ┆ Africa        │
│ Orange     ┆ 130    ┆ orange ┆ true     ┆ Asia          │
│ Papaya     ┆ 1000   ┆ orange ┆ false    ┆ South America │
│ Peach      ┆ 150    ┆ orange ┆ true     ┆ Asia          │
└────────────┴────────┴────────┴──────────┴───────────────┘
```

```python
fruit.group_by(is_orange).len()
```

```
shape: (2, 2)
┌───────────┬─────┐
│ is_orange ┆ len │
│ ---       ┆ --- │
│ bool      ┆ u32 │
╞═══════════╪═════╡
│ false     ┆ 6   │
│ true      ┆ 4   │
└───────────┴─────┘
```

The key take away is that you'll use the same syntax to accomplish different tasks. Which ties into the next property of expressions: reusability.

Reusable

Expressions are Python objects. In the previous example, we created the expression object `is_orange` and reused it by passing it to different methods of the `fruit` DataFrame. Taking this further, there's nothing stopping us from using the same expression on a completely different DataFrame:

```
flowers = pl.DataFrame({
    "name": ["Tiger lily", "Blue flag", "African marigold"],
    "latin": ["Lilium columbianum", "Iris versicolor", "Tagetes erecta"],
    "color": ["orange", "purple", "orange"]
})

flowers.filter(is_orange)
```

```
shape: (2, 3)
┌──────────────────┬────────────────────┬────────┐
│ name             ┆ latin              ┆ color  │
│ ---              ┆ ---                ┆ ---    │
│ str              ┆ str                ┆ str    │
╞══════════════════╪════════════════════╪════════╡
│ Tiger lily       ┆ Lilium columbianum ┆ orange │
│ African marigold ┆ Tagetes erecta     ┆ orange │
└──────────────────┴────────────────────┴────────┘
```

Efficient

Because expressions are lazy you can optimize them before you executed them. Polars will minimize the number of computations required to construct the Series by analyzing the operations in the expression. Moreover, when a function is given multiple expressions, they are executed in parallel.

To summarize, expressions have many favorable properties. Let's continue with creating expressions.

Creating Expressions

Each expression starts with a first operation. Generally speaking, a new expression is created using a function that doesn't depend on another expression. Once you have an expression, you can continue to build on it with many methods and combine it with other expressions using inline operators (discussed in the next two chapters). Let's look at the various ways in which we can create one, starting with existing columns.

From Existing Columns

The most common way to create an expression is to reference one or more existing columns in the DataFrame. After all, most often you want to transform the data you already have. This can be done with the function `pl.col()`, which accepts column names, regular expressions, and data types. Here are a few examples.

For demonstration purposes, we execute the expressions using the method `df.select()` and get the list of column names via the `df.columns` attribute. You can reference a particular column by passing its name:

```
fruit.select(pl.col("color")).columns
```

```
['color']
```

If the DataFrame has no column with that particular name, Polars will throw an error:

```
fruit.select(pl.col("is_smelly")).columns
```

```
ColumnNotFoundError: is_smelly

Error originated just after this operation:
DF ["name", "weight", "color", "is_round"]; PROJECT */5 COLUMNS; SELECTION: "Non
e"
```

Regular expressions are especially useful for referencing multiple columns whose names have a common pattern. To do so, the regular expression has to start with a caret (`^`) and end with a dollar sign (`$`):

```
fruit.select(pl.col("^.*or.*$")).columns
```

```
['color', 'origin']
```

With `pl.col("*")` or the convenient alias `pl.all()` you can reference all columns:

```
fruit.select(pl.all()).columns
```

```
['name', 'weight', 'color', 'is_round', 'origin']
```

You can reference all columns with a particular data type (for example, `pl.String` for strings):

```
fruit.select(pl.col(pl.String)).columns
```

```
['name', 'color', 'origin']
```

You can give `pl.col()` multiple column names or data types:

```
fruit.select(pl.col(pl.Boolean, pl.Int64)).columns
```

```
['weight', 'is_round']
```

Or you can pass them as a list, if that's more convenient:

```
fruit.select(pl.col(["name", "color"])).columns
```

```
['name', 'color']
```

However, you cannot mix column names and data types:

```
fruit.select(pl.col([pl.String, "is_round"])).columns
```

```
TypeError: argument 'dtypes': 'str' is not a Polars data type
```

Referencing Numeric Data Types

To reference all the columns containing numbers, you can use the constant `pl.NUMERIC_DTYPES`, which has all the numerical data types:

```
pl.NUMERIC_DTYPES
```

```
frozenset({Decimal,
           Float32,
           Float64,
           Int16,
           Int32,
           Int64,
           Int8,
           UInt16,
           UInt32,
           UInt64,
           UInt8})
```

You can use this constant directly in `pl.col()`:

```
(
    fruit
    .with_columns((pl.col("weight") / 1000).alias("weight_kg"))
    .select(pl.col(pl.NUMERIC_DTYPES))
    .head()
)
```

```
shape: (5, 2)
┌────────┬───────────┐
│ weight ┆ weight_kg │
│ ---    ┆ ---       │
│ i64    ┆ f64       │
╞════════╪═══════════╡
│ 200    ┆ 0.2       │
│ 120    ┆ 0.12      │
│ 1      ┆ 0.001     │
│ 2500   ┆ 2.5       │
│ 2      ┆ 0.002     │
└────────┴───────────┘
```

From Literal Values

To create a new expression based on some other Python value you can use the function `pl.lit()`. *Lit* is short for "literal". The next few examples execute the expressions using the `pl.select()` function, which starts with a new, empty DataFrame:

```
pl.select(pl.lit(42))
```

```
shape: (1, 1)
┌─────────┐
│ literal │
│ ---     │
│ i32     │
╞═════════╡
│ 42      │
└─────────┘
```

Notice that the column name is literally `literal`. You can give this column a better name using the method `Expr.alias()`:

```
pl.select(pl.lit(42).alias("answer"))
```

```
shape: (1, 1)
┌────────┐
│ answer │
│ ---    │
│ i32    │
╞════════╡
│ 42     │
└────────┘
```

When you execute these expressions using the function `pl.select()`, the constructed Series have only one value. However, when you execute the same expression to a nonempty DataFrame, the length of the Series will be equal to the number of rows:

```
fruit.with_columns(pl.lit("Earth").alias("planet"))
```

```
shape: (10, 6)
┌────────────┬────────┬────────┬──────────┬───────────────┬────────┐
│ name       ┆ weight ┆ color  ┆ is_round ┆ origin        ┆ planet │
│ ---        ┆ ---    ┆ ---    ┆ ---      ┆ ---           ┆ ---    │
│ str        ┆ i64    ┆ str    ┆ bool     ┆ str           ┆ str    │
╞════════════╪════════╪════════╪══════════╪═══════════════╪════════╡
│ Avocado    ┆ 200    ┆ green  ┆ false    ┆ South America ┆ Earth  │
│ Banana     ┆ 120    ┆ yellow ┆ false    ┆ Asia          ┆ Earth  │
│ Blueberry  ┆ 1      ┆ blue   ┆ false    ┆ North America ┆ Earth  │
│ Cantaloupe ┆ 2500   ┆ orange ┆ true     ┆ Africa        ┆ Earth  │
│ Cranberry  ┆ 2      ┆ red    ┆ false    ┆ North America ┆ Earth  │
│ Elderberry ┆ 1      ┆ black  ┆ false    ┆ Europe        ┆ Earth  │
│ Orange     ┆ 130    ┆ orange ┆ true     ┆ Asia          ┆ Earth  │
│ Papaya     ┆ 1000   ┆ orange ┆ false    ┆ South America ┆ Earth  │
│ Peach      ┆ 150    ┆ orange ┆ true     ┆ Asia          ┆ Earth  │
│ Watermelon ┆ 5000   ┆ green  ┆ true     ┆ Africa        ┆ Earth  │
└────────────┴────────┴────────┴──────────┴───────────────┴────────┘
```

As you can see, the value `Earth` is repeated such that the length of the Series `planet` is equal to the number of rows in the DataFrame. Values are only repeated automatically if you pass a single value to the function `pl.lit()`. When you pass more than one value, but fewer values than there are rows, you get an error:

```
fruit.with_columns(pl.lit(pl.Series([False, True])).alias("row_is_even"))
```

```
ShapeError: unable to add a column of length 2 to a DataFrame of height 10
```

Also, the list of values `[False, True]` is first turned into a Series using the `pl.Series()` constructor. Otherwise, Polars will create a list column such that each row has these two values:

```
fruit.with_columns(pl.lit([False, True]).alias("row_is_even"))
```

```
shape: (10, 6)
┌────────────┬────────┬────────┬──────────┬───────────────┬────────────────┐
│ name       ┆ weight ┆ color  ┆ is_round ┆ origin        ┆ row_is_even    │
│ ---        ┆ ---    ┆ ---    ┆ ---      ┆ ---           ┆ ---            │
│ str        ┆ i64    ┆ str    ┆ bool     ┆ str           ┆ list[bool]     │
╞════════════╪════════╪════════╪══════════╪═══════════════╪════════════════╡
│ Avocado    ┆ 200    ┆ green  ┆ false    ┆ South America ┆ [false, true]  │
│ Banana     ┆ 120    ┆ yellow ┆ false    ┆ Asia          ┆ [false, true]  │
│ …          ┆ …      ┆ …      ┆ …        ┆ …             ┆ …              │
│ Peach      ┆ 150    ┆ orange ┆ true     ┆ Asia          ┆ [false, true]  │
│ Watermelon ┆ 5000   ┆ green  ┆ true     ┆ Africa        ┆ [false, true]  │
└────────────┴────────┴────────┴──────────┴───────────────┴────────────────┘
```

To repeat values explicitly, for a fixed number of times, you can use the function `pl.repeat()`. The functions `pl.zeros()` and `pl.ones()` are aliases for `pl.repeat(0.0)` and `pl.repeat(1.0)`, respectively:

```
pl.select(
    pl.repeat("Ello", 3).alias("hello"),
    pl.zeros(3),
    pl.ones(3)
)
```

```
shape: (3, 3)
┌───────┬───────┬──────┐
│ hello ┆ zeros ┆ ones │
│ ---   ┆ ---   ┆ ---  │
│ str   ┆ f64   ┆ f64  │
╞═══════╪═══════╪══════╡
│ Ello  ┆ 0.0   ┆ 1.0  │
│ Ello  ┆ 0.0   ┆ 1.0  │
│ Ello  ┆ 0.0   ┆ 1.0  │
└───────┴───────┴──────┘
```

Keep in mind that the length of each Series must be the same; otherwise you'll get an error:

```
fruit.with_columns(pl.repeat("Earth", 9).alias("planet"))
```

```
ShapeError: unable to add a column of length 9 to a DataFrame of height 10
```

From Ranges

Polars offers a couple of convenient functions for creating ranges of integers, dates, times, and datetimes. They are listed in Table 5-1.

Table 5-1. Functions for creating ranges

Function	Description
`pl.arange(…)`	A range of integers (alias of `pl.int_range(…)`)
`pl.date_range(…)`	A range of dates

Function	Description
`pl.date_ranges(…)`	Each element is a range of dates
`pl.datetime_range(…)`	A range of datetimes
`pl.datetime_ranges(…)`	Each element is a range of datetimes
`pl.int_range(…)`	A range of integers
`pl.int_ranges(…)`	Each element is a range of integers
`pl.time_range(…)`	A range of times
`pl.time_ranges(…)`	Each element is a range of times

The following example demonstrates the functions `pl.int_range()`, its alias `pl.arange()`, and `pl.int_ranges()`. It also includes a sneak peek to the method `Expr.list.len()`, which calculates the number of elements in each list in the `int_range` column:

```
pl.select(
    pl.int_range(0, 5).alias("start"),
    pl.arange(0, 10, 2).pow(2).alias("end")
).with_columns(
    pl.int_ranges("start", "end").alias("int_range")
).with_columns(
    pl.col("int_range").list.len().alias("range_length")
)
```

```
shape: (5, 4)
┌───────┬──────┬───────────────┬──────────────┐
│ start ┆ end  ┆ int_range     ┆ range_length │
│ ---   ┆ ---  ┆ ---           ┆ ---          │
│ i64   ┆ f64  ┆ list[i64]     ┆ u32          │
╞═══════╪══════╪═══════════════╪══════════════╡
│ 0     ┆ 0.0  ┆ []            ┆ 0            │
│ 1     ┆ 4.0  ┆ [1, 2, 3]     ┆ 3            │
│ 2     ┆ 16.0 ┆ [2, 3, … 15]  ┆ 14           │
│ 3     ┆ 36.0 ┆ [3, 4, … 35]  ┆ 33           │
│ 4     ┆ 64.0 ┆ [4, 5, … 63]  ┆ 60           │
└───────┴──────┴───────────────┴──────────────┘
```

Note that the function `pl.int_ranges()` generates a Series where each element is a list of integers. The functions `pl.date_ranges`, `pl.datetime_ranges`, and `pl.time_ranges()` work similarly but then for dates, datetimes, and times, respectively:

```
from datetime import date

pl.select(
    pl.date_range(date(1985, 10, 21), date(1985, 10, 26)).alias("start"),
    pl.repeat(date(2021, 10, 21), 6).alias("end")
).with_columns(
```

```
    pl.datetime_ranges("start", "end", interval="1h").alias("range")
)
```

```
shape: (6, 3)
┌────────────┬────────────┬─────────────────────────────────────────────────┐
│ start      ┆ end        ┆ range                                           │
│ ---        ┆ ---        ┆ ---                                             │
│ date       ┆ date       ┆ list[datetime[μs]]                              │
╞════════════╪════════════╪═════════════════════════════════════════════════╡
│ 1985-10-21 ┆ 2021-10-21 ┆ [1985-10-21 00:00:00, 1985-10-21 01:00:00, …    │
│ 1985-10-22 ┆ 2021-10-21 ┆ [1985-10-22 00:00:00, 1985-10-22 01:00:00, …    │
│ 1985-10-23 ┆ 2021-10-21 ┆ [1985-10-23 00:00:00, 1985-10-23 01:00:00, …    │
│ 1985-10-24 ┆ 2021-10-21 ┆ [1985-10-24 00:00:00, 1985-10-24 01:00:00, …    │
│ 1985-10-25 ┆ 2021-10-21 ┆ [1985-10-25 00:00:00, 1985-10-25 01:00:00, …    │
│ 1985-10-26 ┆ 2021-10-21 ┆ [1985-10-26 00:00:00, 1985-10-26 01:00:00, …    │
└────────────┴────────────┴─────────────────────────────────────────────────┘
```

In Chapter 9 we cover working with temporal data (such as dates and times) in more detail.

Other Functions to Create Expressions

There are many function to create expressions. Unfortunately, we're not able to cover all of them in this chapter. However, to give you an idea of the possibilities, we'll briefly mention a couple of functions, what they do, and where we'll cover them in more detail.

First, the function `pl.count()` is used, as the name implies, for counting the number of rows. It's most often used when aggregating using the method `df.group_by()`. This is covered in Chapter 10.

Second, the function `pl.element()` represents a single element in a list. It is used in combination with the method `Expr.list.eval()` to apply an expression to each element in a list. We explain this is further detail in Chapter 9.

Finally, the function `pl.sql_expr()` is handy if you want to create an expression using SQL.

Renaming Expressions

Renaming an expression—which eventually determines the name of the Series that will be constructed—happens very often. There are various reasons why you would want to rename an expression, including:

- To better express what the column is about
- To avoid duplicate column names
- To clean up a column name

- To change the default column name

Good Names

Having good expression names is just as important as having good variable names in general. They can drastically influence the quality of your code. We personally recommend using column names that are all lowercase using underscores to separate words.

The most common method to change the name of an expression is `Expr.alias()`. Additional methods that are concerned with the name of an expression are available within the `Expr.name` namespace (see Table 5-2). The methods `Expr.name.map_fields()`, `Expr.name.prefix_fields()`, and `Expr.name.suffix_fields()` can only be used when the data type of the expression is `pl.Struct`.

Table 5-2. Methods for renaming expressions

Method	Description
`Expr.alias(…)`	Rename the expression.
`Expr.name.keep()`	Keep the original root name of the expression.
`Expr.name.map(…)`	Rename the expression by mapping a function over the root name.
`Expr.name.prefix(…)`	Add a prefix to the root column name of the expression.
`Expr.name.suffix(…)`	Add a suffix to the root column name of the expression.
`Expr.name.to_lowercase()`	Make the root column name lowercase.
`Expr.name.to_uppercase()`	Make the root column name uppercase.
`Expr.name.map_fields(…)`	Rename fields of a struct by mapping a function over the field name.
`Expr.name.prefix_fields(…)`	Add a prefix to all fields names of a struct.
`Expr.name.suffix_fields(…)`	Add a suffix to all fields names of a struct.

To illustrate, consider this small DataFrame with some arbitrary column names:

```
df = pl.DataFrame({"text": "value", "An integer": 5040, "BOOLEAN": True})
df
```

```
shape: (1, 3)
┌───────┬────────────┬─────────┐
│ text  ┆ An integer ┆ BOOLEAN │
│ ---   ┆ ---        ┆ ---     │
│ str   ┆ i64        ┆ bool    │
╞═══════╪════════════╪═════════╡
│ value ┆ 5040       ┆ true    │
└───────┴────────────┴─────────┘
```

We can change these column names with various methods:

```python
df.select(
    pl.col("text").name.to_uppercase(),
    pl.col("An integer").alias("int"),
    pl.col("BOOLEAN").name.to_lowercase(),
)
```

```
shape: (1, 3)
┌───────┬──────┬─────────┐
│ TEXT  ┆ int  ┆ boolean │
│ ---   ┆ ---  ┆ ---     │
│ str   ┆ i64  ┆ bool    │
╞═══════╪══════╪═════════╡
│ value ┆ 5040 ┆ true    │
└───────┴──────┴─────────┘
```

Chaining Naming Operations

At the time of writing, Polars allows only one naming operation per expression. So the following is not allowed:

```python
df.select(
    pl.all()
    .name.to_lowercase()
    .name.map(lambda s: s.replace(" ", "_"))
)
```

```
PanicException: no `rename_alias` expected at this point
```

A solution is to combine all the operations into one (anonymous) function and then apply that with the `Expr.name.map()` method:

```python
df.select(
    pl.all()
    .name.map(lambda s: s.lower().replace(" ", "_"))
)
```

```
shape: (1, 3)
┌───────┬────────────┬─────────┐
│ text  ┆ an_integer ┆ boolean │
│ ---   ┆ ---        ┆ ---     │
│ str   ┆ i64        ┆ bool    │
╞═══════╪════════════╪═════════╡
│ value ┆ 5040       ┆ true    │
└───────┴────────────┴─────────┘
```

This restriction may be lifted in a future version of Polars.

Expressions Are Idiomatic

You already know that expressions are lazy and that they need to be executed in order to be useful. We understand that it may take time to get used to this, especially if you're used to a nonlazy (eager) way of working using packages, such as Pandas.

So here's a word of caution. All expression methods and inline operations are also available for Series. For instance, the filtering rows example from earlier, which uses expressions, can be rewritten to use Series directly:

```
fruit.filter(
    (fruit["weight"] > 1000) & fruit["is_round"]
)
```

```
shape: (2, 5)
┌────────────┬────────┬────────┬──────────┬────────┐
│ name       ┆ weight ┆ color  ┆ is_round ┆ origin │
│ ---        ┆ ---    ┆ ---    ┆ ---      ┆ ---    │
│ str        ┆ i64    ┆ str    ┆ bool     ┆ str    │
╞════════════╪════════╪════════╪══════════╪════════╡
│ Cantaloupe ┆ 2500   ┆ orange ┆ true     ┆ Africa │
│ Watermelon ┆ 5000   ┆ green  ┆ true     ┆ Africa │
└────────────┴────────┴────────┴──────────┴────────┘
```

If you have experience with Pandas, then this syntax will look familiar, and you might be tempted to write this way when using Polars.

While the code above produces the same results as the original example, it is executed eagerly. Because of this, it doesn't use the Polars query engine and makes no optimizations. Moreover, the two components are executed serially rather than in parallel.

This becomes even more clear when you apply multiple methods to a LazyFrame. Here's an example that uses expressions:

```
(
    fruit
    .lazy()
    .filter((pl.col("weight") > 1000) & pl.col("is_round"))
    .with_columns(pl.col("name").str.ends_with("berry").alias("is_berry"))
    .collect()
)
```

```
shape: (2, 6)
┌────────────┬────────┬────────┬──────────┬────────┬──────────┐
│ name       ┆ weight ┆ color  ┆ is_round ┆ origin ┆ is_berry │
│ ---        ┆ ---    ┆ ---    ┆ ---      ┆ ---    ┆ ---      │
│ str        ┆ i64    ┆ str    ┆ bool     ┆ str    ┆ bool     │
╞════════════╪════════╪════════╪══════════╪════════╪══════════╡
│ Cantaloupe ┆ 2500   ┆ orange ┆ true     ┆ Africa ┆ false    │
│ Watermelon ┆ 5000   ┆ green  ┆ true     ┆ Africa ┆ false    │
└────────────┴────────┴────────┴──────────┴────────┴──────────┘
```

Now an example without expressions:

```
(
    fruit
    .lazy()
    .filter((fruit["weight"] > 1000) & fruit["is_round"])
```

```
        .with_columns(fruit["name"].str.ends_with("berry").alias("is_berry"))
        .collect()
)
```

```
ShapeError: unable to add a column of length 10 to a DataFrame of height 2
```

That's right: Polars can't optimize the execution plan, and **now** you also have to be careful to apply the methods in the correct order to avoid an error. (The reason for the error is that the method `df.filter()` reduces the DataFrame to two rows, whereas the variable `fruit` still refers to a DataFrame with 10 rows.)

For these reasons, we always encourage you to use expressions. Being lazy pays off in Polars.

Conclusion

Expressions are at the heart of Polars. In this first chapter about expressions, we've covered their fundamentals: what they are, where they're used, how they're created, and why they're so elegant and efficient. In the next chapter we explain how you can continue expressions by adding operations.

CHAPTER 6

Continuing Expressions

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

In the previous chapter you learned how to *begin* an expression from scratch. A bare expression only gets you so far. In this chapter, you'll learn how to *continue* an expression by adding additional operations (or methods).

More specifcally, you'll learn how to:

- Perform mathematical transformations
- Work with missing values
- Apply smoothing to values
- Select specific values
- Summarize values using statistics

A Plethora of Methods

There are more than 138 methods discussed in this chapter. It's not possible to explain and demonstrate every single method in full detail. Please refer to the Polars API Reference for more details and examples.

For some code snippets in this chapter we use the `math` and `numpy` modules for accessing certain constants, such as `math.pi`, and for generating random values:

```
import math
import numpy as np

print(f"{math.pi=}")
rng = np.random.default_rng(1729)
print(f"{rng.random()=}")
```

```
math.pi=3.141592653589793
rng.random()=0.03074202960516803
```

Types of Operations

Rather than presenting 138 methods as one long list, we've organized them into five sections according to which inputs they use and the shape of their output. Within those five sections we've grouped methods into categories when applicable. Methods that do not fall into any category are placed in "Others." Figure 6-1 shows the types of operations for continuing expressions.

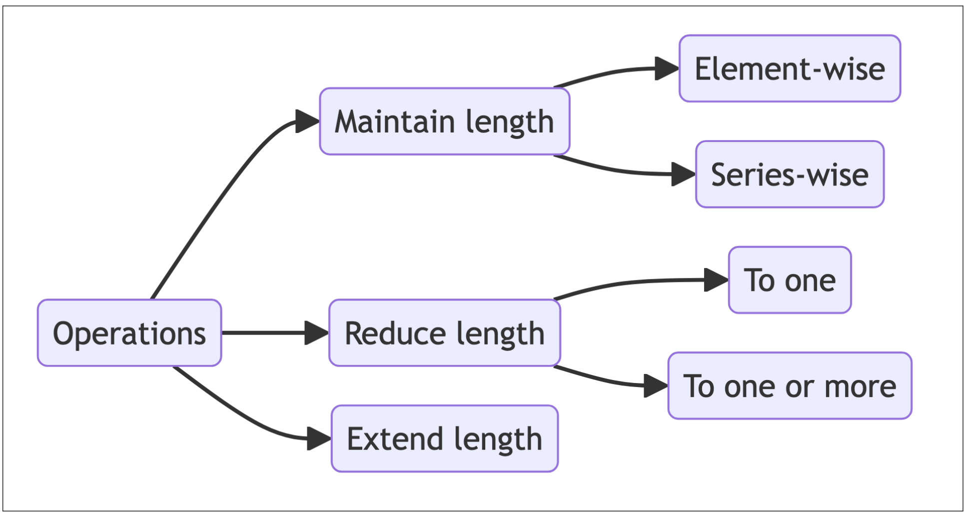


Figure 6-1. Types of operations for continuing expressions

Related Methods, Different Sections

While we trust this organization to be useful, it does cause certain related methods to appear in different sections. For example, both `Expr.unique()` and `Expr.is_unique()` are concerned with unique values, but because the former may reduce the length of the Series while the latter does not, they're in different sections.

Here are four examples to demonstrate what we mean by the various types of operations.

Example A: Element-Wise Operations

In the first example we'll use two methods to create two additional columns: `Expr.sqrt()` and `Expr.interpolate()`. Both methods operate *element-wise* (that is, they consider one element at a time) and maintain the length of the Series.

```
penguins = (
    pl.read_csv("data/penguins.csv", null_values="NA")
    .select(
        "species",
        "island",
        "sex",
        "year",
        pl.col("body_mass_g").alias("mass") / 1000
    ))
penguins.with_columns(
    pl.col("mass").sqrt().alias("mass_sqrt"),  ❶
    pl.col("mass").interpolate().alias("mass_filled")  ❷
)
```

```
shape: (344, 7)
┌───────────┬───────────┬────────┬──────┬───────┬───────────┬─────────────┐
│ species   ┆ island    ┆ sex    ┆ year ┆ mass  ┆ mass_sqrt ┆ mass_filled │
│ ---       ┆ ---       ┆ ---    ┆ ---  ┆ ---   ┆ ---       ┆ ---         │
│ str       ┆ str       ┆ str    ┆ i64  ┆ f64   ┆ f64       ┆ f64         │
╞═══════════╪═══════════╪════════╪══════╪═══════╪═══════════╪═════════════╡
│ Adelie    ┆ Torgersen ┆ male   ┆ 2007 ┆ 3.75  ┆ 1.936492  ┆ 3.75        │
│ Adelie    ┆ Torgersen ┆ female ┆ 2007 ┆ 3.8   ┆ 1.949359  ┆ 3.8         │
│ Adelie    ┆ Torgersen ┆ female ┆ 2007 ┆ 3.25  ┆ 1.802776  ┆ 3.25        │
│ Adelie    ┆ Torgersen ┆ null   ┆ 2007 ┆ null  ┆ null      ┆ 3.35        │
│ Adelie    ┆ Torgersen ┆ female ┆ 2007 ┆ 3.45  ┆ 1.857418  ┆ 3.45        │
│ …         ┆ …         ┆ …      ┆ …    ┆ …     ┆ …         ┆ …           │
│ Chinstrap ┆ Dream     ┆ male   ┆ 2009 ┆ 4.0   ┆ 2.0       ┆ 4.0         │
│ Chinstrap ┆ Dream     ┆ female ┆ 2009 ┆ 3.4   ┆ 1.843909  ┆ 3.4         │
│ Chinstrap ┆ Dream     ┆ male   ┆ 2009 ┆ 3.775 ┆ 1.942936  ┆ 3.775       │
│ Chinstrap ┆ Dream     ┆ male   ┆ 2009 ┆ 4.1   ┆ 2.024846  ┆ 4.1         │
│ Chinstrap ┆ Dream     ┆ female ┆ 2009 ┆ 3.775 ┆ 1.942936  ┆ 3.775       │
└───────────┴───────────┴────────┴──────┴───────┴───────────┴─────────────┘
```

❶ The `Expr.sqrt()` method computes the square root of the `mass` column. Notice how null values remain null.

❷ It would be better to interpolate the missing values per species so that we get a more accurate result.

Example B: Operations that Summarize to One

In the second example, we apply two methods that summarize the Series to a single value:

```
penguins.select(
    pl.col("mass").mean(),
    pl.col("island").mode().first()  ❶
)
```

```
shape: (1, 2)
┌──────────┬────────┐
│ mass     ┆ island │
│ ---      ┆ ---    │
│ f64      ┆ str    │
╞══════════╪════════╡
│ 4.201754 ┆ Biscoe │
└──────────┴────────┘
```

❶ Be careful—the method `Expr.mode()` can, depending on the input, produce more than one value. That's why we continue the expression with the method `Expr.first()` to make sure there's only one value.

Example C: Operations that Summarize to One or More

In the third example, we use the `Expr.unique()` method to get the unique values in a Series. This is a type of operation that summarizes to one or more values.

```
penguins.select(
    pl.col("island").unique()
)
```

```
shape: (3, 1)
┌───────────┐
│ island    │
│ ---       │
│ str       │
╞═══════════╡
│ Dream     │
│ Biscoe    │
│ Torgersen │
└───────────┘
```

Example D: Operations that Extend

In the fourth example we use the `Expr.extend_constant()` method to append a specific value to the end of the Series. This type of operation is used less often. The example is perhaps a bit contrived, but it does illustrate how powerful expressions can be if you add additional methods:

```
penguins.select(
    pl.col("species")
    .unique()  ❶
    .repeat_by(3000)  ❷
    .explode()  ❸
    .extend_constant("Saiyan", n=1)  ❹
)
```

```
shape: (9_001, 1)
┌───────────┐
│ species   │
│ ---       │
│ str       │
╞═══════════╡
│ Chinstrap │
│ Chinstrap │
│ Chinstrap │
│ Chinstrap │
│ Chinstrap │
│ …         │
│ Adelie    │
│ Adelie    │
│ Adelie    │
│ Adelie    │
│ Saiyan    │
└───────────┘
```

❶ Get the three unique values of the Series.

❷ Repeat each value 3,000 times. This produces a Series of three long lists.

❸ Use the explode method to get one long Series of 9,000 values.

❹ Add one more value at the end of the Series. The result is a Series with a length that's just over 9,000.

With these four examples, you should have an idea of the type of operations we can use to continue expressions.

Element-Wise Operations

This section is about operations that consider one element at a time. Each element is computed independently and the order in which they appear doesn't matter. Examples include the `Expr.sqrt()` method for computing the square root of each value and the `Expr.round()` method for rounding values.

In the next five subsections we're looking at element-wise operations for performing mathematical transformations, related to trigonometry, for rounding and binning, concerned with missing or infinite values, and others.

Operations That Perform Mathematical Transformations

Mathematical transformations, such as computing the log or the square root, form the basis of any data-related task. The methods listed in Table 6-1 all perform some mathematical transformation. Arithmetic between two expressions (such as adding and multiplication) is discussed in the next chapter because that's mostly about *combining* expressions.

Table 6-1. Element-wise operations for performing elementary mathematical transformations

Method	Description
`Expr.abs()`	Compute absolute values.
`Expr.cbrt()`	Compute the cube root of the elements.
`Expr.exp()`	Compute the exponential, element-wise.
`Expr.log(…)`	Compute the logarithm to a given base.
`Expr.log10()`	Compute the base 10 logarithm of the input array, element-wise.
`Expr.log1p()`	Compute the natural logarithm of each element plus one.
`Expr.sign()`	Compute the element-wise indication of the sign.
`Expr.sqrt()`	Compute the square root of the elements.

The methods `Expr.abs()`, `Expr.exp()`, `Expr.log()`, `Expr.log10()`, `Expr.log1p()`, `Expr.sign()`, `Expr.sqrt()` are demonstrated in the following code snippet for a variety of numerical values. The method `Expr.cbrt()` is similar in usage.

```
(
    pl.DataFrame({"x": [-2, 0, 0.5, 1, math.e, 1000]})
    .with_columns(
        abs=pl.col("x").abs(),
        exp=pl.col("x").exp(),
        log2=pl.col("x").log(2),  ❶
        log10=pl.col("x").log10(),
        log1p=pl.col("x").log1p(),
        sign=pl.col("x").sign(),
```

```
            sqrt=pl.col("x").sqrt(),
        )
    )
shape: (6, 8)
┌──────────┬──────────┬────────┬────────┬────────┬───────┬──────┬────────┐
│ x        ┆ abs      ┆ exp    ┆ log2   ┆ log10  ┆ log1p ┆ sign ┆ sqrt   │
│ ---      ┆ ---      ┆ ---    ┆ ---    ┆ ---    ┆ ---   ┆ ---  ┆ ---    │
│ f64      ┆ f64      ┆ f64    ┆ f64    ┆ f64    ┆ f64   ┆ i64  ┆ f64    │
╞══════════╪══════════╪════════╪════════╪════════╪═══════╪══════╪════════╡
│ -2.000   ┆ 2.000    ┆ 0.135  ┆ NaN    ┆ NaN    ┆ NaN   ┆ -1   ┆ NaN    │
│ 0.000    ┆ 0.000    ┆ 1.000  ┆ -inf   ┆ -inf   ┆ 0.000 ┆ 0    ┆ 0.000  │
│ 0.500    ┆ 0.500    ┆ 1.649  ┆ -1.000 ┆ -0.301 ┆ 0.405 ┆ 1    ┆ 0.707  │
│ 1.000    ┆ 1.000    ┆ 2.718  ┆ 0.000  ┆ 0.000  ┆ 0.693 ┆ 1    ┆ 1.000  │
│ 2.718    ┆ 2.718    ┆ 15.154 ┆ 1.443  ┆ 0.434  ┆ 1.313 ┆ 1    ┆ 1.649  │
│ 1000.000 ┆ 1000.000 ┆ inf    ┆ 9.966  ┆ 3.000  ┆ 6.909 ┆ 1    ┆ 31.623 │
└──────────┴──────────┴────────┴────────┴────────┴───────┴──────┴────────┘
```

❶ The method `Expr.log()` is the only one here that requires an argument, namely the base of the logarithm.

Operations Related to Trigonometry

Trigonometry is the branch of mathematics that studies the relationships between angles and sides of triangles. It plays a crucial role in various aspects of data science, including signal processing, spacial data, analysis and feature engineering. Table 6-2 lists all methods related to trigonometry that Polars supports.

Table 6-2. Element-wise operations related to trigonometry

Method	Description
`Expr.arccos()`	Compute the element-wise value for the inverse cosine.
`Expr.arccosh()`	Compute the element-wise value for the inverse hyperbolic cosine.
`Expr.arcsin()`	Compute the element-wise value for the inverse sine.
`Expr.arcsinh()`	Compute the element-wise value for the inverse hyperbolic sine.
`Expr.arctan()`	Compute the element-wise value for the inverse tangent.
`Expr.arctanh()`	Compute the element-wise value for the inverse hyperbolic tangent.
`Expr.cos()`	Compute the element-wise value for the cosine.
`Expr.cosh()`	Compute the element-wise value for the hyperbolic cosine.
`Expr.degrees()`	Convert from radians to degrees.
`Expr.radians()`	Convert from degrees to radians.
`Expr.sin()`	Compute the element-wise value for the sine.
`Expr.sinh()`	Compute the element-wise value for the hyperbolic sine.
`Expr.tan()`	Compute the element-wise value for the tangent.
`Expr.tanh()`	Compute the element-wise value for the hyperbolic tangent.

In the code snippet below we apply the methods `Expr.arccos()`, `Expr.cos()`, `Expr.degrees()`, `Expr.radians()`, and `Expr.sin()` to a variety of numerical values. The remaining methods, namely `Expr.arccosh()`, `Expr.arcsin()`, `Expr.arcsinh()`, `Expr.arctan()`, `Expr.arctanh()`, `Expr.cosh()`, `Expr.sinh()`, `Expr.tan()`, and `Expr.tanh()` can be used in a similar way. None of these methods require arguments.

```
(
    pl.DataFrame({"x": [-math.pi, 0, 1, math.pi, 2*math.pi, 90, 180, 360]})
    .with_columns(
        arccos=pl.col("x").arccos(),  ❶
        cos=pl.col("x").cos(),
        degrees=pl.col("x").degrees(),
        radians=pl.col("x").radians(),
        sin=pl.col("x").sin(),
    )
)
```

```
shape: (8, 6)
┌───────────┬──────────┬───────────┬──────────────┬──────────┬─────────────┐
│ x         ┆ arccos   ┆ cos       ┆ degrees      ┆ radians  ┆ sin         │
│ ---       ┆ ---      ┆ ---       ┆ ---          ┆ ---      ┆ ---         │
│ f64       ┆ f64      ┆ f64       ┆ f64          ┆ f64      ┆ f64         │
╞═══════════╪══════════╪═══════════╪══════════════╪══════════╪═════════════╡
│ -3.141593 ┆ NaN      ┆ -1.0      ┆ -180.0       ┆ -0.054831┆ -1.2246e-16 │
│ 0.0       ┆ 1.570796 ┆ 1.0       ┆ 0.0          ┆ 0.0      ┆ 0.0         │
│ 1.0       ┆ 0.0      ┆ 0.540302  ┆ 57.29578     ┆ 0.017453 ┆ 0.841471    │
│ 3.141593  ┆ NaN      ┆ -1.0      ┆ 180.0        ┆ 0.054831 ┆ 1.2246e-16  │
│ 6.283185  ┆ NaN      ┆ 1.0       ┆ 360.0        ┆ 0.109662 ┆ -2.4493e-16 │
│ 90.0      ┆ NaN      ┆ -0.448074 ┆ 5156.620156  ┆ 1.570796 ┆ 0.893997    │
│ 180.0     ┆ NaN      ┆ -0.59846  ┆ 10313.240312 ┆ 3.141593 ┆ -0.801153   │
│ 360.0     ┆ NaN      ┆ -0.283691 ┆ 20626.480625 ┆ 6.283185 ┆ 0.958916    │
└───────────┴──────────┴───────────┴──────────────┴──────────┴─────────────┘
```

❶ With element-wise operations, when an operation results in a `NaN`, the other values are not affected.

Operations That Round and Categorize

Sometimes your data contains too much precision or too many distinct values. In those cases it can be useful to round them or to cut them into discrete categories. Table 6-3 lists the methods that Polars provides for this[1].

1 Technically, the method `Expr.qcut()` is not an element-wise operation because quantiles are based on an entire Series. In this case we thought it's best to keep it close to its cousin `Expr.cut()`.

Table 6-3. Element-wise operations for rounding and binning

Method	Description
`Expr.ceil()`	Round up to the nearest integer value.
`Expr.clip(…)`	Clip (limit) the values in an array to a min and max boundary.
`Expr.cut(…)`	Cut continuous values into discrete categories.
`Expr.floor()`	Round down to the nearest integer value.
`Expr.qcut(…)`	Cut continuous values into discrete categories based on their quantiles.
`Expr.round(…)`	Round underlying floating point data by decimals digits.

Below, we demonstrate these methods (and `Expr.round()` twice) for a range of numbers.

```
(
    pl.DataFrame({"x": [-6, -0.5, 0, 0.5, math.pi, 9.9, 9.99, 9.999]})
    .with_columns(
        ceil=pl.col("x").ceil(),
        clip=pl.col("x").clip(-1, 1),
        cut=pl.col("x").cut([-1, 1], labels=["bad", "neutral", "good"]),  ❶
        floor=pl.col("x").floor(),
        qcut=pl.col("x").qcut([0.5], labels=["below median", "above median"]),
        round2=pl.col("x").round(2),
        round0=pl.col("x").round(0),  ❷
    )
)
```

```
shape: (8, 8)
┌──────────┬──────┬──────┬─────────┬───────┬──────────────┬────────┬────────┐
│ x        ┆ ceil ┆ clip ┆ cut     ┆ floor ┆ qcut         ┆ round2 ┆ round0 │
│ ---      ┆ ---  ┆ ---  ┆ ---     ┆ ---   ┆ ---          ┆ ---    ┆ ---    │
│ f64      ┆ f64  ┆ f64  ┆ cat     ┆ f64   ┆ cat          ┆ f64    ┆ f64    │
╞══════════╪══════╪══════╪═════════╪═══════╪══════════════╪════════╪════════╡
│ -6.0     ┆ -6.0 ┆ -1.0 ┆ bad     ┆ -6.0  ┆ below median ┆ -6.0   ┆ -6.0   │
│ -0.5     ┆ -0.0 ┆ -0.5 ┆ neutral ┆ -1.0  ┆ below median ┆ -0.5   ┆ -1.0   │
│ 0.0      ┆ 0.0  ┆ 0.0  ┆ neutral ┆ 0.0   ┆ below median ┆ 0.0    ┆ 0.0    │
│ 0.5      ┆ 1.0  ┆ 0.5  ┆ neutral ┆ 0.0   ┆ below median ┆ 0.5    ┆ 1.0    │
│ 3.141593 ┆ 4.0  ┆ 1.0  ┆ good    ┆ 3.0   ┆ above median ┆ 3.14   ┆ 3.0    │
│ 9.9      ┆ 10.0 ┆ 1.0  ┆ good    ┆ 9.0   ┆ above median ┆ 9.9    ┆ 10.0   │
│ 9.99     ┆ 10.0 ┆ 1.0  ┆ good    ┆ 9.0   ┆ above median ┆ 9.99   ┆ 10.0   │
│ 9.999    ┆ 10.0 ┆ 1.0  ┆ good    ┆ 9.0   ┆ above median ┆ 10.0   ┆ 10.0   │
└──────────┴──────┴──────┴─────────┴───────┴──────────────┴────────┴────────┘
```

❶ The methods `Expr.cut()` and `Expr.qcut()` construct a Categorical Series. If you want it to be an integer, you can add, for instance, `Expr.cast(pl.Int64)` to the expression.

❷ Even when rounding to zero decimals using `Expr.round(0)` (or by using `Expr.ceil()` or `Expr.floor()`) the type remains float.

Operations for Missing or Infinite Values

When your data is based on the real world, you're bound to have some missing values. NaNs or infinite values are usually the result of some invalid transformation. If you need to deal with these, Polars offers a couple of convenient methods (see Table 6-4). Later in this chapter, there are a few more methods for dealing with missing values in a Series-wise manner.

Table 6-4. Element-wise operations concerned with missing or infinite values

Method	Description
`Expr.fill_nan(…)`	Fill floating point NaN value with a fill value.
`Expr.fill_null(…)`	Fill null values using the specified value or strategy.
`Expr.is_finite()`	Returns a Boolean Series indicating which values are finite.
`Expr.is_infinite()`	Returns a Boolean Series indicating which values are infinite.
`Expr.is_nan()`	Returns a Boolean Series indicating which values are NaN.
`Expr.is_not_nan()`	Returns a Boolean Series indicating which values are not NaN.
`Expr.is_not_null()`	Returns a Boolean Series indicating which values are not null.
`Expr.is_null()`	Returns a Boolean Series indicating which values are null.

The code snippet below applies the methods `Expr.fill_nan()`, `Expr.fill_null()`, `Expr.is_finite()`, `Expr.is_infinite()`, `Expr.is_nan()`, and `Expr.is_null()` to a couple of numerical values, some of which are infinite or missing. The methods `Expr.is_not_nan()` and `Expr.is_not_null()` produce the inverse of `Expr.is_nan()` and `Expr.is_null()`, respectively.

