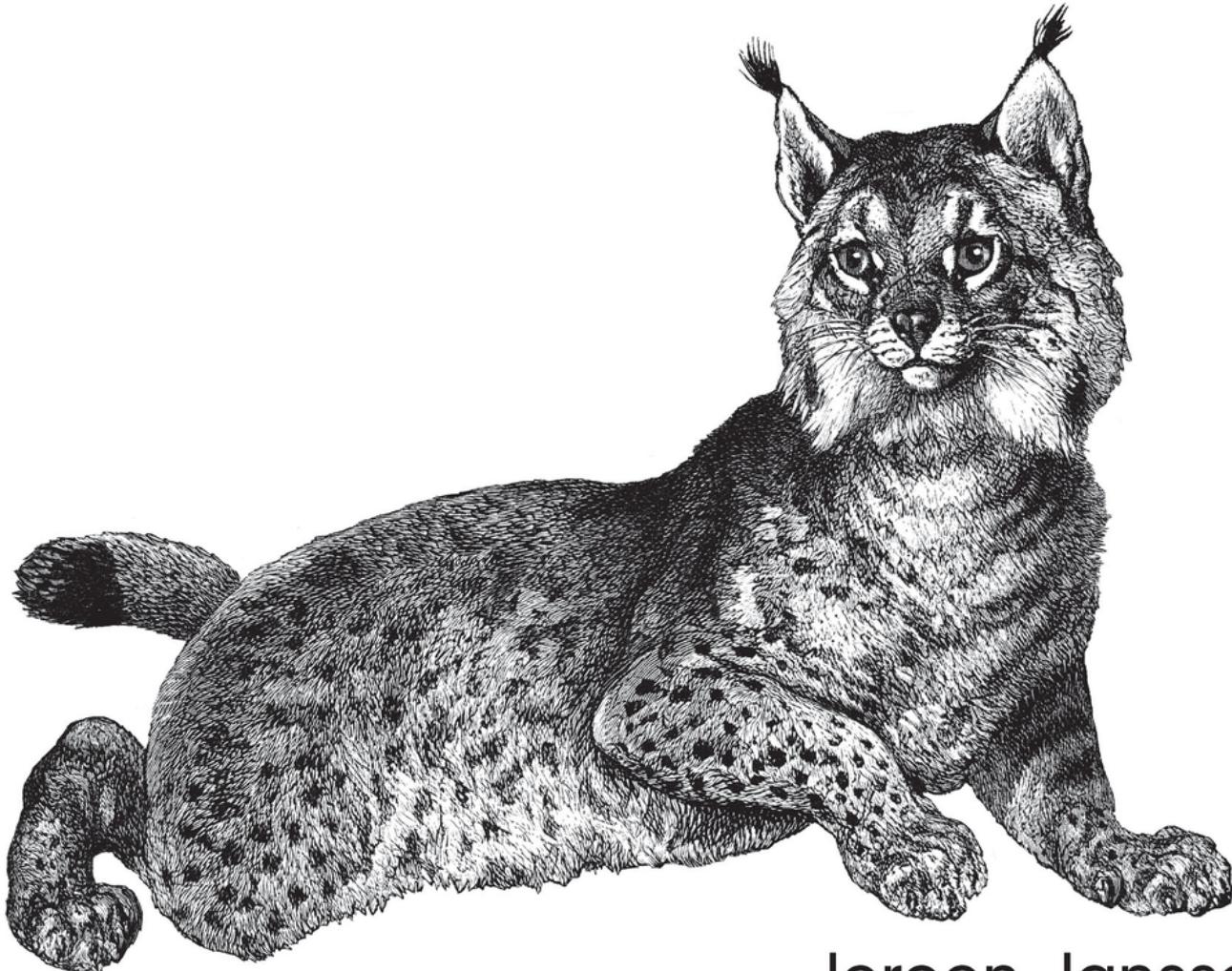


O'REILLY®

# 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

*[OceanofPDF.com](http://OceanofPDF.com)*

# Python Polars: The Definitive Guide

by Jeroen Janssens and Thijs Nieuwdorp

Copyright © 2024 Jeroen Janssens and Thijs Nieuwdorp. All rights reserved.

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](mailto: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

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

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Python Polars: The Definitive Guide*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property

rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-15608-4

[LSI]

[OceanofPDF.com](http://OceanofPDF.com)

# 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](mailto: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 Docker, pull an image and start a container. There will be a quick explanation on the basics of Jupyter. And you’ll get into installing Polars and learn how to compile it from source if necessary. The last thing you’ll get into is the configuration you can tweak in Polars. 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.

## Using Polars in a Docker Container

There are about as many different system configurations in the world as there are systems. To ensure that the commands in this book will run like we intend them to, we recommend that you run them on a Jupyter server that runs in a Docker container. 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. Note that the terminal output will contain an important link that you'll want to copy. To access the Jupyter server that will be running in the container, 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.

To finish the setup, you'll need to install the public repo that contains the notebooks accompanying the book. This can be done by opening the terminal in JupyterLab, which can be started by clicking this tile:



Figure 1-1. The tile to start the terminal in JupyterLab

In the terminal you can run the following command to clone the repository:

```
$ git clone https://github.com/jeroenjanssens/pyt
```

This will clone the repository into the container.

The final step to work with the code in this book is to install the dependencies defined in `requirements.txt`. This can be done by running the following command in the terminal:

```
$ pip install -r python-polars-the-definitive-gu
```

This will install all the required packages to run the code in the notebooks.