```python
x = [42, math.nan, None, math.inf, -math.inf]
(
    pl.DataFrame({"x": x})
    .with_columns(
        fill_nan=pl.col("x").fill_nan(999),
        fill_null=pl.col("x").fill_null(0),
        is_finite=pl.col("x").is_finite(),
        is_infinite=pl.col("x").is_finite(),
        is_nan=pl.col("x").is_nan(),
        is_null=pl.col("x").is_null(),
    )
)
```

```
shape: (5, 7)
┌──────┬──────────┬───────────┬───────────┬─────────────┬────────┬─────────┐
│ x    ┆ fill_nan ┆ fill_null ┆ is_finite ┆ is_infinite ┆ is_nan ┆ is_null │
│ ---  ┆ ---      ┆ ---       ┆ ---       ┆ ---         ┆ ---    ┆ ---     │
│ f64  ┆ f64      ┆ f64       ┆ bool      ┆ bool        ┆ bool   ┆ bool    │
╞══════╪══════════╪═══════════╪═══════════╪═════════════╪════════╪═════════╡
│ 42.0 ┆ 42.0     ┆ 42.0      ┆ true      ┆ true        ┆ false  ┆ false   │
│ NaN  ┆ 999.0    ┆ NaN       ┆ false     ┆ false       ┆ true   ┆ false   │
```

```
│ null │ null   │ 0.0   │ null  │ null  │ null  │ true  │
│ inf  │ inf    │ inf   │ false │ false │ false │ false │
│ -inf │ -inf   │ -inf  │ false │ false │ false │ false │
└──────┴────────┴───────┴───────┴───────┴───────┴───────┘
```

NaN Versus Null

This is a good reminder that `NaN`s and `null`s are not the same type. If you need to fill both types in a Series, you can add `Expr.fill_nan()` and `Expr.fill_null()` to the expression. And if you need to know whether a value is either `Nan` or `null`, you can combine `Expr.is_nan()` and `Expr.is_null()` with the Boolean OR operator (`|`):

```
(
    pl.DataFrame({"x": x})
    .with_columns(
        fill_both=pl.col("x").fill_nan(0).fill_null(0),
        is_either=(
            pl.col("x").is_nan() | pl.col("x").is_null()
        ),
    )
)
```

```
shape: (5, 3)
┌──────┬───────────┬───────────┐
│ x    │ fill_both │ is_either │
│ ---  │ ---       │ ---       │
│ f64  │ f64       │ bool      │
╞══════╪═══════════╪═══════════╡
│ 42.0 │ 42.0      │ false     │
│ NaN  │ 0.0       │ true      │
│ null │ 0.0       │ true      │
│ inf  │ inf       │ false     │
│ -inf │ -inf      │ false     │
└──────┴───────────┴───────────┘
```

You'll learn more about Boolean operators in the next chapter. Whether you actually *want* to treat `NaN`s and `null`s the same depends on the task at hand.

Other Operations

There are three element-wise operators that don't fall in any of the above categories (see Table 6-5).

Table 6-5. Miscellaneous element-wise operations

Method	Description
`Expr.hash(…)`	Hash the elements in the selection.

Method	Description
`Expr.repeat_by(…)`	Repeat the elements in this Series as specified in the given expression.
`Expr.replace(…)`	Replace values in column according to remapping dictionary.

The following code snippet demonstrates these three methods:

```
(
    pl.DataFrame({"x": ["here", "there", "their", "they're"]})
    .with_columns(
        hash=pl.col("x").hash(seed=1337),  ❶
        repeat_by=pl.col("x").repeat_by(3),
        replace=pl.col("x").replace({
            "here": "there",
            "they're": "they are",
        }),
    )
)
```

```
shape: (4, 4)
┌─────────┬──────────────────────┬─────────────────────────────────┬──────────┐
│ x       ┆ hash                 ┆ repeat_by                       ┆ replace  │
│ ---     ┆ ---                  ┆ ---                             ┆ ---      │
│ str     ┆ u64                  ┆ list[str]                       ┆ str      │
╞═════════╪══════════════════════╪═════════════════════════════════╪══════════╡
│ here    ┆ 12695211751326448172 ┆ ["here", "here", "here"]        ┆ there    │
│ there   ┆ 17329794691236705436 ┆ ["there", "there", "there"]     ┆ there    │
│ their   ┆ 2663095961041830581  ┆ ["their", "their", "their"]     ┆ their    │
│ they're ┆ 6743063676290245144  ┆ ["they're", "they're", "they'r… ┆ they are │
└─────────┴──────────────────────┴─────────────────────────────────┴──────────┘
```

❶ With the method `Expr.hash()`, different computers or computers with different versions of Polars will generate different hash values. More information can be found on the AHash website.

Nonreducing Series-Wise Operations

In the remaining sections, we're no longer looking at element wise operations but at *Series-wise operations*. That means that the Series is transformed as a whole and the values themselves (and sometimes also their order) depend on each other. Examples include the `Expr.cum_sum()` method for computing the cumulative sum and the `Expr.forward_fill()` method for filling missing values.

In the next six subsections we're looking at operations which do not change the length of the Series, which includes operations that accumulate, fill, shift, compute rolling statistics, sort, and more.

Operations That Accumulate

Cumulative operations progress through a Series and maintain, for instance, the sum or the maximum. See Table 6-6 for all the cumulative methods that Polars provides.

Table 6-6. Series-wise operations that are cumulative

Method	Description
`Expr.cum_count(…)`	Get an array with the cumulative count computed at every element.
`Expr.cum_max(…)`	Get an array with the cumulative max computed at every element.
`Expr.cum_min(…)`	Get an array with the cumulative min computed at every element.
`Expr.cum_prod(…)`	Get an array with the cumulative product computed at every element.
`Expr.cum_sum(…)`	Get an array with the cumulative sum computed at every element.
`Expr.diff(…)`	Calculate the n-th discrete difference.
`Expr.pct_change(…)`	Computes percentage change between values.

All these methods accept one argument, `reverse`, which indicates whether the Series should be reversed first, i.e., before the operation is applied. The code snippet below applies all methods to a variety of numerical values, including a missing value and a `NaN`:

```
(
    pl.DataFrame({"x": [0, 1, 2, None, 2, np.NaN, -1, 2]})
    .with_columns(
        cum_count=pl.col("x").cum_count(),  ❶
        cum_max=pl.col("x").cum_max(),
        cum_min=pl.col("x").cum_min(),
        cum_prod=pl.col("x").cum_prod(reverse=True),  ❷
        cum_sum=pl.col("x").cum_sum(),
        diff=pl.col("x").diff(),
        pct_change=pl.col("x").pct_change(),
    )
)
```

```
shape: (8, 8)
┌──────┬───────────┬─────────┬─────────┬──────────┬─────────┬──────┬────────────┐
│ x    ┆ cum_count ┆ cum_max ┆ cum_min ┆ cum_prod ┆ cum_sum ┆ diff ┆ pct_change │
│ ---  ┆ ---       ┆ ---     ┆ ---     ┆ ---      ┆ ---     ┆ ---  ┆ ---        │
│ f64  ┆ u32       ┆ f64     ┆ f64     ┆ f64      ┆ f64     ┆ f64  ┆ f64        │
╞══════╪═══════════╪═════════╪═════════╪══════════╪═════════╪══════╪════════════╡
│ 0.0  ┆ 1         ┆ 0.0     ┆ 0.0     ┆ NaN      ┆ 0.0     ┆ null ┆ null       │
│ 1.0  ┆ 2         ┆ 1.0     ┆ 0.0     ┆ NaN      ┆ 1.0     ┆ 1.0  ┆ inf        │
│ 2.0  ┆ 3         ┆ 2.0     ┆ 0.0     ┆ NaN      ┆ 3.0     ┆ 1.0  ┆ 1.0        │
│ null ┆ 3         ┆ null    ┆ null    ┆ null     ┆ null    ┆ null ┆ 0.0        │
│ 2.0  ┆ 4         ┆ 2.0     ┆ 0.0     ┆ NaN      ┆ 5.0     ┆ null ┆ 0.0        │
│ NaN  ┆ 5         ┆ 2.0     ┆ 0.0     ┆ NaN      ┆ NaN     ┆ NaN  ┆ NaN        │
│ -1.0 ┆ 6         ┆ 2.0     ┆ -1.0    ┆ -2.0     ┆ NaN     ┆ NaN  ┆ NaN        │
```

```
│ 2.0 │ 7               │ 2.0          │ -1.0         │ 2.0           │ NaN          │ 3.0      │ -3.0             │
└─────┴─────────────────┴──────────────┴──────────────┴───────────────┴──────────────┴──────────┴──────────────────┘
```

❶ The method `Expr.cum_count()` does not count missing values.

❷ If we didn't reverse this operation, then the entire column would be filled with zeros.

> **Contagious NaNs**
>
> `NaN`s may affect the output of Series-wise operations. In this example:
>
> - The output of `Expr.cum_count()`, `Expr.cum_max()`, and `Expr.cum_min()` is not affected at all.
> - The output of `Expr.cum_prod()` and `Expr.cum_sum()` remains affected once a `NaN` has been seen.
> - The output of `Expr.diff()` and `Expr.pct_change()` is only affected for two values for every `NaN`.

Operations That Fill and Shift

Table 6-7 lists the nonreducing Series-wise methods for filling and shifting.

Table 6-7. Series-wise operations for filling and shifting

Method	Description
`Expr.backward_fill(…)`	Fill missing values with the next to-be-seen value.
`Expr.forward_fill(…)`	Fill missing values with the latest seen value.
`Expr.interpolate(…)`	Fill missing values using interpolation.
`Expr.shift(…)`	Shift the values by a given period.

Let's apply the methods `Expr.backward_fill()`, `Expr.forward_fill()`, `Expr.interpolate()` (twice), and `Expr.shift()` (twice) to some values, including missing values:

```
(
    pl.DataFrame({"x": [-1, 0, 1, None, None, 3, 4, math.nan, 6]})
    .with_columns(
        backward_fill=pl.col("x").backward_fill(),  ❶
        forward_fill=pl.col("x").forward_fill(limit=1),
        interp1=pl.col("x").interpolate(method="linear"),  ❷
        interp2=pl.col("x").interpolate(method="nearest"),
        shift1=pl.col("x").shift(1),
        shift2=pl.col("x").shift(-2),
```

```
    )
)
shape: (9, 7)
┌──────┬───────────────┬──────────────┬──────────┬─────────┬────────┬────────┐
│ x    ┆ backward_fill ┆ forward_fill ┆ interp1  ┆ interp2 ┆ shift1 ┆ shift2 │
│ ---  ┆ ---           ┆ ---          ┆ ---      ┆ ---     ┆ ---    ┆ ---    │
│ f64  ┆ f64           ┆ f64          ┆ f64      ┆ f64     ┆ f64    ┆ f64    │
╞══════╪═══════════════╪══════════════╪══════════╪═════════╪════════╪════════╡
│ -1.0 ┆ -1.0          ┆ -1.0         ┆ -1.0     ┆ -1.0    ┆ null   ┆ 1.0    │
│ 0.0  ┆ 0.0           ┆ 0.0          ┆ 0.0      ┆ 0.0     ┆ -1.0   ┆ null   │
│ 1.0  ┆ 1.0           ┆ 1.0          ┆ 1.0      ┆ 1.0     ┆ 0.0    ┆ null   │
│ null ┆ 3.0           ┆ 1.0          ┆ 1.666667 ┆ 1.0     ┆ 1.0    ┆ 3.0    │
│ null ┆ 3.0           ┆ null         ┆ 2.333333 ┆ 3.0     ┆ null   ┆ 4.0    │
│ 3.0  ┆ 3.0           ┆ 3.0          ┆ 3.0      ┆ 3.0     ┆ null   ┆ NaN    │
│ 4.0  ┆ 4.0           ┆ 4.0          ┆ 4.0      ┆ 4.0     ┆ 3.0    ┆ 6.0    │
│ NaN  ┆ NaN           ┆ NaN          ┆ NaN      ┆ NaN     ┆ 4.0    ┆ null   │
│ 6.0  ┆ 6.0           ┆ 6.0          ┆ 6.0      ┆ 6.0     ┆ NaN    ┆ null   │
└──────┴───────────────┴──────────────┴──────────┴─────────┴────────┴────────┘
```

❶ `NaN`s do not get filled or interpolated.

❷ Note the difference between the two interpolation methods `linear` and `nearest`. The former interpolates between the previous and next non-missing values in the Series, while the latter uses the actual closest non-missing value.

Operations Related to Duplicate Values

There are four nonreducing Series-wise methods that are concerned with unique and duplicate values (see Table 6-8). There are other methods which are concerned with this, but since they reduce the length of the Series, they are discussed later in the chapter.

Table 6-8. Series-wise operations that return a Boolean Series

Method	Description
`Expr.is_duplicated()`	Get a Boolean Series that indicates which values are duplicated.
`Expr.is_first_distinct()`	Get a Boolean Seriers that indicates which values are first unique.
`Expr.is_last_distinct()`	Get a Boolean Seriers that indicates which values are last unique.
`Expr.is_unique()`	Get a Boolean Seriers that indicates which values are unique.

Below, we apply these four methods to a couple of strings:

```
(
    pl.DataFrame({"x": ["A", "C", "D", "C"]})  ❶
    .with_columns(
        is_duplicated=pl.col("x").is_duplicated(),
        is_first_distinct=pl.col("x").is_first_distinct(),
```

```
        is_last_distinct=pl.col("x").is_last_distinct(),
        is_unique=pl.col("x").is_unique(),
    )
)
```

```
shape: (4, 5)
┌─────┬───────────────┬───────────────────┬──────────────────┬───────────┐
│ x   ┆ is_duplicated ┆ is_first_distinct ┆ is_last_distinct ┆ is_unique │
│ --- ┆ ---           ┆ ---               ┆ ---              ┆ ---       │
│ str ┆ bool          ┆ bool              ┆ bool             ┆ bool      │
╞═════╪═══════════════╪═══════════════════╪══════════════════╪═══════════╡
│ A   ┆ false         ┆ true              ┆ true             ┆ true      │
│ C   ┆ true          ┆ true              ┆ false            ┆ false     │
│ D   ┆ false         ┆ true              ┆ true             ┆ true      │
│ C   ┆ true          ┆ false             ┆ true             ┆ false     │
└─────┴───────────────┴───────────────────┴──────────────────┴───────────┘
```

❶ Keep in mind that many of these methods can also be applied to other data types.

Operations That Compute Rolling Statistics

Rolling statistics are used to smooth the values of a Series (see Table 6-9).

Table 6-9. Series-wise operations for rolling statistics

Method	Description
`Expr.ewm_mean(…)`	Exponentially-weighted moving average.
`Expr.ewm_std(…)`	Exponentially-weighted moving standard deviation.
`Expr.ewm_var(…)`	Exponentially-weighted moving variance.
`Expr.rolling_apply(…)`	Apply a custom rolling window function.
`Expr.rolling_map(…)`	Compute a custom rolling window function.
`Expr.rolling_max(…)`	Apply a rolling max (moving max) over the values in this array.
`Expr.rolling_mean(…)`	Apply a rolling mean (moving mean) over the values in this array.
`Expr.rolling_median(…)`	Compute a rolling median.
`Expr.rolling_min(…)`	Apply a rolling min (moving min) over the values in this array.
`Expr.rolling_quantile(…)`	Compute a rolling quantile.
`Expr.rolling_skew(…)`	Compute a rolling skew.
`Expr.rolling_std(…)`	Compute a rolling standard deviation.
`Expr.rolling_sum(…)`	Apply a rolling sum (moving sum) over the values in this array.
`Expr.rolling_var(…)`	Compute a rolling variance.

The following code snippet applies `Expr.ewm_mean()`, `Expr.rolling_mean()`, and `Expr.rolling_min()` to the `close` column of some stock data. The remaining methods work similarly:

```
stock = (
    pl.read_csv("data/stock/nvda/2023.csv", try_parse_dates=True)
    .select("date", "close")
    .with_columns(
        ewm_mean=pl.col("close").ewm_mean(com=7, ignore_nulls=True),   ❶
        rolling_mean=pl.col("close").rolling_mean(window_size=7),
        rolling_min=pl.col("close").rolling_min(window_size=7),
    )
)
stock
```

```
shape: (124, 5)
┌────────────┬────────────┬────────────┬──────────────┬─────────────┐
│ date       ┆ close      ┆ ewm_mean   ┆ rolling_mean ┆ rolling_min │
│ ---        ┆ ---        ┆ ---        ┆ ---          ┆ ---         │
│ date       ┆ f64        ┆ f64        ┆ f64          ┆ f64         │
╞════════════╪════════════╪════════════╪══════════════╪═════════════╡
│ 2023-01-03 ┆ 143.149994 ┆ 143.149994 ┆ null         ┆ null        │
│ 2023-01-04 ┆ 147.490005 ┆ 145.464667 ┆ null         ┆ null        │
│ 2023-01-05 ┆ 142.649994 ┆ 144.398755 ┆ null         ┆ null        │
│ 2023-01-06 ┆ 148.589996 ┆ 145.664782 ┆ null         ┆ null        │
│ 2023-01-09 ┆ 156.279999 ┆ 148.388917 ┆ null         ┆ null        │
│ …          ┆ …          ┆ …          ┆ …            ┆ …           │
│ 2023-06-26 ┆ 406.320007 ┆ 407.54911  ┆ 425.805716   ┆ 406.320007  │
│ 2023-06-27 ┆ 418.76001  ┆ 408.950473 ┆ 424.695718   ┆ 406.320007  │
│ 2023-06-28 ┆ 411.170013 ┆ 409.227915 ┆ 422.445718   ┆ 406.320007  │
│ 2023-06-29 ┆ 408.220001 ┆ 409.101926 ┆ 418.180006   ┆ 406.320007  │
│ 2023-06-30 ┆ 423.019989 ┆ 410.841684 ┆ 417.118574   ┆ 406.320007  │
└────────────┴────────────┴────────────┴──────────────┴─────────────┘
```

Because it's difficult to see the difference between these methods in a table, let's visualize it (see Figure 6-2).

```
from matplotlib.dates import DateFormatter
stock.plot.line(
    x="date",
    y=["close", "ewm_mean", "rolling_mean", "rolling_min"],
    xformatter=DateFormatter("%b %Y")
)
```

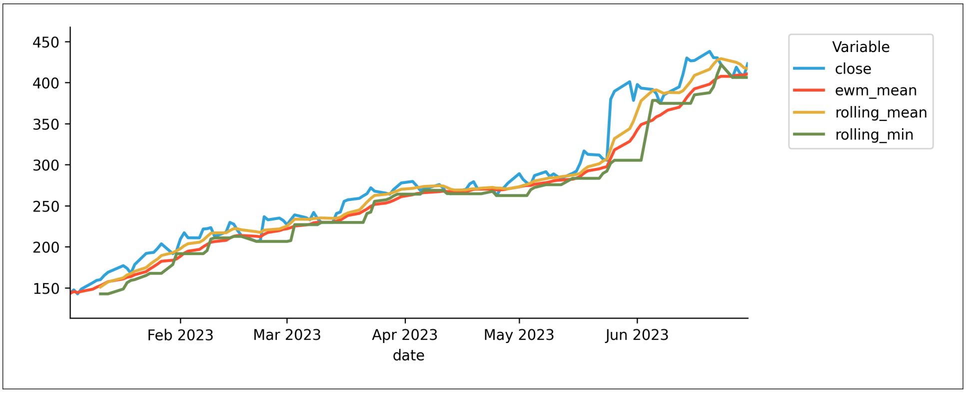


Figure 6-2. Several rolling statistics operations applied to stock data

In Chapter 17 you'll learn more about creating data visualizations with Polars.

Operations That Sort

Table 6-10 lists the methods that Polars provides for sorting expressions.

Table 6-10. Series-wise operations that sort

Method	Description
`Expr.arg_sort(…)`	Get the index values that would sort this column.
`Expr.shuffle(…)`	Shuffle the contents of this expression.
`Expr.sort(…)`	Sort this column.
`Expr.sort_by(…)`	Sort this column by the ordering of other columns.
`Expr.reverse()`	Reverse the selection.
`Expr.rank(…)`	Assign ranks to data, dealing with ties appropriately.

Sorting Expressions is Not That Common

In real-world data sets, a row often represents an observation or event. For that reason, you'll most likely want to sort entire rows so that the measurements of each observation or event stay together. The methods in this section, however, only deal with a single expression or column.

Let's apply those six methods on a couple of numbers. We've also added a column `y` to demonstrate the method `Expr.sort_by()`.

```
(
    pl.DataFrame({
        "x": [1, 3, None, 3, 7],
        "y": ["D", "I", "S", "C", "O"],
```

```
    })
    .with_columns(
        arg_sort=pl.col("x").arg_sort(),
        shuffle=pl.col("x").shuffle(seed=7),
        sort=pl.col("x").sort(nulls_last=True),
        sort_by=pl.col("x").sort_by("y"),
        reverse=pl.col("x").reverse(),
        rank=pl.col("x").rank(),
    )
)
```

```
shape: (5, 8)
┌──────┬─────┬──────────┬─────────┬──────┬─────────┬─────────┬──────┐
│ x    ┆ y   ┆ arg_sort ┆ shuffle ┆ sort ┆ sort_by ┆ reverse ┆ rank │
│ ---  ┆ --- ┆ ---      ┆ ---     ┆ ---  ┆ ---     ┆ ---     ┆ ---  │
│ i64  ┆ str ┆ u32      ┆ i64     ┆ i64  ┆ i64     ┆ i64     ┆ f64  │
╞══════╪═════╪══════════╪═════════╪══════╪═════════╪═════════╪══════╡
│ 1    ┆ D   ┆ 2        ┆ 1       ┆ 1    ┆ 3       ┆ 7       ┆ 1.0  │
│ 3    ┆ I   ┆ 0        ┆ null    ┆ 3    ┆ 1       ┆ 3       ┆ 2.5  │
│ null ┆ S   ┆ 1        ┆ 3       ┆ 3    ┆ 3       ┆ null    ┆ null │
│ 3    ┆ C   ┆ 3        ┆ 7       ┆ 7    ┆ 7       ┆ 3       ┆ 2.5  │
│ 7    ┆ O   ┆ 4        ┆ 3       ┆ null ┆ null    ┆ 1       ┆ 4.0  │
└──────┴─────┴──────────┴─────────┴──────┴─────────┴─────────┴──────┘
```

Other Operations

There's one nonreducing Series-wise operator that doesn't fall in any of the above categories (see Table 6-11).

Table 6-11. One other Series-wise operations

Method	Description
`Expr.rle_id()`	Map values to run IDs.

Let's apply `Expr.rle_id()` to a numerical Series:

```
(
    pl.DataFrame({"x": [33, 33, 27, 33, 60, 60, 60, 33, 60]})
    .with_columns(
        rle_id=pl.col("x").rle_id(),
    )
)
```

```
shape: (9, 2)
┌─────┬────────┐
│ x   ┆ rle_id │
│ --- ┆ ---    │
│ i64 ┆ u32    │
╞═════╪════════╡
│ 33  ┆ 0      │
│ 33  ┆ 0      │
│ 27  ┆ 1      │
```

```
│ 33  │ 2       │
│ 60  │ 3       │
│ 60  │ 3       │
│ 60  │ 3       │
│ 33  │ 4       │
│ 60  │ 5       │
└─────┴─────────┘
```

Series-Wise Operations that Summarize to One

We continue with Series-wise operations but in this section we're looking at operations which summarize all values in the Series into one value. Examples include the `Expr.mean()` method for computing the mean value and the `Expr.null_count()` method for counting the number of missing values.

In the next four subsections we look at operations that summarize to one using quantifiers, by computing statistics, by counting, and others.

Repeated Values

If you use an operation that summarizes to one value and you keep any of the original columns, the computed value gets repeated. For example, here the mean of the Series gets repeated four times:

```
(
    pl.DataFrame({"x": [1, 3, 3, 7]})
    .with_columns(
        mean=pl.col("x").mean(),
    )
)
```

```
shape: (4, 2)
┌─────┬──────┐
│ x   ┆ mean │
│ --- ┆ ---  │
│ i64 ┆ f64  │
╞═════╪══════╡
│ 1   ┆ 3.5  │
│ 3   ┆ 3.5  │
│ 3   ┆ 3.5  │
│ 7   ┆ 3.5  │
└─────┴──────┘
```

Because summarizing operations are most often used in an aggregation context, this is not really an issue. For example, when you compute the mean per group:

```
(
    pl.DataFrame({
        "cluster": ["a", "a", "b", "b"],
        "x": [1, 3, 3, 7]
```

```
        })
        .group_by("cluster")
        .agg(
            mean=pl.col("x").mean(),
        )
    )
```

```
shape: (2, 2)
┌─────────┬──────┐
│ cluster ┆ mean │
│ ---     ┆ ---  │
│ str     ┆ f64  │
╞═════════╪══════╡
│ b       ┆ 5.0  │
│ a       ┆ 2.0  │
└─────────┴──────┘
```

In the remainder of this chapter, we'll use the `df.select()` method to exclude the original columns and thus avoid repeated values.

Operations That Are Quantifiers

Using quantifiers allows you to summarize multiple Boolean values into one. Polars supports the universal and existential quantifiers via `Expr.all()` and `Expr.any()` (see Table 6-12).

Table 6-12. Series-wise operations that summarize to one value using quantifiers

Method	Description
`Expr.all(…)`	Return whether all values in the column are True.
`Expr.any(…)`	Return whether any of the values in the column are True.

Both methods accept one argument `ignore_nulls` that indicates whether missing values should be ignored. The following code snippet applies `Expr.all()` and `Expr.any()` to three Boolean columns, `x`, `y`, and `z`:

```
(
    pl.DataFrame({
        "x": [True, False, False],
        "y": [True, True, True],
        "z": [False, False, False],
        })
    .select(
        pl.all().all().name.suffix("_all"),
        pl.all().any().name.suffix("_any"),
    )
)
```

```
shape: (1, 6)
┌───────┬──────┬───────┬───────┬───────┬───────┐
│ x_all ┆ y_all ┆ z_all ┆ x_any ┆ y_any ┆ z_any │
│ ---   ┆ ---   ┆ ---   ┆ ---   ┆ ---   ┆ ---   │
│ bool  ┆ bool  ┆ bool  ┆ bool  ┆ bool  ┆ bool  │
╞═══════╪═══════╪═══════╪═══════╪═══════╪═══════╡
│ false ┆ true  ┆ false ┆ true  ┆ true  ┆ false │
└───────┴───────┴───────┴───────┴───────┴───────┘
```

Operations That Compute Statistics

Polars supports many methods to compute a variety of statistics of a numerical Series (see Table 6-13).

Table 6-13. Series-wise operations that summarize to one element by computing statistics

Method	Description
`Expr.entropy(…)`	Computes the entropy.
`Expr.kurtosis(…)`	Compute the kurtosis (Fisher or Pearson) of a dataset.
`Expr.max()`	Get maximum value.
`Expr.mean()`	Get mean value.
`Expr.median()`	Get median value using linear interpolation.
`Expr.min()`	Get minimum value.
`Expr.nan_max()`	Get maximum value, but propagate/poison encountered NaN values.
`Expr.nan_min()`	Get minimum value, but propagate/poison encountered NaN values.
`Expr.product()`	Compute the product of an expression.
`Expr.quantile(…)`	Get quantile value.
`Expr.skew(…)`	Compute the sample skewness of a data set.
`Expr.std(…)`	Get standard deviation.
`Expr.sum()`	Get sum value.
`Expr.var(…)`	Get variance.

In the following code snippet, we apply the methods `Expr.max()`, `Expr.mean()`, `Expr.quantile()`, `Expr.skew()`, `Expr.std()`, `Expr.sum()`, and `Expr.var()` to a million values. These values are sampled from a normal distribution with a mean of 5 and a standard deviation of 3.

```
samples = rng.normal(loc=5, scale=3, size=1_000_000)

(
    pl.DataFrame({"x": samples})
    .select(
        max=pl.col("x").max(),
        mean=pl.col("x").mean(),
        quantile=pl.col("x").quantile(quantile=0.95),
```

```
            skew=pl.col("x").skew(),
            std=pl.col("x").std(),
            sum=pl.col("x").sum(),
            var=pl.col("x").var(),
        )
    )

    shape: (1, 7)
    ┌───────────┬──────────┬──────────┬──────────┬──────────┬──────────┬──────────┐
    │ max       ┆ mean     ┆ quantile ┆ skew     ┆ std      ┆ sum      ┆ var      │
    │ ---       ┆ ---      ┆ ---      ┆ ---      ┆ ---      ┆ ---      ┆ ---      │
    │ f64       ┆ f64      ┆ f64      ┆ f64      ┆ f64      ┆ f64      ┆ f64      │
    ╞═══════════╪══════════╪══════════╪══════════╪══════════╪══════════╪══════════╡
    │ 20.752443 ┆ 4.994978 ┆ 9.931565 ┆ 0.003245 ┆ 2.999926 ┆ 4.9950e6 ┆ 8.999558 │
    └───────────┴──────────┴──────────┴──────────┴──────────┴──────────┴──────────┘
```

The other methods `Expr.entropy()`, `Expr.kurtosis()`, `Expr.median()`, `Expr.min()`, `Expr.nan_max()`, `Expr.nan_min()`, and `Expr.product()`, work similarly.

Operations That Count

Polars offers several methods for counting certain things (see Table 6-14).

Table 6-14. Series-wise operations that summarize to one element by counting

Method	Description
`Expr.approx_n_unique()`	Approximate count of unique values.
`Expr.count()`	Count the number of values in this expression.
`Expr.len()`	Count the number of values in this expression.
`Expr.n_unique()`	Count unique values.
`Expr.null_count()`	Count null values.

To demonstrate these methods, let's generate 1,729 random integers between 0 and 10,000 and make one value missing:

```
samples = pl.Series(rng.integers(low=0, high=10_000, size=1_729))
samples[403] = None ❶
df_ints = (
    pl.DataFrame({"x": samples})
    .with_row_index() ❷
)
df_ints.slice(400, 6) ❸

shape: (6, 2)
┌───────┬─────┐
│ index ┆ x   │
│ ---   ┆ --- │
│ u32   ┆ i64 │
╞═══════╪═════╡
│ 400   ┆ 807 │
```

```
│ 401   │ 8634 │
│ 402   │ 2109 │
│ 403   │ null │
│ 404   │ 1740 │
│ 405   │ 3333 │
└───────┴──────┘
```

❶ The 403rd element is made missing.

❷ The DataFrame method `df.with_row_index()` adds a row index as the first column.

❸ We use the DataFrame method `df.slice()` to display a subset of the rows.

Let's apply these five methods to the column `x`:

```
df_ints.select(
    approx_n_unique=pl.col("x").approx_n_unique(),
    count=pl.col("x").count(),
    len=pl.col("x").len(),
    n_unique=pl.col("x").n_unique(),
    null_count=pl.col("x").null_count(),
)
```

```
shape: (1, 5)
┌────────────────┬───────┬──────┬──────────┬────────────┐
│ approx_n_unique ┆ count ┆ len  ┆ n_unique ┆ null_count │
│ ---            ┆ ---   ┆ ---  ┆ ---      ┆ ---        │
│ u32            ┆ u32   ┆ u32  ┆ u32      ┆ u32        │
╞════════════════╪═══════╪══════╪══════════╪════════════╡
│ 1572           ┆ 1728  ┆ 1729 ┆ 1575     ┆ 1          │
└────────────────┴───────┴──────┴──────────┴────────────┘
```

Other Operations

There are eight Series-wise operators that summarize to one that don't fall in any of the above categories (see Table 6-15).

Table 6-15. Several miscellaneous Series-wise operations that summarize to one element

Method	Description
`Expr.arg_max()`	Get the index of the maximal value.
`Expr.arg_min()`	Get the index of the minimal value.
`Expr.first()`	Get the first value.
`Expr.get(…)`	Return a single value by index.
`Expr.implode()`	Aggregate values into a list.
`Expr.last()`	Get the last value.
`Expr.lower_bound()`	Calculate the lower bound.

Method	Description
`Expr.upper_bound()`	Calculate the upper bound.

Below we apply the methods `Expr.arg_min()`, `Expr.first()`, `Expr.get()`, `Expr.implode()`, `Expr.last()`, and `Expr.upper_bound()` to the same values as the previous section. The method `Expr.arg_max()` is similar to `Expr.arg_min()`, and `Expr.lower_bound()` is similar to `Expr.upper_bound()`.

```
df_ints.select(
    arg_min=pl.col("x").arg_min(),
    first=pl.col("x").first(),
    get=pl.col("x").get(403),  ❶
    implode=pl.col("x").implode(),
    last=pl.col("x").last(),
    upper_bound=pl.col("x").upper_bound(),
)
```

```
shape: (1, 6)
┌─────────┬───────┬──────┬───────────────────┬──────┬─────────────────────┐
│ arg_min ┆ first ┆ get  ┆ implode           ┆ last ┆ upper_bound         │
│ ---     ┆ ---   ┆ ---  ┆ ---               ┆ ---  ┆ ---                 │
│ u32     ┆ i64   ┆ i64  ┆ list[i64]         ┆ i64  ┆ i64                 │
╞═════════╪═══════╪══════╪═══════════════════╪══════╪═════════════════════╡
│ 0       ┆ 0     ┆ null ┆ [0, 7245, … 3723] ┆ 3723 ┆ 9223372036854775807 │
└─────────┴───────┴──────┴───────────────────┴──────┴─────────────────────┘
```

❶ The result is `null` because, in the previous section, we made the 403rd element missing.

Series-Wise Operations that Summarize to One or More

Besides Series-wise operations that summarize to one, there are also some that summarize to one or more. The actual length of the output Series depends on the values.

In the next four subsections, we cover Series-wise operations that summarize to one or more based on unique values, by selecting, by dropping missing values, and others.

Operations Related to Unique Values

Table 6-16 lists four methods related with unique values.

Table 6-16. Several Series-wise operations that summarize to one or more elements based on unique values

Method	Description
`Expr.arg_unique()`	Get index of first unique value.
`Expr.unique(…)`	Get unique values of this expression.

Method	Description
`Expr.unique_counts()`	Return a count of the unique values in the order of appearance.
`Expr.value_counts(…)`	Count the occurrences of unique values.

Let's apply those four methods to a Series of strings:

```
(
    pl.DataFrame({"x": ["A", "C", "D", "C"]})
    .select(
        arg_unique=pl.col("x").arg_unique(),
        unique=pl.col("x").unique(maintain_order=True),  ❶
        unique_counts=pl.col("x").unique_counts(),
        value_counts=pl.col("x").value_counts(),  ❷
    )
)
```

```
shape: (3, 4)
┌────────────┬────────┬───────────────┬──────────────┐
│ arg_unique ┆ unique ┆ unique_counts ┆ value_counts │
│ ---        ┆ ---    ┆ ---           ┆ ---          │
│ u32        ┆ str    ┆ u32           ┆ struct[2]    │
╞════════════╪════════╪═══════════════╪══════════════╡
│ 0          ┆ A      ┆ 1             ┆ {"C",2}      │
│ 1          ┆ C      ┆ 2             ┆ {"D",1}      │
│ 2          ┆ D      ┆ 1             ┆ {"A",1}      │
└────────────┴────────┴───────────────┴──────────────┘
```

❶ Maintaining the order of the values is computationally more intensive.

❷ The result of `Expr.value_counts()` is of data type `pl.Struct`; a combination of `Expr.unique()` and `Expr.unique_counts()`, though not necessarily in the same order.

Operations That Select

Table 6-17 lists several methods for selecting specific elements based on their position or value.

Table 6-17. Several Series-wise operations that summarize to one or more elements by selecting

Method	Description
`Expr.bottom_k(…)`	Return the k smallest elements.
`Expr.head(…)`	Get the first n rows.
`Expr.limit(…)`	Get the first n rows (alias for Expr.head()).
`Expr.sample(…)`	Sample from this expression.
`Expr.slice(…)`	Get a slice of this expression.

Method	Description
`Expr.tail(…)`	Get the last n rows.
`Expr.gather(…)`	Take values by index.
`Expr.gather_every(…)`	Take every nth value in the Series and return as a new Series.
`Expr.top_k(…)`	Return the k largest elements.

The following code snippet applies the methods `Expr.bottom_k()`, `Expr.head()`, `Expr.sample()`, `Expr.slice()`, `Expr.gather()`, `Expr.gather_every()`, and `Expr.top_k()` to the samples generated earlier.

The method `Expr.limit()` is an alias for `Expr.head()`. The method `Expr.tail()` works just like `Expr.head()`, except it starts at the bottom.

```
df_ints.select(
    bottom_k=pl.col("x").bottom_k(7),  ❶
    head=pl.col("x").head(7),
    sample=pl.col("x").sample(7),
    slice=pl.col("x").slice(400, 7),
    gather=pl.col("x").gather([1, 1, 2, 3, 5, 8, 13]),
    gather_every=pl.col("x").gather_every(247),  ❷
    top_k=pl.col("x").top_k(7),
)
```

```
shape: (7, 7)
┌──────────┬──────┬────────┬───────┬────────┬──────────────┬───────┐
│ bottom_k ┆ head ┆ sample ┆ slice ┆ gather ┆ gather_every ┆ top_k │
│ ---      ┆ ---  ┆ ---    ┆ ---   ┆ ---    ┆ ---          ┆ ---   │
│ i64      ┆ i64  ┆ i64    ┆ i64   ┆ i64    ┆ i64          ┆ i64   │
╞══════════╪══════╪════════╪═══════╪════════╪══════════════╪═══════╡
│ null     ┆ 0    ┆ 6871   ┆ 807   ┆ 7245   ┆ 0            ┆ 9998  │
│ 0        ┆ 7245 ┆ 2202   ┆ 8634  ┆ 7245   ┆ 8680         ┆ 9988  │
│ 1        ┆ 5227 ┆ 7328   ┆ 2109  ┆ 5227   ┆ 8483         ┆ 9988  │
│ 6        ┆ 2747 ┆ 1648   ┆ null  ┆ 2747   ┆ 8358         ┆ 9986  │
│ 7        ┆ 9816 ┆ 5761   ┆ 1740  ┆ 2657   ┆ 1805         ┆ 9985  │
│ 10       ┆ 2657 ┆ 9315   ┆ 3333  ┆ 5393   ┆ 3638         ┆ 9979  │
│ 21       ┆ 4578 ┆ 8370   ┆ 788   ┆ 8203   ┆ 5843         ┆ 9975  │
└──────────┴──────┴────────┴───────┴────────┴──────────────┴───────┘
```

❶ Note that nulls are first.

❷ Has to match a height of 7, otherwise you get an error saying that the lengths don't match. In this example, taking every 247th value from a Series of length 1,729 yields 7 values.

Operations That Drop Missing Values

Table 6-18 lists two methods for dropping missing values: `Expr.drop_nans()` and `Expr.drop_nulls()`.

Table 6-18. Several Series-wise operations that summarize to one or more elements by dropping missing values

Method	Description
`Expr.drop_nans()`	Drop floating point NaN values.
`Expr.drop_nulls()`	Drop all null values.

Here's how you can apply both methods:

```
x = [None, 1, 2, 3, np.NaN]
(
    pl.DataFrame({"x": x})
    .select(
        drop_nans=pl.col("x").drop_nans(),
        drop_nulls=pl.col("x").drop_nulls()
    )
)
```

```
shape: (4, 2)
┌───────────┬────────────┐
│ drop_nans ┆ drop_nulls │
│ ---       ┆ ---        │
│ f64       ┆ f64        │
╞═══════════╪════════════╡
│ null      ┆ 1.0        │
│ 1.0       ┆ 2.0        │
│ 2.0       ┆ 3.0        │
│ 3.0       ┆ NaN        │
└───────────┴────────────┘
```

Other Operations

There are six Series-wise operators that summarize to one or more that don't fall in any of the above categories (see Table 6-19).

Table 6-19. Miscellaneous Series-wise operations that summarize to one or more elements

Method	Description
`Expr.arg_true()`	Return indices where expression evaluates True.
`Expr.flatten()`	Flatten a list or string column.
`Expr.mode()`	Compute the most occurring value(s).
`Expr.reshape(…)`	Reshape this Expr to a flat Series or a Series of Lists.
`Expr.rle()`	Get the lengths of runs of identical values.
`Expr.search_sorted(…)`	Find indices where elements should be inserted to maintain order.

Below we apply the methods `Expr.arg_true()`, `Expr.mode()`, `Expr.reshape()`, `Expr.rle()`, and `Expr.search_sorted()` to an unsorted Series of integers. We demonstrate the methods separately, because they construct Series of different lengths.

First, the method `Expr.arg_true()` can be applied as follows:

```
numbers = [33, 33, 27, 33, 60, 60, 60, 33, 60]

(
    pl.DataFrame({"x": numbers})
    .select(
        arg_true=(pl.col("x") >= 60).arg_true(),  ❶
    )
)
```

```
shape: (4, 1)
┌──────────┐
│ arg_true │
│ ---      │
│ u32      │
╞══════════╡
│ 4        │
│ 5        │
│ 6        │
│ 8        │
└──────────┘
```

❶ We use the greater-than operator (>=) to get a Boolean Series first. You'll learn more about this and other comparison operators in the next chapter.

Second, the method `Expr.mode()` can be applied as follows:

```
(
    pl.DataFrame({"x": numbers})
    .select(
        mode=pl.col("x").mode(),
    )
)
```

```
shape: (2, 1)
┌──────┐
│ mode │
│ ---  │
│ i64  │
╞══════╡
│ 60   │
│ 33   │
└──────┘
```

Third, the method `Expr.reshape()` can be applied as follows:

```
(
    pl.DataFrame({"x": numbers})
    .select(
        reshape=pl.col("x").reshape((3, 3)),  ❶
```

```
    )
)
```

```
shape: (3, 1)
┌──────────────┐
│ reshape      │
│ ---          │
│ list[i64]    │
╞══════════════╡
│ [33, 33, 27] │
│ [33, 60, 60] │
│ [60, 33, 60] │
└──────────────┘
```

❶ The total number of elements needs to remain the same. For example, it's not possible to reshape this into 5 rows where the last row is a `pl.List` of one element.

Fourth, the method `Expr.rle()` can be applied as follows:

```
(
    pl.DataFrame({"x": numbers})
    .select(
        rle=pl.col("x").rle(),  ❶
    )
)
```

```
shape: (6, 1)
┌───────────┐
│ rle       │
│ ---       │
│ struct[2] │
╞═══════════╡
│ {2,33}    │
│ {1,27}    │
│ {1,33}    │
│ {3,60}    │
│ {1,33}    │
│ {1,60}    │
└───────────┘
```

❶ Compare with `Expr.rle_id()` discussed earlier in this chapter.

Finally, the method `Expr.search_sorted()` can be applied as follows:

```
(
    pl.DataFrame({"x": numbers})
    .select(
        rle=pl.col("x").sort().search_sorted(42),  ❶
    )
)
```

```
shape: (1, 1)
┌─────┐
│ rle │
│ --- │
│ u32 │
╞═════╡
│ 5   │
└─────┘
```

❶ The method `Expr.search_sorted()` is probably most useful on a sorted Series.

Series-Wise Operations that Extend

There are only two operations that can extend the length of a Series (see Table 6-20).

Table 6-20. Two Series-wise operations that extend

Method	Description
`Expr.explode()`	Explode a list expression.
`Expr.extend_constant(…)`	Extend the Series with a contant value.

Below, we use the method `Expr.explode()` to turn a List Series into a regular, flat Series:

```
(
    pl.DataFrame({
        "x": [["a", "b"], ["c", "d"]],
    })
    .select(
        explode=pl.col("x").explode()
    )
)
```

```
shape: (4, 1)
┌─────────┐
│ explode │
│ ---     │
│ str     │
╞═════════╡
│ a       │
│ b       │
│ c       │
│ d       │
└─────────┘
```

We demonstrated the method `Expr.extend_constant()` in the beginning of this chapter.

Conclusion

In this chapter, you've learned about many different methods to continue expressions with additional operations. These operations were organized according to the length of the Series they construct. In the next chapter you're going to learn how to combine expressions.

CHAPTER 7
Combining Expressions

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

Now that you understand the fundamentals of expressions and know various methods to continue them, it's time to learn how to combine them.

Combining expressions is necessary whenever the Series you want to construct is based on more than one value or column. This happens to be the case more often than you might think: for example, when you want to compute the ratio between two float columns, filter rows based on multiple conditions, or concatenate multiple string columns into one.

In fact, you've already combined expressions several times in the previous chapters. Let's look at an example from Chapter 5 to refresh your memory:

```
fruit = pl.read_csv("data/fruit.csv")
fruit.filter(pl.col("is_round") & (pl.col("weight") > 1000))
```

```
shape: (2, 5)
┌─────────────┬────────┬───────┬──────────┬────────┐
│ name        ┆ weight ┆ color ┆ is_round ┆ origin │
```

```
│ ---        ┆ ---  ┆ ---    ┆ ---   ┆ ---    │
│ str        ┆ i64  ┆ str    ┆ bool  ┆ str    │
╞════════════╪══════╪════════╪═══════╪════════╡
│ Cantaloupe ┆ 2500 ┆ orange ┆ true  ┆ Africa │
│ Watermelon ┆ 5000 ┆ green  ┆ true  ┆ Africa │
└────────────┴──────┴────────┴───────┴────────┘
```

This code combines, in two steps, two existing columns (`is_round` and `weight`) and one value (`1000`) into one expression. The `df.filter()` method then uses this expression to filter rows.

Because of how the parentheses are organized, the comparison operator greater-than (`>`) combines the `weight` column and the value `1000`. When the value is larger, it produces the Boolean `True`. Second, the Boolean AND operator (`&`) combines the `is_round` column and the Series constructed in the first step. Only when they're both `True` is the output `True`. The `df.filter()` method interprets `True` as "keep this row."

That's only the tip of the iceberg when it comes to combining expressions in Polars. In this chapter you'll learn about the difference between inline operators and method chaining. You'll also learn how to combine expressions:

- Through arithmetic, such as adding and multiplying
- By comparing, such as greater than and equals
- With Boolean algebra, such as conjunction and negation
- Via bitwise operations, such as AND and XOR
- Using a variety of module-level functions

Inline Operators Versus Methods

In the previous two chapters, you used method chaining to continue expressions. To combine expressions, you can often use inline operators instead of method chaining. Both approaches produce the same result, as illustrated in Figure 7-1.

> While every inline operator has a corresponding `Expr` method, not every method (or function) to combine expressions has a corresponding inline operator. Examples are the method `Expr.dot()` and the function `pl.concat_list()`.

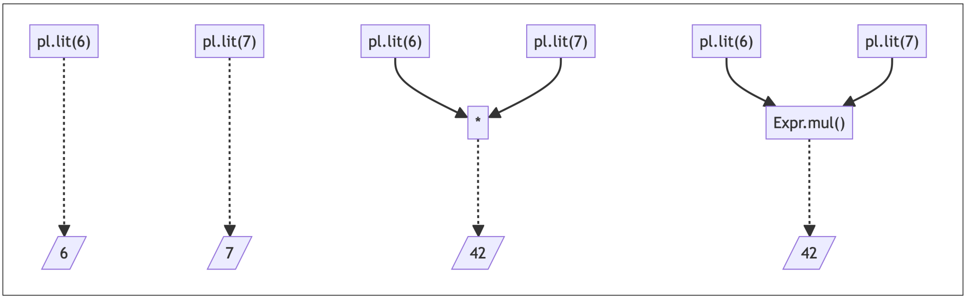


Figure 7-1. Combining expressions using inline operators or method chaining produce the same result

To illustrate this in code, the following snippet multiplies two columns `i` and `j` using both approaches:

```
(
    pl.DataFrame({
        "i": [6, 0, 2, 2.5],
        "j": [7, 1, 2, 3]
    })
    .with_columns(
        (pl.col("i") * pl.col("j")).alias("*"),
        pl.col("i").mul(pl.col("j")).alias("Expr.mul()")
    )
)
```

```
shape: (4, 4)
┌─────┬─────┬──────┬────────────┐
│ i   ┆ j   ┆ *    ┆ Expr.mul() │
│ --- ┆ --- ┆ ---  ┆ ---        │
│ f64 ┆ i64 ┆ f64  ┆ f64        │
╞═════╪═════╪══════╪════════════╡
│ 6.0 ┆ 7   ┆ 42.0 ┆ 42.0       │
│ 0.0 ┆ 1   ┆ 0.0  ┆ 0.0        │
│ 2.0 ┆ 2   ┆ 4.0  ┆ 4.0        │
│ 2.5 ┆ 3   ┆ 7.5  ┆ 7.5        │
└─────┴─────┴──────┴────────────┘
```

As expected, both approaches yield the same result. We think that the spaces around the inline operator make it clear that you're combining two expressions into a new one. For this reason, we generally recommend you use the corresponding inline operator, if one exists. However, note that with the inline-operator approach, you need to wrap the new expression in parentheses in order to continue with additional methods.

That's it for multiplying expressions. Now let's look at some other arithmetic operations.

Arithmetic Operations

Arithmetic is the cornerstone of any data-related task. You can perform arithmetic with both expressions and Python values.

Here's a fruity example that divides the `weight` column (an expression) by `1000` (a Python integer):

```
fruit.select(
    pl.col("name"),
    (pl.col("weight") / 1000)
)
```

```
shape: (10, 2)
┌────────────┬────────┐
│ name       ┆ weight │
│ ---        ┆ ---    │
│ str        ┆ f64    │
╞════════════╪════════╡
│ Avocado    ┆ 0.2    │
│ Banana     ┆ 0.12   │
│ Blueberry  ┆ 0.001  │
│ Cantaloupe ┆ 2.5    │
│ Cranberry  ┆ 0.002  │
│ Elderberry ┆ 0.001  │
│ Orange     ┆ 0.13   │
│ Papaya     ┆ 1.0    │
│ Peach      ┆ 0.15   │
│ Watermelon ┆ 5.0    │
└────────────┴────────┘
```

Table 7-1 lists all inline operators and methods used to perform arithmetic in Polars.

Table 7-1. Inline operators and their corresponding methods for performing arithmetic

Inline Operator	Method	Description
`+`	`Expr.add(…)`	Addition
`-`	`Expr.sub(…)`	Subtraction
`*`	`Expr.mul(…)`	Multiplication
`/`	`Expr.truediv(…)`	Division
`//`	`Expr.floordiv(…)`	Floor division
`**`	`Expr.pow(…)`	Power
`%`	`Expr.mod(…)`	Modulus
N/A	`Expr.dot(…)`	Dot product

The following code snippet demonstrates how to use these inline operators on two integer columns (`i` and `j`). Polars automatically creates a float column when needed.

Because the method `Expr.dot()` has no corresponding inline operator, we use the method instead.

```
pl.Config(float_precision=2, tbl_cell_numeric_alignment="RIGHT") ❶

(
    pl.DataFrame({
        "i": [0, 2, 2, -2, -2],
        "j": [1, 2, 3, 4, -5]
    })
    .with_columns(
        (pl.col("i") + pl.col("j")).alias("i + j"),
        (pl.col("i") - pl.col("j")).alias("i - j"),
        (pl.col("i") * pl.col("j")).alias("i * j"),
        (pl.col("i") / pl.col("j")).alias("i / j"),
        (pl.col("i") // pl.col("j")).alias("i // j"),
        (pl.col("i") ** pl.col("j")).alias("i ** j"),
        (pl.col("j") % 2).alias("j % 2"), ❷
        pl.col("i").dot(pl.col("j")).alias("i · j"), ❸
    )
)
```

```
shape: (5, 10)
┌─────┬─────┬───────┬───────┬───────┬───────┬────────┬────────┬───────┬───────┐
│   i ┆   j ┆ i + j ┆ i - j ┆ i * j ┆ i / j ┆ i // j ┆ i ** j ┆ j % 2 ┆ i · j │
│ --- ┆ --- ┆   --- ┆   --- ┆   --- ┆   --- ┆    --- ┆    --- ┆   --- ┆   --- │
│ i64 ┆ i64 ┆   i64 ┆   i64 ┆   i64 ┆   f64 ┆    i64 ┆    f64 ┆   i64 ┆   i64 │
╞═════╪═════╪═══════╪═══════╪═══════╪═══════╪════════╪════════╪═══════╪═══════╡
│   0 ┆   1 ┆     1 ┆    -1 ┆     0 ┆  0.00 ┆      0 ┆   0.00 ┆     1 ┆    12 │
│   2 ┆   2 ┆     4 ┆     0 ┆     4 ┆  1.00 ┆      1 ┆   4.00 ┆     0 ┆    12 │
│   2 ┆   3 ┆     5 ┆    -1 ┆     6 ┆  0.67 ┆      0 ┆   8.00 ┆     1 ┆    12 │
│  -2 ┆   4 ┆     2 ┆    -6 ┆    -8 ┆ -0.50 ┆     -1 ┆  16.00 ┆     0 ┆    12 │
│  -2 ┆  -5 ┆    -7 ┆     3 ┆    10 ┆  0.40 ┆      0 ┆  -0.03 ┆     1 ┆    12 │
└─────┴─────┴───────┴───────┴───────┴───────┴────────┴────────┴───────┴───────┘
```

❶ We're temporarily changing these two display settings to fit this wide DataFrame on the page.

❷ The modulo operator (%) accepts a second expression, just like the other arithmetic operations.

❸ Because the dot product doesn't have a corresponding inline operator we use a dot character (·) in the column name. The dot product is also the only operation here that uses entire columns instead of elements; that's why the last column contains the same value (namely, 12) five times.

Comparison Operations

Which of these experiments produced a significant result? Which movies released in the 90s have an IMDB score of 8.7 or higher? Are these voltages within the allowed range? These are all data-related questions that involve comparing values.

Comparing values in Polars works pretty much the same as in Python, except that they cannot be chained (explained below).

```
pl.select(pl.lit("a") > pl.lit("b"))
```

```
shape: (1, 1)
┌─────────┐
│ literal │
│ ---     │
│ bool    │
╞═════════╡
│ false   │
└─────────┘
```

You'll most likely compare two numeric columns (such as `pl.Int8` and `pl.Float64`). You can also compare strings and temporal data types (which includes `pl.Date`, `pl.DateTime`, `pl.Duration`, and `pl.Time`).

A comparison always constructs a Boolean Series. This Series can be added as a column to a DataFrame, but it's more often used for filtering rows, using `df.filter()`, or in conditional expressions using, `pl.when()`.

Here's an example using the DataFrame `fruit` that compares the column `weight` with the value `1000` using the greater-than-or-equal operator (>=). The constructed Boolean Series is used to filter the rows.