You can now start by opening this chapter's notebook `ch02.ipynb` in the `python-polars-the-definitive-guide/` directory, which you can find on the left hand side of the screen.

## Crash Course JupyterLab

Jupyter is a web-based interactive development environment for notebooks, code, and data. To get started, you can create a Python 3 Notebook using the button in the top row.

All files will remain saved in the container as long as it is not deleted. When you're done, you can exit the container by navigating to the terminal window in which you've started the container and pressing `Ctrl + C` (which is the universal keyboard interrupt hotkey). This will stop the Jupyter server, and close the container. If you want to start the container up again, use the same `docker run` command.

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 1-1](#) lists shortcuts that can be run in any mode, [Table 1-2](#) lists shortcuts that can be run in command mode, and [Table 1-3](#) lists shortcuts that can be run in edit mode.

## Any Mode

Table 1-1. 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 1-2. 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
Z	Undo cell deletion
M	Change the cell type to Markdown
Y	Change the cell type to Code

## Edit Mode

Table 1-3. 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.

## Installing Polars

Now that you've set up an environment to work in, you need Polars to do some work with. The latest information on how to install Polars can always be found on [the GitHub page](#). The following section is based on those instructions.

Polars works with optional dependencies for different use cases. At the time of writing Polars supports the optional dependencies as shown in [Table 1-4](#).

Table 1-4. 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
connector	Support for reading from SQL databases
x	
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]'
```

---

#### NOTE

Notice the exclamation mark. This tells the Jupyter kernel to run the command following it in a bash session, instead of interpreting it as Python.

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`.

---

---

**TIP**

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 you could use `conda`, although `pip` is the Polars team's preferred way of installing Polars:

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

- 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-  
-C target-cpu=native
```

---

**NOTE**

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 complex 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`.

---

## NOTE

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
```

---

## WARNING

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 1-5](#).

Table 1-5. A few of the notable Polars configuration settings

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

# 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 output
```

```
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 _ 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_10
---	---	---	...	---
str	str	str	...	str
NITxKLUkXv	vrLgRRjGXL	ErZIZfRrEq	...	beyr
y0YxtzSnWQ	QPcPJFsbjj	jUgWnjTSkj	...	LsnE
...	...	...	...	Vg...
MqZmJe0HNK	ubSBwY0gYk	HQdUpgsJus	...	eCko
XceAPNdRb0	fTat0QmkRm	uscqAuvSfP	...	sGft
...	...	...	...	ox...
H1XQdVFTVL	ybaZRpdIJh	VtnYHFRNNA	...	LPZv
DbZHFIWPUw	PzrHJsjsA	KCTLizyVyl	...	Uqt:
...	...	...	...	jW...
dODwyenwQR	BmfJ0HZkA	JbXtfyUyNG	...	NOTe
PMqTnmiEzN	oMfoGlBbBH	DXbgdKpXjo	...	yTiS
...	...	...	...	jZ...
IuXzwotCLy	cZzaPcRNBU	vTlcINzgBB	...	kLti
OyjNrWVpyT	DfuMHNCJGn	zoCwOeoATT	...	xtcs
...	...	...	...	ow...

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.

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

shape: (5, 10)

col	colu...								
u...	---	---	---	---	---	---	---	---	---
---	str								
str									
NIT	vrLg...	ErZI...	aXoG...	zQXd...	BqAF...	F...	...	...	...
X...									
MqZ	ubSB...	HQdU...	jvRg...	zcDr...	Pees...	Z...	...	...	...
m...									
HlX	ybaZ...	VtnY...	JFHN...	EXzX...	aBdy...	C...	...	...	...
Q...									
dOD	BmfJ...	JbXt...	rHAq...	pwJO...	oRCW...	C...	...	...	...
w...									
IuX	cZza...	vTlc...	JNjW...	OEuZ...	AXWe...	E...	...	...	...



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

---

**NOTE**

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:

```
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:

```
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.

```
@pl.Config(ascii_tables=True)
def write_ascii_frame_to_stdout(df: pl.DataFrame):
    print(str(df))

@pl.Config(verbose=True)
def function_that_im_debugging(df: pl.DataFrame):
    # Polars operation you want to see the verbose
    pass
```

## Downloading Datasets and Code Examples

## Conclusion

In this chapter you've learned how to use Docker and Jupyter, two pieces of software that allow us to set up a working environment that guarantees that the code in this book will run. You've also learned how to install Polars and

optional dependencies, and how to compile it from source if necessary. Finally, you've learned how to tweak the configuration of Polars to make it just right. 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.

[OceanofPDF.com](http://OceanofPDF.com)

# 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 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](mailto: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](#).

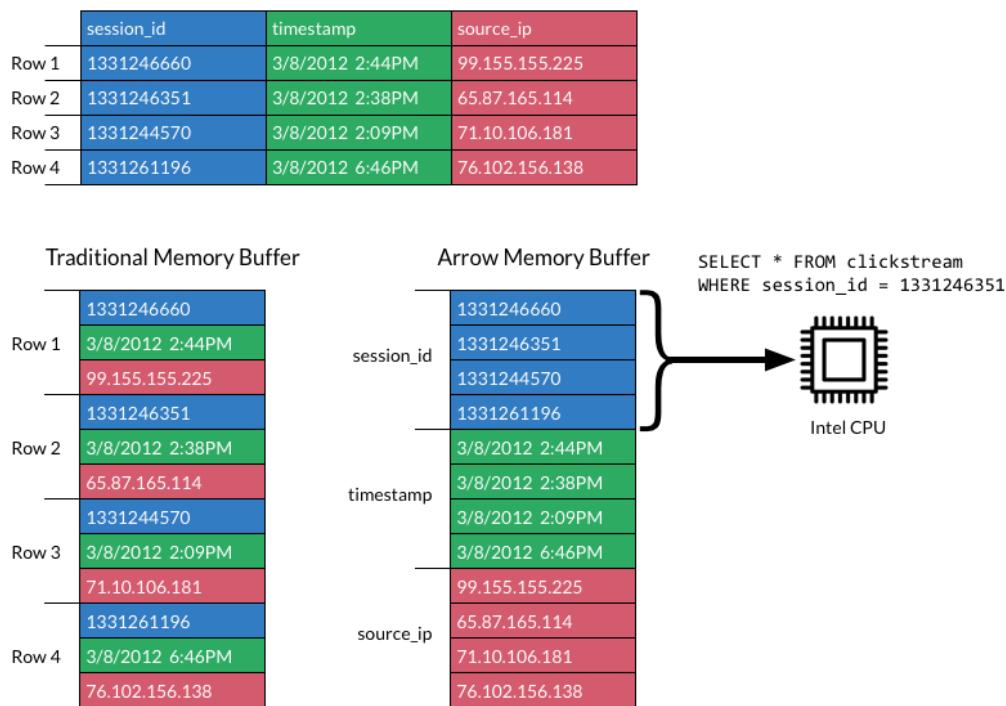


Figure 2-1. Illustration of the Arrow memory buffer and its advantages for computation

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.

---

#### NOTE

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:

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

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 10100000000000000000000000000000

- 0 - means the float is positive.
- 10000010 - The *exponent* in binary, which is 130 in decimal.
- 10100000000000000000000000000000 - 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:

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

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

<b>Group</b>	<b>Type</b>	<b>Details</b>	<b>Range</b>
Base class	<code>DataType</code>	Base class for all Polars data types.	
Numeric	<code>Decimal</code>	Decimal 128-bit type with an optional precision and non-negative scale.	Can exactly represent 128 significant digits
	<code>Float32</code>	32-bit floating point type.	-3.4e+38 to 3.4e+38
	<code>Float64</code>	64-bit floating point type.	-1.7e+308 to 1.7e+308
	<code>Int8</code>	8-bit signed integer type.	-128 to 128
	<code>Int16</code>	16-bit signed integer type.	-32,768 to 32,767

Group	Type	Details	Range
	Int32	32-bit signed integer type.	-2,147,483,648 to 2,147,483,647
	Int64	64-bit signed integer type.	-9,223,372,036,854,77 to 9,223,372,036,854,775
	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	-5877641-06-24 to 5879610-09-09

<b>Group</b>	<b>Type</b>	<b>Details</b>	<b>Range</b>
		data type, days since UNIX epoch 1970- 01-01 as int32.	
	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.	

Group	Type	Details	Range
	rgs)	List(*args, **kwargs) Variable length list type.	
	Struct(*args, **kwargs)	Struct type.	
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.	
	al	Allows for more efficient memory usage if a column	

<b>Group</b>	<b>Type</b>	<b>Details</b>	<b>Range</b>
		contains few unique strings.	
	<b>Null</b>	Type representing Null / None values.	
	<b>Object</b>	Type for wrapping arbitrary Python objects.	
	<b>String</b>	UTF-8 encoded string type of variable length.	
	<b>Unknown</b>	Type representing Datatype values that	

Group	Type	Details	Range
		could not be determined statically.	

---

**NOTE**

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.

---

**NOTE**

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]
<hr/>	
[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. *Structs* 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]  
    },  
)  
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
<hr/>	
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
---	---	---	---	---	---
i64	i64	i64	i64	i64	i64
null	null	2	2	9	5
2	2	2	2	2	2
3	3	3	3	3	3
4	4	4	4	4	4
null	4	7	2	9	5
null	4	7	2	9	5
7	7	7	7	7	7
8	8	8	8	8	8
9	9	9	9	9	9
null	9	null	2	9	5



The third way of filling null values is with an expression like

```
pl.col("value").mean():
```

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

shape: (10, 2)

value	expression_mean
---	---
i64	f64
<hr/>	
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()
```

```
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

---

#### WARNING

`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.

---

#### NOTE

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()}"
```

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:

```
int_df = string_df.select(pl.col("id").cast(pl.U:
print(int_df)
print(f"Estimated size: {int_df.estimated_size('I
```

```
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`:

```
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
10000	1.0	1

---	---	---
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, "v
```

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.U
```

```
shape: (3, 3)
```

id	value	value2
---	---	---
i64	f32	u8
100000	1.0	1
200000	2.0	2
300000	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
100000	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 8](#) 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 as 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.

[OceanofPDF.com](http://OceanofPDF.com)

# 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 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](mailto: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")
sum_per_vendor = trips.groupby("VendorID").sum()

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

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
<hr/>	
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.