```
(
    fruit.select(
        pl.col("name"),
        pl.col("weight"),
    )
    .filter(pl.col("weight") >= 1000)
)
```

```
shape: (3, 2)
┌────────────┬────────┐
│ name       │ weight │
│ ---        │ ---    │
│ str        │ i64    │
╞════════════╪════════╡
│ Cantaloupe │ 2500   │
│ Papaya     │ 1000   │
│ Watermelon │ 5000   │
└────────────┴────────┘
```

Table 7-2 lists all inline operators and methods for performing comparisons in Polars.

Table 7-2. Inline operators and their methods for performing comparisons

Inline Operator	Method	Description
`<`	`Expr.lt(…)`	Less than
`<=`	`Expr.le(…)`	Less than or equal
`==`	`Expr.eq(…)`	Equal
`>=`	`Expr.ge(…)`	Greater than or equal
`>`	`Expr.gt(…)`	Greater than
`!=`	`Expr.ne(…)`	Not equal

Chaining Comparisons

In Python itself, you can chain the inline operators listed in Table 7-2. Consider the following example, which uses the less-than operator (`<`) twice to test whether the value of `x` is between 3 and 5:

```
x = 4
3 < x < 5
```

```
True
```

With Polars, however, if you do this you get an error:

```
pl.select(pl.lit(3) < pl.lit(x) < pl.lit(5))
```

```
TypeError: the truth value of an Expr is ambiguous

You probably got here by using a Python standard library function instea
d of the native expressions API.
Here are some things you might want to try:
- instead of `pl.col('a') and pl.col('b')`, use `pl.col('a') & pl.col('b
')`
- instead of `pl.col('a') in [y, z]`, use `pl.col('a').is_in([y, z])`
- instead of `max(pl.col('a'), pl.col('b'))`, use `pl.max_horizontal(pl.
col('a'), pl.col('b'))`
```

A solution is to perform two separate comparisons and combine them using the AND (`&`) operator:

```
pl.select((pl.lit(3) < pl.lit(x)) & (pl.lit(x) < pl.lit(5))).item()
```

```
True
```

You'll learn more about the AND (`&`) operator in the next section, where we discuss combining expressions using Boolean algebra. Another solution, in this particular case, is to use the method `Expr.is_between()`:

```
pl.select(pl.lit(x).is_between(3, 5)).item()
```

```
True
```

Let's apply a couple of comparison operators to two numerical columns a and b:

```
(
    pl.DataFrame({
        "a": [-273.15, 0, 42, 100],
        "b": [1.4142, 2.7183, 42, 3.1415]
    })
    .with_columns(
        (pl.col("a") == pl.col("b")).alias("a == b"),
        (pl.col("a") <= pl.col("b")).alias("a <= b"),
        (pl.all() > 0).name.suffix(" > 0"),
        ((pl.col("b") - pl.lit(2).sqrt()).abs() < 1e-3).alias("b ≈ √2"), ❶
        ((1 < pl.col("b")) & (pl.col("b") < 3)).alias("1 < b < 3")
    )
)
```

```
shape: (4, 8)
┌─────────┬────────┬────────┬────────┬───────┬───────┬────────┬───────────┐
│ a       ┆ b      ┆ a == b ┆ a <= b ┆ a > 0 ┆ b > 0 ┆ b ≈ √2 ┆ 1 < b < 3 │
│ ---     ┆ ---    ┆ ---    ┆ ---    ┆ ---   ┆ ---   ┆ ---    ┆ ---       │
│ f64     ┆ f64    ┆ bool   ┆ bool   ┆ bool  ┆ bool  ┆ bool   ┆ bool      │
╞═════════╪════════╪════════╪════════╪═══════╪═══════╪════════╪═══════════╡
│ -273.15 ┆ 1.4142 ┆ false  ┆ true   ┆ false ┆ true  ┆ true   ┆ true      │
│ 0.0     ┆ 2.7183 ┆ false  ┆ true   ┆ false ┆ true  ┆ false  ┆ true      │
│ 42.0    ┆ 42.0   ┆ true   ┆ true   ┆ true  ┆ true  ┆ false  ┆ false     │
│ 100.0   ┆ 3.1415 ┆ false  ┆ false  ┆ true  ┆ true  ┆ false  ┆ false     │
└─────────┴────────┴────────┴────────┴───────┴───────┴────────┴───────────┘
```

❶ Here we use both arithmetic and comparison to combine expressions.

The following code snippets demonstrates a few more comparisons between different data types. Two of those are not allowed: String with number and DateTime with Time.

```
pl.select(
    bool_num=pl.lit(True) > 0,
    time_time=pl.time(23, 58) > pl.time(0, 0),
    datetime_date=pl.datetime(1969, 7, 21, 2, 56) < pl.date(1976, 7, 20),
    str_num=pl.lit("5") < pl.lit(3).cast(pl.String), ❶
    datetime_time=pl.datetime(1999, 1, 1).dt.time() != pl.time(0, 0), ❷
).transpose(include_header=True,
            header_name="comparison",
            column_names=["allowed"])
```

```
shape: (5, 2)
┌───────────────┬─────────┐
│ comparison    ┆ allowed │
│ ---           ┆ ---     │
│ str           ┆ bool    │
╞═══════════════╪═════════╡
│ bool_num      ┆ true    │
│ time_time     ┆ true    │
│ datetime_date ┆ true    │
```

```
│ str_num       │ false   │
│ datetime_time │ false   │
└───────────────┴─────────┘
```

❶ You cannot compare a String and a number. A solution is to first cast the number to String using `Expr.cast(pl.String)`.

❷ You also cannot compare a DateTime and a Time. A solution is to first extract the Time component from the DateTime using the method `Expr.dt.time()`.

Boolean Algebra Operations

In the previous section, we combined two comparison expressions to check whether the value of `x` is between two values. Let's use that example again, but set the value of `x` to `7`. We'll also assign the two comparison expressions to two variables ,`p` and `q`.

```
x = 7
p = pl.lit(3) < pl.lit(x)  # True
q = pl.lit(x) < pl.lit(5)  # False
pl.select(p & q).item()
```

```
False
```

We combine the expressions `p` and `q` using the Boolean operator AND (`&`), which evaluates to `True` if and only if both `p` and `q` are `True`. Since `q` is `False` in this case, the result is `False`. This Boolean operation is known as *conjunction*.

Conjunction is one of the three basic operations of Boolean algebra: *conjunction* (`&`), *disjunction* (`|`), and *negation* (`~`).[1] With these three basic operations you can create any *secondary* Boolean operation.

Polars provides one secondary operation: "exclusive or" (`^`). Table 7-3 lists the four Boolean operations with their inline operators and methods.

Table 7-3. Inline operators and their corresponding methods for performing Boolean operations

Operation	Inline Operator	Method	Description
Conjunction	`&`	`Expr.and_(…)`	Logical AND
Disjunction	`|`	`Expr.or_(…)`	Logical OR
Negation	`~`	`Expr.not_()`	Logical NOT
Exclusive OR	`^`	`Expr.xor(…)`	Logical XOR

1 Because negation (~) operates on a single expression, it's not combining expressions, but we're still discussing it here. It's only logical.

Ugly Underscores

In Python itself, `and`, `or`, and `not` are reserved keywords. That's why the first three methods listed in Table 7-3 have underscores (_) at the end. They're ugly, but you'll most likely use the corresponding inline operators (`&`, `|`, and `~`) anyway.

The conjection operation results in `True` only if both expressions are `True`. The following code snippet applies six Boolean operations (the four listed in Table 7-3 and two bonus operations[2] NAND and NOR) to all possible combinations of `p` and `q`. The output is known as a *truth table*.

```
(
    pl.DataFrame({
        "p": [True, True, False, False],
        "q": [True, False, True, False]
    })
    .with_columns(
        (pl.col("p") & pl.col("q")).alias("p & q"),
        (pl.col("p") | pl.col("q")).alias("p | q"),
        (~pl.col("p")).alias("~p"),
        (pl.col("p") ^ pl.col("q")).alias("p ^ q"),
        (~(pl.col("p") & pl.col("q"))).alias("p ↑ q"),  ❶
        ((pl.col("p").or_(pl.col("q"))).not_()).alias("p ↓ q")  ❷
    )
)
```

```
shape: (4, 8)
```

p --- bool	q --- bool	p & q --- bool	p | q --- bool	~p --- bool	p ^ q --- bool	p ↑ q --- bool	p ↓ q --- bool
true	true	true	true	false	false	false	false
true	false	false	true	false	true	true	false
false	true	false	true	true	true	true	false
false	false	false	false	true	false	true	true

❶ The NAND (NOT AND) operator is not part of Polars, but it can be emulated by combining the NOT (`~`) and the AND (`&`) operators.

❷ The same holds for the NOR (NOT OR) operator. Here we use an alternative syntax with methods instead of inline operators.

2 NAND stands for NOT AND. NOR stands for NOT OR.

Being able to combine Boolean expressions via these Boolean operations allows you to express complex relationships between expressions. In the next section we're going to apply the same methods and inline operations to integers instead of Booleans.

Bitwise Operations

You can also apply the AND (`&`), OR (`|`), XOR (`^`), and NOT(`~`) operators to integers. In that case, these operators perform bitwise operations.[3]

Here's an example that applies the bitwise OR operator (`|`) to the values `10` and `34`, which yields, logically[4], `42`:

```
pl.select(pl.lit(10) | pl.lit(34)).item()
```

```
42
```

Under the hood, Polars is applying the OR operator to each pair of bits that makes up the numbers 10 and 34. The output bit is `1` when at least one input bit is `1`:

```
   00001010 (decimal 10)
OR 00100010 (decimal 34)
=  00101010 (decimal 42)
```

So `10 | 34` is `42`, because in either 10 or 34, the second, fourth, and sixth bits from the right are all `1`. You can think of these bits as a sequence of Booleans—it's the same logic.

Table 7-4 lists the four bitwise operations and their inline operators and methods.

Table 7-4. Inline operators and their methods for performing bitwise operations.

Inline Operator	Method	Description
`&`	`Expr.and_(…)`	Bitwise AND
`|`	`Expr.or_(…)`	Bitwise OR
`~`	`Expr.not_(…)`	Bitwise NOT
`^`	`Expr.xor(…)`	Bitwise XOR

The following code snippet applies the bitwise operations listed in Table 7-4 to a couple of integers:

```
bits = (
    pl.DataFrame({
        "x": [1, 1, 0, 0, 7, 10],
        "y": [1, 0, 1, 0, 2, 34]
```

3 Bitwise operations are perhaps a bit niche, but this is *The Definitive Guide* after all.

4 See Douglas Adams' *The Hitchhiker's Guide to the Galaxy* for a comprehensive explanation.

```
    }, schema={"x": pl.UInt8, "y": pl.UInt8})  ❶
    .with_columns(
        (pl.col("x") & pl.col("y")).alias("x & y"),
        (pl.col("x") | pl.col("y")).alias("x | y"),
        (~pl.col("x")).alias("~x"),
        (pl.col("x") ^ pl.col("y")).alias("x ^ y"),
    )
)
bits
```

```
shape: (6, 6)
┌─────┬─────┬───────┬───────┬─────┬───────┐
│ x   ┆ y   ┆ x & y ┆ x | y ┆ ~x  ┆ x ^ y │
│ --- ┆ --- ┆ ---   ┆ ---   ┆ --- ┆ ---   │
│ u8  ┆ u8  ┆ u8    ┆ u8    ┆ u8  ┆ u8    │
╞═════╪═════╪═══════╪═══════╪═════╪═══════╡
│ 1   ┆ 1   ┆ 1     ┆ 1     ┆ 254 ┆ 0     │
│ 1   ┆ 0   ┆ 0     ┆ 1     ┆ 254 ┆ 1     │
│ 0   ┆ 1   ┆ 0     ┆ 1     ┆ 255 ┆ 1     │
│ 0   ┆ 0   ┆ 0     ┆ 0     ┆ 255 ┆ 0     │
│ 7   ┆ 2   ┆ 2     ┆ 7     ┆ 248 ┆ 5     │
│ 10  ┆ 34  ┆ 2     ┆ 42    ┆ 245 ┆ 40    │
└─────┴─────┴───────┴───────┴─────┴───────┘
```

❶ We're using 8-bit unsigned integers (`pl.UInt8`) so that it's easy to reason about the operations on a bit level. You can apply bitwise operators to any integer type.

Let's take a look at the binary string representations of these integers to understand how each operator works:

```
bits.select(pl.all().map_elements("{0:08b}".format))
```

```
MapWithoutReturnDtypeWarning: Calling `map_elements` without specifying `return_
dtype` can lead to unpredictable results. Specify `return_dtype` to silence this
 warning.
shape: (6, 6)
┌──────────┬──────────┬──────────┬──────────┬──────────┬──────────┐
│ x        ┆ y        ┆ x & y    ┆ x | y    ┆ ~x       ┆ x ^ y    │
│ ---      ┆ ---      ┆ ---      ┆ ---      ┆ ---      ┆ ---      │
│ str      ┆ str      ┆ str      ┆ str      ┆ str      ┆ str      │
╞══════════╪══════════╪══════════╪══════════╪══════════╪══════════╡
│ 00000001 ┆ 00000001 ┆ 00000001 ┆ 00000001 ┆ 11111110 ┆ 00000000 │
│ 00000001 ┆ 00000000 ┆ 00000000 ┆ 00000001 ┆ 11111110 ┆ 00000001 │
│ 00000000 ┆ 00000001 ┆ 00000000 ┆ 00000001 ┆ 11111111 ┆ 00000001 │
│ 00000000 ┆ 00000000 ┆ 00000000 ┆ 00000000 ┆ 11111111 ┆ 00000000 │
│ 00000111 ┆ 00000010 ┆ 00000010 ┆ 00000111 ┆ 11111000 ┆ 00000101 │
│ 00001010 ┆ 00100010 ┆ 00000010 ┆ 00101010 ┆ 11110101 ┆ 00101000 │
└──────────┴──────────┴──────────┴──────────┴──────────┴──────────┘
```

Ones and Zeros

When you use ones and zeros to represent Booleans, the result of these operators is the same as if they were Booleans, except for the NOT operator. The inverse of `True` is `False`, whereas the inverse of `1` is `254` (and not `0`), because the 7 left-most bits add up to 254 (128 + 64 + 32 + 16 + 8 + 4 + 2 = 254). We recommend using Booleans whenever an expression or column should be able to take only two values.

Using Functions

Table 7-5 lists all module-level functions that combine existing expressions into a single one.

Table 7-5. Module-level functions to combine expressions

Function	Description
`pl.all_horizontal(…)`	Compute the bitwise AND horizontally across columns.
`pl.any_horizontal(…)`	Compute the bitwise OR horizontally across columns.
`pl.arctan2(…)`	Compute two argument arctan in radians.
`pl.arctan2d(…)`	Compute two argument arctan in degrees.
`pl.arg_sort_by(…)`	Return the row indices that would sort the columns.
`pl.arg_where(…)`	Return indices where condition evaluates True.
`pl.coalesce(…)`	Folds the columns from left to right, keeping the first non-null value.
`pl.concat_list(…)`	Horizontally concatenate columns into a single list column.
`pl.concat_str(…)`	Horizontally concatenate columns into a single string column.
`pl.corr(…)`	Compute the Pearson's or Spearman rank correlation between two columns.
`pl.cov(…)`	Compute the covariance between two columns/ expressions.
`pl.cum_fold(…)`	Cumulatively fold horizontally across columns with a left fold.
`pl.cum_reduce(…)`	Cumulatively reduce horizontally across columns with a left fold.
`pl.cum_sum_horizontal(…)`	Cumulatively sum all values horizontally across columns.
`pl.fold(…)`	Accumulate over multiple columns horizontally / row wise with a left fold.
`pl.format(…)`	Format expressions as a string.
`pl.map_batches(…)`	Map a custom function over multiple columns/expressions.
`pl.max_horizontal(…)`	Get the maximum value horizontally across columns.
`pl.min_horizontal(…)`	Get the minimum value horizontally across columns.
`pl.reduce(…)`	Accumulate over multiple columns horizontally/ row wise with a left fold.
`pl.rolling_corr(…)`	Compute the rolling correlation between two columns/ expressions.
`pl.rolling_cov(…)`	Compute the rolling covariance between two columns/ expressions.

Function	Description
`pl.struct(…)`	Collect columns into a struct column.
`pl.sum_horizontal(…)`	Sum all values horizontally across columns.
`pl.when(…)`	Start a when-then-otherwise expression.

We cannot discuss them all in detail, but here are few noteworthy examples.

First is a two functions that combine the values of multiple expressions into one structure. The functions `pl.concat_list()` and `pl.struct()` create a list and a struct, respectively.

```
scientists = pl.DataFrame({
    'first_name': ['George', 'Grace', 'John', 'Kurt', 'Ada'],
    'last_name': ['Boole', 'Hopper', 'Tukey', 'Gödel', 'Lovelace'],
    'country': ['England', 'United States', 'United States',
    'Austria-Hungary', 'England']
})
scientists
```

```
shape: (5, 3)
┌────────────┬───────────┬─────────────────┐
│ first_name ┆ last_name ┆ country         │
│ ---        ┆ ---       ┆ ---             │
│ str        ┆ str       ┆ str             │
╞════════════╪═══════════╪═════════════════╡
│ George     ┆ Boole     ┆ England         │
│ Grace      ┆ Hopper    ┆ United States   │
│ John       ┆ Tukey     ┆ United States   │
│ Kurt       ┆ Gödel     ┆ Austria-Hungary │
│ Ada        ┆ Lovelace  ┆ England         │
└────────────┴───────────┴─────────────────┘
```

```
scientists.select(
    pl.concat_list(pl.col("^*_name$")).alias("concat_list"),
    pl.struct(pl.all()).alias("struct")
)
```

```
shape: (5, 2)
┌───────────────────────┬─────────────────────────────────────┐
│ concat_list           ┆ struct                              │
│ ---                   ┆ ---                                 │
│ list[str]             ┆ struct[3]                           │
╞═══════════════════════╪═════════════════════════════════════╡
│ ["George", "Boole"]   ┆ {"George","Boole","England"}        │
│ ["Grace", "Hopper"]   ┆ {"Grace","Hopper","United States"}  │
│ ["John", "Tukey"]     ┆ {"John","Tukey","United States"}    │
│ ["Kurt", "Gödel"]     ┆ {"Kurt","Gödel","Austria-Hungary"}  │
│ ["Ada", "Lovelace"]   ┆ {"Ada","Lovelace","England"}        │
└───────────────────────┴─────────────────────────────────────┘
```

Second, the functions `pl.concat_str()` and `pl.format()` create one string based on multiple expressions. The latter gives you a bit more flexibility in how the strings are combined. Here's an example:

```
scientists.select(
    pl.concat_str(pl.all(), separator=" ").alias("concat_str"),
    pl.format("{}, {} from {}",
    "last_name", "first_name", "country").alias("format")
    )
```

```
shape: (5, 2)
┌─────────────────────────────┬──────────────────────────────────┐
│ concat_str                  ┆ format                           │
│ ---                         ┆ ---                              │
│ str                         ┆ str                              │
╞═════════════════════════════╪══════════════════════════════════╡
│ George Boole England        ┆ Boole, George from England       │
│ Grace Hopper United States  ┆ Hopper, Grace from United States │
│ John Tukey United States    ┆ Tukey, John from United States   │
│ Kurt Gödel Austria-Hungary  ┆ Gödel, Kurt from Austria-Hungary │
│ Ada Lovelace England        ┆ Lovelace, Ada from England       │
└─────────────────────────────┴──────────────────────────────────┘
```

The functions `pl.all_horizontal()` and `pl.any_horizontal()` are analogous to using the AND (`&`) and OR (`|`) operators on multiple columns. This is especially useful if you have many columns to combine and you don't want to write them all out. For instance:

```
prefs = pl.DataFrame({
    "id": [1, 7, 42, 101, 999],
    "has_pet": [True, False, True, False, True],
    "likes_travel": [False, False, False, False, True],
    "likes_movies": [True, False, True, False, True],
    "likes_books": [False, False, True, True, True]
}).with_columns(
    pl.all_horizontal(pl.exclude("id")).alias("all"),
    pl.any_horizontal(pl.exclude("id")).alias("any"),
)
prefs
```

```
shape: (5, 7)
┌─────┬─────────┬──────────────┬──────────────┬─────────────┬───────┬───────┐
│ id  ┆ has_pet ┆ likes_travel ┆ likes_movies ┆ likes_books ┆ all   ┆ any   │
│ --- ┆ ---     ┆ ---          ┆ ---          ┆ ---         ┆ ---   ┆ ---   │
│ i64 ┆ bool    ┆ bool         ┆ bool         ┆ bool        ┆ bool  ┆ bool  │
╞═════╪═════════╪══════════════╪══════════════╪═════════════╪═══════╪═══════╡
│ 1   ┆ true    ┆ false        ┆ true         ┆ false       ┆ false ┆ true  │
│ 7   ┆ false   ┆ false        ┆ false        ┆ false       ┆ false ┆ false │
│ 42  ┆ true    ┆ false        ┆ true         ┆ true        ┆ false ┆ true  │
│ 101 ┆ false   ┆ false        ┆ false        ┆ true        ┆ false ┆ true  │
│ 999 ┆ true    ┆ true         ┆ true         ┆ true        ┆ true  ┆ true  │
└─────┴─────────┴──────────────┴──────────────┴─────────────┴───────┴───────┘
```

Related are the functions `pl.sum_horizontal()`, `pl.max_horizontal()`, and `pl.min_horizontal()`, which compute the sum, maximum, and minimum across columns, respectively. They work on both Boolean and numerical columns:

```
prefs.select(
    pl.sum_horizontal(pl.all()).alias("sum"),
    pl.max_horizontal(pl.all()).alias("max"),
    pl.min_horizontal(pl.all()).alias("min"),
)
```

```
shape: (5, 3)
┌──────┬─────┬─────┐
│ sum  ┆ max ┆ min │
│ ---  ┆ --- ┆ --- │
│ i64  ┆ i64 ┆ i64 │
╞══════╪═════╪═════╡
│ 4    ┆ 1   ┆ 0   │
│ 7    ┆ 7   ┆ 0   │
│ 46   ┆ 42  ┆ 0   │
│ 103  ┆ 101 ┆ 0   │
│ 1005 ┆ 999 ┆ 1   │
└──────┴─────┴─────┘
```

The function `pl.when()` creates a conditional expression. Think of it as a vectorized `if` statement. (We'll cover this in Chapter Chapter 9.) Here's an example:

```
prefs.select(
    pl.col("id"),
    pl.when(pl.all_horizontal(pl.col("^likes_.*$")))
    .then(pl.lit("Likes everything"))
    .when(pl.any_horizontal(pl.col("^likes_.*$")))
    .then(pl.lit("Likes something"))
    .otherwise(pl.lit("Likes nothing"))
    .alias("likes_what")
)
```

```
shape: (5, 2)
┌─────┬──────────────────┐
│ id  ┆ likes_what       │
│ --- ┆ ---              │
│ i64 ┆ str              │
╞═════╪══════════════════╡
│ 1   ┆ Likes something  │
│ 7   ┆ Likes nothing    │
│ 42  ┆ Likes something  │
│ 101 ┆ Likes something  │
│ 999 ┆ Likes everything │
└─────┴──────────────────┘
```

For the other functions we refer you to the online documentation.

Conclusion

This concludes the third and last chapter of Part III, *Express*. You can now begin, continue, and combine expressions.

CHAPTER 8

Filtering and Sorting Rows

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

Whereas the previous chapter was about columns, this chapter is all about the rows in a DataFrame[1]. We'll mainly look at two types of operations you can perform on rows: filtering and sorting.

With filtering, you select a subset of the rows, based on their values. With sorting, you reorder the rows based on their values; the number of rows remains the same.

In this chapter you'll learn how to:

- Filter rows using the `df.filter()` method
- Sort rows using the `df.sort()` method
- Apply various other methods that are related to filtering and sorting

1 The methods covered in this chapter can also be applied to LazyFrames.

You'll be working with a small DataFrame about power tools you'll typically find in the garage of an amateur woodworker. For each tool, we have its type, product code, brand, whether it's cordless or not, its price, and RPM (revolutions per minute). Here's what the `tools` DataFrame looks like:

```
tools = pl.read_csv("data/tools.csv")
tools
```

```
shape: (10, 6)
```

tool --- str	product --- str	brand --- str	cordless --- bool	price --- i64	rpm --- i64
Rotary Hammer	HR2230	Makita	false	199	1050
Miter Saw	GCM 8 SJL	Bosch	false	391	5500
Plunge Cut Saw	DSP600ZJ	Makita	true	459	6300
Impact Driver	DTD157Z	Makita	true	156	3000
Jigsaw	PST 900 PEL	Bosch	false	79	3100
Angle Grinder	DGA504ZJ	Makita	true	229	8500
Nail Gun	DPSB2IN1-XJ	DeWalt	true	129	null
Router	POF 1400 ACE	Bosch	false	185	28000
Random Orbital Sander	DBO180ZJ	Makita	true	199	11000
Table Saw	DWE7485	DeWalt	false	516	5800

Let's get filtering.

Filtering Rows

The rows of a DataFrame can be filtered with the `df.filter()` method. Filtering allows us to answer questions that involve phrases such as "is at least" or "is equal to". For example, which tools are by Makita? Or: which tools are cordless? ??? on page 152 illustrates this operation conceptually.

image::drawing-filter.png

To filter rows, you specify which rows you want to keep. This can be done using expressions, column names, and constraints. We'll first discuss expressions, since they're the most flexible of the three.

Filtering Based on Expressions

The first way to filter rows is using expressions. You've already seen them in action in Chapter 5. They're the most flexible way to filter of the three because you can use all types of comparisons (such as equals and greater-than) and combine them using Boolean algebra (such as OR and AND). Comparison and Boolean algebra operations are discussed in Chapter 7 if you need a refresher.

Expressions allow you to conjure up all sorts of filters. Just make sure that the expression evaluates to a Boolean Series. A `True` means that the corresponding row will be kept, and a `False` that it will be discarded.

Let's filter the `tools` DataFrame to keep our favorite tools, which happen to be cordless tools by Makita:

```
tools.filter(
    pl.col("cordless") &  ❶
    (pl.col("brand") == "Makita")
)
```

```
shape: (4, 6)
┌─────────────────────┬──────────┬────────┬──────────┬───────┬───────┐
│ tool                ┆ product  ┆ brand  ┆ cordless ┆ price ┆ rpm   │
│ ---                 ┆ ---      ┆ ---    ┆ ---      ┆ ---   ┆ ---   │
│ str                 ┆ str      ┆ str    ┆ bool     ┆ i64   ┆ i64   │
╞═════════════════════╪══════════╪════════╪══════════╪═══════╪═══════╡
│ Plunge Cut Saw      ┆ DSP600ZJ ┆ Makita ┆ true     ┆ 459   ┆ 6300  │
│ Impact Driver       ┆ DTD157Z  ┆ Makita ┆ true     ┆ 156   ┆ 3000  │
│ Angle Grinder       ┆ DGA504ZJ ┆ Makita ┆ true     ┆ 229   ┆ 8500  │
│ Random Orbital Sander ┆ DBO180ZJ ┆ Makita ┆ true   ┆ 199   ┆ 11000 │
└─────────────────────┴──────────┴────────┴──────────┴───────┴───────┘
```

❶ You don't need to write `pl.col("cordless") == True` because the data type of column `cordless` is already Boolean.

Commas Instead of Ampersands

If your expression is composed of multiple parts that are combined using the AND operator (&), then you can alternatively pass those parts as separate arguments to the `df.filter()` method. That means that the last code snippet can be rewritten as:

```
tools.filter(
    pl.col("cordless"),
    pl.col("brand") == "Makita"
)
```

Depending on your preference, this might improve the readability of your filter. Keep in mind that this doesn't work for the OR operator (|).

Filtering Based on Column Names

The second way to use `df.filter()` is by specifying column names. If a column is Boolean, such as the column `cordless` in the `tools` DataFrame, you can directly use the column name without turning it into an expression. For example, to select all cordless tools (not just those from Makita), you can use:

```python
tools.filter("cordless")
```

```
shape: (5, 6)
┌─────────────────────┬─────────────┬────────┬──────────┬───────┬───────┐
│ tool                ┆ product     ┆ brand  ┆ cordless ┆ price ┆ rpm   │
│ ---                 ┆ ---         ┆ ---    ┆ ---      ┆ ---   ┆ ---   │
│ str                 ┆ str         ┆ str    ┆ bool     ┆ i64   ┆ i64   │
╞═════════════════════╪═════════════╪════════╪══════════╪═══════╪═══════╡
│ Plunge Cut Saw      ┆ DSP600ZJ    ┆ Makita ┆ true     ┆ 459   ┆ 6300  │
│ Impact Driver       ┆ DTD157Z     ┆ Makita ┆ true     ┆ 156   ┆ 3000  │
│ Angle Grinder       ┆ DGA504ZJ    ┆ Makita ┆ true     ┆ 229   ┆ 8500  │
│ Nail Gun            ┆ DPSB2IN1-XJ ┆ DeWalt ┆ true     ┆ 129   ┆ null  │
│ Random Orbital Sander ┆ DBO180ZJ  ┆ Makita ┆ true     ┆ 199   ┆ 11000 │
└─────────────────────┴─────────────┴────────┴──────────┴───────┴───────┘
```

You can specify multiple column names, but keep in mind that they all need to be Boolean.

Polars Can't Handle the Truthy

In the Python language, there and the concepts of *truthy* and *falsy*. Whether a value is truthy or falsy depends on whether it would become `True` or `False` if cast to a Boolean. Falsy values include `False` itself, the number zero, and empty sequences, collections, and strings. Everything else is truthy. This means that Python code such as `(my_name != "") and (len(my_list) > 0)` can be rewritten as `my_name and my_list`.

Because of this language concept, you might think that Python Polars would allow non-Boolean columns and expressions when filtering. However, Polars is built in Rust, and therefore more strict than Python: only Boolean columns and expressions that construct a Boolean Series can be used for filtering.

You can turn expression that's not Boolean into a Boolean one by using comparisons. For example, to test for non-empty strings and non-empty lists, you can use `pl.col("my_name") != ""` and `pl.col("my_list").list.len() > 0`, respectively.

Filtering Based on Constraints

The third way to use `df.filter()` is by specifying constraints. A *constraint* consists of a column name and a value. Filtering again for cordless Makita tools using constraints looks like this:

```python
tools.filter(cordless=True, brand="Makita")
```

```
shape: (4, 6)
┌─────────────────────┬─────────┬────────┬──────────┬───────┬─────┐
│ tool                ┆ product ┆ brand  ┆ cordless ┆ price ┆ rpm │
│ ---                 ┆ ---     ┆ ---    ┆ ---      ┆ ---   ┆ --- │
```

```
│ str                  ┆ str      ┆ str    ┆ bool     ┆ i64  ┆ i64   │
╞══════════════════════╪══════════╪════════╪══════════╪══════╪═══════╡
│ Plunge Cut Saw       ┆ DSP600ZJ ┆ Makita ┆ true     ┆ 459  ┆ 6300  │
│ Impact Driver        ┆ DTD157Z  ┆ Makita ┆ true     ┆ 156  ┆ 3000  │
│ Angle Grinder        ┆ DGA504ZJ ┆ Makita ┆ true     ┆ 229  ┆ 8500  │
│ Random Orbital Sander┆ DBO180ZJ ┆ Makita ┆ true     ┆ 199  ┆ 11000 │
└──────────────────────┴──────────┴────────┴──────────┴──────┴───────┘
```

Effectively, the column names are specified as keyword arguments. Due to how the Python language works, this has a couple of limitations:

- The column name can only contain letters (`a-z`, `A-Z`), digits (`0-9`), and underscores (`_`), cannot start with a digit, and cannot be a reserved keyword in Python (e.g., `if`, `class`, `global`).
- Constraints must appear last if you combine them with the other two ways (expressions and column names).
- The value must always be specified, including `True`.
- Only equality comparisons are supported and must be written with one equals sign (`=`) instead of two. On top of that, the Python style guide states that there shouldn't be any spaces around the equals sign.

Don't Constrain Yourself

Because of their limitations, we advise against using constraints. Our recommendation is to use expressions for filtering. They're more verbose, but at least you won't be constraining your expressiveness.

Sorting Rows

With sorting, you change the order of the rows, based on the values in one or more columns. The number of rows remains the same. Sorting allows us to answer questions that involve phrases such as "the most" or "the lowest". For example, of which brand do we have the most tools? Or: what is the tool with the lowest price? ??? on page 155 illustrates this operation conceptually.

image::drawing-sort.png

Most often you'll be sorting numbers, but you can also sort strings, dates, and times. You can also sort container data types such as structs and lists, as we'll show you later. In the next few sections we'll look at sorting based on a single column, multiple columns, and expressions.

Sorting Based On a Single Column

To sort, you use the `df.sort()` method. The easiest way to invoke this method is by specifying a column name:

```
tools.sort("price")
```

```
shape: (10, 6)
┌───────────────┬─────────────┬────────┬──────────┬───────┬──────┐
│ tool          ┆ product     ┆ brand  ┆ cordless ┆ price ┆ rpm  │
│ ---           ┆ ---         ┆ ---    ┆ ---      ┆ ---   ┆ ---  │
│ str           ┆ str         ┆ str    ┆ bool     ┆ i64   ┆ i64  │
╞═══════════════╪═════════════╪════════╪══════════╪═══════╪══════╡
│ Jigsaw        ┆ PST 900 PEL ┆ Bosch  ┆ false    ┆ 79    ┆ 3100 │
│ Nail Gun      ┆ DPSB2IN1-XJ ┆ DeWalt ┆ true     ┆ 129   ┆ null │
│ …             ┆ …           ┆ …      ┆ …        ┆ …     ┆ …    │
│ Plunge Cut Saw┆ DSP600ZJ    ┆ Makita ┆ true     ┆ 459   ┆ 6300 │
│ Table Saw     ┆ DWE7485     ┆ DeWalt ┆ false    ┆ 516   ┆ 5800 │
└───────────────┴─────────────┴────────┴──────────┴───────┴──────┘
```

As you see, by default the values are sorted in ascending order. Table 8-1 lists the other arguments that the `df.sort()` method accepts.

Table 8-1. Common arguments for the method `df.sort()`

Argument	Description
`descending`	Sort in descending order. When sorting by multiple columns, can be specified per column by passing a sequence of Booleans. Default `False`.
`nulls_last`	Place null values last. Default `False`.
`multithreaded`	Sort using multiple threads. Default `True`.[a]
`maintain_order`	Whether the order should be maintained if elements are equal. Default `False`.

[a] Only set this to `False` when your Polars is code is part of an application that's already multithreaded.

Sorting in Reverse

You can change the default order by setting the `descending` keyword to `True`:

```
tools.sort("price", descending=True)
```

```
shape: (10, 6)
┌───────────────┬─────────────┬────────┬──────────┬───────┬──────┐
│ tool          ┆ product     ┆ brand  ┆ cordless ┆ price ┆ rpm  │
│ ---           ┆ ---         ┆ ---    ┆ ---      ┆ ---   ┆ ---  │
│ str           ┆ str         ┆ str    ┆ bool     ┆ i64   ┆ i64  │
╞═══════════════╪═════════════╪════════╪══════════╪═══════╪══════╡
│ Table Saw     ┆ DWE7485     ┆ DeWalt ┆ false    ┆ 516   ┆ 5800 │
│ Plunge Cut Saw┆ DSP600ZJ    ┆ Makita ┆ true     ┆ 459   ┆ 6300 │
│ …             ┆ …           ┆ …      ┆ …        ┆ …     ┆ …    │
│ Nail Gun      ┆ DPSB2IN1-XJ ┆ DeWalt ┆ true     ┆ 129   ┆ null │
```

Jigsaw	PST 900 PEL	Bosch	false	79	3100

Up or Down?

Make sure to use the `descending` keyword instead of `ascending`, otherwise you get an error:

```
tools.sort("price", ascending=False)
```

```
TypeError: DataFrame.sort() got an unexpected keyword
argument 'ascending'
```

It's easy to forget this, especially if you're used to Pandas, where you can use `ascending = False` to reverse the order.

Sorting Based on Multiple Columns

To sort based on multiple columns, you specify multiple column names as separate arguments:

```
tools.sort("brand", "price")
```

shape: (10, 6)

tool --- str	product --- str	brand --- str	cordless --- bool	price --- i64	rpm --- i64
Jigsaw	PST 900 PEL	Bosch	false	79	3100
Router	POF 1400 ACE	Bosch	false	185	28000
…	…	…	…	…	…
Angle Grinder	DGA504ZJ	Makita	true	229	8500
Plunge Cut Saw	DSP600ZJ	Makita	true	459	6300

Again, the default order is ascending for all columns that you specify. Setting `descending` to `True` will apply to all columns. If you want to have different directions per column, you can pass a list of Booleans to `descending`:

```
tools.sort("brand", "price", descending=[False, True])
```

shape: (10, 6)

tool --- str	product --- str	brand --- str	cordless --- bool	price --- i64	rpm --- i64
Miter Saw	GCM 8 SJL	Bosch	false	391	5500
Router	POF 1400 ACE	Bosch	false	185	28000
…	…	…	…	…	…
Random Orbital Sander	DBO180ZJ	Makita	true	199	11000

```
│ Impact Driver         │ DTD157Z     │ Makita │ true     │ 156   │ 3000  │
└───────────────────────┴─────────────┴────────┴──────────┴───────┴───────┘
```

Make sure that the number of Booleans is equal to the number of columns.

Sorting Based on Expressions

The `df.sort()` method also accepts one or more expressions:

```
tools.sort(pl.col("rpm") / pl.col("price"))
```

```
shape: (10, 6)
┌───────────────────────┬─────────────┬────────┬──────────┬───────┬───────┐
│ tool                  │ product     │ brand  │ cordless │ price │ rpm   │
│ ---                   │ ---         │ ---    │ ---      │ ---   │ ---   │
│ str                   │ str         │ str    │ bool     │ i64   │ i64   │
╞═══════════════════════╪═════════════╪════════╪══════════╪═══════╪═══════╡
│ Nail Gun              │ DPSB2IN1-XJ │ DeWalt │ true     │ 129   │ null  │
│ Rotary Hammer         │ HR2230      │ Makita │ false    │ 199   │ 1050  │
│ …                     │ …           │ …      │ …        │ …     │ …     │
│ Random Orbital Sander │ DBO180ZJ    │ Makita │ true     │ 199   │ 11000 │
│ Router                │ POF 1400 ACE│ Bosch  │ false    │ 185   │ 28000 │
└───────────────────────┴─────────────┴────────┴──────────┴───────┴───────┘
```

Note that expressions will not appear as columns in the DataFrame. Just as with filtering, expressions will give you the most flexibility. However, from our experience, you'll most often sort on columns already present in the DataFrame.

Sorting Nested Data Types

You cannot directly sort nested data types, like Structs, Lists, and Arrays. You first need to extract or create a value that is sortable. To demonstrate this, let's create a new DataFrame `tools_collection` that groups all the tools, by brand, into a List of Structs:

```
tools_collection = tools.group_by("brand").agg(collection=pl.struct(pl.all()))
tools_collection
```

```
shape: (3, 2)
┌────────┬─────────────────────────────────────────────────────────────────┐
│ brand  │ collection                                                      │
│ ---    │ ---                                                             │
│ str    │ list[struct[6]]                                                 │
╞════════╪═════════════════════════════════════════════════════════════════╡
│ DeWalt │ [{"Nail Gun","DPSB2IN1-XJ","DeWalt",true,129,null}, {"Table …   │
│ Makita │ [{"Rotary Hammer","HR2230","Makita",false,199,1050}, {"Plung…   │
│ Bosch  │ [{"Miter Saw","GCM 8 SJL","Bosch",false,391,5500}, {"Jigsaw"…   │
└────────┴─────────────────────────────────────────────────────────────────┘
```

If you try to sort the `collection` column directly, you'll get an error. You *can*, however, sort the Lists by their length, because that's an integer which can be sorted:

```
tools_collection.sort(pl.col("collection").list.len(), descending=True)
```

```
shape: (3, 2)
┌────────┬──────────────────────────────────────────────────────────────────┐
│ brand  ┆ collection                                                       │
│ ---    ┆ ---                                                              │
│ str    ┆ list[struct[6]]                                                  │
╞════════╪══════════════════════════════════════════════════════════════════╡
│ Makita ┆ [{"Rotary Hammer","HR2230","Makita",false,199,1050}, {"Plung… │
│ Bosch  ┆ [{"Miter Saw","GCM 8 SJL","Bosch",false,391,5500}, {"Jigsaw"… │
│ DeWalt ┆ [{"Nail Gun","DPSB2IN1-XJ","DeWalt",true,129,null}, {"Table … │
└────────┴──────────────────────────────────────────────────────────────────┘
```

Another example is to sort on the average price for each brand:

```
tools_collection.sort(
    pl.col("collection").list.eval(
        pl.element().struct.field("price")
    ).list.mean()
)
```

```
shape: (3, 2)
┌────────┬───────────────────────────────────────────────────────────────────────┐
│ brand  ┆ collection                                                            │
│ ---    ┆ ---                                                                   │
│ str    ┆ list[struct[6]]                                                       │
╞════════╪═══════════════════════════════════════════════════════════════════════╡
│ Bosch  ┆ [{"Miter Saw","GCM 8 SJL","Bosch",false,391,5500}, {"Jigsaw","PST… │
│ Makita ┆ [{"Rotary Hammer","HR2230","Makita",false,199,1050}, {"Plunge Cut… │
│ DeWalt ┆ [{"Nail Gun","DPSB2IN1-XJ","DeWalt",true,129,null}, {"Table Saw",… │
└────────┴───────────────────────────────────────────────────────────────────────┘
```

Materialize First, Sort Second

Sometimes, just as with the last code snippet, things can get a bit complicated and make you wonder whether you're sorting correctly. In those cases, it can be helpful to first construct a new column using the `df.with_columns()` method to inspect the values on which you're sorting:

```
tools_collection.with_columns(
    mean_price=pl.col("collection").list.eval(
        pl.element().struct.field("price")
    ).list.mean()
).sort("mean_price")
```

```
shape: (3, 3)
┌────────┬─────────────────────────────────────┬────────────┐
│ brand  ┆ collection                          ┆ mean_price │
│ ---    ┆ ---                                 ┆ ---        │
│ str    ┆ list[struct[6]]                     ┆ f64        │
╞════════╪═════════════════════════════════════╪════════════╡
│ Bosch  ┆ [{"Miter Saw","GCM 8 SJL","Bosch",f… ┆ 218.333333 │
│ Makita ┆ [{"Rotary Hammer","HR2230","Makita"… ┆ 248.4      │
│ DeWalt ┆ [{"Nail Gun","DPSB2IN1-XJ","DeWalt"… ┆ 322.5      │
└────────┴─────────────────────────────────────┴────────────┘
```

Turns out we were sorting on the correct values. Now we can safely turn that `df.with_columns()` back into a `df.sort()`.

Related Row Operations

Besides filtering and sorting, there are a few related row operations worth knowing about:

Filtering Missing Values

Sometimes, your analysis or machine learning algorithm cannot handle missing values. The method `df.drop_nulls()` keeps only rows without missing values. You can specify which columns should be considered. For example:

```
tools.drop_nulls("rpm").height
```

```
9
```

By default all columns are considered, in which case it's effectively the same as:

```
df.filter(pl.all_horizontal(pl.all().is_not_null()))
```

Slicing

Sometimes you want to keep the rows based on their position in the DataFrame, irrespective of the values they contain. This is generally known as *slicing*, and there are several methods for this:

- With `df.head()` and `df.tail()` you keep the first or last few rows, respectively. For example: the first five rows.
- With `df.slice()` you keep a range of rows. For example, from the third to the seventh row.
- With `df.gather()` you keep individual rows. For example, the first, second, and the fifth row.
- With `df.gather_every()` you keep a row every so often. For example, every second row.

  You can of course combine these methods to create complex slices. For example:

```
(
    tools.with_row_index()
    .gather_every(2).head(3)
)
```

```
shape: (3, 7)
┌───────┬────────────────┬─────────────┬───┬──────────┬───────┬──────┐
│ index ┆ tool           ┆ product     ┆ … ┆ cordless ┆ price ┆ rpm  │
│ ---   ┆ ---            ┆ ---         ┆   ┆ ---      ┆ ---   ┆ ---  │
│ u32   ┆ str            ┆ str         ┆   ┆ bool     ┆ i64   ┆ i64  │
╞═══════╪════════════════╪═════════════╪═══╪══════════╪═══════╪══════╡
│ 0     ┆ Rotary Hammer  ┆ HR2230      ┆ … ┆ false    ┆ 199   ┆ 1050 │
│ 2     ┆ Plunge Cut Saw ┆ DSP600ZJ    ┆ … ┆ true     ┆ 459   ┆ 6300 │
│ 4     ┆ Jigsaw         ┆ PST 900 PEL ┆ … ┆ false    ┆ 79    ┆ 3100 │
└───────┴────────────────┴─────────────┴───┴──────────┴───────┴──────┘
```

  The method `df.with_row_index()` is used here to clarify which row positions are kept.

Top and Bottom

With the methods `df.top_k()` and `df.bottom_k()`, you keep the *k* rows with the largest or smallest value. For example, to keep the top three most expensive tools:

```
tools.top_k(3, by="price")
```

```
shape: (3, 6)
┌────────────────┬───────────┬─────────┬──────────┬───────┬──────┐
│ tool           ┆ product   ┆ brand   ┆ cordless ┆ price ┆ rpm  │
│ ---            ┆ ---       ┆ ---     ┆ ---      ┆ ---   ┆ ---  │
│ str            ┆ str       ┆ str     ┆ bool     ┆ i64   ┆ i64  │
╞════════════════╪═══════════╪═════════╪══════════╪═══════╪══════╡
│ Table Saw      ┆ DWE7485   ┆ DeWalt  ┆ false    ┆ 516   ┆ 5800 │
│ Plunge Cut Saw ┆ DSP600ZJ  ┆ Makita  ┆ true     ┆ 459   ┆ 6300 │
│ Miter Saw      ┆ GCM 8 SJL ┆ Bosch   ┆ false    ┆ 391   ┆ 5500 │
└────────────────┴───────────┴─────────┴──────────┴───────┴──────┘
```

This code is essentially `tools.sort("price", descending=True)` followed by `tools.head(3)`.

Sampling

The method `df.sample()` filters the rows based on randomness. For example, to keep only 20% of the rows:

```
tools.sample(fraction=0.2)
```

```
shape: (2, 6)
┌───────────────┬──────────────┬────────┬──────────┬───────┬───────┐
│ tool          ┆ product      ┆ brand  ┆ cordless ┆ price ┆ rpm   │
│ ---           ┆ ---          ┆ ---    ┆ ---      ┆ ---   ┆ ---   │
│ str           ┆ str          ┆ str    ┆ bool     ┆ i64   ┆ i64   │
╞═══════════════╪══════════════╪════════╪══════════╪═══════╪═══════╡
│ Rotary Hammer ┆ HR2230       ┆ Makita ┆ false    ┆ 199   ┆ 1050  │
│ Router        ┆ POF 1400 ACE ┆ Bosch  ┆ false    ┆ 185   ┆ 28000 │
└───────────────┴──────────────┴────────┴──────────┴───────┴───────┘
```

Semi Joins

Another way to filter is to semi-join with another DataFrame. For example, let's say you have a DataFrame `saws` which contains all sorts of saws. You can use this to keep only the saws in the `tools` DataFrame:

```
saws = pl.DataFrame({"tool": ["Table Saw", "Plunge Cut Saw", "Miter Saw",
                              "Jigsaw", "Bandsaw", "Chainsaw", "Seesaw"]})
tools.join(saws, how="semi", on="tool")
```

```
shape: (4, 6)
┌────────────────┬─────────────┬────────┬──────────┬───────┬──────┐
│ tool           ┆ product     ┆ brand  ┆ cordless ┆ price ┆ rpm  │
│ ---            ┆ ---         ┆ ---    ┆ ---      ┆ ---   ┆ ---  │
│ str            ┆ str         ┆ str    ┆ bool     ┆ i64   ┆ i64  │
╞════════════════╪═════════════╪════════╪══════════╪═══════╪══════╡
│ Miter Saw      ┆ GCM 8 SJL   ┆ Bosch  ┆ false    ┆ 391   ┆ 5500 │
│ Plunge Cut Saw ┆ DSP600ZJ    ┆ Makita ┆ true     ┆ 459   ┆ 6300 │
│ Jigsaw         ┆ PST 900 PEL ┆ Bosch  ┆ false    ┆ 79    ┆ 3100 │
│ Table Saw      ┆ DWE7485     ┆ DeWalt ┆ false    ┆ 516   ┆ 5800 │
└────────────────┴─────────────┴────────┴──────────┴───────┴──────┘
```

You'll learn more about joining in general in Chapter 11.

Takeaways

In this chapter we've looked at filtering and sorting rows, and a few related operations. The key takeaways are:

- Filtering based on expressions give you the most flexibility.
- With filtering, expressions must evaluate to a Boolean Series.
- Filtering based on constraints has many limits.
- Expressions, column names, and constraints separated by commas are combined under the hood with the AND operator (`&`).

- Sorting based on a single column is most often sufficient.
- Use `descending = True` to reverse the default sort order.
- To sort nested data types, first create or extract a sortable value from them.
- There are many related row operations, including slicing and sampling.

In the next chapter we're going to look at how to work with special data types such as Strings, Categoricals, and Temporal Data.

CHAPTER 9
Working with Special Data Types

> ### A Note for Early Release Readers
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 12th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

In Chapter 2 we covered the basic data types that Polars supports and how they are used to store information in DataFrames. However, certain data types deserve special attention. Some have special operations that can be performed on them, and some are optimized for specific use cases. In Polars, these special data types have their own namespace in Expressions, meaning that you can access their methods and attributes through the `Expr` namespace.

The special data types in Polars are `String`, `Categorical`, and `Enum`; the Temporal data types `Date`, `DateTime`, `Time`, and `Duration`; and the nested data types `Array`, `List`, and `Struct`. This chapter will dive into these special data types and their operations.

Strings

A *string* is a data type for representing text, consisting of a sequence of characters, digits, or symbols. One of the challenges of strings is their variable length. For example, integers are a fixed length: you can calculate the memory address of the next integer by adding the size of the integer to the current memory address. This is not the case for strings. The length of a string is not known in advance, so the memory address of the next string cannot be predicted purely from the data buffer. This means that strings have to be stored differently than integers: contiguously, in a data buffer. *Contiguous memory* is one long memory block where all the values are stored in a row.

The view layout stores several attributes of a string value:

- Bytes 0 to 3 store the length of the string.
- Bytes 4 to 7 store a copy of the first 4 bytes of a string. This allows for "fast paths," or optimizations, since these 4 bytes frequently contain the information needed to make a quick comparison.
- Bytes 8 to 11 store the index of the data buffer where the string resides.
- Bytes 12 to 15 store the *offset*: the location within that data buffer where the string starts.

With all this information, you can retrieve the string from the data buffer without having to seek through memory!

Polars has another optimization for strings shorter than 12 bytes long. In this case, the string is stored in the view layout itself, instead of in the data buffer. This is called *inlining*. When its length is at most 12 bytes, the string can be stored in the 12 bytes that follow the length. This prevents Polars from having to allocate memory and seek in the data buffer, which are both costly operation.

Figure 9-1 illustrates how this storage works. Aerosmith fits inline and thus is not in the data buffer. "Toots and the Maytals" is stored in the data buffer starting on the 22nd position after "The Velvet UnderGround"(where the first position is 0).

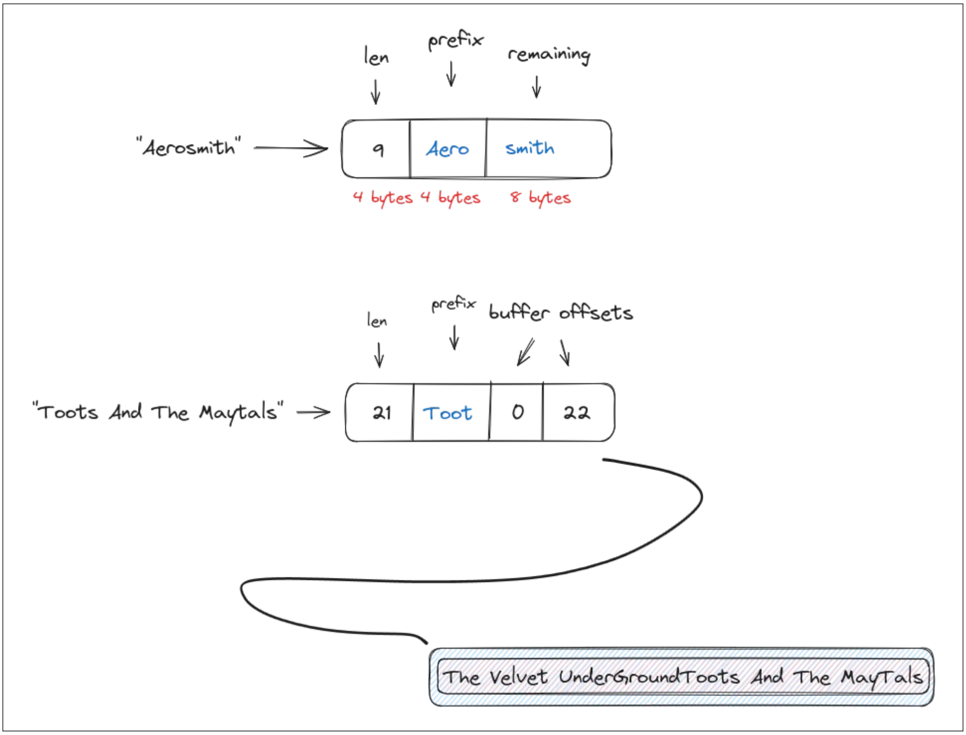


Figure 9-1. How short and long strings are stored in memory

Methods

Now that you know how strings are stored in physical memory, let's look at what operations are available for them.

Conversion

The methods in Table 9-1 allow you to convert strings to and from different data types or formats.

Table 9-1. Conversion methods for the String data type

Function or method	Description
`pl.Expr.str.decode(…)`	Decode a value using the provided encoding.
`pl.Expr.str.encode(…)`	Encode a value using the provided encoding.
`pl.Expr.str.json_decode(…)`	Parse string values as JSON.
`pl.Expr.str.json_extract(…)`	Parse string values as JSON.
`pl.Expr.str.json_path_match(…)`	Extract the first match of a JSON string with the provided JSONPath expression.

Function or method	Description
`pl.Expr.str.strptime(…)`	Convert a `String` column into a `Date`/`Datetime`/`Time` column.
`pl.Expr.str.to_date(…)`	Convert a `String` column into a `Date` column.
`pl.Expr.str.to_datetime(…)`	Convert a `String` column into a `Datetime` column.
`pl.Expr.str.to_decimal(…)`	Convert a `String` column into a `Decimal` column.
`pl.Expr.str.to_integer(…)`	Convert an `String` column into an `Int64` column with base radix[a].
`pl.Expr.str.to_time(…)`	Convert a `String` column into a `Time` column.
`pl.Expr.str.parse_int(…)`	Parse integers with base radix from strings.

[a] *Radix* refers to the base of a number system, specifying how many digits it uses. In the context of converting strings to integers, the radix determines how to interpret the symbols in the string. Without knowing the radix, the conversion can be ambiguous. For example, "101" could represent different values depending on whether it's in decimal (101), binary (5), or hexadecimal (272). The default radix is decimal (10).

Descriptive and Query Methods

The methods in Table 9-2 can return attributes of the string values in a column or allow you to query for certain patterns.

Table 9-2. Descriptive methods for the String data type

Function or method	Description
`pl.Expr.str.contains(…)`	Check if strings in `Series` contain a substring that matches a regex.
`pl.Expr.str.find(…)`	Return the index of the first substring in `Series` strings matching a pattern.
`pl.Expr.str.len_bytes()`	Return the length of each string as the number of bytes.
`pl.Expr.str.len_chars()`	Return the length of each string as the number of characters.
`pl.Expr.str.lengths()`	Return the number of bytes in each string.
`pl.Expr.str.n_chars()`	Return the length of each string as the number of characters.
`pl.Expr.str.starts_with(…)`	Check if string values start with a substring.

Manipulation

The methods in Table 9-3 allow you to manipulate the string values in a column.

Table 9-3. Manipulation methods for the String data type

Function or method	Description
`pl.Expr.str.concat(…)`	Vertically concatenate the string values in the column to a single string value.
`pl.Expr.str.contains_any(…)`	Use the `aho-corasick` algorithm to find matches.
`pl.Expr.str.count_match(…)`	Count all successive non-overlapping regex matches.
`pl.Expr.str.count_matches(…)`	Count all successive non-overlapping regex matches.

Function or method	Description
`pl.Expr.str.ends_with(…)`	Check if string values end with a substring.
`pl.Expr.str.explode()`	Returns a column with a separate row for every string character.
`pl.Expr.str.extract(…)`	Extract the target capture group from provided patterns.
`pl.Expr.str.extract_all(…)`	Extract all matches for the given regex pattern.
`pl.Expr.str.extract_groups(…)`	Extract all capture groups for the given regex pattern.
`pl.Expr.str.ljust(…)`	Return the string left-justified in a string of length `length`.
`pl.Expr.str.lstrip(…)`	Remove leading characters.
`pl.Expr.str.pad_end(…)`	Pad the end of the string until it reaches the given `length`.
`pl.Expr.str.pad_start(…)`	Pad the start of the string until it reaches the given `length`.
`pl.Expr.str.replace(…)`	Replace first matching regex/literal substring with a new string value.
`pl.Expr.str.replace_all(…)`	Replace all matching regex/literal substrings with a new string value.
`pl.Expr.str.replace_many(…)`	Use the `aho-corasick` algorithm to replace many matches.
`pl.Expr.str.reverse()`	Returns string values in reversed order.
`pl.Expr.str.rjust(…)`	Return the string right justified in a string of length length.
`pl.Expr.str.rstrip(…)`	Remove trailing characters.
`pl.Expr.str.slice(…)`	Create subslices of the string values of a string series.
`pl.Expr.str.split(…)`	Split the string by a substring.
`pl.Expr.str.split_exact(…)`	Split the string by a substring using n splits.
`pl.Expr.str.splitn(…)`	Split the string by a substring, restricted to returning at most n items.
`pl.Expr.str.strip_chars(…)`	Remove leading and trailing characters.
`pl.Expr.str.strip_chars_start(…)`	Remove leading characters.
`pl.Expr.str.strip_chars_end(…)`	Remove trailing characters.
`pl.Expr.str.strip_prefix(…)`	Remove prefix.
`pl.Expr.str.strip_suffix(…)`	Remove suffix.
`pl.Expr.str.to_lowercase()`	Modify the strings to their lowercase equivalent.
`pl.Expr.str.to_titlecase()`	Modify the strings to their titlecase equivalent.
`pl.Expr.str.to_uppercase()`	Modify the strings to their uppercase equivalent.
`pl.Expr.str.zfill(…)`	Pad the start of the string with zeros until it reaches the given length.

Examples

Let's dive into some examples. First you'll create a DataFrame with some sample data:

```python
import polars as pl

df = pl.DataFrame({
    "raw_text": [
        "  Data Science is amazing ",
        "Data_analysis > Data entry",
```

```
        " Python&Polars; Fast",
    ]
})
print(df)
```

```
shape: (3, 1)
┌─────────────────────────┐
│ raw_text                │
│ ---                     │
│ str                     │
╞═════════════════════════╡
│   Data Science is amazing │
│ Data_analysis > Data entry │
│  Python&Polars; Fast     │
└─────────────────────────┘
```

This example DataFrame showcases some of the string operations available in Polars. Start by cleaning up the strings:

```
df = df.with_columns(
    pl.col("raw_text")
    .str.strip_chars() ❶
    .str.to_lowercase() ❷
    .str.replace_all("_", " ") ❸
    .alias("processed_text") ❹
)
print(df)
```

```
shape: (3, 2)
┌────────────────────────────┬────────────────────────────┐
│ raw_text                   │ processed_text             │
│ ---                        │ ---                        │
│ str                        │ str                        │
╞════════════════════════════╪════════════════════════════╡
│   Data Science is amazing  │ data science is amazing    │
│ Data_analysis > Data entry │ data analysis > data entry │
│  Python&Polars; Fast       │ python&polars; fast        │
└────────────────────────────┴────────────────────────────┘
```

❶ The `strip_chars()` method removes leading and trailing characters from the string. Since you haven't provided any characters to strip, it defaults to whitespace.

❷ Casting everything to lowercase can make it easier to work with the data, because then it's case-insensitive.

❸ You may want to replace all underscores with spaces when working with filenames or URLs.

❹ Creates a new column with the processed text and name it appropriately.

Now that you have clean data to work with, let's get into manipulating and selecting it.

One common operation is slicing and splitting strings:

```
print(
    df.with_columns(
        pl.col("processed_text")
        .str.slice(0, 5)  ❶
        .alias("first_5_chars"),
        pl.col("processed_text")
        .str.split(" ")  ❷
        .list.get(0)  ❸
        .alias("first_word"),
        pl.col("processed_text")
        .str.split(" ")
        .list.get(1)  ❹
        .alias("second_word"),
    )
)
```

```
shape: (3, 5)
┌────────────────┬────────────────┬───────────────┬────────────────┬─────────────┐
│ raw_text       ┆ processed_tex  ┆ first_5_chars ┆ first_word     ┆ second_word │
│ ---            ┆ t              ┆ ---           ┆ ---            ┆ ---         │
│ str            ┆ ---            ┆ str           ┆ str            ┆ str         │
│                ┆ str            ┆               ┆                ┆             │
╞════════════════╪════════════════╪═══════════════╪════════════════╪═════════════╡
│ Data Science   ┆ data science   ┆ data          ┆ data           ┆ science     │
│ is am…         ┆ is amaz…       ┆               ┆                ┆             │
│ Data_analysis  ┆ data analysis  ┆ data          ┆ data           ┆ analysis    │
│ > Data…        ┆ > data…        ┆               ┆                ┆             │
│ Python&Polars; ┆ python&polars  ┆ pytho         ┆ python&polars  ┆ fast        │
│ Fast           ┆ ; fast         ┆               ┆ ;              ┆             │
└────────────────┴────────────────┴───────────────┴────────────────┴─────────────┘
```

❶ From the string values in the `processed_text` column, you slice the first 5 characters and save them in the `first_5_chars` column.

❷ You split the string values in the `processed_text` column on spaces. This creates a list strings with a length of the amount of spaces in the string + 1.

❸ You get the first element from that list of strings, which will be the first word.

❹ You get the second element from that list of strings.

You can also query the string for some information about it, as follows:

```
print(
    df.with_columns(
        pl.col("processed_text")
```

```
        .str.len_chars()  ❶
        .alias("amount_of_chars"),
        pl.col("processed_text")
        .str.len_bytes()  ❷
        .alias("amount_of_bytes"),
        pl.col("processed_text")
        .str.count_matches("a")  ❸
        .alias("count_a"),
    )
)
```

```
shape: (3, 5)
┌─────────────────┬─────────────────┬─────────────────┬─────────────────┬──────────┐
│ raw_text        ┆ processed_text  ┆ amount_of_char  ┆ amount_of_byte  ┆ count_a  │
│ ---             ┆ ---             ┆ s               ┆ s               ┆ ---      │
│ str             ┆ str             ┆ ---             ┆ ---             ┆ u32      │
│                 ┆                 ┆ u32             ┆ u32             ┆          │
╞═════════════════╪═════════════════╪═════════════════╪═════════════════╪══════════╡
│ Data Science is ┆ data science    ┆ 23              ┆ 23              ┆ 4        │
│ amazing         ┆ is amazing      ┆                 ┆                 ┆          │
│ Data_analysis > ┆ data analysis   ┆ 26              ┆ 26              ┆ 6        │
│ Data entry      ┆ > data entry    ┆                 ┆                 ┆          │
│ Python&Polars;  ┆ python&polars;  ┆ 19              ┆ 19              ┆ 2        │
│ Fast            ┆ fast            ┆                 ┆                 ┆          │
└─────────────────┴─────────────────┴─────────────────┴─────────────────┴──────────┘
```

❶ Calculates the amount of characters in the string.

❷ Calculates the amount of bytes that the string takes in memory.

❸ Counts the times the letter “a” occurs in the string.

> It's good to know that `len_bytes()` is much more performant than `len_chars()`. `len_bytes()` has a time complexity of *O(1)*, whereas `len_chars()` has a time complexity of *O(n)*.
>
> Here, O(1) and O(n) are notations used in computer science to describe the worst-case time complexity of an algorithm. O(1) means that the time it takes to execute the algorithm is constant, regardless of the size of the input. O(n) means that the time it takes to execute the algorithm is linearly proportional to the size of the input. The time complexity of methods differs because the number of bytes can be retrieved from the view layout, whereas the number of characters has to be calculated by iterating over the string in the data buffer.
>
> In the example above, the results for `len_chars()` and `len_bytes()` are the same, because you're working with ASCII text. When working with non-ASCII text, the length in bytes won't be the same as the length in characters, so you may want to use `len_chars()` instead.

Polars also supports regex operations. A *regex*, short for "regular expression", is a sequence of characters that defines a search pattern. Primarily used for string searching and manipulation, regexes let you identify, match, and even modify text based on specific patterns, efficiently processing complex text tasks. The sample code below simply finds all the hashtags in a string.

```
df = pl.DataFrame({
    "post": ["Loving #python and #polars!", "A boomer post without a hashtag"]
})

hashtag_regex = r"#(\w+)"  ❶

df.with_columns(
    pl.col("post").str.extract_all(hashtag_regex).alias("hashtags")  ❷
)
```

```
shape: (2, 2)
┌─────────────────────────────────┬───────────────────────┐
│ post                            ┆ hashtags              │
│ ---                             ┆ ---                   │
│ str                             ┆ list[str]             │
╞═════════════════════════════════╪═══════════════════════╡
│ Loving #python and #polars!     ┆ ["#python", "#polars"] │
│ A boomer post without a hashtag ┆ []                    │
└─────────────────────────────────┴───────────────────────┘
```

❶ You define a regex pattern that matches a hashtag followed by a word. Here the `\w` matches any word character. A *word character* is a character a-z, A-Z, 0-9, including _ (underscore). The + means that the previous character can occur one or more times, capturing the entire word and not just the first character.

❷ You extract all matches of the regex pattern from the `post` column and save them in the `hashtags` column.

Categoricals

The `Categorical` data type encodes columns of string values efficiently. With the `String` data type, all the values are stored in physical memory separately, even if they are the same. The `Categorical` type uses a string cache.

A *string cache* is a dictionary behind the scenes that stores the unique string values and accompanying `UInt32` representations for all unique strings in that column. Instead of storing the string for all values in the column, the smaller `int` representation is used for efficient storage. The `int` is called the *physical representation*, whereas the string is called the *lexical representation*.

If a column of data contains a lot of string values but few unique string values, this allows for more efficient storage and faster operations (because string comparisons are expensive). Categoricals are stored in two parts: a dictionary and indices.

Let's explore the `Categorical` data type and its methods. First, you'll create a DataFrame with a `Categorical` column. Additionally you'll also create a column with it's physical representation.

```
df1 = pl.DataFrame(
    {"categorical_column": ["value1", "value2", "value3"]},
    schema={"categorical_column": pl.Categorical},
)

print(
    df1.with_columns(
        pl.col("categorical_column")
        .to_physical()
        .alias("categorical_column_physical")
    )
)
```

```
shape: (3, 2)
┌────────────────────┬────────────────────┐
│ categorical_column ┆ categorical_column_p… │
│ ---                ┆ ---                │
│ cat                ┆ u32                │
╞════════════════════╪════════════════════╡
│ value1             ┆ 0                  │
│ value2             ┆ 1                  │
│ value3             ┆ 2                  │
└────────────────────┴────────────────────┘
```

Methods

The `Categorical` data type has the following two methods:

Table 9-4. Methods for the Categorical data type

Function or method	Description
`Expr.cat.get_categories()`	Get the categories stored in this data type.
`Expr.cat.set_ordering(…)`	Determine how this categorical series should be sorted.

Examples

The order of strings in the column determines what the `Categorical` and the dictionary will look like. Even if a column of a different DataFrame contains the same unique strings, the `Categorical` will be different if the order is not the same. That's because the order of the dictionary, and thus its physical representation (the `int`), is different:

```
df2 = pl.DataFrame(
    {"categorical_column": ["value4", "value3", "value2"]},
    schema={"categorical_column": pl.Categorical},
)

print(
    df2.with_columns(
        pl.col("categorical_column")
        .to_physical()
        .alias("categorical_column_physical")
    )
)
```

```
shape: (3, 2)
┌────────────────────┬───────────────────────┐
│ categorical_column ┆ categorical_column_p… │
│ ---                ┆ ---                   │
│ cat                ┆ u32                   │
╞════════════════════╪═══════════════════════╡
│ value4             ┆ 0                     │
│ value3             ┆ 1                     │
│ value2             ┆ 2                     │
└────────────────────┴───────────────────────┘
```

For this reason, trying to combine two different Categoricals will cause a `CategoricalRemappingWarning`:

```
df1.join(df2, on="categorical_column")
```

```
CategoricalRemappingWarning: Local categoricals have different encodings, expens
ive re-encoding is done to perform this merge operation. Consider using a String
Cache or an Enum type if the categories are known in advance
shape: (2, 1)
```

```
┌───────────────────┐
│ categorical_column │
│ ---                │
│ cat                │
╞════════════════════╡
│ value3             │
│ value2             │
└────────────────────┘
```

To combine two Categoricals, you need to make their string caches match by creating them under the same string cache. You can do this with a global string cache. A *global string cache* is a string cache that is shared across all Categoricals. This way, all Categoricals tap into the same string cache, preventing any mismatch. The global string cache is turned off by default, because using the same string cache for all Categoricals incurs a performance penalty. If the string cache is a global object, it needs to be locked while it's accessed, making threads wait for each other, which results in longer loading times.

The example beneath shows how to create categoricals under the same string cache with a `StringCache` context manager:

```python
with pl.StringCache():
    df1 = pl.DataFrame(
        {
            "categorical_column": ["value3", "value2", "value1"],
            "other": ["a", "b", "c"],
        },
        schema={"categorical_column": pl.Categorical, "other": pl.String},
    )
    df2 = pl.DataFrame(
        {
            "categorical_column": ["value2", "value3", "value4"],
            "other": ["d", "e", "f"],
        },
        schema={"categorical_column": pl.Categorical, "other": pl.String},
    )

# Even outside the global string cache's scope, you can now join the
# two dataframes containing Categorical columns
df1.join(df2, on="categorical_column")
```

```
shape: (2, 3)
┌────────────────────┬───────┬─────────────┐
│ categorical_column │ other │ other_right │
│ ---                │ ---   │ ---         │
│ cat                │ str   │ str         │
╞════════════════════╪═══════╪═════════════╡
│ value2             │ b     │ d           │
│ value3             │ a     │ e           │
└────────────────────┴───────┴─────────────┘
```

You can also enable the global string cache, using:

```
pl.enable_string_cache()
```

Note, however, that this means the global string cache will *always* be used, which can be a suboptimal solution compared to using the context manager.

To retrieve the unique categories that the `Categorical` column contains, enter the Categorical namespace, and call `.get_categories()`.

```
df2.select(pl.col("categorical_column").cat.get_categories())
```

```
shape: (3, 1)
┌────────────────────┐
│ categorical_column │
│ ---                │
│ str                │
╞════════════════════╡
│ value2             │
│ value3             │
│ value4             │
└────────────────────┘
```

The last relevant attribute is the way the column is ordered in a `sort()`. There are two options:

- Physical (default): The physical (`int`) representation is used to sort.
- Lexical: The string value is used to sort.

You can set these options as soon as you create the Categorical datatype. You can swap by casting the Categorical to the other variant.

First, prepare one of the dataframes:

```
sorting_comparison_df = (
    df2
    .select(
        pl.col("categorical_column")
        .alias("categorical_lexical")
    )
    .with_columns(
        pl.col("categorical_lexical")
        .to_physical()
        .alias("categorical_physical")
    )
)
print(sorting_comparison_df)
```

```
shape: (3, 2)
┌─────────────────────┬──────────────────────┐
│ categorical_lexical ┆ categorical_physical │
│ ---                 ┆ ---                  │
│ cat                 ┆ u32                  │
╞═════════════════════╪══════════════════════╡
```

```
│ value2               │ 1                    │
│ value3               │ 0                    │
│ value4               │ 3                    │
└──────────────────────┴──────────────────────┘
```

Below, the `Categorical` column is sorted on physical representation, which can be seen in the `categorical_physical` column:

```
print(
    sorting_comparison_df
    .with_columns(
        pl.col("categorical_lexical")
        .cast(pl.Categorical("physical"))  # The default option
    )
    .sort(by="categorical_lexical")
)
```

```
shape: (3, 2)
┌─────────────────────┬──────────────────────┐
│ categorical_lexical │ categorical_physical │
│ ---                 │ ---                  │
│ cat                 │ u32                  │
╞═════════════════════╪══════════════════════╡
│ value3              │ 0                    │
│ value2              │ 1                    │
│ value4              │ 3                    │
└─────────────────────┴──────────────────────┘
```

Here it is sorted on lexical representation, which can be seen in the `categorical_column` column:

```
print(
    sorting_comparison_df
    .with_columns(
        pl.col("categorical_lexical")
        .cast(pl.Categorical("lexical"))
    )
    .sort(by="categorical_lexical")
)
```

```
shape: (3, 2)
┌─────────────────────┬──────────────────────┐
│ categorical_lexical │ categorical_physical │
│ ---                 │ ---                  │
│ cat                 │ u32                  │
╞═════════════════════╪══════════════════════╡
│ value2              │ 1                    │
│ value3              │ 0                    │
│ value4              │ 3                    │
└─────────────────────┴──────────────────────┘
```

Enum

If you know the categories of a column in advance, you can use the `Enum` data type. This data type currently uses the Categorical data type under the hood, but may later get its own implementation.

```
enum_dtype = pl.Enum(["Polar", "Panda", "Brown"])
enum_series = pl.Series(
    ["Polar", "Panda", "Brown", "Brown", "Polar"], dtype=enum_dtype
)

cat_series = pl.Series(
    ["Polar", "Panda", "Brown", "Brown", "Polar"], dtype=pl.Categorical
)
```

Enums are a new data type in Polars and at the time of writing don't have their own namespace yet.

Temporal Data

Temporal data types are specialized formats for working with time-based information, like points and intervals in time. These types allow for comparison, arithmetic, and other time-specific operations.

Polars uses several data types to store temporal data show in Table 9-5.

Table 9-5. Temporal data types

Data type	Description	Example	Storage
`Date`	Represents a calendar date without a time of day.	Birthdays	`int32` representing the amount of days since the UNIX epoch (1970-01-01).
`Datetime`	Represents a calendar date and also a time of day on that date.	Timestamps in logging	`int64` since the Unix epoch and can have different units such as ns, us, ms.
`Duration`	Represents a time interval, the difference between two points in time. It's similar to `timedelta` in Python.	Elapsed time between two events	`int64` that is created when subtracting Date/Datetime.
`Time`	Focuses only on time of day.	Scheduling of daily tasks.	`int64` representing nanoseconds since midnight.

Methods

The `Temporal` data namespace has a variety of methods for converting, describing, and manipulating data.

Conversion

The following methods allow you to convert temporal data to and from other data types or formats.

Table 9-6. Methods for conversion to and from other data types.

Function or method	Description
`Expr.dt.cast_time_unit(…)`	Cast the underlying data to another time unit.
`Expr.dt.strftime(…)`	Convert a `Date`/`Time`/`Datetime` column into a `String` column with the given format.
`Expr.dt.to_string(…)`	Convert a `Date`/`Time`/`Datetime` column into a `String` column with the given format.

Descriptive

The following methods can return attributes of the temporal data.

Table 9-7. Methods for describing temporal data.

Function or method	Description
`Expr.dt.base_utc_offset()`	Base offset from UTC.
`Expr.dt.date()`	Extract date from date(time).
`Expr.dt.datetime()`	Return `Datetime`.
`Expr.dt.day()`	Extract day from underlying `Date` representation.
`Expr.dt.days()`	Extract the total days from a `Duration` type.
`Expr.dt.dst_offset()`	Additional offset currently in effect (typically due to daylight saving time).
`Expr.dt.epoch(…)`	Get the time passed since the Unix epoch in the given time unit.
`Expr.dt.hour()`	Extract the hour from underlying `DateTime` representation.
`Expr.dt.hours()`	Extract the total hours from a `Duration` type.
`Expr.dt.is_leap_year()`	Determine whether the year of the underlying date is a leap year.
`Expr.dt.iso_year()`	Extract ISO year from underlying `Date` representation.
`Expr.dt.microsecond()`	Extract microseconds from underlying `DateTime` representation.
`Expr.dt.microseconds()`	Extract the total microseconds from a `Duration` type.
`Expr.dt.millisecond()`	Extract milliseconds from underlying `DateTime` representation.
`Expr.dt.milliseconds()`	Extract the total milliseconds from a `Duration` type.
`Expr.dt.minute()`	Extract minutes from underlying `DateTime` representation.
`Expr.dt.minutes()`	Extract the total minutes from a `Duration` type.
`Expr.dt.month()`	Extract the month from underlying `Date` representation.
`Expr.dt.nanosecond()`	Extract nanoseconds from underlying `DateTime` representation.
`Expr.dt.nanoseconds()`	Extract the total nanoseconds from a `Duration` type.
`Expr.dt.ordinal_day()`	Extract ordinal day from underlying `Date` representation.

Function or method	Description
`Expr.dt.quarter()`	Extract quarter from underlying `Date` representation.
`Expr.dt.second(…)`	Extract seconds from underlying `DateTime` representation.
`Expr.dt.seconds()`	Extract the total seconds from a `Duration` type.
`Expr.dt.time()`	Extract time.
`Expr.dt.timestamp(…)`	Return a timestamp in the given time unit.
`Expr.dt.total_days()`	Extract the total days from a `Duration` type.
`Expr.dt.total_hours()`	Extract the total hours from a `Duration` type.
`Expr.dt.total_microseconds()`	Extract the total microseconds from a `Duration` type.
`Expr.dt.total_milliseconds()`	Extract the total milliseconds from a `Duration` type.
`Expr.dt.total_minutes()`	Extract the total minutes from a `Duration` type.
`Expr.dt.total_nanoseconds()`	Extract the total nanoseconds from a `Duration` type.
`Expr.dt.total_seconds()`	Extract the total seconds from a `Duration` type.
`Expr.dt.year()`	Extract year from underlying `Date` representation.

Manipulation

The following methods allow you to manipulate temporal data.

Table 9-8. Methods for manipulating temporal data.

Function or method	Description
`Expr.dt.replace_time_zone(…)`	Replace time zone for an expression of type `Datetime`.
`Expr.dt.combine(…)`	Create a naive `Datetime` from an existing `Date`/`Datetime` expression and a `Time`.
`Expr.dt.month_start()`	Roll backward to the first day of the month.
`Expr.dt.month_end()`	Roll forward to the last day of the month.
`Expr.dt.offset_by(…)`	Offset this date by a relative time offset.
`Expr.dt.round(…)`	Divide the `Date`/`DateTime` range into buckets.
`Expr.dt.truncate(…)`	Divide the `Date`/`DateTime` range into buckets.
`Expr.dt.week()`	Extract the week from the underlying `Date` representation.
`Expr.dt.weekday()`	Extract the week day from the underlying `Date` representation.
`Expr.dt.with_time_unit(…)`	Set time unit of an expression of type `DateTime` or `Duration`.
`Expr.dt.convert_time_zone(…)`	Convert to given time zone for an expression of type `DateTime`.

Examples

The field of time series is grand, and we can't cover it all. However, we can cover some of the more common operations used in time-series analysis and illustrate how they are handled in Polars. In the coming example you'll mostly work with dates, but the methods we're about to show you should work for other temporal data types as well.

Loading from CSV

To get started with temporal data in Polars, you first need to load it. You can load temporal data from a CSV file. Use the `read_csv` method and set the `try_parse_dates` parameter to `True`:

```
pl.read_csv("data/all_stocks.csv", try_parse_dates=True)
```

```
shape: (18_476, 8)
┌────────┬────────────┬────────────┬───┬────────────┬────────────┬──────────┐
│ symbol ┆ date       ┆ open       ┆ … ┆ close      ┆ adj close  ┆ volume   │
│ ---    ┆ ---        ┆ ---        ┆   ┆ ---        ┆ ---        ┆ ---      │
│ str    ┆ date       ┆ f64        ┆   ┆ f64        ┆ f64        ┆ i64      │
╞════════╪════════════╪════════════╪═══╪════════════╪════════════╪══════════╡
│ ASML   ┆ 1999-01-04 ┆ 11.765625  ┆ … ┆ 12.140625  ┆ 7.535689   ┆ 1801867  │
│ ASML   ┆ 1999-01-05 ┆ 11.859375  ┆ … ┆ 13.96875   ┆ 8.670406   ┆ 8241600  │
│ ASML   ┆ 1999-01-06 ┆ 14.25      ┆ … ┆ 16.875     ┆ 10.474315  ┆ 16400267 │
│ ASML   ┆ 1999-01-07 ┆ 14.742188  ┆ … ┆ 16.851563  ┆ 10.459769  ┆ 17722133 │
│ ASML   ┆ 1999-01-08 ┆ 16.078125  ┆ … ┆ 15.796875  ┆ 9.805122   ┆ 10696000 │
│ …      ┆ …          ┆ …          ┆ … ┆ …          ┆ …          ┆ …        │
│ TSM    ┆ 2023-06-26 ┆ 102.019997 ┆ … ┆ 100.110001 ┆ 99.125954  ┆ 8560000  │
│ TSM    ┆ 2023-06-27 ┆ 101.150002 ┆ … ┆ 102.080002 ┆ 101.076591 ┆ 9732000  │
│ TSM    ┆ 2023-06-28 ┆ 100.5      ┆ … ┆ 100.919998 ┆ 99.927986  ┆ 8160900  │
│ TSM    ┆ 2023-06-29 ┆ 101.339996 ┆ … ┆ 100.639999 ┆ 99.650742  ┆ 7383900  │
│ TSM    ┆ 2023-06-30 ┆ 101.400002 ┆ … ┆ 100.919998 ┆ 99.927986  ┆ 11701700 │
└────────┴────────────┴────────────┴───┴────────────┴────────────┴──────────┘
```

Here, you can see by the data type in the column header that the `date` column has been read in the correct format.

Converting to and from string

Alternatively, to parse a date from a string, you can do the following:

```
df = pl.DataFrame({
    "date_str": ["2023-12-31", "2024-02-29"]
})

df = df.with_columns(
    pl.col("date_str").str.strptime(pl.Date, "%Y-%m-%d").alias("date")
)
print(df)
```

```
shape: (2, 2)
┌────────────┬────────────┐
│ date_str   ┆ date       │
│ ---        ┆ ---        │
│ str        ┆ date       │
╞════════════╪════════════╡
│ 2023-12-31 ┆ 2023-12-31 │
```

```
│ 2024-02-29 │ 2024-02-29 │
└────────────┴────────────┘
```

If you want to write a date to a string in a certain format, you can do the following:

```
df = df.with_columns(
    pl.col("date").dt.to_string("%d-%m-%Y").alias("formatted_date")
)

print(df)
```

```
shape: (2, 3)
┌────────────┬────────────┬────────────────┐
│ date_str   ┆ date       ┆ formatted_date │
│ ---        ┆ ---        ┆ ---            │
│ str        ┆ date       ┆ str            │
╞════════════╪════════════╪════════════════╡
│ 2023-12-31 ┆ 2023-12-31 ┆ 31-12-2023     │
│ 2024-02-29 ┆ 2024-02-29 ┆ 29-02-2024     │
└────────────┴────────────┴────────────────┘
```

Here, the formatting you provide to the `to_string()` method is `%d-%m-%Y`, which means that the day, month, and year are separated by hyphens. The options for formatting are defined in the chrono strftime documentation, which Polars uses.

Generating Ranges

Instead of loading data from other sources, it's also possible to generate date ranges and datetime ranges directly in Polars:

```
from datetime import date
df = pl.DataFrame(
    {
        "date": pl.date_range(
            start=date(2023,12,31),  ❶
            end=date(2024,1,15),
            interval="1w",  ❷
            eager=True,  ❸
        ),
    }
)
print(df)
```

```
shape: (3, 1)
┌────────────┐
│ date       │
│ ---        │
│ date       │
╞════════════╡
│ 2023-12-31 │
│ 2024-01-07 │
│ 2024-01-14 │
└────────────┘
```

❶ For the `start` and `end` parameters, you can use the `datetime.date` type from the Python standard library.

❷ The `interval` parameter can be set to a string that represents the interval: for example, "1w" for one week, "1d" for one day, "1h" for one hour, and so on.

❸ Set the `eager` parameter to `True` to return the range as a Series object, or `False` to return an Expression instead. Since we're working with a DataFrame constructor here, we can't use an Expression because it would lead to a `TypeError: passing Expr objects to the DataFrame constructor is not supported`.

Time Zones

One of the most unpleasant things about working with temporal data is time zones. Daylight saving time in particular can be a real pain. For this reason, Universal Time Coordinated (UTC) is often used in time-series analysis, because it's a universal fixed timezone. From there you can convert to any timezone you want.

In the next example we've got a dataset that's in UTC, and we want to convert it to the timezone of Amsterdam: Central European Time (CEST).

```
df = pl.DataFrame(  ❶
    {
        "utc_mixed_offset_data": [
            "2021-03-27T00:00:00+0100",
            "2021-03-28T00:00:00+0100",
            "2021-03-29T00:00:00+0200",
            "2021-03-30T00:00:00+0200",
        ]
    }
)
df = (
    df.with_columns(
        pl.col("utc_mixed_offset_data")
        .str.to_datetime("%Y-%m-%dT%H:%M:%S%z")  ❷
        .alias("parsed_data")
    ).with_columns(
        pl.col("parsed_data")
        .dt.convert_time_zone("Europe/Amsterdam")  ❸
        .alias("converted_data")
    )
)
print(df)
```

```
shape: (4, 3)
┌───────────────────────┬──────────────────────────┬──────────────────────────┐
│ utc_mixed_offset_data ┆ parsed_data              ┆ converted_data           │
│ ---                   ┆ ---                      ┆ ---                      │
│ str                   ┆ datetime[μs, UTC]        ┆ datetime[μs,             │
```

```
│                          │                         │ Europe/Amsterdam]        │
╞══════════════════════════╪═════════════════════════╪══════════════════════════╡
│ 2021-03-27T00:00:00+0100 │ 2021-03-26 23:00:00 UTC │ 2021-03-27 00:00:00 CET  │
│ 2021-03-28T00:00:00+0100 │ 2021-03-27 23:00:00 UTC │ 2021-03-28 00:00:00 CET  │
│ 2021-03-29T00:00:00+0200 │ 2021-03-28 22:00:00 UTC │ 2021-03-29 00:00:00 CEST │
│ 2021-03-30T00:00:00+0200 │ 2021-03-29 22:00:00 UTC │ 2021-03-30 00:00:00 CEST │
└──────────────────────────┴─────────────────────────┴──────────────────────────┘
```

❶ We create a DataFrame with a column that contains dates with mixed offsets from strings.

❷ We parse the strings to a datetime with the `str.to_datetime()` method. The `%z` in the format string is used to parse the timezone offset.

❸ We convert the parsed datetime to the timezone of Amsterdam with the `dt.convert_time_zone()` method.

In the resulting DataFrame, you can see that the dates have been converted to the timezone of Amsterdam. The offset has been parsed according to Central Eastern Time (CET) and Central European Summer Time (CEST).

In Chapter 10 we'll show you how to summarize and aggregate temporal data using window functions, dynamic group by operations, and more!

List

There are three ways to store a collection of data points in a single column: using an `Array`, a `List`, or a `Struct`.

The `List` type can contain lists of varying lengths with values of the same data type.

> **`pl.List != list`**
>
> The Polars `List`, which holds only values of the *same data type*, is different from the Python `list`, which can contain values of *different data types*. It is possible to achieve the same in Polars by using the `Object` type to store a Python list, but this is not recommended, because the contents will be binary objects of serialized Python data. This means that there are no special list manipulations, there's no room for the optimizations that normally apply to Polars data types, and all functions performed on it have to be done in Python, which is slower than running them in Rust.

The List type is implemented in memory as Arrow's *Variable Size List Layout*. Similar to the String type, it has a contiguous data buffer and an offset buffer pointing to the memory locations of the values in the data buffer.

Methods

Table 9-9. Methods for the List type

Function or method	Description
`Expr.list.all()`	Evaluate whether all Boolean values in a list are true.
`Expr.list.any()`	Evaluate whether any Boolean value in a list is true.
`Expr.list.drop_nulls()`	Drop all null values in the list.
`Expr.list.arg_max()`	Retrieve the index of the maximum value in every sublist.
`Expr.list.arg_min()`	Retrieve the index of the minimal value in every sublist.
`Expr.list.concat(…)`	Concatenate the arrays in a Series in linear time.
`Expr.list.contains(…)`	Check if sublists contain the given item.
`Expr.list.count_match(…)`	Count how often the value produced by element occurs.
`Expr.list.count_matches(…)`	Count how often the value produced by element occurs.
`Expr.list.diff(…)`	Calculate the first discrete difference between shifted items of every sublist.
`Expr.list.eval(…)`	Run any Polars expression against the list's elements.
`Expr.list.explode()`	Return a column with a separate row for every list element.
`Expr.list.first()`	Get the first value of the sublists.
`Expr.list.gather(…)`	Take sublists by multiple indices.
`Expr.list.get(…)`	Get the value by index in the sublists.
`Expr.list.head(…)`	Slice the first n values of every sublist.
`Expr.list.join(…)`	Join all string items in a sublist and place a separator between them.
`Expr.list.last()`	Get the last value of the sublists.
`Expr.list.len()`	Return the number of elements in each list.
`Expr.list.lengths()`	Return the number of elements in each list.
`Expr.list.max()`	Compute the max value of the lists in the array.
`Expr.list.mean()`	Compute the mean value of the lists in the array.
`Expr.list.min()`	Compute the min value of the lists in the array.
`Expr.list.reverse()`	Reverse the arrays in the list.
`Expr.list.sample(…)`	Sample from this list.
`Expr.list.set_difference(…)`	Compute the set difference between the elements in this list and the elements of other.
`Expr.list.set_intersection(…)`	Compute the set intersection between the elements in this list and the elements of other.
`Expr.list.set_symmetric_difference(…)`	Compute the set summetric difference between the elements in this list and the elements of other.
`Expr.list.set_union(…)`	Compute the set union between the elements in this list and the elements of other.
`Expr.list.shift(…)`	Shift list values by the given number of indices.

Function or method	Description
`Expr.list.slice(…)`	Slice every sublist.
`Expr.list.sort(…)`	Sort the lists in this column.
`Expr.list.sum()`	Sum all the lists in the array.
`Expr.list.tail(…)`	Slice the last n values of every sublist.
`Expr.list.take(…)`	Take sublists by multiple indices.
`Expr.list.to_array(…)`	Convert a `List` column into an `Array` column with the same inner data type.
`Expr.list.to_struct(…)`	Convert the `Series` of type `List` to a `Series` of type `Struct`.
`Expr.list.unique(…)`	Get the unique/distinct values in the list.

Examples

Let's show you some of the methods you can use with the `List` type.

You can use the `all` and `any` methods to evaluate whether all or any Boolean values in a list are true:

```
bool_df = pl.DataFrame({
    "values": [[True, True], [False, False, True], [False]]
})
print(
    bool_df
    .with_columns(
        pl.col("values")
        .list.all()
        .alias("all values"),
        pl.col("values")
        .list.any()
        .alias("any values")
    )
)
```

```
shape: (3, 3)
┌─────────────────────┬────────────┬────────────┐
│ values              ┆ all values ┆ any values │
│ ---                 ┆ ---        ┆ ---        │
│ list[bool]          ┆ bool       ┆ bool       │
╞═════════════════════╪════════════╪════════════╡
│ [true, true]        ┆ true       ┆ true       │
│ [false, false, true]┆ false      ┆ true       │
│ [false]             ┆ false      ┆ false      │
└─────────────────────┴────────────┴────────────┘
```

A powerful method that combines well with `any()` and `all()` is the `eval` method. This method allows you to run any Polars expression against the list's elements. In the following example, we'll use the `eval` method to multiply the list elements by 10:

```
df = pl.DataFrame({
    "values": [[10, 20], [30, 40, 50], [60]]
})
print(
    df
    .with_columns(
        pl.col("values")
        .list.eval(
            pl.element() > 40,  ❶
            parallel=True,  ❷
        )
        .alias("values > 40")
    )
    .with_columns(  ❸
        pl.col("values > 40")
        .list.all()  ❹
        .alias("all values > 40")
    )
)
```

```
shape: (3, 3)
┌──────────────┬───────────────────────┬─────────────────┐
│ values       ┆ values > 40           ┆ all values > 40 │
│ ---          ┆ ---                   ┆ ---             │
│ list[i64]    ┆ list[bool]            ┆ bool            │
╞══════════════╪═══════════════════════╪═════════════════╡
│ [10, 20]     ┆ [false, false]        ┆ false           │
│ [30, 40, 50] ┆ [false, false, true]  ┆ false           │
│ [60]         ┆ [true]                ┆ true            │
└──────────────┴───────────────────────┴─────────────────┘
```

❶ The `element()` method is used to access the elements of the list.

❷ Because the `parallel` parameter is set to `True`, the `eval()` method will run the expression in parallel. This is off by default, but can seriously speed up your calculations if the expression you run allows for parallelism.

❸ To ensure parallel processing within `with_columns()`, any further modifications to a newly created column require a separate subsequent `with_columns()` call.

❹ The `all()` method is used to evaluate whether all Boolean values in a list are true.

You can unpack a list to separate rows using the `explode` method:

```
df.explode("values")
```

```
shape: (6, 1)
┌────────┐
│ values │
│ ---    │
│ i64    │
```

```
└───────┘
┌───────┐
│ 10    │
│ 20    │
│ 30    │
│ 40    │
│ 50    │
│ 60    │
└───────┘
```

Array

The `Array` type can hold arrays of fixed lengths with values of the same data type. It is analogous to Numpy's `ndarray` type. The `Array` type is implemented in memory by Arrow's Fixed Size List Layout. For this type, the data buffer is also contiguous, just like `List`, but the offset buffer isn't needed because the length is constant. This make the type more memory-efficient and performant, because there are fewer lookups to load the relevant data.

Methods

The `Array` type has a variety of methods for converting, describing, and manipulating data.

Table 9-10. Methods for the Array type

Function or method	Description
`Expr.arr.max()`	Compute the max values of the sub-arrays.
`Expr.arr.min()`	Compute the min values of the sub-arrays.
`Expr.arr.median()`	Compute the median of the values of the sub-arrays.
`Expr.arr.sum()`	Compute the sum values of the sub-arrays.
`Expr.arr.std(…)`	Compute the std of the values of the sub-arrays.
`Expr.arr.to_list()`	Convert an `Array` column into a `List` column with the same inner data type.
`Expr.arr.unique(…)`	Get the unique/distinct values in the array.
`Expr.arr.var(…)`	Compute the variance of the values of the sub-arrays.
`Expr.arr.all()`	Evaluate whether all Boolean values are true for every subarray.
`Expr.arr.any()`	Evaluate whether any Boolean value is true for every subarray.
`Expr.arr.sort(…)`	Sort the arrays in this column.
`Expr.arr.reverse()`	Reverse the arrays in this column.
`Expr.arr.arg_min()`	Retrieve the index of the minimal value in every sub-array.
`Expr.arr.arg_max()`	Retrieve the index of the maximum value in every sub-array.
`Expr.arr.get(…)`	Get the value by index in the sub-arrays.
`Expr.arr.first()`	Get the first value of the sub-arrays.

Function or method	Description
`Expr.arr.last()`	Get the last value of the sub-arrays.
`Expr.arr.join(…)`	Join all string items in a sub-array and place a separator between them.
`Expr.arr.explode()`	Returns a column with a separate row for every array element.
`Expr.arr.contains(…)`	Check if sub-arrays contain the given item.
`Expr.arr.count_matches(…)`	Count how often the value produced by element occurs.
`Expr.arr.to_struct(…)`	Convert the `Series` of type `Array` to a `Series` of type `Struct`.
`Expr.arr.shift(…)`	Shift array values by the given number of indices.

Examples

In order to showcase the `Array` type, create a DataFrame with an `Array` column. In the following example, you'll create a DataFrame with an `Array` column that contains arrays of integers that represent temperatures in different locations:

```
df = pl.DataFrame([
    pl.Series(
        "location",
        ["Paris", "Amsterdam", "Barcelona"],
        dtype=pl.String
    ),
    pl.Series(
        "temperatures",
        [
            [23, 27, 21, 22, 24, 23, 22],
            [17, 19, 15, 22, 18, 20, 21],
            [30, 32, 28, 29, 34, 33, 31]
        ],
        dtype=pl.Array(pl.Int64, width=7),
    ),
])
print(df)
```

```
shape: (3, 2)
┌───────────┬────────────────┐
│ location  ┆ temperatures   │
│ ---       ┆ ---            │
│ str       ┆ array[i64, 7]  │
╞═══════════╪════════════════╡
│ Paris     ┆ [23, 27, … 22] │
│ Amsterdam ┆ [17, 19, … 21] │
│ Barcelona ┆ [30, 32, … 31] │
└───────────┴────────────────┘
```

Some methods that are available for the `Array` type are `median`, `max`, and `arg_max`.

```
print(
    df
    .with_columns(
```

```
            pl.col("temperatures")
            .arr.median()
            .alias("median"),
            pl.col("temperatures")
            .arr.max()
            .alias("max"),
            pl.col("temperatures")
            .arr.arg_max()
            .alias("warmest_weekday")
        )
    )
```

```
shape: (3, 5)
┌───────────┬────────────────┬────────┬─────┬─────────────────┐
│ location  ┆ temperatures   ┆ median ┆ max ┆ warmest_weekday │
│ ---       ┆ ---            ┆ ---    ┆ --- ┆ ---             │
│ str       ┆ array[i64, 7]  ┆ f64    ┆ i64 ┆ u32             │
╞═══════════╪════════════════╪════════╪═════╪═════════════════╡
│ Paris     ┆ [23, 27, … 22] ┆ 23.0   ┆ 27  ┆ 1               │
│ Amsterdam ┆ [17, 19, … 21] ┆ 19.0   ┆ 22  ┆ 3               │
│ Barcelona ┆ [30, 32, … 31] ┆ 31.0   ┆ 34  ┆ 4               │
└───────────┴────────────────┴────────┴─────┴─────────────────┘
```

In the resulting DataFrame, you can see that the `median` column contains the median temperature for each location, the `max` column contains the maximum temperature for each location, and the `warmest_weekday` column contains the index of the warmest weekday for each location.

Structs

Structs are a nested type for storing multiple columns in a single column. On the row level, this can be seen as a dictionary. The keys are the column names, which are called *fields*, and the values are the values of the field for that row. The struct data type is the idiomatic way of working with multiple columns in Polars. Polars runs on expressions, and by definition, an expression can only take one column as input and give one column as output (`Fn(Series) -> Series`).

By encapsulating multiple columns within a struct, you can still do multi column operations, while keeping the expression paradigm intact. Casting multiple columns to a struct does not duplicate data, but allows the new struct type to point to existing data buffers in memory, ensuring efficient memory usage.

Methods

The `Struct` type has the following methods:

Table 9-11. Methods for the struct type

Function or method	Description
`Expr.struct.field(…)`	Retrieve a Struct field as a new Series.
`Expr.struct.json_encode()`	Convert this struct to a string column with json values.
`Expr.struct.rename_fields(…)`	Rename the fields of the struct.

Examples

To play around with structs, you have to make them first. There are a number of methods that return structs, or you can create them by constructing a DataFrame using a dictionary:

```
df = pl.DataFrame({
    "struct_column": [
        {"a": 1, "b": 2},
        {"a": 3, "b": 4},
        {"a": 5, "b": 6},
    ]
})
print(df)
```

```
shape: (3, 1)
┌───────────────┐
│ struct_column │
│ ---           │
│ struct[2]     │
╞═══════════════╡
│ {1,2}         │
│ {3,4}         │
│ {5,6}         │
└───────────────┘
```

You can retrieve values from a struct using the `field` method:

```
df.select(pl.col("struct_column").struct.field("a"))
```

```
shape: (3, 1)
┌─────┐
│ a   │
│ --- │
│ i64 │
╞═════╡
│ 1   │
│ 3   │
│ 5   │
└─────┘
```

To return multiple columns, you can use the `unnest` method. Note that the `unn est` method is not part of the struct namespace and should be called on the Data‐Frame/LazyFrame/Series level.