- ③ 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_"
lf.select(pl.col("trip_distance")).show_graph()
```

- 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  $\pi$  that only 1 in 19 columns will be read into memory in the first place.

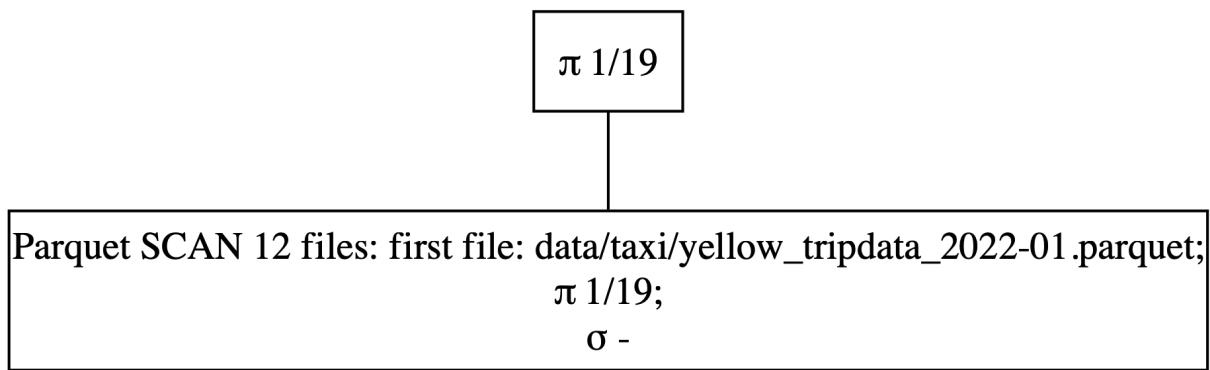


Figure 3-1. The resulting query plan when scanning the taxi dataset and selecting a single column.

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  $\sigma$  stands for SELECTION and indicates any row filter conditions.
- The  $\pi$  stands for PROJECTION and indicates choosing a subset of columns.

In [Figure 3-1](#) you can see that  $\pi$  contains a selection of 1 out of the 19 available columns. In [Figure 3-2](#) you can see that the  $\sigma$  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.

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

```
[Parquet SCAN 12 files: first file: data/taxi/yellow_tripdata_2022-01.parquet;
    π * / 19;
    σ (col("trip_distance")) > ...]
```

*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  $\sigma$ . 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:

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

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

VendorID	tpep_pickup p_datetime	tpep_dropoff ff_datetime	...	total_
---	---	...	---	nt
i64	---	...	---	---
	datetime[n s]	datetime[n s]		f64
1	2022-01-01 00:35:40	2022-01-01 00:53:29	...	21.95
1	2022-01-01 00:33:43	2022-01-01 00:42:07	...	13.3

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").alias("f1"))
```