```
df = df.unnest("struct_column")
print(df)
```

```
shape: (3, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ i64 │
╞═════╪═════╡
│ 1   ┆ 2   │
│ 3   ┆ 4   │
│ 5   ┆ 6   │
└─────┴─────┘
```

If you want to do the opposite and combine multiple columns, cast them to a struct:

```
df.select(
    "a",
    "b",
    pl.struct(
        pl.col("a"),
        pl.col("b")
    ).alias("struct_column"),
)
```

```
shape: (3, 3)
┌─────┬─────┬───────────────┐
│ a   ┆ b   ┆ struct_column │
│ --- ┆ --- ┆ ---           │
│ i64 ┆ i64 ┆ struct[2]     │
╞═════╪═════╪═══════════════╡
│ 1   ┆ 2   ┆ {1,2}         │
│ 3   ┆ 4   ┆ {3,4}         │
│ 5   ┆ 6   ┆ {5,6}         │
└─────┴─────┴───────────────┘
```

One of the common functions that returns a struct is `value_counts()`. This function is used to count the occurrences of unique values in a column. Since an expression can only return one Series, `value_counts()` returns a `Struct` column with two fields: the original column name being counted and `count`.

First, create a DataFrame with a struct column:

```
df = pl.DataFrame({
    "fruit": ["cherry", "apple", "banana", "banana", "apple", "banana"],
})
print(df)
```

```
shape: (6, 1)
┌────────┐
│ fruit  │
│ ---    │
│ str    │
╞════════╡
```

```
│ cherry │
│ apple  │
│ banana │
│ banana │
│ apple  │
│ banana │
└────────┘
```

You can count the number of occurrences per unique element in the `fruit` column using `value_counts()`:

```python
print(
    df
    .select(
        pl.col("fruit")
        .value_counts(sort=True)
    )
)
```

```
shape: (3, 1)
┌─────────────┐
│ fruit       │
│ ---         │
│ struct[2]   │
╞═════════════╡
│ {"banana",3} │
│ {"apple",2}  │
│ {"cherry",1} │
└─────────────┘
```

In the resulting DataFrame, you can see that the `value_counts` method is called with the `sort` parameter set to `True`. This means that the values are sorted in descending order by their counts.

You can then unnest these structs to separate columns:

```python
print(
    df.select(
        pl.col("fruit")
        .value_counts(sort=True)
    )
    .unnest("fruit")
)
```

```
shape: (3, 2)
┌────────┬───────┐
│ fruit  ┆ count │
│ ---    ┆ ---   │
│ str    ┆ u32   │
╞════════╪═══════╡
│ banana ┆ 3     │
│ apple  ┆ 2     │
│ cherry ┆ 1     │
└────────┴───────┘
```

In the resulting DataFrame, you can see that the `value_counts` method has been unnested to separate columns.

Conclusion

This chapter covers the special data types in Polars that have their own namespaces and how to work with them. You learned about:

- Strings, with a focus on optimizing for variable length using an optimized memory layout
- Categoricals' and Enums' memory-efficient way of working with repeated string data
- How the temporal data types `Date`, `DateTime`, `Time`, and `Duration` address the challenges of working with time-based information.
- How nested data types like `List`, `Array`, and `Struct` allow you to store collections of data points in a single column

With these data types, you can work with a wide variety of data using rich set of methods Polars provides to work with them. In the next chapter, you'll learn how to summarize and aggregate data.

CHAPTER 10

Summarizing and Aggregating

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 13th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

In most analyses you will want to eventually summarize or aggregate your data to answer a question or gain insights. Besides the basic aggregations like `sum`, `mean`, `min`, and `max`, that offer you insights over the whole dataset, Polars also offers several functions to analyse subsets of your data. These functions are part of the `group_by` context, which allows you to group your data based on one or more columns, or an expression, and then apply specific calculations or transformations to each group separately. This is a powerful way to analyze large datasets and gain valuable insights.

This chapter will teach you about the `group_by` context and its available methods, and show you how to use them to analyze your data. It will also go into working with grouping data based on temporal values using `group_by_dynamic`, `rolling`, and `over`. And additionally we'll show some optimizations you can use to improve performance.

Group by Context

Think of a "group by" operation in Polars as similar to asking guests at a big party to gather into groups based on something they have in common, like their birth month. Each group (or "group by" category) represents a month, and all guests born in that month join that group. Once everyone is grouped, you can do things like count how many guests are in each group, find out who is the tallest in each group, or calculate the average height of guests in each group. This is similar to how "group by" works in Polars: it groups data by one or more columns, or an expression, and then allows you to apply specific calculations or transformations to each group separately. You can organize your data into these groups using Polars' `group_by` function.

After grouping the data, you can use aggregation functions to summarize the data within each group and gain valuable insights. For instance, you can calculate the average purchase amount for each customer category by grouping them by customer ID. Similarly, you can find the total number of sensor readings taken at each location by grouping them by location. The group_by function is a game-changer for analyzing large datasets. It allows you to:

- Organize and categorize data based on specific attributes.
- Apply aggregation functions to extract meaningful insights from each group.
- Gain a deeper understanding of your data by focusing on specific aspects.

Table 10-1 lists the methods you can use out of the box to analyze data.

Table 10-1. The available methods in the Group By context

Method	Description
`pl.GroupBy.__iter__()`	Allows iteration over the groups of the group by operation.
`pl.GroupBy.agg(…)`	Compute aggregations for each group of a group by operation.
`pl.GroupBy.all()`	Aggregate the groups into Series.
`pl.GroupBy.apply(…)`	Apply a custom/user-defined function (UDF) over the groups as a sub-DataFrame.
`pl.GroupBy.first()`	Aggregate the first values in the group.
`pl.GroupBy.head(…)`	Get the first n rows of each group.
`pl.GroupBy.last()`	Aggregate the last values in the group.
`pl.GroupBy.len()`	Return the number of rows in each group.
`pl.GroupBy.map_groups(…)`	Apply a custom/user-defined function (UDF) over the groups as a sub-DataFrame.
`pl.GroupBy.max()`	Reduce the groups to the maximal value.
`pl.GroupBy.mean()`	Reduce the groups to the mean values.
`pl.GroupBy.median()`	Return the median per group.
`pl.GroupBy.min()`	Reduce the groups to the minimal value.

Method	Description
`pl.GroupBy.n_unique()`	Count the unique values per group.
`pl.GroupBy.quantile(…)`	Compute the quantile per group.
`pl.GroupBy.sum()`	Reduce the groups to the sum.
`pl.GroupBy.tail(…)`	Get the last n rows of each group.

To showcase the `group_by` function, let's start by loading a dataset of the Top2000 hit list for 2023 from the Netherlands. This dataset contains information about the top 2000 songs of all times as chosen by the Dutch in 2023, including their position in the Top2000, the artist, song title, and year of release.

```
import polars as pl

top2000 = pl.read_excel(
    "data/top2000-2023.xlsx",
    read_options={"skip_rows": 1},
    engine="calamine"
).set_sorted("positie")
```

`set_sorted` enabling fast path algorithms

Here we use the `set_sorted` to tell Polars the `"positie"` column is sorted. This is information we know, but Polars doesn't. By telling Polars this, it can make use of some fast path optimizations that are only possible when it knows the data is sorted. A *fast path optimization* is a way to make a program run faster by taking advantage of some special knowledge about the data.

You can show the values the group_by aggregation functions are applied to by turning the groups into lists, which you can do by running the following code:

```
(
    top2000
    .group_by("jaar")
    .agg(  ❶
        (
            pl.concat_str(
                pl.col("artiest"),
                pl.lit(" - "),
                pl.col("titel")
            )  ❷
        ).alias("songs"),
    )
    .sort("jaar", descending=True)
)

shape: (67, 2)
```

```
│ jaar │ songs                                                │
│ ---  │ ---                                                  │
│ i64  │ list[str]                                            │
╞══════╪══════════════════════════════════════════════════════╡
│ 2022 │ ["Son Mieux - Multicolor", "Bankzitters - Je Blik …  │
│ 2021 │ ["Goldband - Noodgeval", "Bankzitters - Stapelgek"…  │
│ 2020 │ ["DI-RECT - Soldier On", "Miss Montreal - Door De …  │
│ 2019 │ ["Danny Vera - Roller Coaster", "Floor Jansen & He…  │
│ 2018 │ ["Lady Gaga & Bradley Cooper - Shallow", "White Li…  │
│ …    │ …                                                    │
│ 1960 │ ["Etta James - At Last", "Shadows - Apache"]         │
│ 1959 │ ["Jacques Brel - Ne Me Quitte Pas", "Elvis Presley…  │
│ 1958 │ ["Chuck Berry - Johnny B. Goode", "Ella Fitzgerald…  │
│ 1957 │ ["Johnny Cash - I Walk The Line", "Elvis Presley -…  │
│ 1956 │ ["Elvis Presley - Love Me Tender", "Elvis Presley …  │
└──────┴──────────────────────────────────────────────────────┘
```

❶ `.agg()` allows you to list the aggregations you want to apply to the group data. We will come back to this after going over the standard group_by aggregations.

❷ You create a list of song titles by concatenating the strings of the artist, a separating dash, and then the title.

The Descriptives

In the following example you'll get the top 3 songs per year of release for the 3 most recent years of release, using the `head()` method.

```
(
    top2000
    .group_by("jaar", maintain_order=True)  ❶
    .head(3)  ❷
    .sort("jaar", descending=True)
    .head(9)  ❸
)
```

```
shape: (9, 4)
┌──────┬─────────┬──────────────────────┬──────────────────────┐
│ jaar │ positie │ titel                │ artiest              │
│ ---  │ ---     │ ---                  │ ---                  │
│ i64  │ i64     │ str                  │ str                  │
╞══════╪═════════╪══════════════════════╪══════════════════════╡
│ 2022 │ 179     │ Multicolor           │ Son Mieux            │
│ 2022 │ 370     │ Je Blik Richting Mij │ Bankzitters          │
│ 2022 │ 395     │ L'enfer              │ Stromae              │
│ 2021 │ 55      │ Noodgeval            │ Goldband             │
│ 2021 │ 149     │ Stapelgek            │ Bankzitters          │
│ 2021 │ 210     │ Dat Heb Jij Gedaan   │ Meau                 │
│ 2020 │ 19      │ Soldier On           │ DI-RECT              │
│ 2020 │ 38      │ Door De Wind         │ Miss Montreal        │
```

2020	77	Impossible (Orchestr…	Nothing But Thieves

❶ The Top2000 dataset is sorted by position, and since you are using this sort order, you want to maintain it. By setting the `maintain_order` parameter to `True` you make sure to preserve that order. When set to `False`, its default value, this order can be lost because of the parallel processing of groups.

❷ You want to get the top 3 songs per year of release, so use the `head` method. This `head(3)` method is applied to the `group_by` context, which means it will return the first 3 songs per group.

❸ You use the `head(9)` method again, but this time to get the top 3 songs per year of release for the 3 most recent years of release.

In the same vein, you can get the lowest 3 positions per year of release for the 3 most recent years of release, using the `tail()` method.

```
(
    top2000
    .group_by("jaar", maintain_order=True)
    .tail(3)
    .sort("jaar", descending=True)
    .head(9)
)
```

```
shape: (9, 4)
```

jaar --- i64	positie --- i64	titel --- str	artiest --- str
2022	1391	De Diepte	S10
2022	1688	Zeit	Rammstein
2022	1716	THE LONELIEST	Måneskin
2021	1865	Bon Gepakt	Donnie & Rene Froger
2021	1978	Hold On	Armin van Buuren ft.…
2021	2000	Drivers License	Olivia Rodrigo
2020	1824	Smoorverliefd	Snelle
2020	1879	The Business	Tiësto
2020	1902	Levitating	Dua Lipa ft. DaBaby

Aliases for head and `tail`

The `first()` method is the same as the `head(1)` method, but it's more explicit and easier to read. The `last()` method is also the same as the `tail(1)` method.

Now say you want to know the top 10 of artists based on the number of songs in the Top2000. You can accomplish this by grouping the data by the artist and then getting the length of the groups using the `len()` method.

```
(
    top2000
    .group_by("artiest")
    .len()
    .sort("len", descending=True)
    .head(10)
)
```

```
shape: (10, 2)
┌────────────────────┬─────┐
│ artiest            ┆ len │
│ ---                ┆ --- │
│ str                ┆ u32 │
╞════════════════════╪═════╡
│ Queen              ┆ 34  │
│ The Beatles        ┆ 31  │
│ ABBA               ┆ 25  │
│ The Rolling Stones ┆ 22  │
│ Bruce Springsteen  ┆ 22  │
│ Michael Jackson    ┆ 20  │
│ Fleetwood Mac      ┆ 20  │
│ Coldplay           ┆ 20  │
│ David Bowie        ┆ 18  │
│ U2                 ┆ 18  │
└────────────────────┴─────┘
```

It looks like Dutch people really like Queen, The Beatles, and ABBA!

The next methods are better explained with a different dataset. This dataset contains sales data which allows us to showcase all kinds of analyses.

```
df = pl.read_csv("data/sales_data.csv")
df.columns
```

```
['Date',
 'Age_Group',
 'Country',
 'Product_Category',
 'Sub_Category',
 'Product',
 'Order_Quantity',
 'Unit_Cost',
 'Unit_Price',
 'Profit',
 'Cost',
 'Revenue']
```

Let's kick it off by demonstrating the `min` and `max` methods. Say you want to know the most expensive category and subcategory. You can accomplish this by grouping the data by the product category and subcategory and then getting the maximum unit price using the `max()` method.

```
(
    df
    .select("Product_Category", "Sub_Category", "Unit_Price") ❶
    .group_by("Product_Category", "Sub_Category") ❷
    .max()
    .sort("Unit_Price", descending=True) ❸
    .head(10)
)
```

```
shape: (10, 3)
┌──────────────────┬─────────────────┬────────────┐
│ Product_Category ┆ Sub_Category    ┆ Unit_Price │
│ ---              ┆ ---             ┆ ---        │
│ str              ┆ str             ┆ i64        │
╞══════════════════╪═════════════════╪════════════╡
│ Bikes            ┆ Road Bikes      ┆ 3578       │
│ Bikes            ┆ Mountain Bikes  ┆ 3400       │
│ Clothing         ┆ Vests           ┆ 2384       │
│ Bikes            ┆ Touring Bikes   ┆ 2384       │
│ Accessories      ┆ Bike Stands     ┆ 159        │
│ Accessories      ┆ Bike Racks      ┆ 120        │
│ Clothing         ┆ Socks           ┆ 70         │
│ Clothing         ┆ Shorts          ┆ 70         │
│ Accessories      ┆ Hydration Packs ┆ 55         │
│ Clothing         ┆ Jerseys         ┆ 54         │
└──────────────────┴─────────────────┴────────────┘
```

❶ You select the relevant columns so you can focus on the data you need.

❷ You group the data by the product category and sub category. Unlike the previous examples, you group by two columns. This means that the `max()` method will return the maximum unit price for each combination of product category and sub category.

❸ You sort the data by unit price in descending order and get the top 10 most expensive sub categories.

Now let's say you want to know the total profit per country. You can accomplish this by grouping the data by country, then getting the sum of the profit using the `sum()` method.

```
(
    df
    .select("Country", "Profit")
    .group_by("Country")
    .sum()
    .sort("Profit", descending=True)
)
```

```
shape: (6, 2)
┌────────────────┬──────────┐
│ Country        ┆ Profit   │
│ ---            ┆ ---      │
│ str            ┆ i64      │
╞════════════════╪══════════╡
│ United States  ┆ 11073644 │
│ Australia      ┆ 6776030  │
│ United Kingdom ┆ 4413853  │
│ Canada         ┆ 3717296  │
│ Germany        ┆ 3359995  │
│ France         ┆ 2880282  │
└────────────────┴──────────┘
```

How about the subcategories with the most unique products? You can accomplish this by grouping the data by subcategory and then getting the number of unique products using the `n_unique()` method.

```
(
    df
    .select("Sub_Category", "Product")
    .group_by("Sub_Category")
    .n_unique()
    .sort("Product", descending=True)
    .head(10)
)
```

```
shape: (10, 2)
┌───────────────────┬─────────┐
│ Sub_Category      ┆ Product │
│ ---               ┆ ---     │
│ str               ┆ u32     │
╞═══════════════════╪═════════╡
│ Road Bikes        ┆ 38      │
│ Mountain Bikes    ┆ 28      │
│ Touring Bikes     ┆ 22      │
│ Tires and Tubes   ┆ 11      │
│ Jerseys           ┆ 8       │
│ Vests             ┆ 4       │
│ Gloves            ┆ 4       │
│ Socks             ┆ 3       │
│ Bottles and Cages ┆ 3       │
│ Helmets           ┆ 3       │
└───────────────────┴─────────┘
```

Say you want to know the average order quantity per age group. You can accomplish this by grouping the data by the age group and then getting the mean of the order quantity using the `mean()` method.

```
(
    df
    .select("Age_Group", "Order_Quantity")
    .group_by("Age_Group")
```

```
    .mean()
    .sort("Order_Quantity", descending=True)
)
```

```
shape: (4, 2)
┌────────────────────┬────────────────┐
│ Age_Group          ┆ Order_Quantity │
│ ---                ┆ ---            │
│ str                ┆ f64            │
╞════════════════════╪════════════════╡
│ Seniors (64+)      ┆ 13.530137      │
│ Youth (<25)        ┆ 12.124018      │
│ Adults (35-64)     ┆ 12.045303      │
│ Young Adults (25-34) ┆ 11.560899    │
└────────────────────┴────────────────┘
```

Additionally you can use the `quantile` method to get the 25th, 50th, and 75th percentiles of the order quantity per age group.

```
(
    df
    .select("Age_Group", "Revenue")
    .group_by("Age_Group")
    .quantile(0.9)
    .sort("Revenue", descending=True)
)
```

```
shape: (4, 2)
┌──────────────────────┬─────────┐
│ Age_Group            ┆ Revenue │
│ ---                  ┆ ---     │
│ str                  ┆ f64     │
╞══════════════════════╪═════════╡
│ Young Adults (25-34) ┆ 2227.0  │
│ Adults (35-64)       ┆ 2217.0  │
│ Youth (<25)          ┆ 1997.0  │
│ Seniors (64+)        ┆ 943.0   │
└──────────────────────┴─────────┘
```

It seems like the Young Urban Professionals are at it again! In the Netherlands the Young Urban Professionals, known as "*Yuppies*", are often associated with high income and high spending. It seems like the Yuppies are spending a lot on orders, with the 90th percentile of their order quantity being 2,227!

Another method is the `median` method, which is an alias to `quantile(0.5)`.

Now that we've seen the basic aggregations available in the `group_by` context, it's time to get weird with it and move on to the advanced stuff.

The Advanced

All the methods we've discussed so far are great for simple aggregations. However, sometimes you want to do more complex aggregations, do multiple aggregations at the same time, or even apply your own custom aggregation functions. This is where the multi-functional `agg` comes in.

The `agg` method allows you to:

- Aggregate column elements into a list per group.
- Control what the resulting column names will be.
- Use expressions to apply multiple aggregation functions at the same time, and to multiple columns.
- Apply your own custom aggregation functions.

Let's go through these one by one.

First, `agg` allows you to aggregate column elements into a list per group. This is done by passing a column selector to the `agg` method. Let's see how by aggregating the profit and revenue per country:

```
(
    df
    .select("Country", "Profit", "Revenue")
    .group_by("Country")
    .agg(
        pl.col("Profit"),
        pl.col("Revenue"),
    )
)
```

```
shape: (6, 3)
┌────────────────┬─────────────────────┬──────────────────────┐
│ Country        ┆ Profit              ┆ Revenue              │
│ ---            ┆ ---                 ┆ ---                  │
│ str            ┆ list[i64]           ┆ list[i64]            │
╞════════════════╪═════════════════════╪══════════════════════╡
│ Canada         ┆ [590, 590, … 630]   ┆ [950, 950, … 1014]   │
│ Germany        ┆ [160, 53, … 746]    ┆ [295, 98, … 1250]    │
│ United Kingdom ┆ [1053, 1053, … 112] ┆ [1728, 1728, … 184]  │
│ Australia      ┆ [1366, 1188, … 655] ┆ [2401, 2088, … 1183] │
│ United States  ┆ [524, 407, … 542]   ┆ [929, 722, … 878]    │
│ France         ┆ [427, 427, … 655]   ┆ [787, 787, … 1207]   │
└────────────────┴─────────────────────┴──────────────────────┘
```

Gathering results like this is the first step of the aggregation process and is normally followed by applying a function to these values.

The second thing you can do with the `agg` method is name the resulting columns. This can be done using the `alias` method on the aggregation expression or the `name` namespace. You can refer to Table 5-2 in Chapter 5 for more information on the `name` namespace.

```
(
    df
    .select("Country", "Profit", "Revenue")
    .group_by("Country")
    .agg(
        pl.col("Profit").alias("All Profits Per Transactions"),
        pl.col("Revenue").name.prefix("All "),
    )
)
```

```
shape: (6, 3)
┌────────────────┬─────────────────────┬────────────────────────┐
│ Country        ┆ All Profits Per Tran… ┆ All Revenue          │
│ ---            ┆ ---                 ┆ ---                    │
│ str            ┆ list[i64]           ┆ list[i64]              │
╞════════════════╪═════════════════════╪════════════════════════╡
│ United States  ┆ [524, 407, … 542]   ┆ [929, 722, … 878]      │
│ Germany        ┆ [160, 53, … 746]    ┆ [295, 98, … 1250]      │
│ Canada         ┆ [590, 590, … 630]   ┆ [950, 950, … 1014]     │
│ United Kingdom ┆ [1053, 1053, … 112] ┆ [1728, 1728, … 184]    │
│ France         ┆ [427, 427, … 655]   ┆ [787, 787, … 1207]     │
│ Australia      ┆ [1366, 1188, … 655] ┆ [2401, 2088, … 1183]   │
└────────────────┴─────────────────────┴────────────────────────┘
```

Third, `agg` allows you to apply multiple aggregation functions at the same time, including multiple columns. You can do this by passing a list of expressions to the `agg` method.

```
(
    df
    .select("Country", "Profit", "Revenue")
    .group_by("Country")
    .agg(
        pl.col("Profit").sum().alias("Total Profit"),
        pl.col("Profit").mean().alias("Average Profit per Transaction"),
        pl.col("Revenue").sum().alias("Total Revenue"),
        pl.col("Revenue").mean().alias("Average Revenue per Transaction"),
    )
)
```

```
shape: (6, 5)
┌────────────────┬──────────────┬─────────────────┬───────────────┬─────────────────┐
│ Country        ┆ Total Profit ┆ Average         ┆ Total Revenue ┆ Average         │
│ ---            ┆ ---          ┆ Profit per T…   ┆ ---           ┆ Revenue per …   │
│ str            ┆ i64          ┆ ---             ┆ i64           ┆ ---             │
│                ┆              ┆ f64             ┆               ┆ f64             │
╞════════════════╪══════════════╪═════════════════╪═══════════════╪═════════════════╡
```

Germany	3359995	302.756803	8978596	809.028293
Canada	3717296	262.187615	7935738	559.721964
United States	11073644	282.447687	27975547	713.552696
United Kingdom	4413853	324.071439	10646196	781.659031
Australia	6776030	283.089489	21302059	889.959016
France	2880282	261.891435	8432872	766.764139

Alternatively, you can use column selectors in combination with these expressions to apply an aggregation function to multiple columns at the same time in a single expression.

```
(
    df
    .select("Country", "Profit", "Revenue")
    .group_by("Country")
    .agg(
        pl.all().sum().name.prefix("Total "),
        pl.all().mean().name.prefix("Average "),
    )
)
```

```
shape: (6, 5)
```

Country --- str	Total Profit --- i64	Total Revenue --- i64	Average Profit --- f64	Average Revenue --- f64
Australia	6776030	21302059	283.089489	889.959016
France	2880282	8432872	261.891435	766.764139
United Kingdom	4413853	10646196	324.071439	781.659031
Germany	3359995	8978596	302.756803	809.028293
Canada	3717296	7935738	262.187615	559.721964
United States	11073644	27975547	282.447687	713.552696

Because you're using expressions, it's even possible to work with conditions. The condition returns a Boolean mask, and the `sum` method will sum the `true` values. A *Boolean mask* is an array-like structure of Boolean values used to filter rows or columns based on specific conditions.

For example, we can find the number of transactions with a large profit grouped by country by running the example beneath. To show what the Boolean mask looks like of the expression, you can only aggregate using the expression. This creates a list of Boolean values. Since this is a list of 0 and 1 values, you can sum them to get the number of `true` values!

```
(
    df
    .select("Country", "Profit")
```

```
    .group_by("Country")
    .agg(
        (pl.col("Profit") > 1000)
        .alias("Profit > 1000"),
        (pl.col("Profit") > 1000)
        .sum()
        .alias("Number of Transactions with Profit > 1000"),
    )
)
```

```
shape: (6, 3)
┌────────────────┬───────────────────────┬─────────────────────────┐
│ Country        ┆ Profit > 1000         ┆ Number of Transactio…   │
│ ---            ┆ ---                   ┆ ---                     │
│ str            ┆ list[bool]            ┆ u32                     │
╞════════════════╪═══════════════════════╪═════════════════════════╡
│ Canada         ┆ [false, false, … fal… ┆ 868                     │
│ Australia      ┆ [true, true, … false… ┆ 1233                    │
│ United States  ┆ [false, false, … fal… ┆ 2623                    │
│ Germany        ┆ [false, false, … fal… ┆ 659                     │
│ United Kingdom ┆ [true, true, … false… ┆ 788                     │
│ France         ┆ [false, false, … fal… ┆ 482                     │
└────────────────┴───────────────────────┴─────────────────────────┘
```

Because you can use expressions in the `agg` function, you can also put in Python functions that return expressions. While you should normally only combine Polars and Python when absolutely necessary, this is one of the exceptions. This is because the Python function runs before Polars does and returns a Polars expression, which Polars then runs in Rust.

```
def custom_agg(column: str) -> pl.Expr:
    return (column > 1000).alias("Profit > 1000"), (column > 1000).sum().alias("Number of Transact

(
    df
    .select("Country", "Profit")
    .group_by("Country")
    .agg(
        custom_agg(pl.col("Profit"))
    )
)
```

```
shape: (6, 3)
┌────────────────┬───────────────────────┬─────────────────────────┐
│ Country        ┆ Profit > 1000         ┆ Number of Transactio…   │
│ ---            ┆ ---                   ┆ ---                     │
│ str            ┆ list[bool]            ┆ u32                     │
╞════════════════╪═══════════════════════╪═════════════════════════╡
│ Germany        ┆ [false, false, … fal… ┆ 659                     │
│ United Kingdom ┆ [true, true, … false… ┆ 788                     │
│ France         ┆ [false, false, … fal… ┆ 482                     │
│ United States  ┆ [false, false, … fal… ┆ 2623                    │
│ Canada         ┆ [false, false, … fal… ┆ 868                     │
```

```
│ Australia        │ [true, true, … false… │ 1233                     │
└──────────────────┴───────────────────────┴──────────────────────────┘
```

Allowing plugging in expressions like this shows the versatility of the `agg` method. However, what should you do if you want to apply a Python function to your data?

User-Defined Functions

Polars has an extensive set of expressions that allow you to perform a wide range of operations. However, sometimes you need to perform an operation that isn't covered by the available expressions, or is performed by an external package. To leave you this option, Polars allows for *user defined functions* (UDFs). The Polars functions that allow you to do this are:

map_elements
: Apply a Python function to each element of a column.

map_batches
: Apply a Python function to a Series, or sequence of Series.

map_groups
: Apply a Python function to each group in the GroupBy context.

Let's dive into how you can use these functions to apply your own custom Python functions to your data.

The `map_elements` function allows you to apply a Python function to each element in a column in case you don't need to know anything about the other elements in the column.

This example is a sentiment analysis on a DataFrame text with reviews:

```python
from textblob import TextBlob

def analyze_sentiment(review):
    return TextBlob(review).sentiment.polarity

df = pl.DataFrame({
    "reviews": [
        "This product is great!",
        "Terrible service.",
        "Okay, but not what I expected.",
        "Excellent! I love it."
    ]
})

df = df.with_columns(
    pl.col("reviews")
    .map_elements(
        analyze_sentiment,
```

```
            return_dtype=pl.Float64
        )
        .alias("sentiment_score")
    )
    df
```

```
shape: (4, 2)
┌─────────────────────────────────┬─────────────────┐
│ reviews                         ┆ sentiment_score │
│ ---                             ┆ ---             │
│ str                             ┆ f64             │
╞═════════════════════════════════╪═════════════════╡
│ This product is great!          ┆ 1.0             │
│ Terrible service.               ┆ -1.0            │
│ Okay, but not what I expected.  ┆ 0.2             │
│ Excellent! I love it.           ┆ 0.75            │
└─────────────────────────────────┴─────────────────┘
```

In this example, we use the `map_elements` function to apply the `analyze_sentiment` function to each element in the "reviews" column. The resulting values range from -1.0 (very negative) to 1.0 (very positive) with 0.0 being neutral.

Warning For Inefficient Mappings

When you use a Python function in Polars, it's important to know that it won't be as fast as the native Polars functions. Polars normally runs its operations in Rust. However when it has to apply a custom Python function a few things happen:

- The function executes slower Python bytecode instead of faster Rust bytecode.
- The Python function is constrained by the *global interpreter lock* (GIL), which means it can't run in parallel. This is especially detrimental to speed in `group_by` operations, where the aggregation function is normally called in parallel for each group.

Mapping Python lambdas or custom functions to Polars data should be treated as a last resort. When Polars raises a `PolarsInefficientMapWarning`, it's a sign that there are probably alternative ways to use a native Polars expression instead. Only if you've gone through the Polars documentation and found that there's no native expression or combination of expressions that does what you want should you consider using a Python function.

In the following example, you'll see the `PolarsInefficientMapWarning` by mapping a simple function to a column.

```
df = pl.DataFrame({
    "x": [1,2,3,4]
})

def add_one(x):
    return x + 1

df.with_columns(
    pl.col('x')
    .map_elements(
        add_one,
        return_dtype=pl.Int64,
    )
    .alias("x + 1")
)
```

```
PolarsInefficientMapWarning:
Expr.map_elements is significantly slower than the native expressions API.
Only use if you absolutely CANNOT implement your logic otherwise.
Replace this expression...
  - pl.col("x").map_elements(add_one)
with this one instead:
  + pl.col("x") + 1
shape: (4, 2)
┌─────┬───────┐
│ x   ┆ x + 1 │
│ --- ┆ ---   │
│ i64 ┆ i64   │
╞═════╪═══════╡
│ 1   ┆ 2     │
│ 2   ┆ 3     │
│ 3   ┆ 4     │
│ 4   ┆ 5     │
```

@lru_cache

The `@lru_cache` decorator from the `functools` module in Python is a handy tool for optimizing functions that are computationally intensive. By caching the results of function calls, it can significantly reduce execution time, especially when the function is repeatedly called with the same arguments. This is particularly useful in scenarios where you map a function over a DataFrame column containing repeated values. `@lru_cache` stores the outcomes of your function calls. When the function is invoked again with the same parameters, it retrieves the result from the cache instead of computing it again.

You can give the `@lru_cache` decorator a `maxsize` parameter, which determines the number of results that are cached. By default this is set to `128` cache entries, but you can set it higher to prevent cache misses depending on your data size. `@lru_cache` discards the least recently used entries when it fills up. You can set `maxsize` to `None` if you want to store all results at the cost of high memory usage. You can clear the cache using the `cache_clear()` method when it's no longer needed. Let's apply this to the `map_elements` you did earlier with the cosine similarity function:

```python
from functools import lru_cache


df = pl.DataFrame({
    "x": [1,1,3,3]
})

@lru_cache(maxsize=None)
def add_one(x):
    return x + 1

df.with_columns(
    pl.col('x')
    .map_elements(
        add_one,
        return_dtype=pl.Int64,
    )
    .alias("x + 1")
)
```

```
shape: (4, 2)
┌─────┬───────┐
│ x   ┆ x + 1 │
│ --- ┆ ---   │
│ i64 ┆ i64   │
╞═════╪═══════╡
│ 1   ┆ 2     │
│ 1   ┆ 2     │
│ 3   ┆ 4     │
│ 3   ┆ 4     │
└─────┴───────┘
```

The `map_batches` function allows you to apply a Python function to a Series or sequence of Series. This is useful when you need to know something about the other elements in the column, or when you need to apply a function to multiple columns at the same time. `map_batches` has the following arguments:

function
: Function to apply to the Series.

return_dtype
: The data type of the Series that is returned by the function.

is_elementwise
: Whether the function is elementwise or not. If it is, it can run in the streaming engine, but it might return incorrect group_by results.

agg_list
: Aggregate the values of the expression into a list before applying the function in a group-by context. The function will be invoked only once on a list of groups, rather than once per group.

In the following example we'll demonstrate the `map_batches` function by applying a softmax normalization function to the columns "feature1" and "feature2". The softmax normalization function turns a list of numbers into probabilities that add up to 100%.

```
import polars.selectors as cs
import numpy as np
from scipy.special import softmax

df = pl.DataFrame({
    "feature1": [0.3, 0.2, 0.4, 0.1, 0.2, 0.3, 0.5],
    "feature2": [32, 50, 70, 65, 0, 10, 15],
    "label": [1, 0, 1, 0, 1, 0, 0]
})

result = df.select(
    "label",
    cs.starts_with("feature").map_batches(
        lambda x: softmax(x.to_numpy()),
    )
)
result
```

```
shape: (7, 3)
┌───────┬──────────┬────────────┐
│ label ┆ feature1 ┆ feature2   │
│ ---   ┆ ---      ┆ ---        │
│ i64   ┆ f64      ┆ f64        │
╞═══════╪══════════╪════════════╡
│ 1     ┆ 0.143782 ┆ 3.1181e-17 │
```

```
│ 0     │ 0.130099 │ 2.0474e-9  │
│ 1     │ 0.158904 │ 0.993307   │
│ 0     │ 0.117719 │ 0.006693   │
│ 1     │ 0.130099 │ 3.9488e-31 │
│ 0     │ 0.143782 │ 8.6979e-27 │
│ 0     │ 0.175616 │ 1.2909e-24 │
└───────┴──────────┴────────────┘
```

Finally, the `map_groups` function allows you to apply a Python function to each group in the GroupBy context.

Say you have a DataFrame with temperatures measured in different locations, where the American locations' temperatures are in Fahrenheit and the European locations' in Celsius. If only the variation in temperature is relevant for your analysis, you can scale the features within each group to make them comparable:

```python
from sklearn.preprocessing import StandardScaler

def scale_temperature(group):
    scaler = StandardScaler()
    scaled_values = scaler.fit_transform(group[['temperature']].to_numpy())
    return group.with_columns(pl.Series(values=scaled_values.flatten(), name="scaled_feature"))

df = pl.DataFrame({
    "group": ["USA", "USA", "USA", "USA", "NL", "NL", "NL"],
    "temperature": [32, 50, 70, 65, 0, 10, 15]
})

result = df.group_by("group").map_groups(scale_temperature)
result
```

```
shape: (7, 3)
┌───────┬─────────────┬────────────────┐
│ group ┆ temperature ┆ scaled_feature │
│ ---   ┆ ---         ┆ ---            │
│ str   ┆ i64         ┆ f64            │
╞═══════╪═════════════╪════════════════╡
│ USA   ┆ 32          ┆ -1.502872      │
│ USA   ┆ 50          ┆ -0.287066      │
│ USA   ┆ 70          ┆ 1.063831       │
│ USA   ┆ 65          ┆ 0.726107       │
│ NL    ┆ 0           ┆ -1.336306      │
│ NL    ┆ 10          ┆ 0.267261       │
│ NL    ┆ 15          ┆ 1.069045       │
└───────┴─────────────┴────────────────┘
```

Lastly, if you need fine-grained control over the individual groups in the GroupBy context, you can also iterate over them. This can be useful when you need to apply a different custom function per group, or when you want to inspect the groups individually. Iterating over the groups returns a tuple containing the group identifiers (or a single identifier if there's only one) and the DataFrame for that group.

```python
df = pl.DataFrame({
    "group": ["USA", "USA", "USA", "USA", "NL", "NL", "NL"],
    "temperature": [32, 50, 70, 65, 0, 10, 15]
})

for group in df.group_by(["group"]):
    print(group)
```

```
(('NL',), shape: (3, 2)
┌───────┬─────────────┐
│ group ┆ temperature │
│ ---   ┆ ---         │
│ str   ┆ i64         │
╞═══════╪═════════════╡
│ NL    ┆ 0           │
│ NL    ┆ 10          │
│ NL    ┆ 15          │
└───────┴─────────────┘)
(('USA',), shape: (4, 2)
┌───────┬─────────────┐
│ group ┆ temperature │
│ ---   ┆ ---         │
│ str   ┆ i64         │
╞═══════╪═════════════╡
│ USA   ┆ 32          │
│ USA   ┆ 50          │
│ USA   ┆ 70          │
│ USA   ┆ 65          │
└───────┴─────────────┘)
```

In summary, Polars offers functions to apply custom Python functions to your data through `map_elements`, `map_batches`, and `map_groups`. While these user-defined functions allow for extensive customization, it's important to think about performance drawbacks compared to native Polars expressions. If you still need to work with Python functions, but the input is often the same, the @lru_cache decorator can help optimize repeated computations. By understanding and leveraging these tools, you can tailor your data transformations to meet specific needs while maintaining optimal performance.

Row-wise Aggregations with `reduce` and `fold`

Polars provides a lot of standard horizontal aggregations out of the box. These expressions are shown in Table 7-5. Two methods that allow you to build more complex horizontal aggregations within the Polars API are the `reduce` and `fold` methods. These methods operate on a whole column at the same time, often in a vectorized manner, keeping it performant.

Here's how `reduce` and `fold` work: First, they create a new column called the *accumulator*. This accumulator is a new column with initial values to which the aggregation is applied. The other input is the value resulting from the expression that is being aggregated over. This accumulator is updated with the result of a function that gets as input the accumulator and that value.

Both `reduce` and `fold` take these arguments:

`function`
: The function to apply over the accumulator and the value that gets folded.

`exprs`
: The expression to aggregate over.

Additionally, while `reduce` uses the first value it comes across as the accumulator, the `fold` method allows you to set an initial value for the accumulator with the following parameter:

`acc`
: The initial value of the accumulator.

Let's look at a simple example to understand how `fold` works.

```
df = pl.DataFrame({
    "col1": [2],
    "col2": [3],
    "col3": [4]
})

df.with_columns(
    pl.fold(
        acc=pl.lit(0),  ❶
        function=lambda acc, x: acc + x,  ❷
        exprs=pl.col("*")  ❸
    ).alias("sum")
)
```

```
shape: (1, 4)
┌──────┬──────┬──────┬─────┐
│ col1 ┆ col2 ┆ col3 ┆ sum │
│ ---  ┆ ---  ┆ ---  ┆ --- │
│ i64  ┆ i64  ┆ i64  ┆ i64 │
╞══════╪══════╪══════╪═════╡
│ 2    ┆ 3    ┆ 4    ┆ 9   │
└──────┴──────┴──────┴─────┘
```

❶ Because you are summing the values of the columns, you set the initial value of the accumulator to 0. Using the `reduce` method would have set the accumulator to the first value in the column.

❷ This is the simple summing function. The value in the accumulator column is added to the value in the next column you are aggregating over.

❸ Since `pl.col("*")` functions as a wildcard representing any column, you are aggregating over all columns in the DataFrame without changing them in any way.

The execution would look like this:

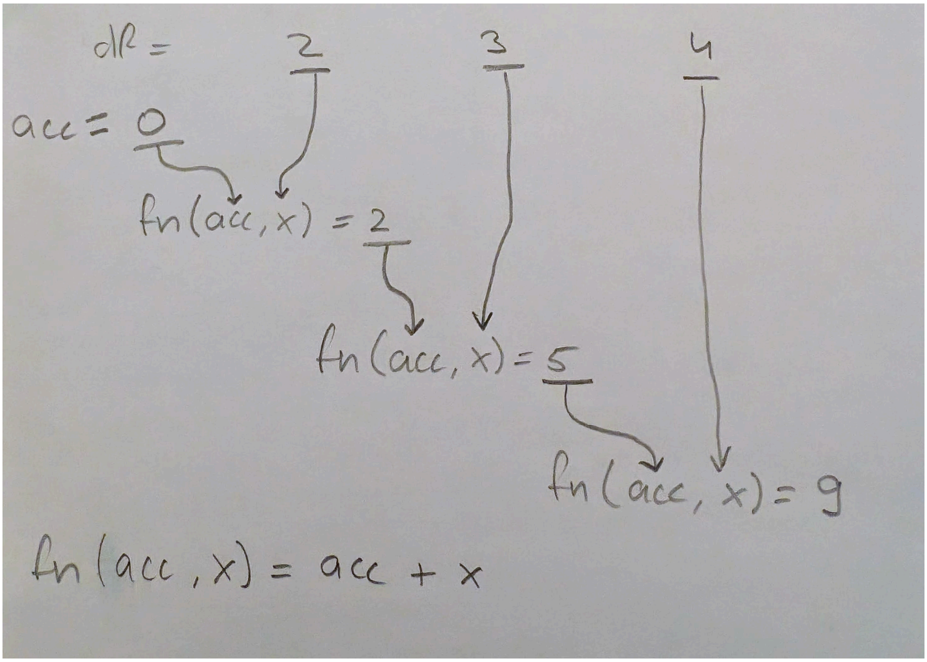


Figure 10-1. How a fold function is executed.

```
df = pl.DataFrame({
    "col1": [2],
    "col2": [3],
    "col3": [4]
})

df.with_columns(
    pl.fold(
        acc=pl.lit(0), ❶
        function=lambda acc, x: acc + x, ❷
        exprs=pl.col("*") ❸
    ).alias("sum")
)
```

```
shape: (1, 4)
┌──────┬──────┬──────┬─────┐
│ col1 │ col2 │ col3 │ sum │
│ ---  │ ---  │ ---  │ --- │
```

```
| i64 | i64 | i64 | i64 |
╞═════╪═════╪═════╪═════╡
| 2   | 3   | 4   | 9   |
└─────┴─────┴─────┴─────┘
```

One possible use case for `fold` is when you want to sum with weights per column. For example, you have a DataFrame with sales data for different products, and you want to calculate the weighted sum of the sales:

```python
df = pl.DataFrame({
    "product_A": [10, 20, 30],
    "product_B": [20, 30, 40],
    "product_C": [30, 40, 50]
})


weights = {  ❶
    "product_A": 0.5,
    "product_B": 1.5,
    "product_C": 2.0
}

weighted_exprs = [  ❷
    (pl.col(product) * weight).alias(product)
    for product, weight in weights.items()
]

df_with_weighted_sum = df.with_columns(
    pl.fold(  ❸
        acc=pl.lit(0),  ❹
        function=lambda acc, x: acc + x,  ❺
        exprs=weighted_exprs  ❻
    ).alias("weighted_sum")
)

df_with_weighted_sum
```

```
shape: (3, 4)
┌───────────┬───────────┬───────────┬──────────────┐
│ product_A ┆ product_B ┆ product_C ┆ weighted_sum │
│ ---       ┆ ---       ┆ ---       ┆ ---          │
│ i64       ┆ i64       ┆ i64       ┆ f64          │
╞═══════════╪═══════════╪═══════════╪══════════════╡
│ 10        ┆ 20        ┆ 30        ┆ 95.0         │
│ 20        ┆ 30        ┆ 40        ┆ 135.0        │
│ 30        ┆ 40        ┆ 50        ┆ 175.0        │
└───────────┴───────────┴───────────┴──────────────┘
```

❶ Define weights for each product.

❷ Create a Polars expression that multiplies each column by its respective weight.

❸ Apply the fold function to calculate the weighted sum.

❹ Start with an initial value of 0 for the accumulator.

❺ Once again use a summing function

❻ Apply the weighted expressions to the fold function.

over() Expressions in Selection Context

Sometimes, instead of aggregating data into groups, you want to add information to the frame. This is where `over()` comes in. The `over()` function allows you to perform aggregations on groups in the `select` context! Additionally, it allows you to map the results back to the original DataFrame, keeping its original dimensions. This is practical when you need the context of individual rows, and want to enrich it with information from the group. The `over()` function has the following parameters:

`expr` *and* `*more_exprs`
: the column(s) to group by. Accepts both expressions and strings, which will be parsed to column names.

`mapping_strategy`
- *group_to_rows* (default): Maps the results back to the row from which they originate. The result is the same size as the original DataFrame.
- *join*: Aggregates results to a list that is joined back to the original DataFrame.
- *explode*: Creates a new row for each element in the result list. This alters the size of the DataFrame.

Let's return to the Top2000 dataset from the beginning of this chapter. If you want to add information to the frame instead of aggregating results for an analysis, you can use the `over()` function. For example, let's try is calculating the position of a song for its release year.

```
(top2000
    .select(
        "jaar",
        "artiest",
        "titel",
        "positie",
        pl.col("positie")
        .rank()
        .over("jaar")
        .alias("year_rank")
    )
    .sample(10, seed=42)
)
```

```
shape: (10, 5)
┌──────┬──────────────────────┬──────────────────────┬──────────┬───────────┐
│ jaar ┆ artiest              ┆ titel                ┆ positie  ┆ year_rank │
│ ---  ┆ ---                  ┆ ---                  ┆ ---      ┆ ---       │
│ i64  ┆ str                  ┆ str                  ┆ i64      ┆ f64       │
╞══════╪══════════════════════╪══════════════════════╪══════════╪═══════════╡
│ 2013 ┆ Stromae              ┆ Papaoutai            ┆ 318      ┆ 6.0       │
│ 1969 ┆ John Denver          ┆ Leaving On A Jet Pla… ┆ 607      ┆ 16.0      │
│ 1971 ┆ Led Zeppelin         ┆ Immigrant Song       ┆ 590      ┆ 19.0      │
│ 2009 ┆ Anouk                ┆ For Bitter Or Worse  ┆ 1453     ┆ 23.0      │
│ 2015 ┆ Snollebollekes       ┆ Links Rechts         ┆ 1076     ┆ 14.0      │
│ 1984 ┆ Alphaville           ┆ Forever Young        ┆ 302      ┆ 11.0      │
│ 1977 ┆ ABBA                 ┆ Take A Chance On Me  ┆ 636      ┆ 23.0      │
│ 1975 ┆ Rod Stewart          ┆ Sailing              ┆ 918      ┆ 20.0      │
│ 1986 ┆ Metallica            ┆ Master Of Puppets    ┆ 29       ┆ 1.0       │
│ 2005 ┆ Alderliefste & Ramse… ┆ Laat Me/Vivre        ┆ 463      ┆ 5.0       │
└──────┴──────────────────────┴──────────────────────┴──────────┴───────────┘
```

Here we can see that Stromae's "Papaoutai" was ranked 6th best song in 2013 according to Top2000 voters, while it's ranked 318th overall.

Dynamic Grouping with `group_by_dynamic`

When you're working with temporal data, it can be practical to create groups based on a time window. This is where the `group_by_dynamic` function comes in. *group_by_dynamic* calculates a time window of a fixed size and width, to which it assigns the rows in your DataFrame. This is different from a normal group-by, because rows can occur in multiple time windows, depending on the window size and the time column. This is useful for calculating yearly or quarterly sales data, where you want to divide data into specific time periods. These windows can be defined by the following parameters:

`every`
: the interval at which the windows start.

`period`
: the length of the time window. It matches `every` if not specified, resulting in adjacent, non-overlapping groups. However, if you want to create overlapping windows, you can set `period` to a value larger than `every`.

`offset`
: used to shift the start of the window. For example, if you want to start our time window at 9AM every day to align with business hours, you can set `every=1d`, and `offset=9h`.

`start_by`
: sets the strategy for determining the start of the first window, allowing you to align the start with the earliest data point, with a specific day of the week, or by adjusting to the earliest timestamp and then applying an offset based on your specified `every` interval.

The `every`, `period` and `offset` arguments can be specified using the following strings:

Table 10-2. Duration strings and their meaning

Duration string	Description
1ns	1 nanosecond
1us	1 microsecond
1ms	1 millisecond
1s	1 second
1m	1 minute
1h	1 hour
1d	1 calendar day
1w	1 calendar week
1mo	1 calendar month
1q	1 calendar quarter
1y	1 calendar year
1i	1 index count

These can also be combined. For example: `"1y6m1w5d"` would be 1 year, 6 months, 1 week, and 5 days. With these settings you can create regular time windows and group your data into them. There are three types of window configurations you can create, as shown in Figure 10-2.

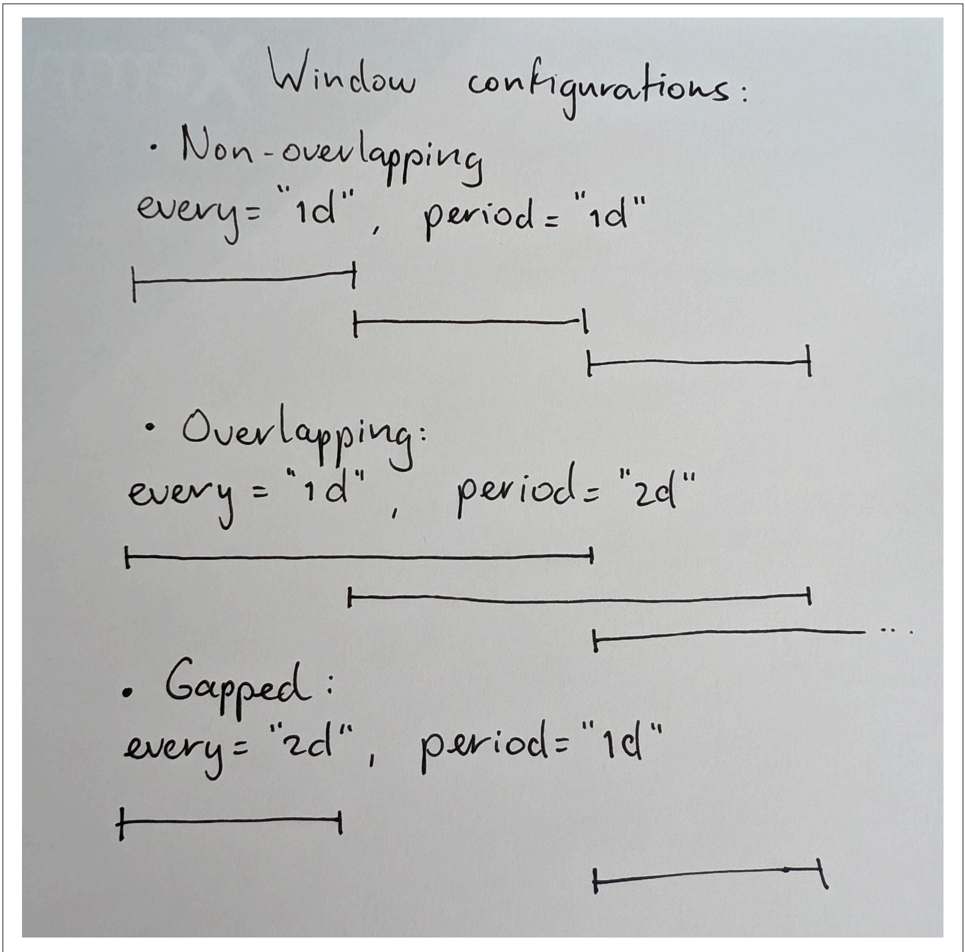


Figure 10-2. The different types of windows.

Additionally, the `closed` parameter determines whether values that are exactly the lower or upper bound are included or excluded. The options provided are shown in Table 10-3.

Table 10-3. Closed interval options

Parameter	Description	Interval	Contains a	Contains b
`left`	The lower bound is inclusive, the upper bound is exclusive.	[a, b)	✓	✗
`right`	The lower bound is exclusive, the upper bound is inclusive.	(a, b]	✗	✓
`both`	Both the lower and upper bounds are inclusive.	[a, b]	✓	✓
`none`	Both the lower and upper bounds are exclusive.	(a, b)	✗	✗

Tell Polars your data is sorted for a boost

If your index column is already sorted in ascending order you can set `check_sorted` to `False` to speed up the grouping process. Otherwise Polars will check if the index is sorted, which is a costly operation in the GroupBy context, because it cannot use the sorted flags that are normally available in the DataFrame.

Rolling Aggregations with `rolling`

Where `group_by_dynamic` creates time windows of fixed size and width, `rolling` creates windows tailored around values in the DataFrame itself. This is useful when you want to calculate rolling aggregations, such as moving averages or cumulative sums. The `rolling` function has the following parameters:

`index_column`
: the column that contains values that will be used as the anchor point of the window.

`period`
: the size of the window.

`offset`
: shifts the window backward or forward.

`closed`
: defines how boundary values are handled. Works exactly like explained earlier in `group_by_dynamic`.

`group_by`
: groups the data by the specified columns before applying the rolling aggregation.

For a dataset with timestamps, the rolling operation will create a window for each timestamp that extends backwards by the specified `period`. If `offset` is set, it shifts the entire window forward or backward, offering a way to adjust the focus of the analysis. This is illustrated in Figure 10-3.

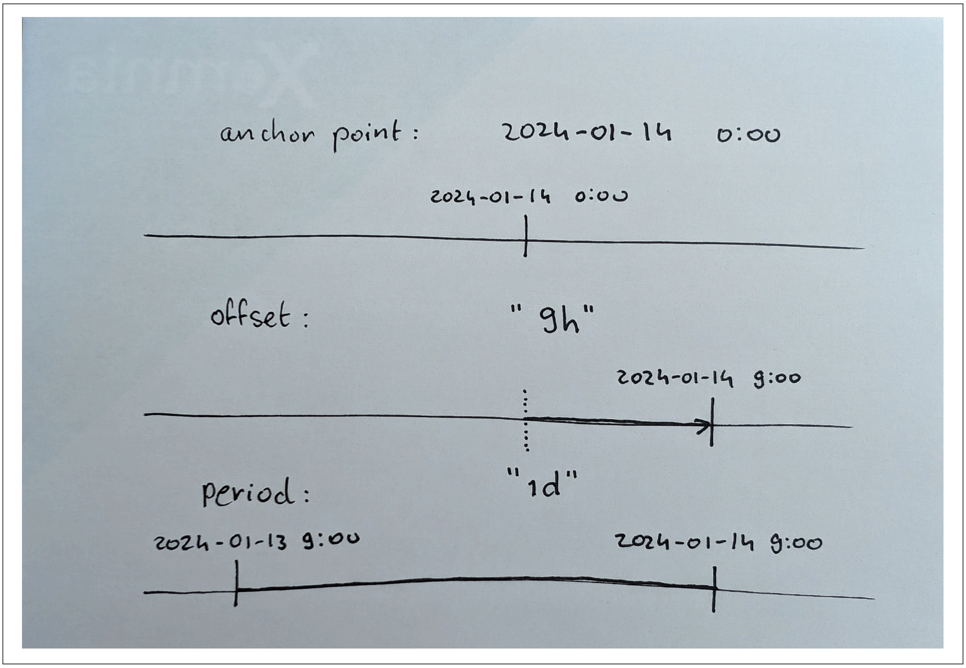


Figure 10-3. How a time window is determined using the `rolling` method.

The `group_by` parameter allows you to perform rolling aggregation within groups of data.

Imagine you're analyzing a dataset from a chain of retail stores. You have sales data from multiple locations and want to understand the rolling average sales over a 7-day period for each store. This will help us identify trends, such as which stores are consistently performing well and which might be experiencing declines or variability in sales.

Let's create a small DataFrame with simple sales numbers. The DataFrame will contain 2 weeks of sales data for 2 stores that are only open on weekdays. You'll calculate the rolling sum of the last week of sales for each store:

```
from datetime import date

dates = pl.date_range(  ❶
    start=date(2024, 4, 1),
    end=date(2024, 4, 14),
    interval='1d',
    eager=True,  ❷
)
dates  = dates.filter(dates.dt.weekday() < 6)  ❸
dates_repeated = pl.concat([dates, dates]).sort()  ❹
```

```
df = pl.DataFrame({
    "date": dates_repeated,
    "store": ["Store A", "Store B"] * dates.len(),
    "sales": [
        200, 150, 220, 160, 250, 180, 270, 190, 280, 210,
        210, 170, 220, 180, 240, 190, 250, 200, 260, 210,
    ]
}).set_sorted("date", "store")  ❺
```

❶ Create a date range from April 1st to April 14th.

❷ The `eager` parameter is set to `True` to create the date range immediately.

❸ Filter out weekend days.

❹ Repeat the dates for the two weeks and sort them.

❺ Indicate that the date and store columns are sorted.

Now that you have a nice dataset, you can calculate the rolling sum of the last 7 days of sales for each store.

```
result = (
    df.rolling(  ❶
        index_column="date",
        period="7d",
        group_by="store",
        check_sorted=False,  ❷
    ).agg(  ❸
        pl.sum("sales").alias("sum_of_last_7_days_sales")
    )
)

final_df = df.join(result, on=["date", "store"])  ❹

final_df
```

```
shape: (20, 4)
┌────────────┬─────────┬───────┬─────────────────────┐
│ date       ┆ store   ┆ sales ┆ sum_of_last_7_days_s… │
│ ---        ┆ ---     ┆ ---   ┆ ---                 │
│ date       ┆ str     ┆ i64   ┆ i64                 │
╞════════════╪═════════╪═══════╪═════════════════════╡
│ 2024-04-01 ┆ Store A ┆ 200   ┆ 200                 │
│ 2024-04-02 ┆ Store A ┆ 220   ┆ 420                 │
│ 2024-04-03 ┆ Store A ┆ 250   ┆ 670                 │
│ 2024-04-04 ┆ Store A ┆ 270   ┆ 940                 │
│ 2024-04-05 ┆ Store A ┆ 280   ┆ 1220                │
│ 2024-04-08 ┆ Store A ┆ 210   ┆ 1230                │
│ 2024-04-09 ┆ Store A ┆ 220   ┆ 1230                │
│ 2024-04-10 ┆ Store A ┆ 240   ┆ 1220                │
```

```
│ 2024-04-11 │ Store A │ 250   │ 1200                   │
│ 2024-04-12 │ Store A │ 260   │ 1180                   │
│ 2024-04-01 │ Store B │ 150   │ 150                    │
│ 2024-04-02 │ Store B │ 160   │ 310                    │
│ 2024-04-03 │ Store B │ 180   │ 490                    │
│ 2024-04-04 │ Store B │ 190   │ 680                    │
│ 2024-04-05 │ Store B │ 210   │ 890                    │
│ 2024-04-08 │ Store B │ 170   │ 910                    │
│ 2024-04-09 │ Store B │ 180   │ 930                    │
│ 2024-04-10 │ Store B │ 190   │ 940                    │
│ 2024-04-11 │ Store B │ 200   │ 950                    │
│ 2024-04-12 │ Store B │ 210   │ 950                    │
└────────────┴─────────┴───────┴────────────────────────┘
```

❶ The `rolling` function creates windows that contain the current row and rows that are within 7 days before the current row.

❷ The `check_sorted` parameter is set to `False` to speed up processing because you know the data is already sorted.

❸ Calculate the sum of the created time windows with the `rolling` function.

❹ Join rolling results back to the original DataFrame.

Here you see the rolling sum of the last 7 days is calculated for each store. The first 7 days have a rolling sum of only the days available before it in the dataset, because there are no more days to include in the window. This rolling aggregation allows you to see how the sales of each store are developing over time.

Conclusion

In this chapter you learned how to perform aggregations on your data. You learned about:

- the basic aggregations available in the `group_by` context, such as `sum`, `mean`, `quantile`, and `median`.
- the advanced aggregations available in the `agg` method, which allow you to aggregate column elements into a list per group, control the resulting column names, apply multiple aggregation functions at the same time, and apply your own custom aggregation functions.
- User-defined functions (or UDFs), which allow you to apply your own custom Python functions to your data using `map_elements`, `map_batches`, and `map_groups`.
- creating groups based on a time window using the `group_by_dynamic` function.

- creating rolling aggregations around the values in your DataFrame using the `rolling` function.
- performing aggregations on groups in the `select` context using the `over()` expression.
- some optimizations you can apply to your Python functions to speed up the process, like `set_sorted` and `@lru_cache`.

In the next chapter you'll look at how you can combine multiple DataFrames using joins and unions.

CHAPTER 11
Joining and Concatenating

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 14th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

Data often comes from multiple sources that you will have to connect and combine in a meaningful way. There are multiple ways to combine DataFrames, which we'll go over in this chapter.

Joining

To combine different DataFrames, Polars offers the `join()` method. It takes the following arguments:

- `other`: The DataFrame to join with.
- `on`: The column(s) to join on when the name is the same in the left and right frames.
- `left_on` and `right_on`: The column(s) to join if they have different names in the left and right frame.

- `how`: The join strategy to use.
- `suffix`: The suffix which will be appended to columns that appear in both frames.
- `validate`: Validates that the join is of a certain type.
- `join_nulls`: Join null values. By default, null values are not joined.

Join Strategies

Joining can be done according to different strategies. Depending on your situation you need to combine the two datasets in a different way. The strategies that Polars supports are: **`inner` (default): Only keep rows that have a match in both DataFrames.** `outer`: Keep all rows from both DataFrames. **`left`: Keep all rows from the left DataFrame and only the rows from the right DataFrame that have a match.** `cross`: Create a *Cartesian product* of both DataFrames. The Cartesian product comes from set theory and represents the set of all possible combinations of the elements of two sets. You'll see an example of this later in this section. **`semi`: Keep all rows from the left DataFrame that have a match in the right DataFrame.** `anti`: Keep all rows from the left DataFrame that do not have a match in the right DataFrame.

Throughout this section you'll use the following DataFrames to demonstrate the different join strategies:

```python
import polars as pl

df_left = pl.DataFrame({
    "key": ["A", "B", "C", "D"],
    "value": [1, 2, 3, 4]
})

df_right = pl.DataFrame({
    "key": ["B", "C", "D", "E"],
    "value": [5, 6, 7, 8]
})
```

`inner`
: The default join strategy in Polars is the `inner` join. This join strategy only keeps rows that have a match in both DataFrames, discarding any rows that do not. You'll see in the following example that the row of the left frame with the key `A` is not present in the resulting DataFrame, as is the row of the right frame with the key `E`. All the other rows are there, as they have a match in both DataFrames.

```python
df_left.join(df_right, on="key", how="inner")
```

```
shape: (3, 3)
┌─────┬───────┬─────────────┐
│ key ┆ value ┆ value_right │
```

```
│ ---  ┆ ---  ┆ ---         │
│ str  ┆ i64  ┆ i64         │
╞══════╪══════╪═════════════╡
│ B    ┆ 2    ┆ 5           │
│ C    ┆ 3    ┆ 6           │
│ D    ┆ 4    ┆ 7           │
└──────┴──────┴─────────────┘
```

`outer`

The `outer` join strategy keeps all rows from both DataFrames, filling the missing values with nulls. Additionally you can change the default suffixes for columns with a duplicate name in the right DataFrame. In the following example we'll change the suffix to `_other`.

```
df_left.join(df_right, on="key", how="outer", suffix="_other")
```

```
shape: (5, 4)
┌──────┬───────┬───────────┬─────────────┐
│ key  ┆ value ┆ key_other ┆ value_other │
│ ---  ┆ ---   ┆ ---       ┆ ---         │
│ str  ┆ i64   ┆ str       ┆ i64         │
╞══════╪═══════╪═══════════╪═════════════╡
│ B    ┆ 2     ┆ B         ┆ 5           │
│ C    ┆ 3     ┆ C         ┆ 6           │
│ D    ┆ 4     ┆ D         ┆ 7           │
│ null ┆ null  ┆ E         ┆ 8           │
│ A    ┆ 1     ┆ null      ┆ null        │
└──────┴───────┴───────────┴─────────────┘
```

`left`

The `left` join strategy keeps all rows from the left DataFrame and only the rows from the right DataFrame that have a match, filling the missing values with nulls. Note that Polars doesn't have a right join. This can be achieved by switching the DataFrames in the join operation.

```
df_left.join(df_right, on="key", how="left")
```

```
shape: (4, 3)
┌──────┬───────┬─────────────┐
│ key  ┆ value ┆ value_right │
│ ---  ┆ ---   ┆ ---         │
│ str  ┆ i64   ┆ i64         │
╞══════╪═══════╪═════════════╡
│ A    ┆ 1     ┆ null        │
│ B    ┆ 2     ┆ 5           │
│ C    ┆ 3     ┆ 6           │
│ D    ┆ 4     ┆ 7           │
└──────┴───────┴─────────────┘
```

`cross`

The `cross` join strategy creates a Cartesian product of both DataFrames. This means that the resulting DataFrame will have a length equal to the length of the left DataFrame multiplied by the length of the right DataFrame, resulting in potentially huge DataFrames! The `on` argument is not needed for this join, as all rows will be joined with each other.

```python
df_left.join(df_right, how="cross")
```

```
shape: (16, 4)
┌─────┬───────┬───────────┬─────────────┐
│ key ┆ value ┆ key_right ┆ value_right │
│ --- ┆ ---   ┆ ---       ┆ ---         │
│ str ┆ i64   ┆ str       ┆ i64         │
╞═════╪═══════╪═══════════╪═════════════╡
│ A   ┆ 1     ┆ B         ┆ 5           │
│ A   ┆ 1     ┆ C         ┆ 6           │
│ A   ┆ 1     ┆ D         ┆ 7           │
│ A   ┆ 1     ┆ E         ┆ 8           │
│ B   ┆ 2     ┆ B         ┆ 5           │
│ …   ┆ …     ┆ …         ┆ …           │
│ C   ┆ 3     ┆ E         ┆ 8           │
│ D   ┆ 4     ┆ B         ┆ 5           │
│ D   ┆ 4     ┆ C         ┆ 6           │
│ D   ┆ 4     ┆ D         ┆ 7           │
│ D   ┆ 4     ┆ E         ┆ 8           │
└─────┴───────┴───────────┴─────────────┘
```

`semi`

A `semi` join is a special join that doesn't add any data from the right DataFrame to the resulting DataFrame. Instead, it only keeps the rows from the left DataFrame that have a match in the right DataFrame. This make the semi join one of the additional ways to filter the left DataFrame.

```python
df_left.join(df_right, on="key", how="semi")
```

```
shape: (3, 2)
┌─────┬───────┐
│ key ┆ value │
│ --- ┆ ---   │
│ str ┆ i64   │
╞═════╪═══════╡
│ B   ┆ 2     │
│ C   ┆ 3     │
│ D   ┆ 4     │
└─────┴───────┘
```

`anti`

The `anti` join strategy is the opposite of the `semi` join. It only keeps the rows from the left DataFrame that do not have a match in the right DataFrame.

```
df_left.join(df_right, on="key", how="anti")
```

```
shape: (1, 2)
┌─────┬───────┐
│ key ┆ value │
│ --- ┆ ---   │
│ str ┆ i64   │
╞═════╪═══════╡
│ A   ┆ 1     │
└─────┴───────┘
```

Joining on Multiple Columns

You can join DataFrames on multiple columns by passing a list of column names to the `on` argument. This will join the DataFrames on all the columns in the list.

To try this, you'll need two DataFrames with more columns. For this example, you'll use the following example DataFrames. In these frames you'll join on the `name` and `city` columns.

```
df_left = pl.DataFrame({
    "name": ["Alice", "Bob", "Charlie", "Dave"],
    "city": ["NY", "LA", "NY", "SF"],
    "age": [25, 30, 35, 40]
})

df_right = pl.DataFrame({
    "name": ["Alice", "Bob", "Charlie", "Dave"],
    "city": ["NY", "LA", "NY", "Chicago"],
    "department": ["Finance", "Marketing", "Engineering", "Operations"]
})

df_left.join(df_right, on=["name", "city"], how="inner")
```

```
shape: (3, 4)
┌─────────┬──────┬─────┬─────────────┐
│ name    ┆ city ┆ age ┆ department  │
│ ---     ┆ ---  ┆ --- ┆ ---         │
│ str     ┆ str  ┆ i64 ┆ str         │
╞═════════╪══════╪═════╪═════════════╡
│ Alice   ┆ NY   ┆ 25  ┆ Finance     │
│ Bob     ┆ LA   ┆ 30  ┆ Marketing   │
│ Charlie ┆ NY   ┆ 35  ┆ Engineering │
└─────────┴──────┴─────┴─────────────┘
```

Validation

After joining data you can validate whether the join was of a certain *cardinality*. This involves checking the nature of the relationships between the joined tables to make sure that they joined according to the expected relationships. The following relationships can be validated:

Many-to-many (`m:m`)
: A *many-to-many join* is when multiple rows in the left DataFrame match multiple rows in the right DataFrame. An example of this would be joining a table of employees to a table of projects. Each employee can be involved in multiple projects, and projects usually have multiple employees working on them. In Polars this is the default option, and it doesn't result in checks.

One-to-many (`1:m`)
: A *one-to-many join* is when a single row in the left DataFrame matches multiple rows in the right DataFrame. An example of this relationship would be joining a list of departments with a list of employees. Each department has multiple employees, but each employee only belongs to one department. Polars validates whether the join keys are unique in the left DataFrame.

Many-to-one (`m:1`)
: A *many-to-one join* is when multiple rows in the left DataFrame match a single row in the right DataFrame. An example of this would be joining a table of employees with a table of cities they live in. Each employee can only live in one city, but a city can contain multiple employees. Polars validates whether the join keys are unique in the right DataFrame.

One-to-one (`1:1`)
: A *one-to-one join* is when a single row in the left DataFrame has a match with a single row in the right DataFrame. An example of this would be joining a table of employees with a table of employee IDs. Polars validates whether the join keys are unique in both DataFrames.

To validate the join, you can pass the `validate` argument to the `join` method. This argument takes a string with the relationship you want to validate.

In the example below, you'll make two DataFrames containing a set of employees and a set of departments which you will join in a many-to-one fashion. Each employee only belongs to one department, but each department can have multiple employees:

```
df_employees = pl.DataFrame({
    "employee_id": [1, 2, 3, 4],
    "name": ["Alice", "Bob", "Charlie", "Dave"],
    "department_id": [10, 10, 30, 10],
})

df_departments = pl.DataFrame({
    "department_id": [10, 20, 30],
    "department_name": ["Information Technology", "Finance", "Human Resources"],
})

df_employees.join(
    df_departments,
    on="department_id",
```

```
    how="left",
    validate="m:1"
)
```

```
shape: (4, 4)
┌─────────────┬─────────┬───────────────┬────────────────────────┐
│ employee_id ┆ name    ┆ department_id ┆ department_name        │
│ ---         ┆ ---     ┆ ---           ┆ ---                    │
│ i64         ┆ str     ┆ i64           ┆ str                    │
╞═════════════╪═════════╪═══════════════╪════════════════════════╡
│ 1           ┆ Alice   ┆ 10            ┆ Information Technolo…  │
│ 2           ┆ Bob     ┆ 10            ┆ Information Technolo…  │
│ 3           ┆ Charlie ┆ 30            ┆ Human Resources        │
│ 4           ┆ Dave    ┆ 10            ┆ Information Technolo…  │
└─────────────┴─────────┴───────────────┴────────────────────────┘
```

The moment there are multiple departments sharing the same ID, the validation will fail:

```
df_departments = pl.DataFrame({
    "department_id": [10, 20, 10],
    "department_name": ["Information Technology", "Finance", "Human Resources"],
})

df_employees.join(
    df_departments,
    on="department_id",
    how="left",
    validate="m:1"
)
```

```
ComputeError: the join keys did not fulfil m:1 validation
```

Inexact Joining

When joining DataFrames, you might want to connect two tables based on values that are close to each other but not exactly the same. An example of this would be joining datasets of sales from different sources where one system timestamps it on writing the sale into a database, while the other system timestamps it on the time of payment. This creates an inconsistent discrepancy in the timestamps, which can be solved by joining the DataFrames on the closest value. You can do this with Polars by using the `join_asof` function.

`join_asof` takes the following arguments:

- `other`: The DataFrame to join with.
- on: The column(s) to join on when the name is the same in the left and right frame.

- `left_on` and `right_on`: The column(s) to join on when there are different names in the left and right frame.
- `by`: Columns to join on before doing the `join_asof`.
- `by_left` and `by_right`: Columns to join on before doing the `join_asof` in case they have different names in the left and right frame.
- `strategy`: The strategy to use when joining.
- `suffix`: Suffix to use for columns appearing in both DataFrames with the same name.
- `tolerance`: The maximum difference between the values to consider them a match.
- `allow_parallel`: Allow Polars to calculate the DataFrames up to the join in parallel. `True` by default.
- `force_parallel`: Force parallel frame computation before the join. `False` by default.

Before we dive into some examples, you'll need to know that `join_asof` only works if both DataFrames are sorted on the column(s) you want to join on. If the DataFrames are not sorted, you'll get an error:

```
df_left = pl.DataFrame({
    "int_id": [5, 10],
    "value": ["1", "2"]
})

df_right = pl.DataFrame({
    "int_id": [4, 7, 12],
    "value": [1, 2, 3]
})

df_left.join_asof(df_right, on="int_id", tolerance=3)
```

```
InvalidOperationError: argument in operation 'asof_join' is not explicitly sorte
d

- If your data is ALREADY sorted, set the sorted flag with: '.set_sorted()'.
- If your data is NOT sorted, sort the 'expr/series/column' first.
```

In this example the DataFrames are already sorted, which we can indicate to Polars by calling the `set_sorted("int_id")` method on the DataFrames. `set_sorted()` is a free operation, because you provide Polars with the knowledge that the column is already sorted. In other cases you can sort the DataFrames by calling the `sort("int_id")` method. Even when the data is already sorted, this will take some time to check that it's sorted and, if it is not, to sort it.

```python
df_left = df_left.set_sorted("int_id")
df_right = df_right.set_sorted("int_id")

df_left.join_asof(df_right, on="int_id")
```

```
shape: (2, 3)
┌────────┬───────┬─────────────┐
│ int_id ┆ value ┆ value_right │
│ ---    ┆ ---   ┆ ---         │
│ i64    ┆ str   ┆ i64         │
╞════════╪═══════╪═════════════╡
│ 5      ┆ 1     ┆ 1           │
│ 10     ┆ 2     ┆ 2           │
└────────┴───────┴─────────────┘
```

Note that this also drops the right join column if they have the same name. If this is not what you want you can rename the column beforehand and use the `on_left` and `on_right` arguments:

```python
df_right = df_right.rename({"int_id": "int_id_right"})

df_left.join_asof(
    df_right,
    left_on="int_id",
    right_on="int_id_right",
)
```

```
shape: (2, 4)
┌────────┬───────┬──────────────┬─────────────┐
│ int_id ┆ value ┆ int_id_right ┆ value_right │
│ ---    ┆ ---   ┆ ---          ┆ ---         │
│ i64    ┆ str   ┆ i64          ┆ i64         │
╞════════╪═══════╪══════════════╪═════════════╡
│ 5      ┆ 1     ┆ 4            ┆ 1           │
│ 10     ┆ 2     ┆ 7            ┆ 2           │
└────────┴───────┴──────────────┴─────────────┘
```

`join_asof` Strategies

The `join_asof` function has three strategies to join DataFrames: * `backward` (default): Join with the last row that has an equal or smaller value. * `forward`: Join with the first row that has an equal or larger value. * `nearest`: Join with the row that has the closest value.

The default strategy is the `backward` strategy. This strategy joins the row on the first value in the other DataFrame's join columns that is equal, or smaller than the value in the left DataFrame, while still falling in the range defined in `tolerance`.

```python
df_left.join_asof(
    df_right,
    left_on="int_id",
    right_on="int_id_right",
```

```
    tolerance=3,
    strategy="backward"
)
shape: (2, 4)
┌────────┬───────┬──────────────┬─────────────┐
│ int_id ┆ value ┆ int_id_right ┆ value_right │
│ ---    ┆ ---   ┆ ---          ┆ ---         │
│ i64    ┆ str   ┆ i64          ┆ i64         │
╞════════╪═══════╪══════════════╪═════════════╡
│ 5      ┆ 1     ┆ 4            ┆ 1           │
│ 10     ┆ 2     ┆ 7            ┆ 2           │
└────────┴───────┴──────────────┴─────────────┘
```

The `forward` strategy looks for the first value in the other DataFrame's join columns that is equal or larger than the value in the left DataFrame.

```
df_left.join_asof(
    df_right,
    left_on="int_id",
    right_on="int_id_right",
    tolerance=3,
    strategy="forward"
)
shape: (2, 4)
┌────────┬───────┬──────────────┬─────────────┐
│ int_id ┆ value ┆ int_id_right ┆ value_right │
│ ---    ┆ ---   ┆ ---          ┆ ---         │
│ i64    ┆ str   ┆ i64          ┆ i64         │
╞════════╪═══════╪══════════════╪═════════════╡
│ 5      ┆ 1     ┆ 7            ┆ 2           │
│ 10     ┆ 2     ┆ 12           ┆ 3           │
└────────┴───────┴──────────────┴─────────────┘
```

Finally, the `nearest` strategy joins the row on the closest value:

```
df_left.join_asof(
    df_right,
    left_on="int_id",
    right_on="int_id_right",
    tolerance=3,
    strategy="nearest"
)
shape: (2, 4)
┌────────┬───────┬──────────────┬─────────────┐
│ int_id ┆ value ┆ int_id_right ┆ value_right │
│ ---    ┆ ---   ┆ ---          ┆ ---         │
│ i64    ┆ str   ┆ i64          ┆ i64         │
╞════════╪═══════╪══════════════╪═════════════╡
│ 5      ┆ 1     ┆ 4            ┆ 1           │
│ 10     ┆ 2     ┆ 12           ┆ 3           │
└────────┴───────┴──────────────┴─────────────┘
```

Additional Finetuning with `tolerance` and `by`

When you set the `tolerance` parameter, rows are only joined if the nearest matching value falls within a certain range. You can set the tolerance for numeric and temporal data types. For temporal data types, use a `datetime.timedelta` or the duration strings (as we previously talked about in Table 10-2), such as `"7d12h30m"`.

If you want to ensure rows are joined only with those in the other DataFrame that share the same value in the specified column(s), instead of just joining with any nearest match, you can use the `by` keyword. If the names of the columns don't match, use the combination of `by_left` and `by_right`.

Now let's put these parameters to good use in a use case.

Use Case: Marketing Campaign Attribution

Imagine: you're tasked with finding out the efficacy of your company's marketing campaigns. You've gathered two datasets: one containing the Sales data, the other containing the Marketing campaigns for the past year:

```
marketing_lf = pl.scan_csv("data/marketing use case/marketing_campaigns.csv")
marketing_lf.fetch(1)
```

```
shape: (1, 3)
┌───────────────┬─────────────────────┬──────────────┐
│ Campaign Name ┆ Campaign Date       ┆ Product Type │
│ ---           ┆ ---                 ┆ ---          │
│ str           ┆ str                 ┆ str          │
╞═══════════════╪═════════════════════╪══════════════╡
│ Launch        ┆ 2023-01-01 20:00:00 ┆ Electronics  │
└───────────────┴─────────────────────┴──────────────┘
```

```
marketing_lf.select(pl.col("Product Type").unique()).collect()
```

```
shape: (4, 1)
┌──────────────┐
│ Product Type │
│ ---          │
│ str          │
╞══════════════╡
│ Books        │
│ Furniture    │
│ Electronics  │
│ Clothing     │
└──────────────┘
```

```
sales_lf = pl.scan_csv("data/marketing use case/sales_data.csv")
sales_lf.fetch(1)
```

```
shape: (1, 3)
┌─────────────────────────┬──────────────┬──────────┐
│ Sale Date               ┆ Product Type ┆ Quantity │
```

```
│ ---                  │ ---             │ ---         │
│ str                  │ str             │ i64         │
╞══════════════════════╪═════════════════╪═════════════╡
│ 2023-01-01 02:00:00.… │ Books           │ 7           │
└──────────────────────┴─────────────────┴─────────────┘
```

You can see that the timestamps are still a string data type. To work with this data you need to format it to a matching temporal data type first. Also, since the dates don't match exactly, use the `join_asof` function to join the two DataFrames. Additionally, match the campaigns with the sales data of the same product category, which you can do by setting the `by` parameter accordingly. And last, campaigns don't work forever. You can assume in this instance that a campaign is in effect for 2 months, so set the `tolerance` to 2 months.

```python
sales_lf = sales_lf.with_columns(
    pl.col("Sale Date")
    .str.to_datetime("%Y-%m-%d %H:%M:%S%.f")
    .cast(pl.Datetime("us")),
)
marketing_lf = marketing_lf.with_columns(
    pl.col("Campaign Date").str.to_datetime("%Y-%m-%d %H:%M:%S"),
)

sales_with_campaign_df = (
    sales_lf
    .sort("Sale Date")
    .join_asof(
        marketing_lf
        .sort("Campaign Date"),
        left_on="Sale Date",
        right_on="Campaign Date",
        by="Product Type",
        strategy="backward",
        tolerance="60d"
    )
    .collect()
)
sales_with_campaign_df
```

```
shape: (20_000, 5)
┌──────────────┬──────────────┬──────────┬───────────────┬───────────────┐
│ Sale Date    ┆ Product Type ┆ Quantity ┆ Campaign Name ┆ Campaign Date │
│ ---          ┆ ---          ┆ ---      ┆ ---           ┆ ---           │
│ datetime[μs] ┆ str          ┆ i64      ┆ str           ┆ datetime[μs]  │
╞══════════════╪══════════════╪══════════╪═══════════════╪═══════════════╡
│ 2023-01-01   ┆ Electronics  ┆ 2        ┆ null          ┆ null          │
│ 01:26:12.…   ┆              ┆          ┆               ┆               │
│ 2023-01-01   ┆ Books        ┆ 7        ┆ null          ┆ null          │
│ 02:00:00     ┆              ┆          ┆               ┆               │
│ 2023-01-01   ┆ Toys         ┆ 9        ┆ null          ┆ null          │
│ 06:14:30.…   ┆              ┆          ┆               ┆               │
│ 2023-01-01   ┆ Clothing     ┆ 9        ┆ null          ┆ null          │
```

```
│ 06:52:25.…  │             │    │      │      │
│ 2023-01-01  │ Books       │ 7  │ null │ null │
│ 07:44:50.…  │             │    │      │      │
│ …           │ …           │ …  │ …    │ …    │
│ 2023-12-31  │ Clothing    │ 10 │ null │ null │
│ 15:45:29.…  │             │    │      │      │
│ 2023-12-31  │ Toys        │ 4  │ null │ null │
│ 18:15:09.…  │             │    │      │      │
│ 2023-12-31  │ Electronics │ 7  │ null │ null │
│ 18:33:47.…  │             │    │      │      │
│ 2023-12-31  │ Books       │ 6  │ null │ null │
│ 18:37:54.…  │             │    │      │      │
│ 2023-12-31  │ Furniture   │ 4  │ null │ null │
│ 19:41:22.…  │             │    │      │      │
└─────────────┴─────────────┴────┴──────┴──────┘
```

Now, if you want to find out whether the campaigns led to a higher average sales quantity, you can group the data by `Product Type` and `Campaign Name`. This lets you compare the products sold with versus without the campaign and calculate the average quantity, like discussed in Chapter 10.

```
(
    sales_with_campaign_df
    .group_by("Product Type", "Campaign Name")
    .agg(pl.col("Quantity").mean())
    .sort("Product Type", "Campaign Name")
)
```

```
shape: (9, 3)
┌──────────────┬───────────────┬──────────┐
│ Product Type ┆ Campaign Name ┆ Quantity │
│ ---          ┆ ---           ┆ ---      │
│ str          ┆ str           ┆ f64      │
╞══════════════╪═══════════════╪══════════╡
│ Books        ┆ null          ┆ 5.527716 │
│ Clothing     ┆ null          ┆ 5.433385 │
│ Clothing     ┆ New Arrivals  ┆ 8.200581 │
│ Electronics  ┆ null          ┆ 5.486832 │
│ Electronics  ┆ Launch        ┆ 8.080775 │
│ Electronics  ┆ Seasonal Sale ┆ 8.471406 │
│ Furniture    ┆ null          ┆ 5.430222 │
│ Furniture    ┆ Discount      ┆ 8.191888 │
│ Toys         ┆ null          ┆ 5.50318  │
└──────────────┴───────────────┴──────────┘
```

From this result we can see that campaigns generally lead to a higher sales quantity, with the exception of the Books and Toys categories. The Toys category never ran a campaign, which explains it, but what about the Books category?

```
marketing_lf.filter(pl.col("Product Type") == "Books").collect()
```

```
shape: (1, 3)
┌────────────┬──────────────┬──────────┐
```

```
│ Campaign Name │ Campaign Date       │ Product Type │
│ ---           │ ---                 │ ---          │
│ str           │ datetime[μs]        │ str          │
╞═══════════════╪═════════════════════╪══════════════╡
│ Clearance     │ 2023-12-31 21:00:00 │ Books        │
└───────────────┴─────────────────────┴──────────────┘
```

It seems that the Books category only ran one campaign: a New Year's Eve clearance sale. Let's see if there are any sales after that moment:

```
(
    sales_lf
    .filter(
        (pl.col("Product Type") == "Books") &
        (
            pl.col("Sale Date") >
            pl.lit("2023-12-31 21:00:00").str.to_datetime()
        )
    )
    .collect()
)
```

```
shape: (0, 3)
┌──────────────┬──────────────┬──────────┐
│ Sale Date    │ Product Type │ Quantity │
│ ---          │ ---          │ ---      │
│ datetime[μs] │ str          │ i64      │
╞══════════════╪══════════════╪══════════╡
└──────────────┴──────────────┴──────────┘
```

It seems that after the clearance started, no more books were sold. Since our `join_asof` strategy was `backward`, this campaign wasn't joined to any of the values, which explains why it's missing in the results! This means that the sales it might've caused are not in the dataset, making it look ineffective!

Vertical and Horizontal Concatenation

The `join` function combines DataFrames based on the values in a DataFrame, but sometimes you just want to add DataFrames together without regard to their the values. Usually DataFrames are stored in different locations in memory. When you want to combine them, you can do three things:

- Combine the data in a new DataFrame by copying it to a new location.
- Point the new DataFrame to the locations where the data is stored.
- Copy the second DataFrame's data behind the data of the first DataFrame.

The first way is to copy data to a new location. This is the default behavior of `pl.concat(…)`. This function takes a list of DataFrames, LazyFrames, or Series and can concatenate them vertically, horizontally or diagonally. After combining the frames it rechunks the resulting DataFrame, copying the data to a new location into a single chunk. This guarantees optimal querying performance afterwards.

As explained in Chapter 2, *rechunking* copies data to a new location in memory to make it contiguous again. This improves the performance of queries, and is especially helpful when the resulting DataFrame is queried multiple times.

`concat` has the following keywords:

- `items`: The list of DataFrames, LazyFrames, or Series to concatenate.
- `how`: The strategy to combine these items.
- `rechunk`: Whether to rechunk the resulting DataFrame. `True` by default.
- `parallel`: Determines if LazyFrames should be computed in parallel. `True` by default.

You can choose the following concatenation strategies:

- `vertical` (default): Concatenate the items vertically. This applies multiple `vstack` operations. It will fail if the DataFrames don't having matching columns, including data type.
- `vertical_relaxed`: Concatenate the items vertically, and additionally coerces columns to a *supertype* if their types don't match. This will fail if the DataFrames don't have matching column names, and disregards their data types.
- `horizontal`: Concatenate the items horizontally. Fills with `null` if the lengths don't match.
- `diagonal`: Combines the items by creating a union of their columns. Fills missing values with `null`.
- `diagonal_relaxed`: Same as `diagonal`, and additionally coerces columns to a *supertype* if their types don't match.
- `align`: Combines frames horizontally in a smart way. It aligns the rows based on the values in the columns that are available in both DataFrames. Missing values are padded with `null`.

The first strategy is *`vertical` concatenation*. This is the default strategy of `concat`. It combines the DataFrames vertically, meaning that the rows of the DataFrames are stacked on top of each other. How it works is shown in Figure 11-1.

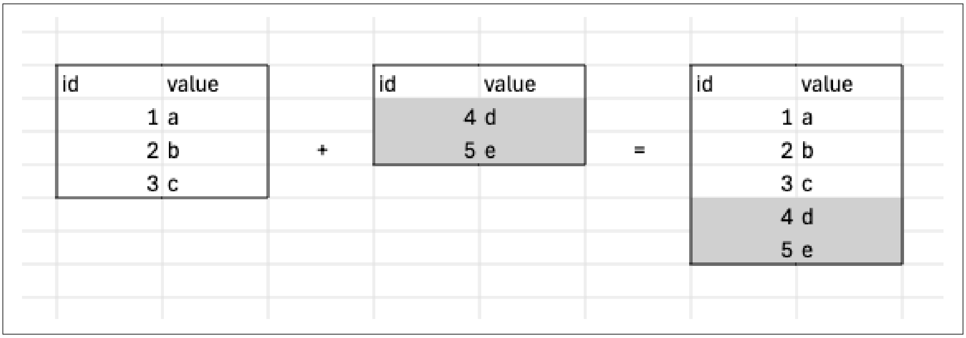


Figure 11-1. Vertical concatenation.

```python
df1 = pl.DataFrame({
    "id": [1, 2, 3],
    "value": ["a", "b", "c"],
})
df2 = pl.DataFrame({
    "id": [4, 5],
    "value": ["d", "e"],
})
pl.concat([df1,df2], how="vertical")
```

```
shape: (5, 2)
┌─────┬───────┐
│ id  ┆ value │
│ --- ┆ ---   │
│ i64 ┆ str   │
╞═════╪═══════╡
│ 1   ┆ a     │
│ 2   ┆ b     │
│ 3   ┆ c     │
│ 4   ┆ d     │
│ 5   ┆ e     │
└─────┴───────┘
```

The second strategy is *`horizontal` concatenation*. This strategy combines the DataFrames horizontally, meaning that the columns of the DataFrames are stacked next to each other. When the lengths of the DataFrames don't match, the resulting DataFrame will be filled with `null` values. Columns cannot have the same name, and if they do, the operation will fail. You can circumvent this by renaming the columns before concatenating. How it works is shown in Figure 11-2.

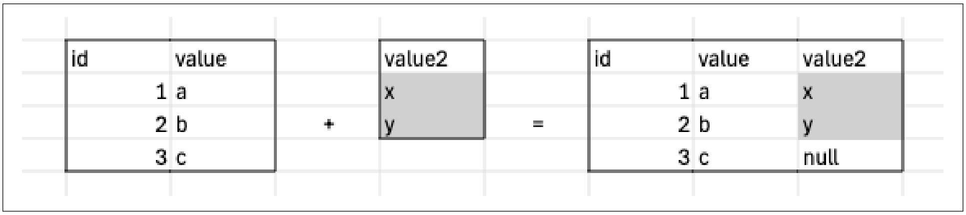


Figure 11-2. Horizontal concatenation.

```
df1 = pl.DataFrame({
    "id": [1, 2, 3],
    "value": ["a", "b", "c"],
})
df2 = pl.DataFrame({
    "value2": ["x", "y"],
})
pl.concat([df1,df2], how="horizontal")
```

```
shape: (3, 3)
┌─────┬───────┬────────┐
│ id  ┆ value ┆ value2 │
│ --- ┆ ---   ┆ ---    │
│ i64 ┆ str   ┆ str    │
╞═════╪═══════╪════════╡
│ 1   ┆ a     ┆ x      │
│ 2   ┆ b     ┆ y      │
│ 3   ┆ c     ┆ null   │
└─────┴───────┴────────┘
```

The third strategy is `` `*diagonal*` *concatenation* ``. This strategy combines the DataFrames by creating a union of their columns. Any column values that are missing in the DataFrames will be filled with `null` values. How it works is shown in Figure 11-3.

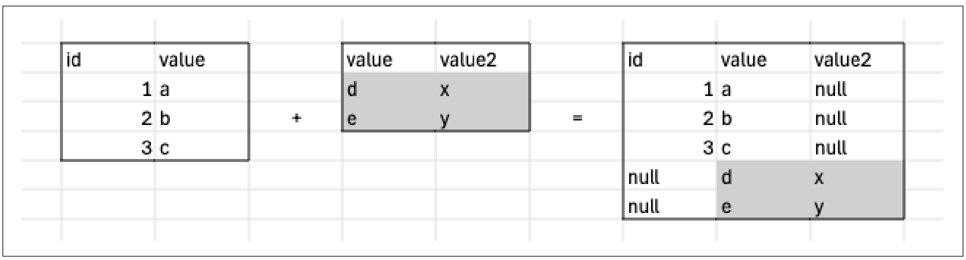


Figure 11-3. Diagonal concatenation.

```
df1 = pl.DataFrame({
    "id": [1, 2, 3],
    "value": ["a", "b", "c"],
})
df2 = pl.DataFrame({
    "value": ["d", "e"],
```

```
    "value2": ["x", "y"],
})
pl.concat([df1,df2], how="diagonal")
```

```
shape: (5, 3)
┌──────┬───────┬────────┐
│ id   ┆ value ┆ value2 │
│ ---  ┆ ---   ┆ ---    │
│ i64  ┆ str   ┆ str    │
╞══════╪═══════╪════════╡
│ 1    ┆ a     ┆ null   │
│ 2    ┆ b     ┆ null   │
│ 3    ┆ c     ┆ null   │
│ null ┆ d     ┆ x      │
│ null ┆ e     ┆ y      │
└──────┴───────┴────────┘
```

The fourth and last strategy is *`align` concatenation*. This strategy doesn't simply tape rows or columns together. Instead it finds matching values in columns that are available in both DataFrames, and aligns the rows based on these values, like shown in Figure 11-4.

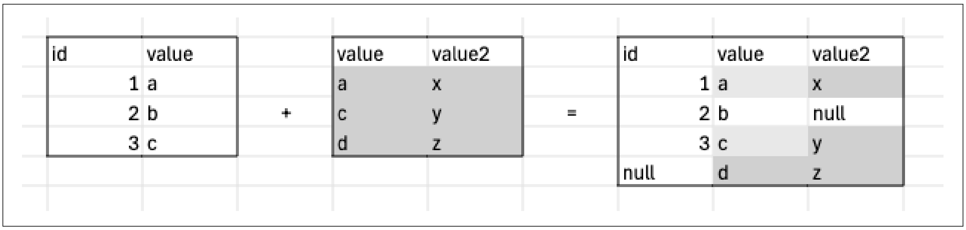


Figure 11-4. Aligned concatenation.

```
df1 = pl.DataFrame({
    "id": [1, 2, 3],
    "value": ["a", "b", "c"],
})
df2 = pl.DataFrame({
    "value": ["a", "c", "d"],
    "value2": ["x", "y", "z"],
})
pl.concat([df1,df2], how="align")
```

```
shape: (4, 3)
┌─────┬───────┬────────┐
│ id  ┆ value ┆ value2 │
│ --- ┆ ---   ┆ ---    │
│ i64 ┆ str   ┆ str    │
╞═════╪═══════╪════════╡
│ 1   ┆ a     ┆ x      │
│ 2   ┆ b     ┆ null   │
│ 3   ┆ c     ┆ y      │
```

```
│ null ┆ d     ┆ z     │
└──────┴───────┴───────┘
```

In addition, the `vertical`, `horizontal`, and `diagonal` strategies each have a `relaxed` version. This means that if the types of columns with the same names in both frames don't match, the columns will be coerced to become a *supertype*. For example, a column with integers and floats will be coerced to a float column, and a column with integers and strings will be coerced to a string column. This is useful when you want to concatenate DataFrames that have the same columns but different data types.

The example below shows what happens when you try to concatenate two DataFrames with the same columns but different data types:

```
df1 = pl.DataFrame({
    "id": [1, 2, 3],
    "value": ["a", "b", "c"],
})
df2 = pl.DataFrame({
    "id": [4.0, 5.0],
    "value": [1, 2],
})
pl.concat([df1,df2], how="vertical")
```

```
SchemaError: type Float64 is incompatible with expected type Int64
```

When you use the `vertical_relaxed` strategy, the concatenation will succeed:

```
pl.concat([df1,df2], how="vertical_relaxed")
```

```
shape: (5, 2)
┌─────┬───────┐
│ id  ┆ value │
│ --- ┆ ---   │
│ f64 ┆ str   │
╞═════╪═══════╡
│ 1.0 ┆ a     │
│ 2.0 ┆ b     │
│ 3.0 ┆ c     │
│ 4.0 ┆ 1     │
│ 5.0 ┆ 2     │
└─────┴───────┘
```

`align_frames`

The align strategy is based on the `align_frames` function. This function lets you pick a column and aligns the rows of a set of DataFrames according to the values in that column. If a value is missing in one of the DataFrames, the resulting DataFrame will have a `null` value in that row. If values appear multiple times the Cartesian product of the rows will be created. The function returns the same DataFrames, but with their rows aligned to each other.

The keywords of `align_frames` are:

- *frames: The frames you want to align to each other.
- *on: The column(s) to align the frames on.
- how: The join strategy used to determine the resulting values. The default is `outer`.
- select: Columns and their order to select from the resulting DataFrames.
- descending: Whether to sort the resulting DataFrame in descending order. This can also be a list of Booleans with a matching length of the columns provided in `on`.

```
df1 = pl.DataFrame({
    "id": [1, 2, 2],
    "value": ["a", "c", "b"],
})
df2 = pl.DataFrame({
    "id": [2, 2],
    "value": ["x", "y"],
})
pl.align_frames(df1,df2, on="id")
```

```
[shape: (5, 2)
 ┌─────┬───────┐
 │ id  ┆ value │
 │ --- ┆ ---   │
 │ i64 ┆ str   │
 ╞═════╪═══════╡
 │ 1   ┆ a     │
 │ 2   ┆ c     │
 │ 2   ┆ c     │
 │ 2   ┆ b     │
 │ 2   ┆ b     │
 └─────┴───────┘,
 shape: (5, 2)
 ┌─────┬───────┐
 │ id  ┆ value │
 │ --- ┆ ---   │
 │ i64 ┆ str   │
 ╞═════╪═══════╡
 │ 1   ┆ null  │
 │ 2   ┆ x     │
 │ 2   ┆ y     │
 │ 2   ┆ x     │
 │ 2   ┆ y     │
 └─────┴───────┘]
```


Note that in the second frame, which is missing the ID "1", the

The vstack and hstack functions use the second way of combining DataFrames. (A comparable function exists for Series, called append.) They combine two DataFrames without moving the data in memory. Instead they create a new DataFrame or Series containing multiple chunks that can be located in different parts of memory. This makes stack operations quick, with the drawback that querying could be slower because data has to be read from multiple locations in memory. The concat vertical strategy uses the vstack operation, but this can also be called by itself. concat allows you to rechunk the resulting DataFrame to prevent this performance hit, while vstack does not.

These are the preferred methods when you append multiple DataFrames one after the other. Note that stack operations only work on DataFrames, not LazyFrames, since they need to combine existing chunks.

vstack requires the width, column names, and their dtypes to match.

```
df1 = pl.DataFrame({
    "id": [1, 2],
    "value": ["a", "b"],
})
df2 = pl.DataFrame({
    "id": [3, 4],
    "value": ["c", "d"],
})
df1.vstack(df2)
```

```
shape: (4, 2)
┌─────┬───────┐
│ id  ┆ value │
│ --- ┆ ---   │
│ i64 ┆ str   │
╞═════╪═══════╡
│ 1   ┆ a     │
│ 2   ┆ b     │
│ 3   ┆ c     │
│ 4   ┆ d     │
└─────┴───────┘
```

Exactly like vstack, hstack combines DataFrames horizontally. This operation requires the height of the DataFrames to match.