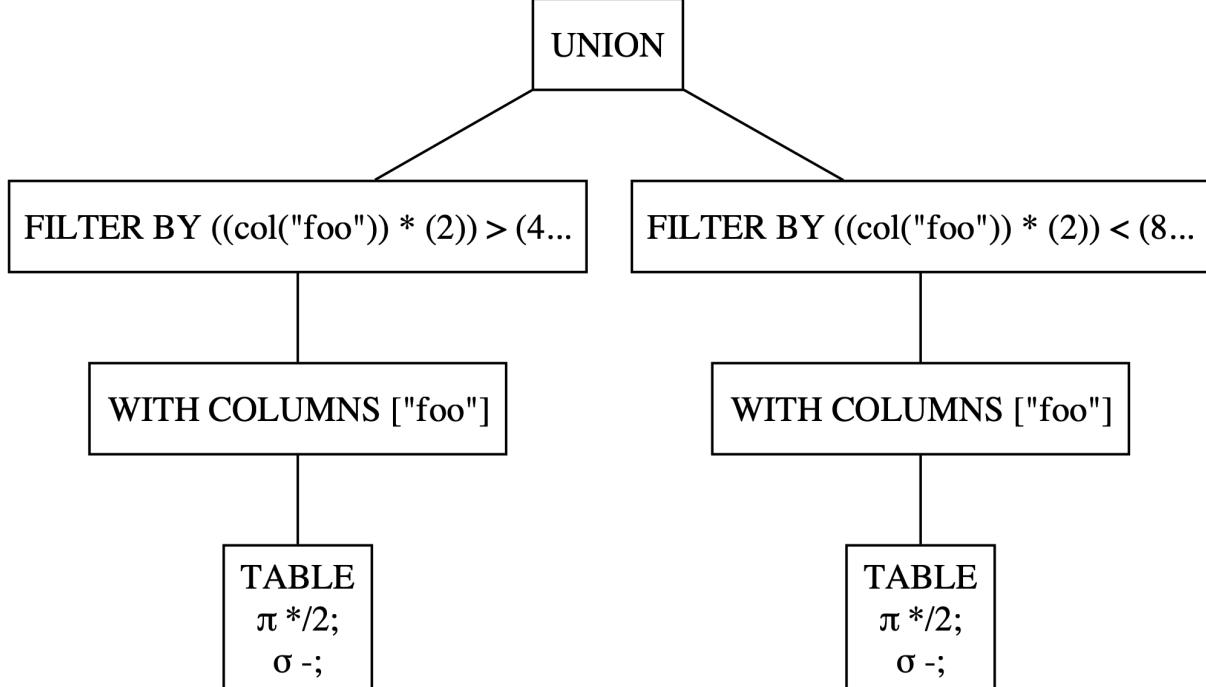
```

# 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)

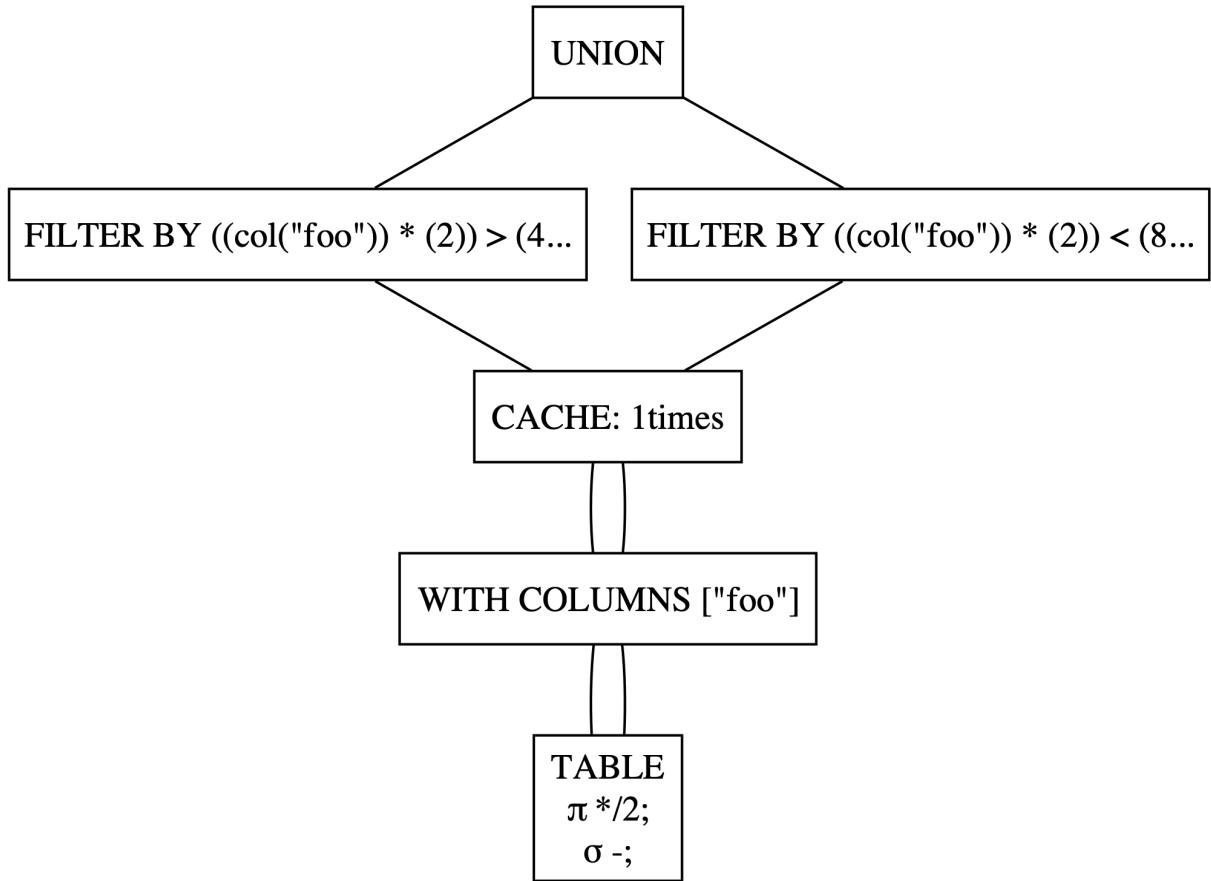
result = pl.concat([expr1, expr2])

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.

---

#### NOTE

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.

```
l df = pl.LazyFrame({  
    "name": ["Alice", "Bob", "Charlie"],  
    "age": [25, 30, 35]  
})  
  
erroneous_query = l df.with_columns(  
    pl.col("age").str.slice(1,3).alias("sliced_age")  
)  
  
result_df = erroneous_query.collect()
```



```
SchemaError: invalid series dtype: expected `Str`
```

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()
income_per_distance_per_vendor = sum_per_vendor \
    .with_column(pl.col("total_amount") / pl.col("distance"))
income_per_distance_per_vendor = income_per_distance_per_vendor \
    .group_by("VendorID",
              income_per_distance=pl.col("total_amount") / pl.col("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  
Wall time: 592 ms  
shape: (3, 2)
```

VendorID	income_per_distance
---	---
i64	f64
<hr/>	
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.

```
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())
```

shape: (3, 2)

col1	col2
1	4
2	5
3	6

shape: (3, 2)

col1	col2
1	4
2	5
3	6

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
<code>.columns</code>	✓	✓
<code>.dtypes</code>	✓	✓
<code>.flags</code>	✓	
<code>.height</code>	✓	
<code>.schema</code>	✓	✓
<code>.shape</code>	✓	
<code>.width</code>	✓	✓

# Computation

DataFrames have the *computation* methods `fold()` and `hash_rows()` where a LazyFrame 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
<code>.approx_n_unique()</code>	✓	
<code>.describe(...)</code>	✓	
<code>.estimated_size(...)</code>	✓	
<code>.explain(...)</code>		✓
<code>.glimpse()</code>	✓	
<code>.is_duplicated()</code>	✓	
<code>.is_empty()</code>	✓	
<code>.is_unique()</code>	✓	
<code>.n_chunks()</code>	✓	
<code>.n_unique(...)</code>	✓	
<code>.show_graph(...)</code>		✓