```
df1 = pl.DataFrame({
    "id": [1, 2],
    "value": ["a", "b"],
})
df2 = pl.DataFrame({
    "value2": ["x", "y"],
})
df1.hstack(df2)
```

```
shape: (2, 3)
┌─────┬───────┬────────┐
│ id  ┆ value ┆ value2 │
│ --- ┆ ---   ┆ ---    │
│ i64 ┆ str   ┆ str    │
╞═════╪═══════╪════════╡
│ 1   ┆ a     ┆ x      │
│ 2   ┆ b     ┆ y      │
└─────┴───────┴────────┘
```

For Series, you can use `append`. This keeps the name of the first Series.

```python
s1 = pl.Series("a", [1, 2])
s2 = pl.Series("b", [3, 4])
s1.append(s2)
```

```
shape: (4,)
Series: 'a' [i64]
[
	1
	2
	3
	4
]
```

The third way to combine DataFrames is `extend`. When there's enough space available in memory behind the original DataFrame, `extend` copies the data of the second DataFrame behind the first one. This eliminates the need to copy the data to a new location, which can be faster, and still keeps the data contiguous in memory. This works best when you want to add a smaller DataFrame to a larger one. Note that `extend` modifies the DataFrame in place! It does return the resulting DataFrame as well, but just as a convenience.

```python
df1 = pl.DataFrame({
    "id": [1, 2],
    "value": ["a", "b"],
})
df2 = pl.DataFrame({
    "id": [3, 4],
    "value": ["c", "d"],
})
df1.extend(df2)
```

```
shape: (4, 2)
┌─────┬───────┐
│ id  ┆ value │
│ --- ┆ ---   │
│ i64 ┆ str   │
╞═════╪═══════╡
│ 1   ┆ a     │
│ 2   ┆ b     │
│ 3   ┆ c     │
```

```
│ 4   ┆ d    │
└─────┴──────┘
```

Conclusion

In this chapter you learned how to combine DataFrames.

- You can use `join` to combine DataFrames based on the values in the DataFrames and the strategies outlined here. You can to finetune the join with the `tolerance` and `by` parameters.
- `join_asof` is a special join that joins DataFrames based on the nearest value in the other DataFrame.
- `concat` combines DataFrames without regarding the values, which has multiple strategies to combine the DataFrames, and you can optimize performance of the resulting DataFrame with the `rechunk` parameter.
- `vstack` and `hstack` stack DataFrames vertically and horizontally, respectively, and `append` appends Series to each other by creating a new DataFrame with multiple chunks, which is quick, but less performant when querying.
- `extend` adds a DataFrame to another DataFrame by copying the data behind the original DataFrame, which is faster than copying the data to a new location.

In the next chapter we'll look into reshaping DataFrames, which is useful when you want to change the structure of your data.

CHAPTER 12
Reshaping

> ### A Note for Early Release Readers
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 15th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

In the last chapter we focussed on aggregating data to create informative summaries. However, what should you do if the data is not in the right shape to perform these aggregations? In this chapter we will learn how to reshape data to make it more suitable for analysis.

Reshaping data is a crucial step in the data analysis process. It involves changing the dimensions of the data to make it more suitable for analysis, improve computational performance, or prepare it for visualization. Polars offers a variety of functions to reshape the data exactly like you need it, using functions like `pivot`, `melt`, `transpose`, `explode`, and `partition_by`.

Wide Versus Long DataFrames

There once was a wise man called Hadley Wickham. He is a statistician known for, among other things, his work on the R programming language and the "tidyverse", which is a popular suite of packages to process and visualize data in R. Related to that, in 2014 he wrote a paper titled "Tidy Data" in which he introduced the concept of wide and long DataFrames. These concepts are to this day widely used in the data science community to describe the shape of DataFrames, and we will use it in this chapter as well.

Wickham stated that DataFrames can be represented in two extreme forms: wide and long. Wide DataFrames have many columns and few rows. The idea is that every row contains a column with an identifier, and the data is spread over many columns. This format is often used when there are multiple measurements per observation. An example of wide data would be the following:

```python
import polars as pl

df = pl.DataFrame({
    "student": ["Alice", "Bob", "Charlie"],
    "math": [85, 78, 92],
    "science": [90, 82, 85],
    "history": [88, 80, 87]
})
df
```

```
shape: (3, 4)
┌─────────┬──────┬─────────┬─────────┐
│ student ┆ math ┆ science ┆ history │
│ ---     ┆ ---  ┆ ---     ┆ ---     │
│ str     ┆ i64  ┆ i64     ┆ i64     │
╞═════════╪══════╪═════════╪═════════╡
│ Alice   ┆ 85   ┆ 90      ┆ 88      │
│ Bob     ┆ 78   ┆ 82      ┆ 80      │
│ Charlie ┆ 92   ┆ 85      ┆ 87      │
└─────────┴──────┴─────────┴─────────┘
```

This DataFrame has three columns with the subjects as the column names.

Where wide DataFrames have many columns, long DataFrames have few columns and many rows. Instead of having multiple values and variables per row, long DataFrames have multiple rows with each one variable and corresponding value. The last example in long format would look like the following:

```python
df = pl.DataFrame({
    "student": ["Alice", "Alice", "Alice", "Bob", "Bob", "Bob", "Charlie",
        "Charlie", "Charlie"],
    "subject": ["Math", "Science", "History", "Math", "Science", "History",
        "Math", "Science", "History"],
    "grade": [85, 90, 88, 78, 82, 80, 92, 85, 87]
})
df
```

```
shape: (9, 3)
┌─────────┬─────────┬───────┐
│ student ┆ subject ┆ grade │
```

```
│ ---     │ ---     │ ---   │
│ str     │ str     │ i64   │
╞═════════╪═════════╪═══════╡
│ Alice   │ Math    │ 85    │
│ Alice   │ Science │ 90    │
│ Alice   │ History │ 88    │
│ Bob     │ Math    │ 78    │
│ Bob     │ Science │ 82    │
│ Bob     │ History │ 80    │
│ Charlie │ Math    │ 92    │
│ Charlie │ Science │ 85    │
│ Charlie │ History │ 87    │
└─────────┴─────────┴───────┘
```

This formats the DataFrame so that every row contains only one observation.

The implications of the used format on memory usage and computational performance are significant. Since Polars uses a columnar storage format, long DataFrames tend to be more efficient in terms of memory usage and computational performance.

Pivot to Wider DataFrame

If you want to go from a long format to a wide DataFrame, you can use the `pivot` function. The `pivot` function takes the following arguments:

`index`
: Columns to use as identifiers for the rows.

`columns`
: Columns containing what will be the column names.

`values`
: Columns containing the values that will end up in the cells.

`aggregate_function`
: The function used to aggregate the values if there are multiple values for a single cell. If left empty, the function will throw an error if there are multiple values for a single cell.

`maintain_order`
: Sort the grouped keys to make the outcome predictable.

`sort_columns`
: Sort the transposed values of `columns`, also the transposed columns, by name. The default is to sort by the order of appearance in the DataFrame.

`separator`
: The string used as separator in generated column names.

Let's break it down with an example. You've got to store the grades of a group of students. As the grades come in one by one, you store them in a long DataFrame:

```
import polars as pl

df = pl.DataFrame({
    "student": ["Alice", "Alice", "Alice", "Bob", "Bob", "Bob", "Charlie",
        "Charlie", "Charlie"],
    "subject": ["Math", "Science", "History", "Math", "Science", "History",
        "Math", "Science", "History"],
    "grade": [85, 90, 88, 78, 82, 80, 92, 85, 87]
})

df
```

```
shape: (9, 3)
┌─────────┬─────────┬───────┐
│ student ┆ subject ┆ grade │
│ ---     ┆ ---     ┆ ---   │
│ str     ┆ str     ┆ i64   │
╞═════════╪═════════╪═══════╡
│ Alice   ┆ Math    ┆ 85    │
│ Alice   ┆ Science ┆ 90    │
│ Alice   ┆ History ┆ 88    │
│ Bob     ┆ Math    ┆ 78    │
│ Bob     ┆ Science ┆ 82    │
│ Bob     ┆ History ┆ 80    │
│ Charlie ┆ Math    ┆ 92    │
│ Charlie ┆ Science ┆ 85    │
│ Charlie ┆ History ┆ 87    │
└─────────┴─────────┴───────┘
```

Neat. Now at the time of the student's report cards being handed out, you want to create one row per student. You can do this by pivoting on the `subject` column:

```
df.pivot(index="student", columns="subject", values="grade")
```

```
shape: (3, 4)
┌─────────┬──────┬─────────┬─────────┐
│ student ┆ Math ┆ Science ┆ History │
│ ---     ┆ ---  ┆ ---     ┆ ---     │
│ str     ┆ i64  ┆ i64     ┆ i64     │
╞═════════╪══════╪═════════╪═════════╡
│ Alice   ┆ 85   ┆ 90      ┆ 88      │
│ Bob     ┆ 78   ┆ 82      ┆ 80      │
│ Charlie ┆ 92   ┆ 85      ┆ 87      │
└─────────┴──────┴─────────┴─────────┘
```

You can see how you went to a wide DataFrame with the student names as the index and the subjects as the columns!

In reality, students don't just get one grade, but multiple grades. This means you'll have to aggregate the grades! Luckily, the `pivot` function can handle this for us.

By default, the `pivot` function will not aggregate and throw an error if there are multiple values. In our example this didn't happen, because all the values are unique. You can change this by passing the `aggregate_function` argument with the desired aggregation function. You can select from the following aggregation functions: `min`, `max`, `first`, `last`, `sum`, `mean`, `median`, and `len`. In our example to calculate the average grade, you can use the `mean` aggregation function. First let's update the DataFrame to have multiple grades per student:

```
df = pl.DataFrame({
    "student": ["Alice", "Alice", "Alice", "Alice", "Alice", "Alice",
                "Bob", "Bob", "Bob", "Bob", "Bob", "Bob"],
    "subject": ["Math", "Math", "Math", "Science", "Science", "Science",
                "Math", "Math", "Math", "Science", "Science", "Science"],
    "grade": [85, 88, 85, 60, 66, 63,
              51, 79, 62, 82, 85, 82]
})

df
```

```
shape: (12, 3)
┌─────────┬─────────┬───────┐
│ student ┆ subject ┆ grade │
│ ---     ┆ ---     ┆ ---   │
│ str     ┆ str     ┆ i64   │
╞═════════╪═════════╪═══════╡
│ Alice   ┆ Math    ┆ 85    │
│ Alice   ┆ Math    ┆ 88    │
│ Alice   ┆ Math    ┆ 85    │
│ Alice   ┆ Science ┆ 60    │
│ Alice   ┆ Science ┆ 66    │
│ …       ┆ …       ┆ …     │
│ Bob     ┆ Math    ┆ 79    │
│ Bob     ┆ Math    ┆ 62    │
│ Bob     ┆ Science ┆ 82    │
│ Bob     ┆ Science ┆ 85    │
│ Bob     ┆ Science ┆ 82    │
└─────────┴─────────┴───────┘
```

Now you can pivot the DataFrame to calculate the average grade per student:

```
df.pivot(
    index="student",
    columns="subject",
    values="grade",
    aggregate_function="mean"
)
```

```
shape: (2, 3)
┌─────────┬──────┬─────────┐
│ student ┆ Math ┆ Science │
│ ---     ┆ ---  ┆ ---     │
│ str     ┆ f64  ┆ f64     │
```

```
├─────────┼──────┼─────────┤
│ Alice   ┆ 86.0 ┆ 63.0    │
│ Bob     ┆ 64.0 ┆ 83.0    │
└─────────┴──────┴─────────┘
```

In addition to this list of standard aggregation functions, you can also pass a custom aggregation function through an expression. By creating an expression that can be run against a lists elements, like `pl.col(…).list.eval(<your_expression>)`, you can make use of extended flexibility. For example, you can calculate the difference between the maximum and minimum grade to show the stability of the students' grades:

```
df.pivot(
    index="student",
    columns="subject",
    values="grade",
    aggregate_function=pl.element().max() - pl.element().min()
)
```

```
shape: (2, 3)
┌─────────┬──────┬─────────┐
│ student ┆ Math ┆ Science │
│ ---     ┆ ---  ┆ ---     │
│ str     ┆ i64  ┆ i64     │
╞═════════╪══════╪═════════╡
│ Alice   ┆ 3    ┆ 6       │
│ Bob     ┆ 28   ┆ 3       │
└─────────┴──────┴─────────┘
```

Here you can see Bob has a much larger difference between his maximum and minimum grade than Alice in Math. In our use case, you could use this to approach Bob's mentor to make sure he's okay, because he seems to have slipped up on one of his tests!

Now that's a whole lot easier than getting a couch up the stairs!

Melt to Longer DataFrame

If instead you want to go from a wide format to a long DataFrame, or *unpivot*, you can use the `melt` function. The `melt` function takes the following arguments: `id_vars`:: Columns to use as identifiers for the rows. These will remain columns in the resulting frame. `value_vars`:: Columns to melt. If not specified, uses all columns not set in `id_vars`. `variable_name`:: Name of the resulting column that will contain the names of the column that were melted. `value_name`:: Name of the resulting column that contains the values of the columns that were melted.

Let's illustrate this with an example. Let's take the report card set-up of the data from the previous section. Every student has a row with their grades for Math, Science, and History.

```python
df = pl.DataFrame({
    "student": ["Alice", "Bob", "Charlie"],
    "math": [85, 78, 92],
    "science": [90, 82, 85],
    "history": [88, 80, 87]
})
df
```

```
shape: (3, 4)
┌─────────┬──────┬─────────┬─────────┐
│ student ┆ math ┆ science ┆ history │
│ ---     ┆ ---  ┆ ---     ┆ ---     │
│ str     ┆ i64  ┆ i64     ┆ i64     │
╞═════════╪══════╪═════════╪═════════╡
│ Alice   ┆ 85   ┆ 90      ┆ 88      │
│ Bob     ┆ 78   ┆ 82      ┆ 80      │
│ Charlie ┆ 92   ┆ 85      ┆ 87      │
└─────────┴──────┴─────────┴─────────┘
```

You can melt this DataFrame to get a long DataFrame with one row per student per subject:

```python
df.melt(
    id_vars=["student"],
    value_vars=["math", "science", "history"],
    variable_name="subject",
    value_name="grade"
)
```

```
shape: (9, 3)
┌─────────┬─────────┬───────┐
│ student ┆ subject ┆ grade │
│ ---     ┆ ---     ┆ ---   │
│ str     ┆ str     ┆ i64   │
╞═════════╪═════════╪═══════╡
│ Alice   ┆ math    ┆ 85    │
│ Bob     ┆ math    ┆ 78    │
│ Charlie ┆ math    ┆ 92    │
│ Alice   ┆ science ┆ 90    │
│ Bob     ┆ science ┆ 82    │
│ Charlie ┆ science ┆ 85    │
│ Alice   ┆ history ┆ 88    │
│ Bob     ┆ history ┆ 80    │
│ Charlie ┆ history ┆ 87    │
└─────────┴─────────┴───────┘
```

Here the way we identify the rows in the resulting frame is by the `student` column, containing the name of the student. All the columns that contain what will be the values in the returning frame are the columns with subjects (Math, Science, and History). The way we'll call the columns that contain these subjects is by the `variable_name` column we'll call `subject`, and the `value_name` will be stored in the `grade` column.

```python
df = pl.DataFrame({
    "student": ["Alice", "Bob", "Charlie", "Alice", "Bob", "Charlie"],
    "class": ["Math101", "Math101", "Math101", "Math102", "Math102", "Math102"],
    "age": [20, 21, 22, 20, 21, 22],
    "semester": ["Fall", "Fall", "Fall", "Spring", "Spring", "Spring"],
    "math": [85, 78, 92, 88, 79, 95],
    "science": [90, 82, 85, 92, 81, 87],
    "history": [88, 80, 87, 85, 82, 89]
})
df
```

shape: (6, 7)

student --- str	class --- str	age --- i64	semester --- str	math --- i64	science --- i64	history --- i64
Alice	Math101	20	Fall	85	90	88
Bob	Math101	21	Fall	78	82	80
Charlie	Math101	22	Fall	92	85	87
Alice	Math102	20	Spring	88	92	85
Bob	Math102	21	Spring	79	81	82
Charlie	Math102	22	Spring	95	87	89

```python
df.melt(
    id_vars=["student", "class", "age", "semester"],
    value_vars=["math", "science", "history"],
    variable_name="subject",
    value_name="grade"
)
```

shape: (18, 6)

student --- str	class --- str	age --- i64	semester --- str	subject --- str	grade --- i64
Alice	Math101	20	Fall	math	85
Bob	Math101	21	Fall	math	78
Charlie	Math101	22	Fall	math	92
Alice	Math102	20	Spring	math	88
Bob	Math102	21	Spring	math	79
Charlie	Math102	22	Spring	math	95
Alice	Math101	20	Fall	science	90
Bob	Math101	21	Fall	science	82
Charlie	Math101	22	Fall	science	85
Alice	Math102	20	Spring	science	92
Bob	Math102	21	Spring	science	81
Charlie	Math102	22	Spring	science	87
Alice	Math101	20	Fall	history	88
Bob	Math101	21	Fall	history	80
Charlie	Math101	22	Fall	history	87
Alice	Math102	20	Spring	history	85

```
│ Bob     │ Math102 │ 21  │ Spring   │ history │ 82   │
│ Charlie │ Math102 │ 22  │ Spring   │ history │ 89   │
└─────────┴─────────┴─────┴──────────┴─────────┴──────┘
```

Transposing

If you want to flip all the columns into rows diagonally, without keeping some columns as identifiers, you can use the `transpose` function. The `transpose` function only works on DataFrames, and takes the following arguments: `include_header`:: Whether to set the column names to the first column in the resulting DataFrame. `header_name`:: If `include_header` is set to `True`, this will be the name of the column containing the original column names. It defaults to `column`. `column_names`:: You can pass a list of column names (or another iterable) that will be used as the column names in the resulting DataFrame.

Time for an example. Let's take the DataFrame from the previous section:

```
df = pl.DataFrame({
    "student": ["Alice", "Bob", "Charlie"],
    "math": [85, 78, 92],
    "science": [90, 82, 85],
    "history": [88, 80, 87]
})
df
```

```
shape: (3, 4)
┌─────────┬──────┬─────────┬─────────┐
│ student │ math │ science │ history │
│ ---     │ ---  │ ---     │ ---     │
│ str     │ i64  │ i64     │ i64     │
╞═════════╪══════╪═════════╪═════════╡
│ Alice   │ 85   │ 90      │ 88      │
│ Bob     │ 78   │ 82      │ 80      │
│ Charlie │ 92   │ 85      │ 87      │
└─────────┴──────┴─────────┴─────────┘
```

Now let's flip this frame diagonally:

```
df.transpose(
    include_header=True,
    header_name="original_headers",
    column_names=(f"report_{count}" for count in range(1, len(df.columns) + 1))
)
```

```
shape: (4, 4)
┌──────────────────┬──────────┬──────────┬──────────┐
│ original_headers │ report_1 │ report_2 │ report_3 │
│ ---              │ ---      │ ---      │ ---      │
│ str              │ str      │ str      │ str      │
╞══════════════════╪══════════╪══════════╪══════════╡
│ student          │ Alice    │ Bob      │ Charlie  │
│ math             │ 85       │ 78       │ 92       │
```

```
│ science          │ 90     │ 82     │ 85     │
│ history          │ 88     │ 80     │ 87     │
└──────────────────┴────────┴────────┴────────┘
```

All the columns are now rows, and the original column names are stored in the `original_headers` column!

> **Generators in Python**
>
> Generators are a special type of function that will return an iterable sequence of items. They can be defined in multiple ways, from functions with the `yield` keyword to generator expressions. In the example above, we used a generator expression to create a sequence of strings. This is done by using a for loop inside parentheses, which results in an list of column names that can be used by Polars' `transpose`.

Exploding

When you have a `List` or `Array` in your columns, it isn't quite the wide format we talked about earlier, but it's also not a long format. In case you want to unpack these nested values into a long format, you can use the `explode` function. Instead of blowing stuff up, this function safely creates a row for every value in the nested column copying the values from the other columns. The only arguments `explode` takes are the columns it is supposed to unpack to separate rows. Sticking to the student example, let's list the scores for one subject.

```
df = pl.DataFrame({
    "student": ["Alice", "Bob", "Charlie"],
    "math": [[85, 90, 88], [78, 82, 80], [92, 85, 87]]
})
df
```

```
shape: (3, 2)
┌─────────┬──────────────┐
│ student │ math         │
│ ---     │ ---          │
│ str     │ list[i64]    │
╞═════════╪══════════════╡
│ Alice   │ [85, 90, 88] │
│ Bob     │ [78, 82, 80] │
│ Charlie │ [92, 85, 87] │
└─────────┴──────────────┘
```

In order to turn this frame into a long format we can apply `explode` to the `math` column:

```
df.explode("math")
```

```
shape: (9, 2)
┌─────────┬──────┐
│ student ┆ math │
│ ---     ┆ ---  │
│ str     ┆ i64  │
╞═════════╪══════╡
│ Alice   ┆ 85   │
│ Alice   ┆ 90   │
│ Alice   ┆ 88   │
│ Bob     ┆ 78   │
│ Bob     ┆ 82   │
│ Bob     ┆ 80   │
│ Charlie ┆ 92   │
│ Charlie ┆ 85   │
│ Charlie ┆ 87   │
└─────────┴──────┘
```

And in the case of multiple columns:

```
df = pl.DataFrame({
    "student": ["Alice", "Bob", "Charlie"],
    "math": [[85, 90, 88], [78, 82, 80], [92, 85, 87]],
    "science": [[85, 90, 88], [78, 82], [92, 85, 87]],
    "history": [[85, 90, 88], [78, 82], [92, 85, 87]],
})
df
```

```
shape: (3, 4)
┌─────────┬──────────────┬──────────────┬──────────────┐
│ student ┆ math         ┆ science      ┆ history      │
│ ---     ┆ ---          ┆ ---          ┆ ---          │
│ str     ┆ list[i64]    ┆ list[i64]    ┆ list[i64]    │
╞═════════╪══════════════╪══════════════╪══════════════╡
│ Alice   ┆ [85, 90, 88] ┆ [85, 90, 88] ┆ [85, 90, 88] │
│ Bob     ┆ [78, 82, 80] ┆ [78, 82]     ┆ [78, 82]     │
│ Charlie ┆ [92, 85, 87] ┆ [92, 85, 87] ┆ [92, 85, 87] │
└─────────┴──────────────┴──────────────┴──────────────┘
```

In order to turn this frame into a long format we can apply `explode` to the `math` column:

```
df.explode("math", "science", "history")
```

```
ShapeError: exploded columns must have matching element counts
```

Please note in the above example that the order of values in the lists is important! The items that are lined up end up on the same row in the results. We've discussed sorting of lists in Chapter 11. Additionally, the exploded columns must all yield the same number of resulting rows, otherwise it'll raise a `ShapeError`:

```
df = pl.DataFrame({
    "id": [1,2],
    "value1": [["a", "b"], ["c"]],
    "value2": [["a"], ["b"]],
```

```
})
df.explode("value1", "value2")
```

```
ShapeError: exploded columns must have matching element counts
```

`explode` can even deal with nested lists:

```
df = pl.DataFrame({
    "id": [1,2],
    "nested_value": [["a", "b"], [["c"], ["d", "e"]]],
}, strict=False)
df
```

```
shape: (2, 2)
┌─────┬─────────────────────┐
│ id  ┆ nested_value        │
│ --- ┆ ---                 │
│ i64 ┆ list[list[str]]     │
╞═════╪═════════════════════╡
│ 1   ┆ [["a"], ["b"]]      │
│ 2   ┆ [["c"], ["d", "e"]] │
└─────┴─────────────────────┘
```

Note that it with a nested structure it will only explode one layer at a time:

```
df.explode("nested_value")
```

```
shape: (4, 2)
┌─────┬──────────────┐
│ id  ┆ nested_value │
│ --- ┆ ---          │
│ i64 ┆ list[str]    │
╞═════╪══════════════╡
│ 1   ┆ ["a"]        │
│ 1   ┆ ["b"]        │
│ 2   ┆ ["c"]        │
│ 2   ┆ ["d", "e"]   │
└─────┴──────────────┘
```

In case you want to get the string values, you'll have to call it two times:

```
df.explode("nested_value").explode("nested_value")
```

```
shape: (5, 2)
┌─────┬──────────────┐
│ id  ┆ nested_value │
│ --- ┆ ---          │
│ i64 ┆ str          │
╞═════╪══════════════╡
│ 1   ┆ a            │
│ 1   ┆ b            │
│ 2   ┆ c            │
│ 2   ┆ d            │
│ 2   ┆ e            │
└─────┴──────────────┘
```

Partition into Multiple DataFrames

Previously we discussed the `group_by` operation in Chapter 10. You can use a comparable function to split the DataFrame into multiple partitions. By using `partition_by` you group a DataFrame by some given columns and return the groups as separate DataFrames: `partition_by` takes the following arguments: `by` and `*more_by`:: The column(s) to group by. `maintain_order`:: Ensure that the order of the groups is deterministic. `include_key`:: Instead of a list of DataFrames, return a list of tuples with the group key and the DataFrame. `as_dict`:: Return the group by key(s) as a dictionary.

Let's create an example with some fictional sales data for different regions:

```
df = pl.DataFrame({
    "OrderID": [1, 2, 3, 4, 5, 6],
    "Product": ["A", "B", "A", "C", "B", "A"],
    "Quantity": [10, 5, 8, 7, 3, 12],
    "Region": ["North", "South", "North", "West", "South", "West"]
})
```

Now you can partition the DataFrame by the `Region` column:

```
df.partition_by("Region")
```

```
[shape: (2, 4)
┌─────────┬─────────┬──────────┬────────┐
│ OrderID ┆ Product ┆ Quantity ┆ Region │
│ ---     ┆ ---     ┆ ---      ┆ ---    │
│ i64     ┆ str     ┆ i64      ┆ str    │
╞═════════╪═════════╪══════════╪════════╡
│ 1       ┆ A       ┆ 10       ┆ North  │
│ 3       ┆ A       ┆ 8        ┆ North  │
└─────────┴─────────┴──────────┴────────┘,
 shape: (2, 4)
┌─────────┬─────────┬──────────┬────────┐
│ OrderID ┆ Product ┆ Quantity ┆ Region │
│ ---     ┆ ---     ┆ ---      ┆ ---    │
│ i64     ┆ str     ┆ i64      ┆ str    │
╞═════════╪═════════╪══════════╪════════╡
│ 2       ┆ B       ┆ 5        ┆ South  │
│ 5       ┆ B       ┆ 3        ┆ South  │
└─────────┴─────────┴──────────┴────────┘,
 shape: (2, 4)
┌─────────┬─────────┬──────────┬────────┐
│ OrderID ┆ Product ┆ Quantity ┆ Region │
│ ---     ┆ ---     ┆ ---      ┆ ---    │
│ i64     ┆ str     ┆ i64      ┆ str    │
╞═════════╪═════════╪══════════╪════════╡
│ 4       ┆ C       ┆ 7        ┆ West   │
│ 6       ┆ A       ┆ 12       ┆ West   │
└─────────┴─────────┴──────────┴────────┘]
```

If you want to drop the column you're partitioning by, you can set the `include_key` to `False`:

```
df.partition_by("Region", include_key=False)
```

```
[shape: (2, 3)
┌─────────┬─────────┬──────────┐
│ OrderID ┆ Product ┆ Quantity │
│ ---     ┆ ---     ┆ ---      │
│ i64     ┆ str     ┆ i64      │
╞═════════╪═════════╪══════════╡
│ 1       ┆ A       ┆ 10       │
│ 3       ┆ A       ┆ 8        │
└─────────┴─────────┴──────────┘,
 shape: (2, 3)
┌─────────┬─────────┬──────────┐
│ OrderID ┆ Product ┆ Quantity │
│ ---     ┆ ---     ┆ ---      │
│ i64     ┆ str     ┆ i64      │
╞═════════╪═════════╪══════════╡
│ 2       ┆ B       ┆ 5        │
│ 5       ┆ B       ┆ 3        │
└─────────┴─────────┴──────────┘,
 shape: (2, 3)
┌─────────┬─────────┬──────────┐
│ OrderID ┆ Product ┆ Quantity │
│ ---     ┆ ---     ┆ ---      │
│ i64     ┆ str     ┆ i64      │
╞═════════╪═════════╪══════════╡
│ 4       ┆ C       ┆ 7        │
│ 6       ┆ A       ┆ 12       │
└─────────┴─────────┴──────────┘]
```

And finally, if you want to get the results as a dictionary using a tuple with the group keys as key, and the DataFrames as value, you can set the `as_dict` argument to `True`:

```
dfs = df.partition_by(["Region"], as_dict=True)
dfs
```

```
{('North',): shape: (2, 4)
┌─────────┬─────────┬──────────┬────────┐
│ OrderID ┆ Product ┆ Quantity ┆ Region │
│ ---     ┆ ---     ┆ ---      ┆ ---    │
│ i64     ┆ str     ┆ i64      ┆ str    │
╞═════════╪═════════╪══════════╪════════╡
│ 1       ┆ A       ┆ 10       ┆ North  │
│ 3       ┆ A       ┆ 8        ┆ North  │
└─────────┴─────────┴──────────┴────────┘,
 ('South',): shape: (2, 4)
┌─────────┬─────────┬──────────┬────────┐
│ OrderID ┆ Product ┆ Quantity ┆ Region │
│ ---     ┆ ---     ┆ ---      ┆ ---    │
│ i64     ┆ str     ┆ i64      ┆ str    │
```

```
┝━━━━━━━━━┿━━━━━━━━━┿━━━━━━━━━━┿━━━━━━━━┥
│ 2       ┆ B       ┆ 5        ┆ South  │
│ 5       ┆ B       ┆ 3        ┆ South  │
└─────────┴─────────┴──────────┴────────┘,
('West',): shape: (2, 4)
┌─────────┬─────────┬──────────┬────────┐
│ OrderID ┆ Product ┆ Quantity ┆ Region │
│ ---     ┆ ---     ┆ ---      ┆ ---    │
│ i64     ┆ str     ┆ i64      ┆ str    │
╞═════════╪═════════╪══════════╪════════╡
│ 4       ┆ C       ┆ 7        ┆ West   │
│ 6       ┆ A       ┆ 12       ┆ West   │
└─────────┴─────────┴──────────┴────────┘}
```

You can then get the DataFrames by accessing the dictionary with the key you want:

```
dfs[("North",)]
```

```
shape: (2, 4)
┌─────────┬─────────┬──────────┬────────┐
│ OrderID ┆ Product ┆ Quantity ┆ Region │
│ ---     ┆ ---     ┆ ---      ┆ ---    │
│ i64     ┆ str     ┆ i64      ┆ str    │
╞═════════╪═════════╪══════════╪════════╡
│ 1       ┆ A       ┆ 10       ┆ North  │
│ 3       ┆ A       ┆ 8        ┆ North  │
└─────────┴─────────┴──────────┴────────┘
```

And that's how you can partition your DataFrame into multiple DataFrames!

Conclusion

In this chapter you learned how to reshape your data.

- We discussed the wide and long formats of data.
- We showed you how to pivot your data from a long to wide format.
- We showed you how to melt your data from a wide to long format.
- We showed you how to transpose your data, flipping the DataFrames diagonally.
- We showed you how to explode nested values into a long format.
- We showed you how to partition your DataFrame into multiple DataFrames.

Now that you're ready to reshape your data like a pro, you can now prepare your data for visualization, which we'll discuss in the next chapter!

CHAPTER 13

Visualizing Data

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 16th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

The previous chapters have given you all the tools you need to transform raw data into a polished DataFrame. But how do you turn such a DataFrame into something insightful?

One way is through data visualization, and Python provides a plethora of packages for that. Packages include hvPlot for quick visualizations, Matplotlib for low-level plotting, Bokeh for interactive graphs, and Plotnine for leveraging the grammar of graphics in Python.

This is both a blessing and a curse, because it's likely there's a package that fits your needs, while it's challenging to choose the right package. Moreover, each package comes with its own set of features, assumptions, and pitfalls.

Figure 13-1 illustrates Python's elaborate data visualization landscape. In this chapter we're focusing on hvPlot, because that's built into Polars, and we demonstrate a couple of alternative packages.

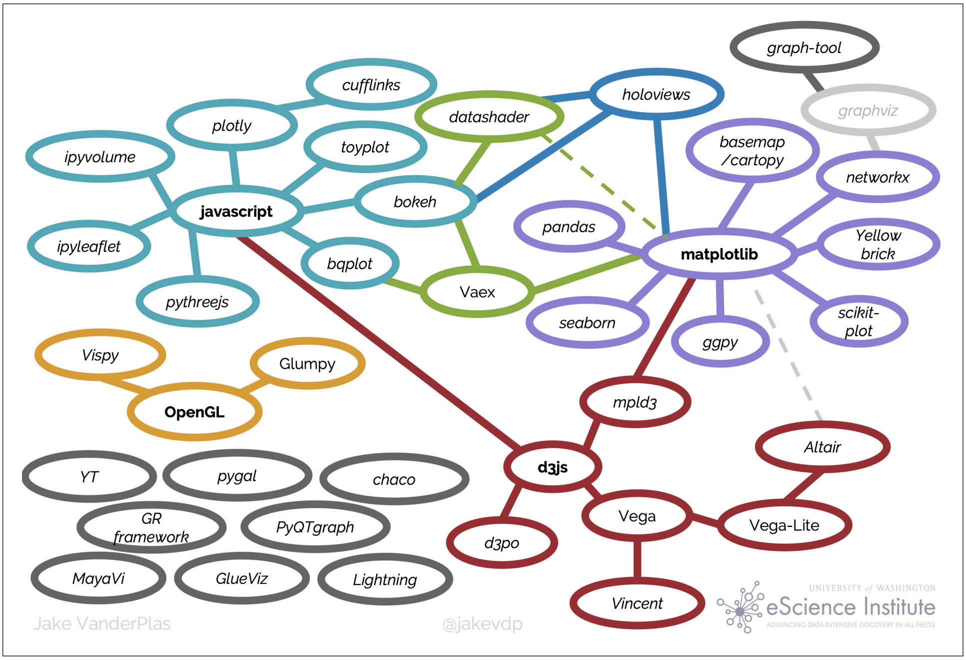


Figure 13-1. Python's data visualization landscape by Jake VanderPlas (reproduced with permission)

Data visualization isn't just about making pretty pictures; it's a fundamental part of data science. By transforming a DataFrame into graphical form, you improve your ability to understand trends, spot outliers, and tell stories that can influence decision making. Effective data visualizations clarify the obscure and simplify the complicated, making your data more accessible.

In this chapter you'll learn how to:

- Quickly create bar charts, scatter plots, density plots, and histograms using the built-in plotting functionality of Polars
- Compose and layer multiple plots
- Customize plots
- Create interactive visualizations
- Plot millions of points on a map
- Use alternative visualization packages such as Plotnine
- Create beautiful tables with nanoplots using the Great Tables package

By the end of this chapter, you'll have a good idea of what each package has to offer, when to use which, and how to use them in combination with Polars. But first, we need to talk about the data set that we will be using.

NYC Bike Trips

Throughout the chapter we're going to use the same data, namely bike trips made with Citi Bike rental bikes in New York City. Here's what the DataFrame looks like.[1]

```python
import polars as pl

trips = pl.read_parquet("data/biketrips/*.parquet")

print(trips[:,:4])
print(trips[:,4:7])
print(trips[:,7:11])
print(trips[:,11:])
```

```
shape: (2_735_398, 4)
┌───────────┬────────────┬─────────────────────┬─────────────────────┐
│ bike_type ┆ rider_type ┆ datetime_start      ┆ datetime_end        │
│ ---       ┆ ---        ┆ ---                 ┆ ---                 │
│ cat       ┆ cat        ┆ datetime[μs]        ┆ datetime[μs]        │
╞═══════════╪════════════╪═════════════════════╪═════════════════════╡
│ electric  ┆ member     ┆ 2024-03-01 00:00:02 ┆ 2024-03-01 00:27:39 │
│ electric  ┆ member     ┆ 2024-03-01 00:00:04 ┆ 2024-03-01 00:09:29 │
│ …         ┆ …          ┆ …                   ┆ …                   │
│ electric  ┆ casual     ┆ 2024-03-31 23:59:57 ┆ 2024-04-01 00:15:39 │
│ classic   ┆ casual     ┆ 2024-03-31 23:59:58 ┆ 2024-04-01 00:01:38 │
└───────────┴────────────┴─────────────────────┴─────────────────────┘
shape: (2_735_398, 3)
┌──────────────┬────────────────────────────────┬─────────────────────────┐
│ duration     ┆ station_start                  ┆ station_end             │
│ ---          ┆ ---                            ┆ ---                     │
│ duration[μs] ┆ str                            ┆ str                     │
╞══════════════╪════════════════════════════════╪═════════════════════════╡
│ 27m 37s      ┆ W 30 St & 8 Ave                ┆ Maiden Ln & Pearl St    │
│ 9m 25s       ┆ Longwood Ave & Southern Blvd   ┆ Lincoln Ave & E 138 St  │
│ …            ┆ …                              ┆ …                       │
│ 15m 42s      ┆ Hart St & Wyckoff Ave          ┆ Monroe St & Bedford Ave │
│ 1m 40s       ┆ 5 Ave & E 30 St                ┆ 5 Ave & E 30 St         │
└──────────────┴────────────────────────────────┴─────────────────────────┘
shape: (2_735_398, 4)
┌────────────────────┬──────────────────┬───────────────┬─────────────┐
│ neighborhood_start ┆ neighborhood_end ┆ borough_start ┆ borough_end │
│ ---                ┆ ---              ┆ ---           ┆ ---         │
│ str                ┆ str              ┆ str           ┆ str         │
```

1 We're printing it in parts because it has 16 columns, which is too wide. If you open up the notebook of this chapter from the our public repository, then you'll be able to see the DataFrame in full in you execute `trips`.

```
╞═════════════════════╪═════════════════════╪═══════════╪═══════════╡
│ Chelsea             │ Financial District  │ Manhattan │ Manhattan │
│ Longwood            │ Mott Haven          │ Bronx     │ Bronx     │
│ …                   │ …                   │ …         │ …         │
│ Bushwick            │ Bedford-Stuyvesant  │ Brooklyn  │ Brooklyn  │
│ Midtown             │ Midtown             │ Manhattan │ Manhattan │
└─────────────────────┴─────────────────────┴───────────┴───────────┘
```

```
shape: (2_735_398, 5)
┌───────────┬────────────┬───────────┬────────────┬──────────┐
│ lat_start ┆ lon_start  ┆ lat_end   ┆ lon_end    ┆ distance │
│ ---       ┆ ---        ┆ ---       ┆ ---        ┆ ---      │
│ f64       ┆ f64        ┆ f64       ┆ f64        ┆ f64      │
╞═══════════╪════════════╪═══════════╪════════════╪══════════╡
│ 40.749614 ┆ -73.995071 ┆ 40.707065 ┆ -74.007319 ┆ 4.83703  │
│ 40.816459 ┆ -73.896576 ┆ 40.810893 ┆ -73.927311 ┆ 2.665806 │
│ …         ┆ …          ┆ …         ┆ …          ┆ …        │
│ 40.704865 ┆ -73.919904 ┆ 40.685144 ┆ -73.953809 ┆ 3.606664 │
│ 40.745985 ┆ -73.986295 ┆ 40.745985 ┆ -73.986295 ┆ 0.0      │
└───────────┴────────────┴───────────┴────────────┴──────────┘
```

In March 2024, over 2.7 million Citi Bike rides were made across four boroughs of New York City: the Bronx, Brooklyn, Manhattan, and Queens. (Staten Island, the fifth borough, doesn't have any Citi Bike stations.) Each borough has many neighborhoods.

The `bike_type` is either "electric" or "classic" and the `rider_type` is either "member" or "casual". The `duration` is the difference between `datetime_start` and `datetime_end`.

The four columns `lat_start`, `lon_start`, `lat_end`, and `lon_end` are the start and end GPS coordinates, respectively. The `distance` is in kilometers as the crow flies between these two coordinates, not the actual distance traveled.

The only data cleaning that we'll do at this point is to remove all bike rides that started and ended at the same bike station and had a duration of less than five minutes, as those are not actually trips:

```
trips = trips.filter(
    ~((pl.col("station_start") == pl.col("station_end")) &
    (pl.col("duration").dt.total_seconds() < 5*60))
)
trips.height
```

```
2672594
```

So, as an example, the last trip shown above, which started and ended at "5 Ave & E 30 St" and lasted 1 minute and 40 seconds, will be removed.

Once we've done that, the DataFrame `trips` still has nearly 2.7 million rows and a variety of columns, including timestamps, categories, names, and coordinates. This will allow us to produce plenty of interesting data visualizations.

Alright, let's figure out when people ride, how far they went, and which stations are most popular by making some data visualizations.

Built-in Plotting with hvPlot

The quickest way to turn a DataFrame into a data visualization is to use the built-in methods that Polars provides. These methods are available through the `df.plot` namespace, for example: `df.plot.scatter()` and `df.plot.bar()`. Under the hood, these methods are being forwarded to another package called hvPlot.

hvPlot is unlike most data visualization packages in that it doesn't do any visualization by itself. Instead, it offers a unified interface to several other data visualization packages, without locking the user in. Figure 13-2 shows an overview of hvPlot's architecture. Let's go over this architecture step by step.

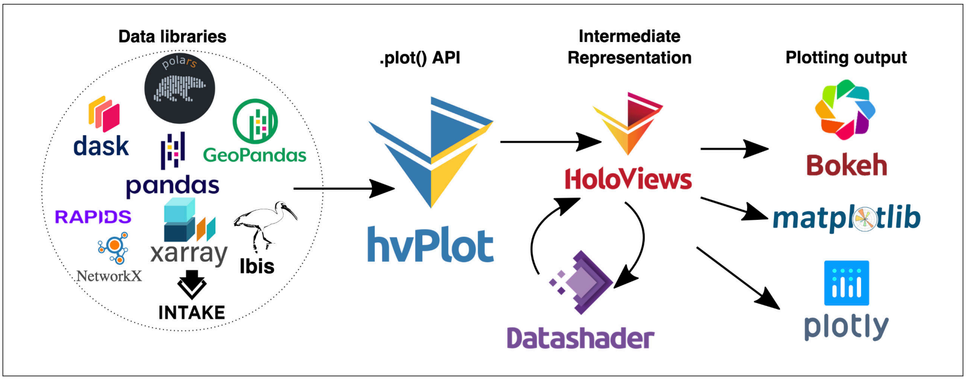


Figure 13-2. hvPlot offers a unified interface to Bokeh, Matplotlib, and Plotly

First, hvPlot's plotting methods accept Polars, Pandas, Ibis DataFrames, and many other data structures of the PyData ecosystem. Second, hvPlot constructs an intermediate, package-agnostic representation of the visualization using the HoloViews package. Think of this representation as a description of how to create a plot.

It optionally uses the Datashader package when needed, for example to plot millions of points on a map, which we'll do later. Third, hvPlot translates the intermediate representation into a specification for either Bokeh, Matplotlib, or Plotly At this point, the user has the opportunity to customize the plot using the package-specific syntax. Finally, the output package renders the plot, meaning it turns the raw data into pixels.

A First Plot

Let's start with a scatter plot, which is a good way to visualize the relationship between two continuous values. Because our `trips` DataFrame is rather large, we'll keep only the trips that started at station "W 21 St & 6 Ave", which happens to be the busiest one of all:

```
trips_speed = (
    trips
    .filter(pl.col("station_start") == "W 21 St & 6 Ave")
    .select(  ❶
        pl.col("distance"),
        pl.col("duration").dt.total_seconds() / 3600,  ❷
        pl.col("bike_type")
    )
)
trips_speed
```

```
shape: (10_981, 3)
┌──────────┬──────────┬───────────┐
│ distance ┆ duration ┆ bike_type │
│ ---      ┆ ---      ┆ ---       │
│ f64      ┆ f64      ┆ cat       │
╞══════════╪══════════╪═══════════╡
│ 0.452909 ┆ 0.026944 ┆ electric  │
│ 0.993271 ┆ 0.089444 ┆ electric  │
│ …        ┆ …        ┆ …         │
│ 0.992056 ┆ 0.059167 ┆ electric  │
│ 3.690942 ┆ 0.326389 ┆ electric  │
└──────────┴──────────┴───────────┘
```

❶ We are keeping only the columns that are needed for the visualization. This is not necessary, but it helps us show you what data we're using.

❷ The unit of the `duration` column is hours. Alternatively, there's the `Expr.dt.total_hours()` method, but this only returns whole hours.

The snippet below constructs the scatter plot using the method `df.plot.scatter`:

```
trips_speed.plot.scatter(x="distance", y="duration", color="bike_type",  ❶
                         xlabel="distance (km)", ylabel="duration (h)",  ❷
                         ylim=(0, 2))  ❸
```

❶ These three arguments are the most important, as they determine which columns are used for the position and color of each point.

❷ Adding or changing labels isn't necessary, but can clarify what the axes represent.

❸ We manually limit the range of the y-axis, because there are some much longer trips that would impact the visualization, making it more difficult to see smaller values. You can also fix these kinds of issues by applying a filter to the DataFrame.

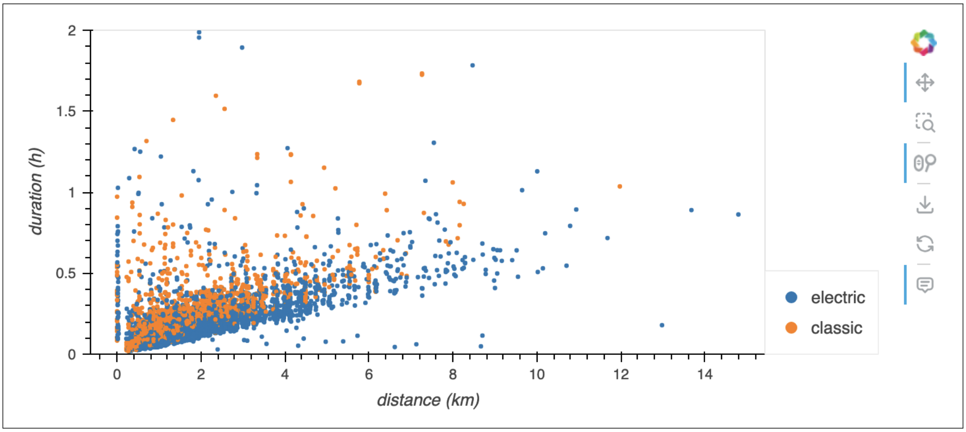


Figure 13-3. The relationship between distance and duration per bike type

Figure 13-3 shows that, generally speaking, electric bikes are faster and travel a greater distance than classic ones.

Methods in the Plot Namespace

The method `df.plot.scatter()` is just one of the many methods available in the `df.plot` namespace. To see which methods are available, you can use tab completion (that is, press the TAB key after `df.plot.`):

```
trips.plot.<TAB>
```

Available methods include:

`df.plot.area()`
: Plots a area chart similar to a line chart except for filling the area under the curve and optionally stacking.

`df.plot.bar()`
: Plots a bar chart that can be stacked or grouped.

`df.plot.bivariate()`
: Plots 2D density of a set of points.

`df.plot.box()`
: Plots a box-whisker chart comparing the distribution of one or more variables.

`df.plot.density()`
: Plots the kernel density estimate of one or more variables.

`df.plot.heatmap()`
: Plots a heatmap to visualizing a variable across two independent dimensions.

`df.plot.hexbins()`
: Plots hex bins.

`df.plot.hist()`
: Plots the distribution of one or histograms as a set of bins.

`df.plot.line()`
: Plots a line chart (such as for a time series).

`df.plot.scatter()`
: Plots a scatter chart comparing two variables.

`df.plot.violin()`
: Plots a violin plot comparing the distribution of one or more variables using the kernel density estimate.

Getting Help for a Method

Normally, to get help for a certain method, you'd type `help(<method>)` or `?<method>`. We don't recommend this for the methods in the `df.plot` namespace. The documentation for the method `df.plot.scatter()`, for example, consists of 334 lines of text, which is overwhelming. It contains a lot of information and arguments that are most likely not always relevant.

Luckily, hvPlot offers its own `help()` function that allows you to disable certain portions:

```
import hvplot
hvplot.help('scatter', generic=False, style=False)

The `scatter` plot visualizes your points as markers in 2D space. You can visua…
one more dimension by using colors.

The `scatter` plot is a good first way to plot data with non continuous axes.