## 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

<b>Method</b>	<b>DataFrame</b>	<b>LazyFrame</b>
<code>.__iter__()</code>	✓	
<code>.agg(...)</code>	✓	✓
<code>.all()</code>	✓	✓
<code>.apply(...)</code>	✓	✓
<code>.count()</code>	✓	✓
<code>.first()</code>	✓	✓
<code>.head(...)</code>	✓	✓
<code>.last()</code>	✓	✓
<code>.map_groups(...)</code>	✓	✓
<code>.max()</code>	✓	✓
<code>.mean()</code>	✓	✓
<code>.median()</code>	✓	✓
<code>.min()</code>	✓	✓
<code>.n_unique()</code>	✓	✓

Method	DataFrame	LazyFrame
.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
<code>.approx_n_unique()</code>		✓
<code>.bottom_k(...)</code>	✓	✓
<code>.cast(...)</code>	✓	✓
<code>.clear(...)</code>	✓	✓
<code>.clone()</code>	✓	✓
<code>.drop(...)</code>	✓	✓
<code>.drop_in_place(...)</code>	✓	
<code>.drop_nulls(...)</code>	✓	✓
<code>.explode(...)</code>	✓	✓
<code>.extend(...)</code>	✓	
<code>.fill_nan(...)</code>	✓	✓
<code>.fill_null(...)</code>	✓	✓
<code>.filter(...)</code>	✓	✓
<code>.find_idx_by_name(...)</code>	✓	

Method	DataFrame	LazyFrame
.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()	✓	✓

Method	DataFrame	LazyFrame
<code>.item(...)</code>	✓	
<code>.iter_columns()</code>	✓	
<code>.iter_rows()</code>	✓	
<code>.iter_slices(...)</code>	✓	
<code>.join(...)</code>	✓	✓
<code>.join_asof(...)</code>	✓	✓
<code>.last()</code>		✓
<code>.limit(...)</code>	✓	✓
<code>.melt(...)</code>	✓	✓
<code>.merge_sorted(...)</code>	✓	✓
<code>.partition_by()</code>	✓	
<code>.pipe(...)</code>	✓	
<code>.pivot(...)</code>	✓	
<code>.rechunk()</code>	✓	

Method	DataFrame	LazyFrame
.rename(...)	✓	✓
.replace(...)	✓	
.replace_at_idx(...)	✓	
.replace_column(...)	✓	
.reverse()	✓	✓
.rolling(...)	✓	✓
.row()	✓	
.rows()	✓	
.rows_by_key(...)	✓	
.sample(...)	✓	
.select(...)	✓	✓
.select_seq(...)	✓	✓
.set_sorted(...)	✓	✓
.shift(...)	✓	✓

Method	DataFrame	LazyFrame
<code>.shift_and_fill(...)</code>	✓	✓
<code>.shrink_to_fit(...)</code>	✓	
<code>.slice(...)</code>	✓	✓
<code>.sort(...)</code>	✓	✓
<code>.tail(...)</code>	✓	✓
<code>.take_every(...)</code>	✓	✓
<code>.top_k(...)</code>	✓	✓
<code>.to_dummies(...)</code>	✓	
<code>.to_series(...)</code>	✓	
<code>.transpose(...)</code>	✓	
<code>.unique(...)</code>	✓	✓
<code>.unnest(...)</code>	✓	✓
<code>.unstack(...)</code>	✓	
<code>.update(...)</code>	✓	✓

Method	DataFrame	LazyFrame
<code>.upsample(...)</code>	✓	
<code>.vstack(...)</code>	✓	
<code>.with_columns(...)</code>	✓	✓
<code>.with_columns_seq(...)</code>	✓	✓
<code>.with_context(...)</code>		✓
<code>.with_row_count(...)</code>	✓	✓

## 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
<code>.cache()</code>		✓
<code>.collect(...)</code>		✓
<code>.collect_async()</code>		✓
<code>.corr(...)</code>	✓	
<code>.equals(...)</code>	✓	
<code>.fetch(...)</code>		✓
<code>.frame_equal(...)</code>	✓	
<code>.lazy()</code>	✓	✓
<code>.map(...)</code>		✓
<code>.map_batches(...)</code>		✓
<code>.map_rows(...)</code>	✓	
<code>.pipe(...)</code>		✓
<code>.profile(...)</code>		✓