Reference: https://hvplot.holoviz.org/reference/tabular/scatter.html

Parameters
----------
x : string, optional
    Field name(s) to draw x-positions from. If not specified, the index is
    used. Can refer to continuous and categorical data.
y : string or list, optional
    Field name(s) to draw y-positions from. If not specified, all numerical
    fields are used.
marker : string, optional
    The marker shape specified above can be any supported by matplotlib, e.g. s…
    See https://matplotlib.org/stable/api/markers_api.html.
c : string, optional
… with 83 more lines
```

hvPlots' website provides great documentation as well, including many examples. Keep in mind that many pages and examples are based on Pandas. This is understandable because Pandas is ten years older than Polars. Some examples assume that your DataFrame has an Index, or even a MultiIndex, which Polars DataFrames do not. The next section offers advice for when this happens.

Pandas as Backup

Under the hood, hvPlot first converts the Polars DataFrame to a Pandas DataFrame. It only copies the columns that are needed for the plot. This works well most of the time, but not always.

Here's an example where we want to create a heatmap. hvPlot's documentation mentions the special `.hour` and `.day` modifiers to extract the hour and day of a DateTime, respectively. Unfortunately, this is not (yet) supported for Polars DataFrames, so the following yields an error:

```
trips_per_day_hour = (
    trips
    .sort("datetime_start")
    .group_by_dynamic("datetime_start", every="1h")
    .agg(pl.len())
)

trips_per_day_hour.plot.heatmap(x='datetime_start.hour',
                                y='datetime_start.day',
                                C='len', cmap='reds')
```

```
ValueError: 'datetime_start.hour' is not in list
```

We get an error because hvPlot attempts to copy the column `datetime_start.day`, which doesn't exist in our DataFrame. Fret not; we can always fall back to Pandas by using the `df.to_pandas()` method:

```
import hvplot.pandas
trips_per_day_hour.to_pandas().hvplot.heatmap(x='datetime_start.hour',
                                              y='datetime_start.day',
                                              C='len', cmap='reds')
```

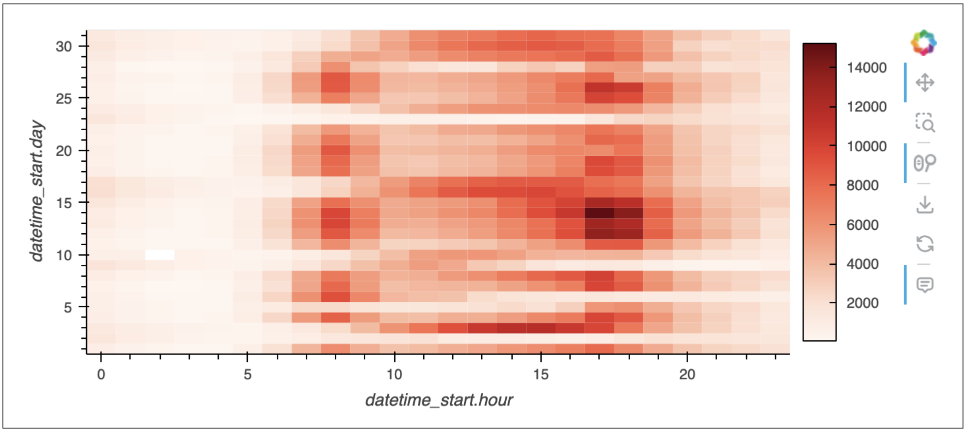


Figure 13-4. Pandas is a good backup in situations where Polars is not yet supported

Manual Transformations

Let's try another plot, the bar chart. The bar chart is particularly useful for showing counts for different groups. hvPlot expects the DataFrame you provide to contain the actual values to be used; it doesn't do any transformation for you.

We're interested in the number of trips per bike type and rider type. Because these counts are not literally in our DataFrame, we need to calculate them manually:

```
trips_type_counts = trips.group_by("rider_type", "bike_type").len()
trips_type_counts
```

```
shape: (4, 3)
┌────────────┬───────────┬─────────┐
│ rider_type ┆ bike_type ┆ len     │
│ ---        ┆ ---       ┆ ---     │
│ cat        ┆ cat       ┆ u32     │
╞════════════╪═══════════╪═════════╡
│ casual     ┆ electric  ┆ 295530  │
│ member     ┆ electric  ┆ 1420012 │
│ casual     ┆ classic   ┆ 120888  │
│ member     ┆ classic   ┆ 836164  │
└────────────┴───────────┴─────────┘
```

Once we have that, we can create a stacked bar chart as follows:

```
trips_type_counts.plot.bar(x="rider_type", y="len", by="bike_type",
                           ylabel="count", stacked=True,
                           color=["orange", "green"])
```

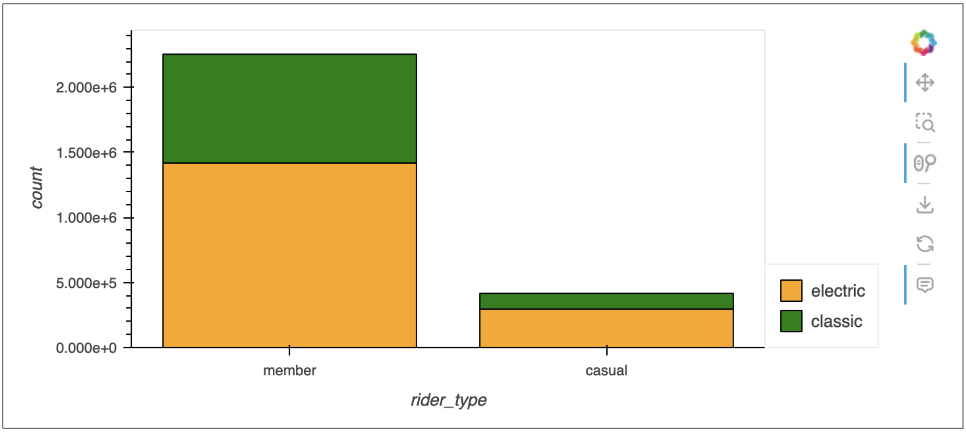


Figure 13-5. A stacked bar chart showing the number of trips per bike type and rider type

Figure 13-5 shows that most bike trips are performed by Citi Bike members and that the majority use an electric bike.

Changing the Plotting Backend

The default plotting backend in hvPlot is Bokeh. In most cases this suffices, but there are situations where changing the backend to Plotly or Matplotlib is useful. For example, Matplotlib is useful when there's no need for an interactive visualization. Or maybe you need to match the style of other visualizations in a report.

The backend can be changed by using the `hvplot.extension()` method and passing either `"matplotlib"` or `"plotly"`. For this, you first need to explicitly import the `hvplot` package:

```
import hvplot
hvplot.extension("matplotlib")
```

Let's create the same bar chart as in the previous section, but now with Matplotlib as the backend:

```
trips_type_counts.plot.bar(x="rider_type", y="len", by="bike_type",
                           ylabel="count", stacked=True,
                           color=["orange", "green"])
```

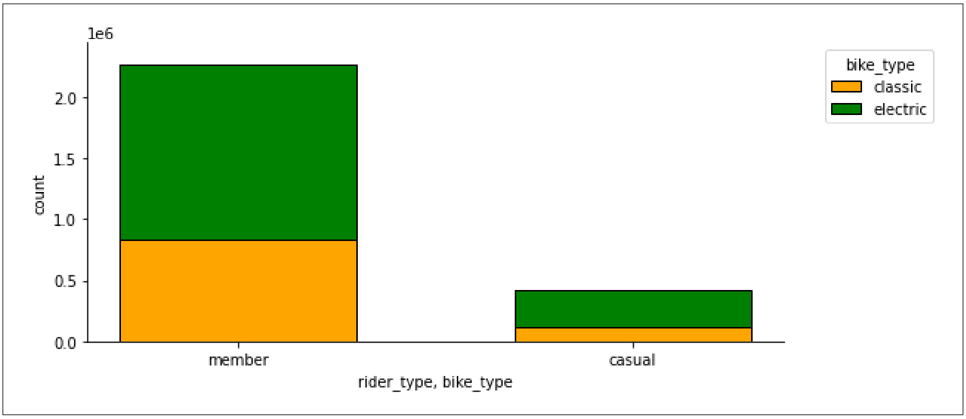


Figure 13-6. The same bar chart as before, but now created by the Matplotlib backend

Figure 13-6 shows that, apart from some minor visual differences, the same unchanged code can produce a Matplotlib image.

Let's reset the plotting backend to Bokeh for the purpose of this chapter:

```
hvplot.extension("bokeh")
```

Plotting Points on a Map

We haven't really used the coordinates in our DataFrame. Let's change that by creating a map. You might be tempted to create a scatter plot, just as before:

```
trips.plot.scatter(x='lon_start', y='lat_start', color='borough_start',
                   width=600, height=600)
```

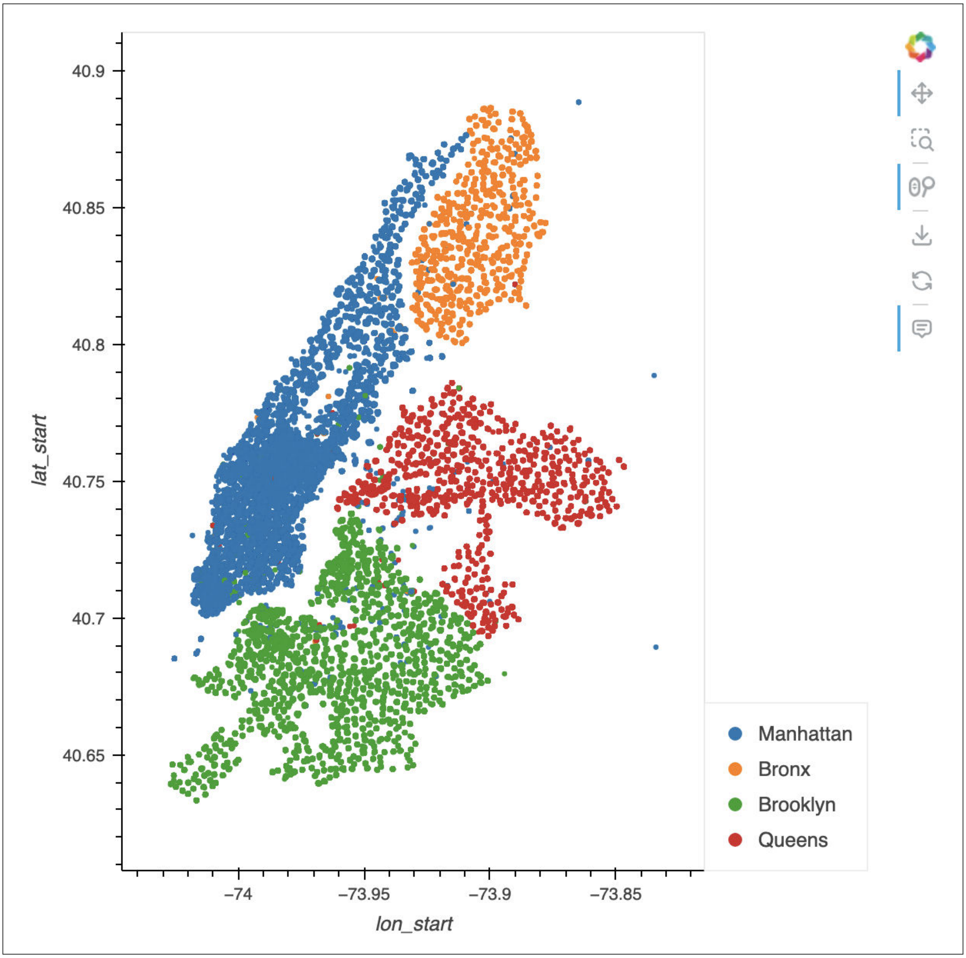


Figure 13-7. A plain scatter plot is not well suited for geographical data.

However, this is not a very good plot. It lacks context, the coordinates are not properly projected, and because there's way too much data, it can take up to a minute to generate.

But, data visualization is meant to be an iterative process. A better way to visualize coordinates is to use the `df.plot.points()` method, with the `geo` argument set to `True`. This ensures that the coordinates are properly projected onto a proper map:

```
trips.plot.points(x="lon_start", y="lat_start",
                  datashade=True, geo=True,
                  tiles="CartoLight",
                  width=800, height=600)
```

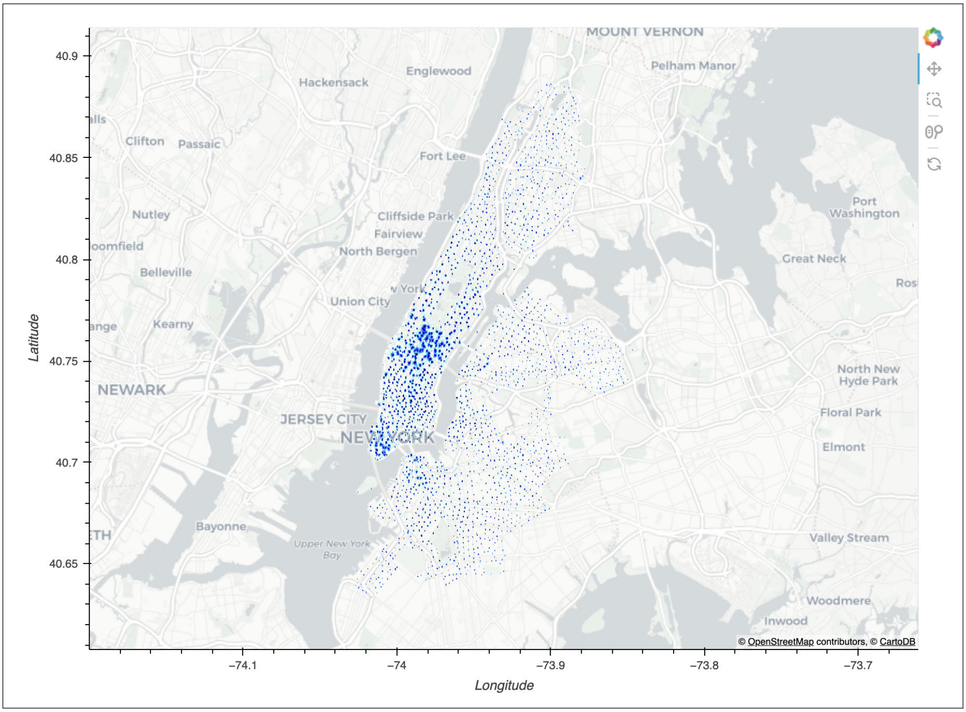


Figure 13-8. An interactive geographical plot

It's difficult to convey in a book, but the map shown statically in Figure 13-8 is actually an interactive visualization. You can pan and zoom. Because we've set the `datashade` argument to `True`, only the necessary data is used, maximizing efficiency.

Composing Plots

Sometimes a single plot is not sufficient. Using hvPlot, you can compose multiple plots into one. There are two types of composition: stacking and layering.

First, we'll prepare some data that allows us to draw a line plot:

```
trips_hour_num_speed = (
    trips
    .sort("datetime_start")
    .group_by_dynamic("datetime_start", every="1h")
    .agg(num_trips=pl.len(),
         speed=(pl.col("distance") / (pl.col("duration").dt.total_seconds() / 3600)).median())
    .filter(pl.col("datetime_start") > pl.date(2024, 3 , 26))
)

trips_hour_num_speed
```

```
shape: (143, 3)
┌─────────────────────┬───────────┬───────────┐
│ datetime_start      ┆ num_trips ┆ speed     │
│ ---                 ┆ ---       ┆ ---       │
│ datetime[μs]        ┆ u32       ┆ f64       │
╞═════════════════════╪═══════════╪═══════════╡
│ 2024-03-26 01:00:00 ┆ 298       ┆ 13.797211 │
│ 2024-03-26 02:00:00 ┆ 182       ┆ 14.312177 │
│ 2024-03-26 03:00:00 ┆ 124       ┆ 12.851984 │
│ 2024-03-26 04:00:00 ┆ 235       ┆ 13.787607 │
│ 2024-03-26 05:00:00 ┆ 888       ┆ 13.920884 │
│ …                   ┆ …         ┆ …         │
│ 2024-03-31 19:00:00 ┆ 5216      ┆ 10.696787 │
│ 2024-03-31 20:00:00 ┆ 3687      ┆ 11.058714 │
│ 2024-03-31 21:00:00 ┆ 2878      ┆ 11.445669 │
│ 2024-03-31 22:00:00 ┆ 2354      ┆ 11.422508 │
│ 2024-03-31 23:00:00 ┆ 1603      ┆ 11.884897 │
└─────────────────────┴───────────┴───────────┘
```

In the first code snippet, we combine two plots using the + operator, which places two plots next to each other:

```
(
    trips_hour_num_speed.plot.line(x="datetime_start", y="num_trips") +
    trips_hour_num_speed.plot.line(x="datetime_start", y="speed")
).cols(1)  ❶
```

❶ With the `.cols()` method, we ensure that the two plots are placed beneath each other.

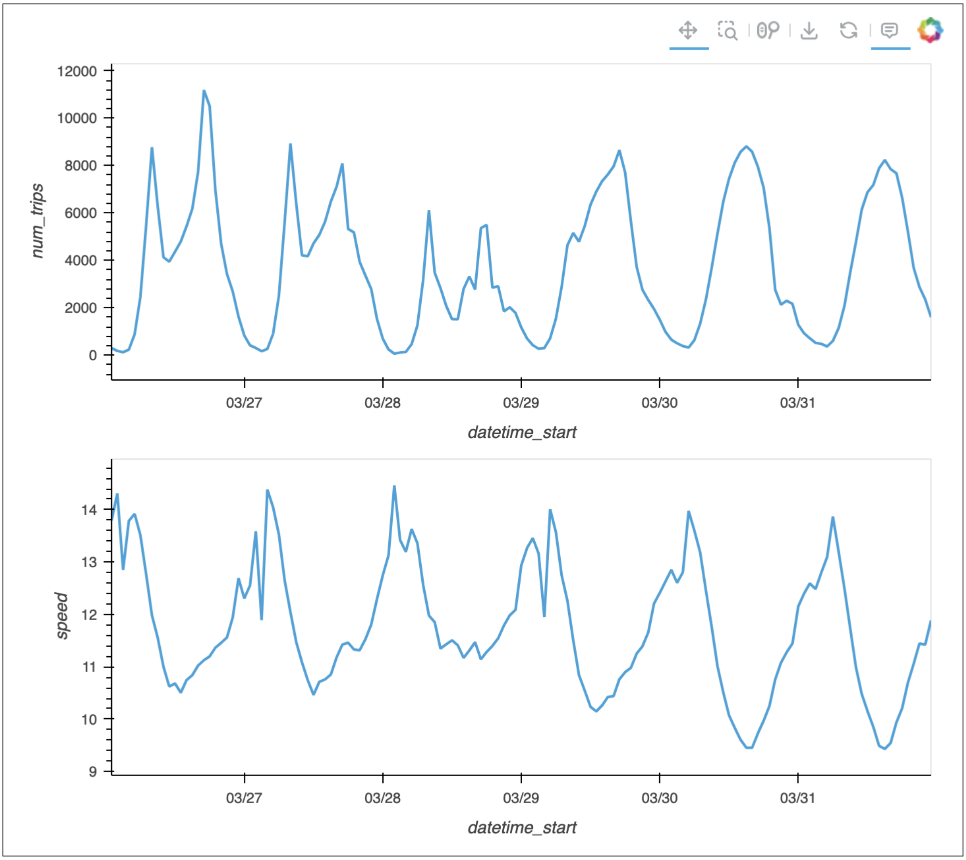


Figure 13-9. Two plots can be placed next to each other

In the second code snippet, we combine two plots:

```
(
    trips_hour_num_speed.plot.line(x="datetime_start", y="num_trips") *
    trips_hour_num_speed
        .filter(pl.col("num_trips") > 9000)
        .plot.scatter(x="datetime_start", y="num_trips", c="red", s=50)
)
```

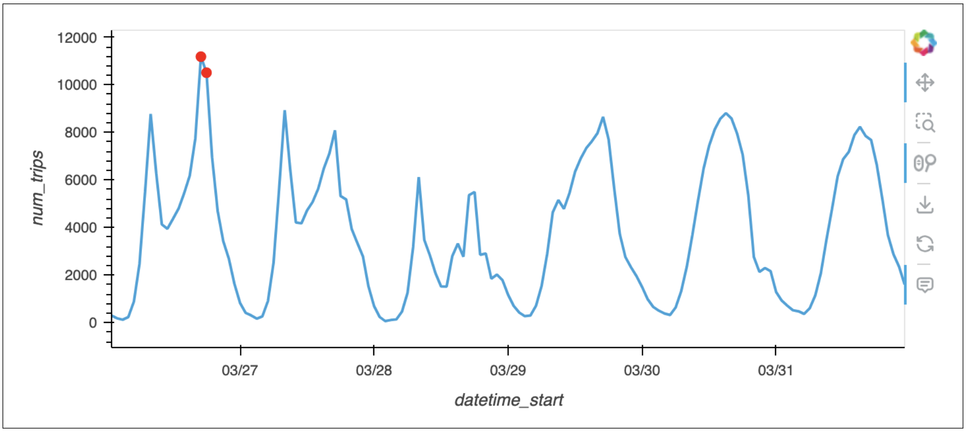


Figure 13-10. Two plots can be placed on top of each other

Notice that we're using two different DataFrames (the second is a subset of the first) and two different plot types (a line plot and a scatter plot).

Adding Interactive Widgets

The Bokeh backend already offers interactivity: you can zoom in and out, pan to move to different parts of the plot, and hover over elements to get more information.

Using the `groupby` keyword argument, you can add one or more widgets. The column name or names passed to this argument slices the data into multiple subsets. With the widgets you can select which subset of the data is used for the plot.

Here's an example where we group by date. The widget type is based on the type of the column. As you can see in Figure 13-11, the widget for selecting the date is a slider.

```
trips_per_hour = (
    trips
    .sort("datetime_start")
    .group_by_dynamic("datetime_start", group_by="borough_start", every="1h")
    .agg(pl.len())
    .with_columns(date=pl.col("datetime_start").dt.date())
)
trips_per_hour
```

```
shape: (2_972, 4)
┌───────────────┬─────────────────────┬─────┬────────────┐
│ borough_start ┆ datetime_start      ┆ len ┆ date       │
│ ---           ┆ ---                 ┆ --- ┆ ---        │
│ str           ┆ datetime[μs]        ┆ u32 ┆ date       │
╞═══════════════╪═════════════════════╪═════╪════════════╡
│ Manhattan     ┆ 2024-03-01 00:00:00 ┆ 480 ┆ 2024-03-01 │
│ Manhattan     ┆ 2024-03-01 01:00:00 ┆ 294 ┆ 2024-03-01 │
```

```
│ Manhattan     │ 2024-03-01 02:00:00 │ 187 │ 2024-03-01 │
│ Manhattan     │ 2024-03-01 03:00:00 │ 100 │ 2024-03-01 │
│ Manhattan     │ 2024-03-01 04:00:00 │ 126 │ 2024-03-01 │
│ …             │ …                   │ …   │ …          │
│ Queens        │ 2024-03-31 19:00:00 │ 366 │ 2024-03-31 │
│ Queens        │ 2024-03-31 20:00:00 │ 336 │ 2024-03-31 │
│ Queens        │ 2024-03-31 21:00:00 │ 211 │ 2024-03-31 │
│ Queens        │ 2024-03-31 22:00:00 │ 176 │ 2024-03-31 │
│ Queens        │ 2024-03-31 23:00:00 │ 144 │ 2024-03-31 │
└───────────────┴─────────────────────┴─────┴────────────┘
```

```
trips_per_hour.plot.line(x="datetime_start", by="borough_start",
                         groupby="date", widget_location='left_top')
```

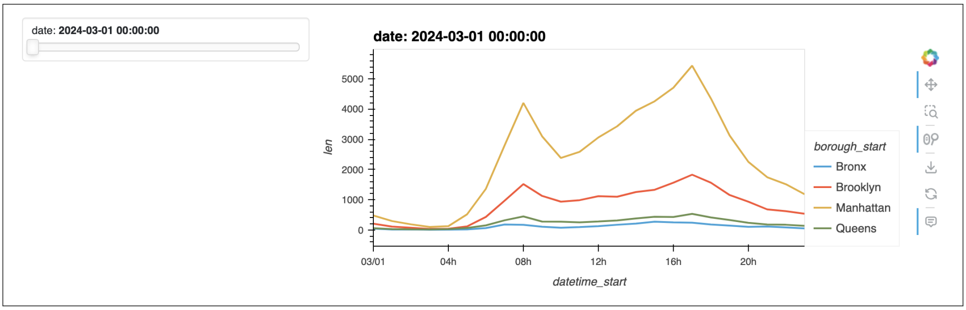


Figure 13-11. Interactive widgets are easily added with the groupby keyword argument

You can also pass a list of column names to the `groupby` keyword argument, to create multiple widgets.

Common Customizations

hvPlot offers many ways to customize a plot through additional keyword arguments. Those keyword arguments are common across all plot types. We've already used a couple, such as `ylim`, `width`, and `height`.

The keyword arguments we use most often to improve the readability of a plot are listed in Table 13-1

Table 13-1. Common arguments for hvPlot

Argument	Description
`cmap`	Sets the colormap. Default None. Common ones are `Category10`, `viridis`, and `fire`. [a]
`fontscale`	Scales the size of all fonts by the same amount. For example, `fontscale=1.5` enlarges all fonts (title, xticks, labels etc.) by 50%. Default value is `1`.
`grid`	Whether to show a grid. Default value is `False`.
`logx`, `logy`	Enables logarithmic x- and y-axis respectively. Default value for both is `False`.
`rot`	Rotates the axis ticks along the x-axis by the specified number of degrees. Default value is `0`.

Argument	Description
`title`	Title for the plot. Default is `""`.
`width`, `height`	The width and height of the plot in pixels. Default values are `700` and `300`, respectively.
`xlabel`, `ylabel`, `clabel`	Axis labels for the x-axis, y-axis, and colorbar, respectively. Default value is None, in which case the column name is used as the label.
`xlim`, `ylim`	Plot limits of the x- and y-axis, respectively. Default value is None. Use either a tuple or list of two numerical values.

[a] See the Holoviews User Guide about Colormaps for more information.

Here's an example that demonstrates these keyword arguments. Because of all the customizations, Figure 13-12 is arguably the ugliest plot in the book. It also includes a couple of faux-pas: redundant use of color, y-axis not starting at zero, and a nonsensical logarithmic scale. But that's the price we have to pay to demonstrate how to customize a plot. (There is indeed such a thing such as too much customization.) Let's try it:

```
busiest_stations = (
    trips
    .group_by(station="station_start").agg(
        num_trips=pl.len(),
    )
    .sort("num_trips", descending=True)
    .head(20)
)

busiest_stations
```

```
shape: (20, 2)
┌───────────────────────┬───────────┐
│ station               ┆ num_trips │
│ ---                   ┆ ---       │
│ str                   ┆ u32       │
╞═══════════════════════╪═══════════╡
│ W 21 St & 6 Ave       ┆ 10981     │
│ Forsyth St & Broome … ┆ 9988      │
│ Broadway & W 58 St    ┆ 9771      │
│ 8 Ave & W 31 St       ┆ 8977      │
│ Delancey St & Eldrid… ┆ 8947      │
│ …                     ┆ …         │
│ Ave A & E 14 St       ┆ 7194      │
│ West St & Chambers S… ┆ 6709      │
│ W 30 St & 10 Ave      ┆ 6505      │
│ Amsterdam Ave & W 73… ┆ 6484      │
│ W 43 St & 10 Ave      ┆ 6451      │
└───────────────────────┴───────────┘
```

```
fig = busiest_stations.plot.bar(x="station", y="num_trips", color="num_trips",
                                cmap="viridis",
                                fontscale=1.2,
                                grid=True,
```

```
                  logx=False, logy=True,
                  rot=45,
                  title="Busiest Citi Bike Stations",
                  width=800, height=400,
                  xlabel="", ylabel="Number of trips",
                  xlim=None, ylim=(4000, None))

fig
```

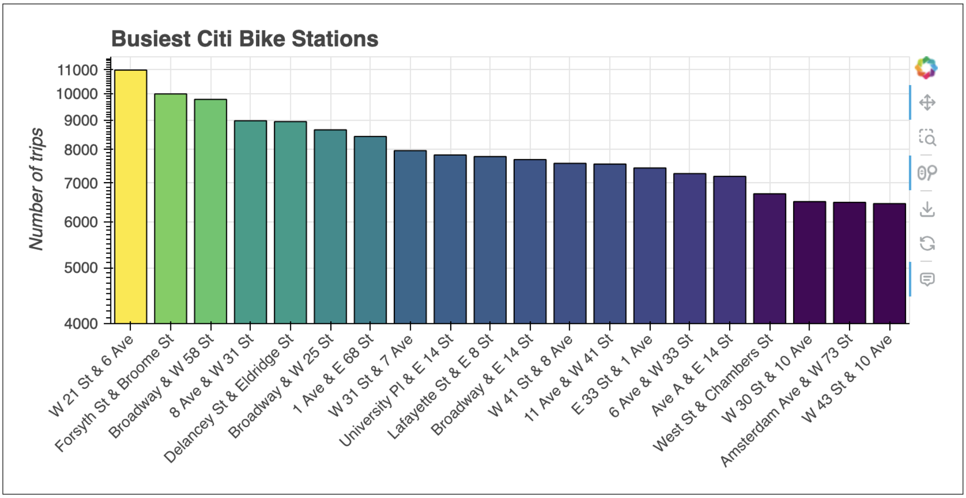


Figure 13-12. There is a thing such a too much customization

See `hv.plot.help(kind="...")` for all available keyword arguments. Some are for common customizations and some belong to a particular kind of plot.

The HoloViews User Guide offers information on how to customize the underlying HoloViews representation using the `.opts()` method. Within that method, using the `hooks` argument, it's possible to specify further customizations that are handled by the backend (Bokeh, Matplotlib, and Plotly). To expand on the above figure (called `fig`), here's a code snippet that uses both of these customization approaches, producing Figure 13-13:

```
def bokeh_hook(plot, element):
    plot.handles["yaxis"].major_label_text_color = "blue"
    plot.handles["plot"].title.align = "right"

fig.opts(invert_axes=True, invert_yaxis=True, hooks=[bokeh_hook])
```

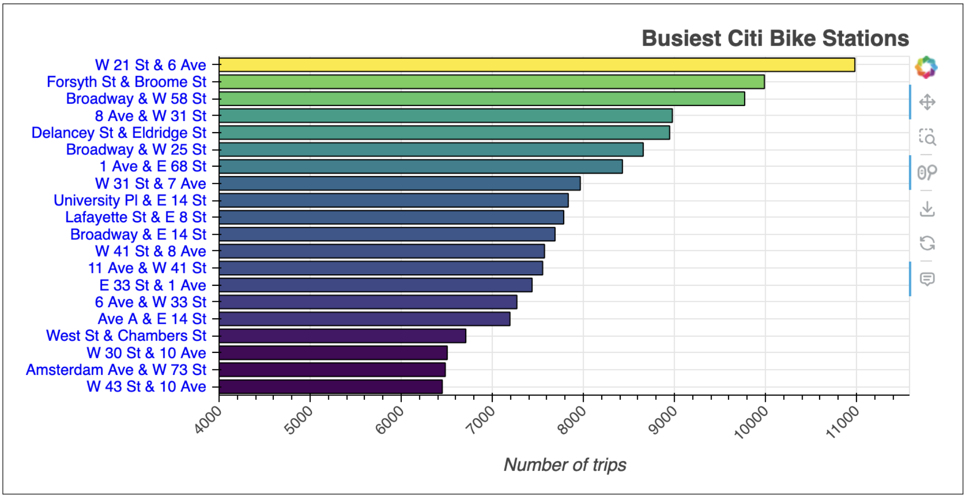


Figure 13-13. And just when you thought you couldn't customize it further!

Alternative Packages

hvPlot is not the only package you can use to visualize your Polars DataFrame. In this section we would like to highight two alternatives: Plotnine and Great Tables.

Plotnine

Plotnine is a data visualisation package based on the layered grammar of graphics, created by Hassan Kibirige. Its API is similar to ggplot2, a widely successful R package by Hadley Wickham and others.

The underlying grammar of graphics is accompanied by a consistent API that allows you to quickly and iteratively create different types of beautiful data visualisations while rarely having to consult the documentation.

The Plotnine package can be installed from PyPI with:

```
$ pip install plotnine[all]
```

And imported as follows:

```
from plotnine import *
```

Import All The Things

While it's generally considered to be bad practice to import everything into the global namespace, we think it's fine to do this in an ad-hoc environment such as a notebook, as it makes using Plotnine's many functions more convenient.

If you'd rather not clutter your global namespace, we advise you to use `import plotnine as p9` and prefix every function with `p9`.

We're going to use Plotnine to create scatter plots with a twist. First, we'll add an additional layer—it is based on the *layered* grammar of graphics, after all. Then, we'll turn the scatter plot into a plot with four panels.

The following code snippet prepares a DataFrame to show, once again, the relationship between distance and duration. This time, however, we'll operate on the level of bike stations, using the median distance and duration per station. The question we want to answer here is to what extent distance and duration are correlated. We're only considering bike trips within the same borough:

```
trips_speed = (
    trips.group_by("neighborhood_start", "neighborhood_end").agg(
        pl.col("duration").dt.total_seconds().median() / 3600,
        pl.col("distance").median(),
        pl.col("borough_start").first(),
        pl.col("borough_end").first(),
        pl.len(),
    ).filter(
        (pl.col("len") > 30) &
        (pl.col("distance") > 0.2) &
        (pl.col("neighborhood_start") != pl.col("neighborhood_end")),
    ).with_columns(
        speed=pl.col("distance") / pl.col("duration")
    ).sort("borough_start")
)
trips_speed
```

shape: (2_971, 8)

neighborhood_start --- str	neighborhood_end --- str	duration --- f64	…	borough_end --- str	len --- u32	speed --- f64
Morris Heights	East Morrisania	0.252778	…	Bronx	38	12.14096
Tremont	West Farms	0.080833	…	Bronx	121	12.197868
…	…	…	…	…	…	…
Long Island City	Clinton Hill	0.469861	…	Brooklyn	58	13.09114
Long Island	West Village	0.539444	…	Manhattan	61	10.26295

City						

Here's the Plotnine code needed to create the first scatter plot. Each row (and in Figure 13-14 each point) is a pair of start and end stations within the same borough.

```
(
    ggplot(data=trips_speed
        .filter(pl.col("borough_start") == pl.col("borough_end")),
        mapping=aes(x="distance", y="duration", color="borough_end")) +
    geom_point(size=0.25, alpha=0.5) +
    geom_smooth(method="lowess", size=2, se=False, alpha=0.8) +
    scale_color_brewer(type="qualitative", palette="Set1") +
    labs(title="Trip distance and duration within each borough",
         x="Distance (km)", y="Duration (m)", color="Borough") +
    theme_linedraw() +
    theme(figure_size=(8, 6))
)
```

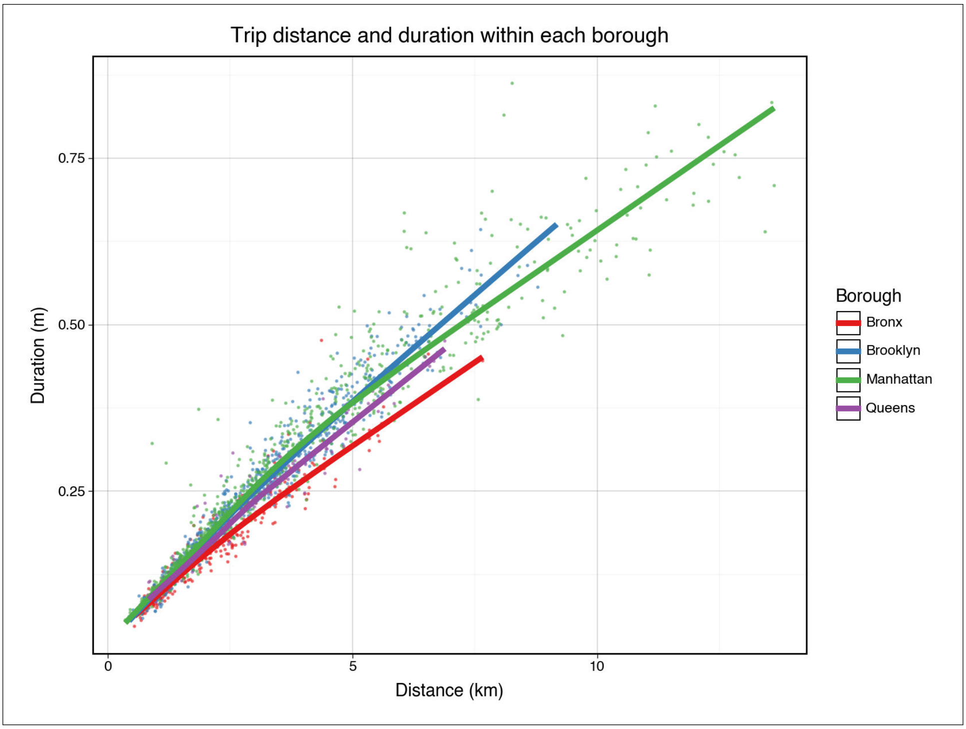


Figure 13-14. Trip distance and duration within each borough

By negating the filter, we can investigate the relationship between trip distance and duration for trips across boroughs. Because we have four different starting boroughs, it makes sense to create four scatter plots. We use the `facet_wrap()` function to create four panels, one for each borough.

```
(
    ggplot(data=trips_speed
        .filter(pl.col("borough_start") != pl.col("borough_end"))
        .with_columns(
            ("From " + pl.col("borough_start")).alias("borough_start")),
        mapping=aes(x="distance", y="duration", color="borough_end")) +
    geom_point(size=0.25, alpha=0.5) +
    geom_smooth(method="lowess", size=2, se=False, alpha=0.8) +
    scale_color_brewer(type="qualitative", palette="Set1") +
    facet_wrap("borough_start") +
    labs(title="Trip distance and duration cross borough",
         x="Distance (km)", y="Duration (m)", color="To Borough") +
    theme_linedraw() +
    theme(figure_size=(8, 6))
)
```

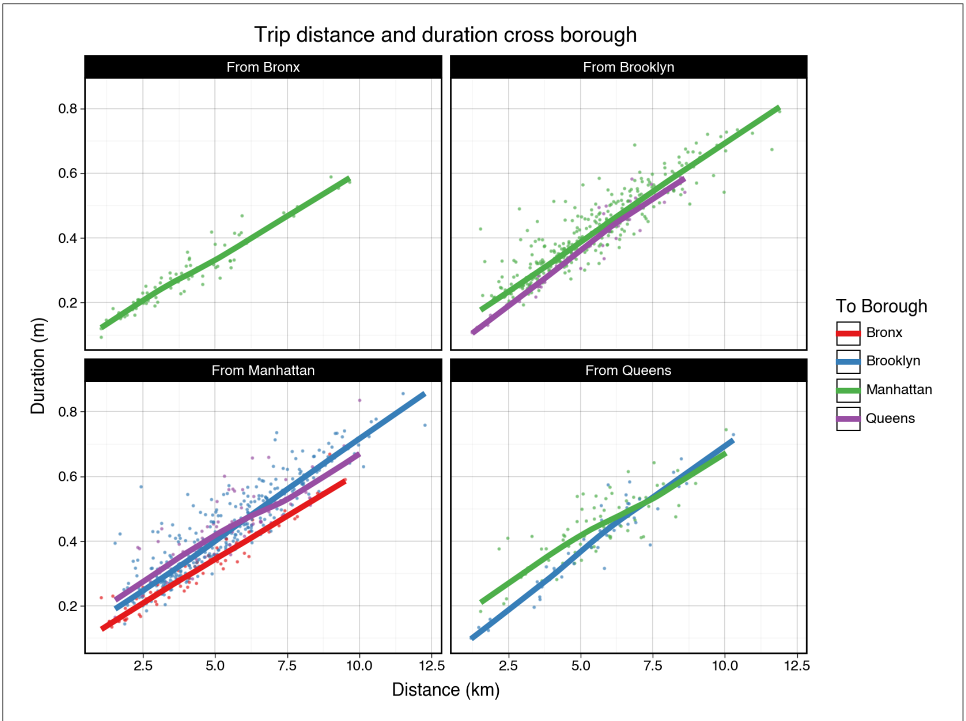


Figure 13-15. Cross-borough trip distance and duration

If you compare the two code snippets above, you'll find that they share a lot of code. We only changed the `data` argument of the `ggplot()` function, added `facet_wrap` in the second snippet to create the four panels, and updated some of the labels. This is possible thanks to Plotnine's composable API. It creates a plot by chaining methods, rather than adding keyword arguments to one method.

For more information about Plotnine, refer to its website or Jeroen's blogpost.

Great Tables

So far, we've gone from a DataFrame all the way to a data visualization, turning raw numbers into colorful pixels. There's a third approach that sits halfway between these two extremes. We're talking about tables. The Great Tables package by Rich Iannone and Michael Chow enables you to create, well, *great* tables.

A table is great when it presents information in a clear and structured way. That may include:

- Readable column names
- Numerical values with proper formatting
- Row grouping
- Styling to draw attention to important values
- Annotations such as titles, labels, and footnotes

Great Tables' underlying philosophy is based on a cohesive set of table components (see Figure 13-16). Starting with a DataFrame as input, you can iteratively chain methods to add elements and apply formatting.

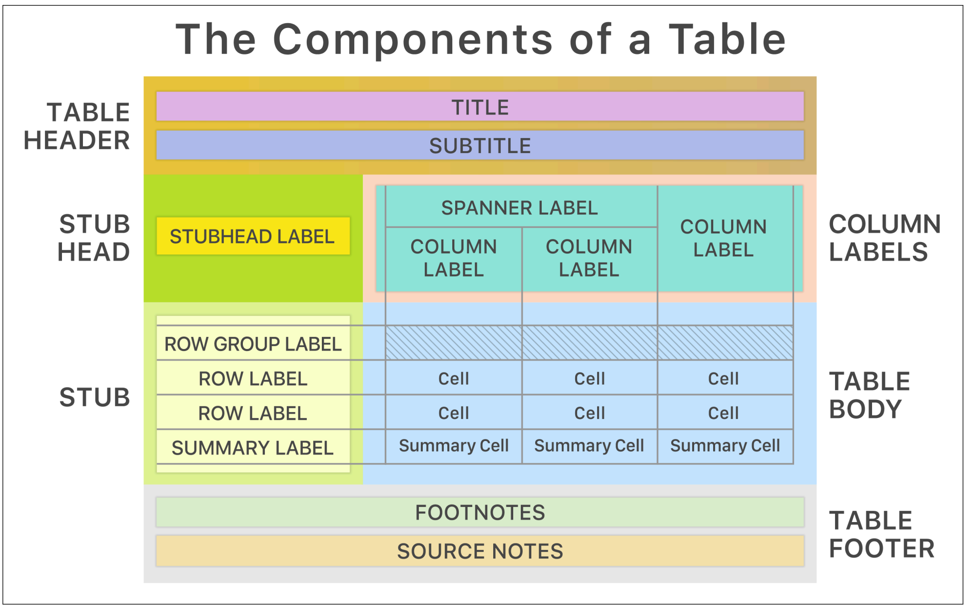


Figure 13-16. Components of a Great Table (reproduced with permission from the Great Tables authors)

The Great Tables package can be installed from PyPI with:

```
$ pip install great_tables
```

Let's prepare a DataFrame for a great-looking table. The code snippet below calculates the three busiest stations per borough:

```
busiest_stations = (
    trips
    .group_by(  ❶
        station=pl.col("station_start"),
        date=pl.col("datetime_start").dt.date()
    )
    .agg(
        borough=pl.col("borough_start").first(),
        neighborhood=pl.col("neighborhood_start").first(),
        num_rides=pl.len(),
        percent_member=(pl.col("rider_type") == "member").mean(),
        percent_electric=(pl.col("bike_type") == "electric").mean()
    )
    .sort("date")
    .group_by("station")
    .agg(
        pl.col(pl.String).first(),
        pl.col(pl.NUMERIC_DTYPES).mean(),
        pl.col("num_rides").alias("rides_per_day")  ❷
    )
    .sort("num_rides", descending=True)
    .group_by("borough", maintain_order=True).head(3)
)
busiest_stations
```

```
shape: (12, 7)
┌───────────┬───────────┬──────────┬──────────┬──────────┬──────────┬──────────┐
│ borough   ┆ station   ┆ neighbor ┆ num_ride ┆ percent_ ┆ percent_ ┆ rides_pe │
│ ---       ┆ ---       ┆ hood     ┆ s        ┆ member   ┆ electric ┆ r_day    │
│ str       ┆ str       ┆ ---      ┆ ---      ┆ ---      ┆ ---      ┆ ---      │
│           ┆           ┆ str      ┆ f64      ┆ f64      ┆ f64      ┆ list[u32 │
│           ┆           ┆          ┆          ┆          ┆          ┆ ]        │
╞═══════════╪═══════════╪══════════╪══════════╪══════════╪══════════╪══════════╡
│ Manhattan ┆ W 21 St & ┆ Chelsea  ┆ 354.2258 ┆ 0.913584 ┆ 0.583709 ┆ [325,    │
│           ┆ 6 Ave     ┆          ┆ 06       ┆          ┆          ┆ 88, …    │
│           ┆           ┆          ┆          ┆          ┆          ┆ 308]     │
│ …         ┆ …         ┆ …        ┆ …        ┆ …        ┆ …        ┆ …        │
│ Bronx     ┆ Plaza Dr  ┆ Mount    ┆ 31.70967 ┆ 0.837427 ┆ 0.948925 ┆ [30, 19, │
│           ┆ & W 170   ┆ Eden     ┆ 7        ┆          ┆          ┆ … 33]    │
│           ┆ St        ┆          ┆          ┆          ┆          ┆          │
└───────────┴───────────┴──────────┴──────────┴──────────┴──────────┴──────────┘
```

❶ This first aggregation is needed because we want to display counts per station per day using nanoplots (we'll get to these later).

❷ The values in this column will make sense once we use them to create a nanoplot.

The following code uses Great Tables to produce a table:

```
import polars.selectors as cs
from great_tables import GT, style, md

(
    GT(busiest_stations, rowname_col="station", groupname_col="borough") ❶
    .cols_label(        ❷
        neighborhood="Neighborhood",
        num_rides="Mean Daily Rides",
        percent_member="Members",
        percent_electric="E-Bikes",
        rides_per_day="Rides Per Day",
    )
    .tab_header(
        title="Busiest Bike Stations in NYC",
        subtitle="In March 2024, Per Borough"
    )
    .tab_stubhead(label="Station")
    .fmt_number(columns="num_rides", decimals=1)
    .fmt_percent(columns=cs.starts_with("percent_"), decimals=0)  ❸
    .fmt_nanoplot(columns="rides_per_day", reference_line="mean")
    .data_color(columns="num_rides", palette="Blues")
    .tab_options(row_group_font_weight="bold")
    .tab_source_note(source_note=md(
        "Source: [NYC Citi Bike](https://citibikenyc.com/system-data)"
    ))
)
```

❶ Stations are grouped by borough to add structure.

❷ We can give table columns proper names without needing to change the underlying DataFrame.

❸ Great Tables accepts column selectors, making our code more compact.

Busiest Bike Stations in NYC

In March 2024, Per Borough

Station	Neighborhood	Mean Daily Rides	Members	E-Bikes	Rides Per Day
Manhattan					
W 21 St & 6 Ave	Chelsea	354.2	91%	58%	
Forsyth St & Broome St	Lower East Side	322.2	95%	25%	
Broadway & W 58 St	Midtown	315.2	80%	70%	
Brooklyn					
Metropolitan Ave & Bedford Ave	Williamsburg	185.4	85%	68%	
N 7 St & Driggs Ave	Williamsburg	146.0	86%	66%	
Hanson Pl & Ashland Pl	Fort Greene	144.1	84%	61%	
Queens					
Queens Plaza North & Crescent St	Long Island City	127.0	85%	57%	
Vernon Blvd & 50 Ave	Long Island City	95.7	89%	66%	
31 St & Newtown Ave	Astoria	78.2	88%	60%	
Bronx					
Melrose Ave & E 150 St	Melrose	41.7	83%	89%	
E 161 St & River Ave	Concourse	35.6	75%	88%	
Plaza Dr & W 170 St	Mount Eden	31.7	84%	95%	

Source: NYC Citi Bike

Figure 13-17. A table showing information about the three busiest stations per borough

The output table is shown in Figure 13-17. The line plots in the right-most column are known as *nanoplots*. They visualize the daily number of rides per station.

A nice property of having these methods as building blocks is that you can quickly create a first table, then iteratively improve on it.

Takeaways

In this chapter we've looked at several ways to turn DataFrames into graphs and tables. The key takeaways are:

- There are many data visualization packages
- Polars has built-in plotting capabilities that use hvPlot under the hood
- hvPlot uses either Bokeh, Matplotlib, or Plotly to produce plots

- hvPlot can combine multiple plots either next to each other or on top of each other
- hvPlot can create widgets for interactively select and plott subsets of data
- hvPlot allows you to create geographical visualizations
- You can always use Pandas if a certain data visualization package doesn't fully support Polars yet
- Plotnine is a great data visualization package if you prefer to use the grammar of graphics
- A table can be a valuable alternative to a plot and the Great Tables helps you produce one that looks great

In the next chapter we're going to look at extending Polars.

About the Authors

Jeroen Janssens is a Senior Machine Learning Engineer at Xomnia in Amsterdam, where he uses Polars on a daily basis. He enjoys wrangling data, implementing machine learning models, and building solutions using Python, R, JavaScript, and Bash. Previously, he ran Data Science Workshops, a training and coaching firm. Jeroen is the author of Data Science at the Command Line (O'Reilly, 2021). He has been an assistant professor at Jheronimus Academy of Data Science and a data scientist at various startups in New York City. Jeroen holds a PhD in machine learning from Tilburg University and an MSc in artificial intelligence from Maastricht University. He lives with his wife and two kids in Rotterdam, the Netherlands.

Thijs Nieuwdorp is the Lead Data Scientist at Xomnia in Amsterdam. His interest in the interaction between human and computer led him to an education in Artificial Intelligence at the Radboud University, after which he dove straight into the field of Data Science. At Xomnia he witnessed the birth of Polars as Ritchie Vink started working on it during his employment there , and has been using it in his projects ever since. He enjoys figuring out complex data problems, optimizing existing solutions, and putting them to good use by implementing them into business processes. Outside work Thijs enjoys exploring our world through hiking and traveling, and exploring other worlds through books, games, and movies. He lives in Amsterdam with his partner.