# 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
```

---

**NOTE**

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:

$$\begin{aligned}\text{thread\_factor} &= \max \left\{ \frac{12}{\text{n\_threads}}, 1 \right\} \\ \text{chunk\_size} &= \max \left\{ \frac{50000}{\text{n\_cols} * \text{thread\_factor}}, 1000 \right\}\end{aligned}$$

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.

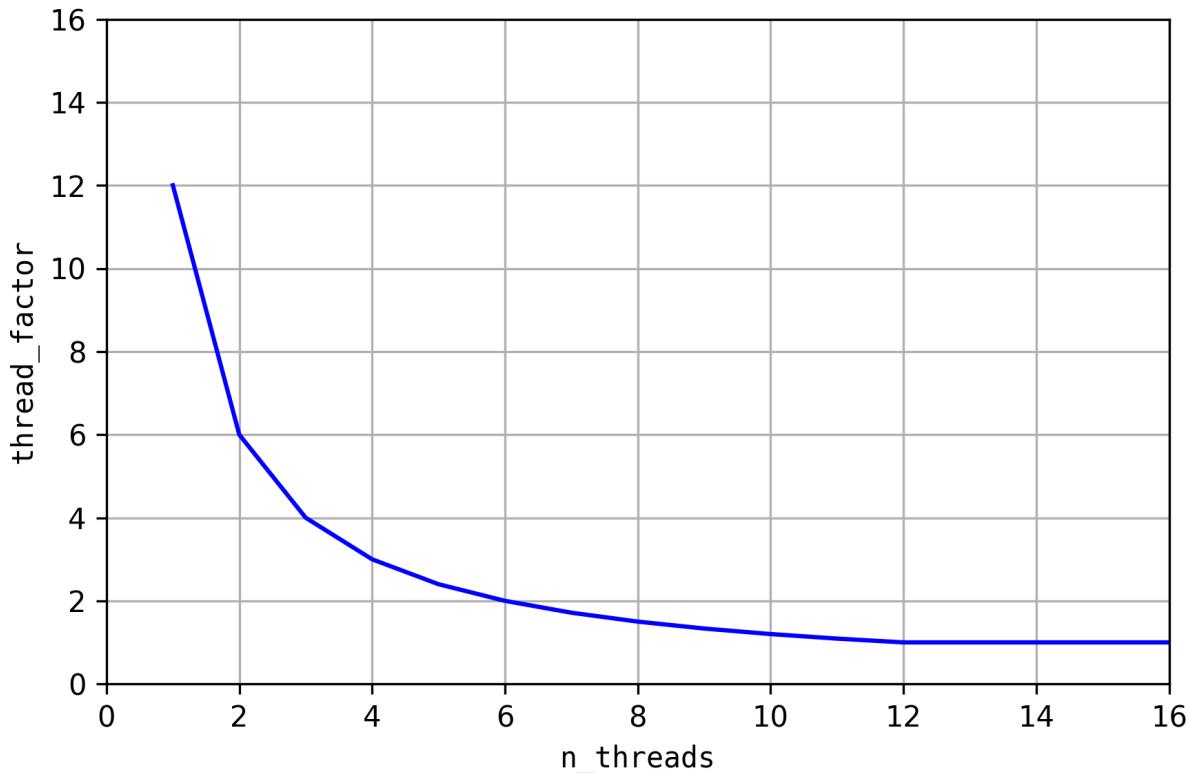


Figure 3-5. Thread factor

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

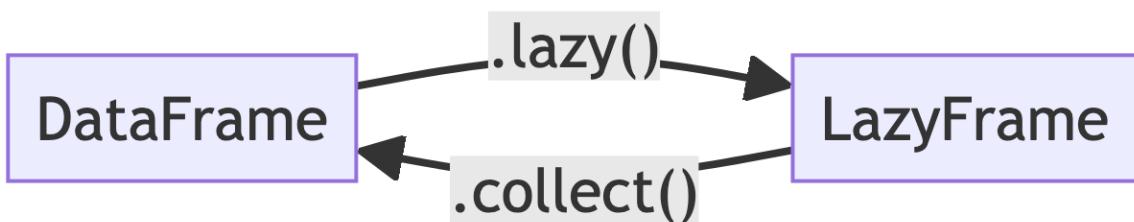


Figure 3-6. The operations to swap to the other API

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 or 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]
```

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())
```

shape: (3, 2)

col1	col2
---	---
i64	i64

1	4
2	5
3	6

shape: (3, 2)

col1	col2
---	---
i64	i64
<hr/>	
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 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](mailto: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 1](#) 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 1](#). 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","body_mass_g","sex","species_id","island_id"
"1","Adelie","Torgersen",39.1,18.7,181,3750,"male",0,0,0
"2","Adelie","Torgersen",39.5,17.4,186,3800,"female",0,0,0
"3","Adelie","Torgersen",40.3,18,195,3250,"female",0,0,0
"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:

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

shape: (344, 9)

rowid	species	island	...	body_mass_g
---	---	---	...	---
i64	str	str	...	str
1	Adelie	Torgersen	...	3750
2	Adelie	Torgersen	...	3800
3	Adelie	Torgersen	...	3250
4	Adelie	Torgersen	...	NA
5	Adelie	Torgersen	...	3450
...	...	...	...	...
340	Chinstrap	Dream	...	4000
341	Chinstrap	Dream	...	3400
342	Chinstrap	Dream	...	3775
343	Chinstrap	Dream	...	4100
344	Chinstrap	Dream	...	3775

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
<code>source</code>	Path to a file or a file-like object.
<code>has_header</code>	Indicate if the first row of dataset is a header or not.
<code>r</code>	
<code>columns</code>	Columns to select. Accepts a list of column indices (starting at zero) or a list of column names.
<code>separator</code>	Single byte character to use as delimiter in the file.
<code>skip_rows</code>	Start reading after a certain number of lines.
<code>null_values</code>	Values to interpret as null values.
<code>encoding</code>	Default: <code>utf8</code> . <code>utf8-lossy</code> means that invalid UTF-8 values are replaced with <code>?</code> characters. When using other encodings than <code>utf8</code> or <code>utf8-lossy</code> , 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="")
```

```
shape: (344, 9)
```

rowid	species	island	...	body_mass_g
---	---	---		---
i64	str	str		i64
1	Adelie	Torgersen	...	3750
2	Adelie	Torgersen	...	3800
3	Adelie	Torgersen	...	3250
4	Adelie	Torgersen	...	null

5	Adelie	Torgersen	...	3450
...	...	...	...	...
340	Chinstrap	Dream	...	4000
341	Chinstrap	Dream	...	3400
342	Chinstrap	Dream	...	3775
343	Chinstrap	Dream	...	4100
344	Chinstrap	Dream	...	3775

---

#### NOTE

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=[])
)
```

```
shape: (9, 2)
```

column	null_count
---	---
str	u32
<hr/>	
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<sup>1</sup> get an error, just like we get here with *data/directors.csv*:

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

```
ComputeError: could not parse '⁇⁇⁇⁇' as dtype
ber 1)
```

The current offset in the file is 19 bytes.

You might want to try:

- increasing `infer\_schema\_length` (e.g. `infer\_s
- specifying correct dtype with the `dtypes` argu
- 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:

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

```
shape: (4, 3)
```

name	born	country
---	---	---
str	i64	str
	1930	
Verhoeven	1938	

	1942	
Tarantino	1963	

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:

```
import chardet

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

    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
<hr/>		
	1930	
Verhoeven	1938	
	1942	
Tarantino	1963	

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 1](#).) 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")  
songs
```

```
shape: (2_001, 4)  
| positie | titel | artiest  
| --- | --- | ---  
| i64 | str | str
```

null	null	null	
1	Bohemian Rhapsody	Queen	
2	Roller Coaster	Danny Vera	
3	Hotel California	Eagles	
4	Piano Man	Billy Joel	
...	...	...	
1996	Charlie Brown	Coldplay	
1997	Beast Of Burden	Bette Midler	
1998	It Was A Very Good Y...	Frank Sinatra	
1999	Hou Van Mij	3JS	
2000	Drivers License	Olivia Rodrigo	

- ➊ 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={"skiprows": 1})  
songs_fixed
```

```
shape: (2_000, 4)
```

positie	titel	artiest
---	---	---
i64	str	str
1	Bohemian Rhapsody	Queen
2	Roller Coaster	Danny Vera
3	Hotel California	Eagles
4	Piano Man	Billy Joel
5	Fix You	Coldplay
...	...	...
1996	Charlie Brown	Coldplay
1997	Beast Of Burden	Bette Midler
1998	It Was A Very Good Y...	Frank Sinatra
1999	Hou Van Mij	3JS
2000	Drivers License	Olivia Rodrigo

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
<code>source</code>	Path to a file or a file-like object.
<code>sheet_id</code>	Sheet number to convert (0 for all sheets). Defaults to 1 if neither this nor <code>sheet_name</code> are specified.
<code>sheet_name</code>	Sheet name to convert. Cannot be used in conjunction with <code>sheet_id</code> .
<code>xlsx2csv_options</code>	Extra options passed to <code>xlsx2csv.Xlsx2csv()</code> . e.g.: <code>{"skip_empty_lines": True}</code>
<code>read_csv_options</code>	Extra options passed to <code>pl.read_csv()</code> for parsing the CSV file returned by <code>xlsx2csv.Xlsx2csv().convert()</code>

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](#)”, 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
---	---	---	...	---
str	str	f64		f64
NVDA	2010-01-04	4.6275	...	4.6225
NVDA	2010-01-05	4.605	...	4.69
NVDA	2010-01-06	4.6875	...	4.72
NVDA	2010-01-07	4.695	...	4.6275
NVDA	2010-01-08	4.59	...	4.6375
...	...	...	...	...
NVDA	2019-12-24	59.549999	...	59.654999
NVDA	2019-12-26	59.689999	...	59.797501
NVDA	2019-12-27	59.950001	...	59.217499
NVDA	2019-12-30	58.997501	...	58.080002
NVDA	2019-12-31	57.724998	...	58.825001

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	date	open	...	close
---	---	---	...	---
str	str	f64	...	f64
ASML	1999-01-04	11.765625	...	12.140625
ASML	1999-01-05	11.859375	...	13.96875
ASML	1999-01-06	14.25	...	16.875
ASML	1999-01-07	14.742188	...	16.851562
ASML	1999-01-08	16.078125	...	15.796875
...	...	...	...	...
TSM	2023-06-26	102.019997	...	100.110002
TSM	2023-06-27	101.150002	...	102.080002
TSM	2023-06-28	100.5	...	100.919996
TSM	2023-06-29	101.339996	...	100.639996
TSM	2023-06-30	101.400002	...	100.919996

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)
]

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']
```

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

shape: (1\_512, 8)

symbol	date	open	...	close
---	---	---	---	---
str	str	f64		f64
ASML	2000-01-03	43.875	...	43.64062
ASML	2000-01-04	41.953125	...	40.73437
ASML	2000-01-05	39.28125	...	39.60937
ASML	2000-01-06	36.75	...	37.17187
ASML	2000-01-07	36.867188	...	38.01562
...	...	...	...	...
ASML	2020-12-24	478.950012	...	483.0899
ASML	2020-12-28	487.140015	...	480.2399
ASML	2020-12-29	489.450012	...	484.0100
ASML	2020-12-30	488.130005	...	489.9100
ASML	2020-12-31	490.0	...	487.7200

## 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:

```
trips = pl.read_parquet("data/taxi/yellow_tripdat
trips
```

```
shape: (39_656_098, 19)
```

VendorID	tpep_pickup_datetime	...	congestion
---	---		---
i64	datetime[ns]		f64
<hr/>			
1	2022-01-01 00:35:40	...	2.5
1	2022-01-01 00:33:43	...	0.0
2	2022-01-01 00:53:21	...	0.0
2	2022-01-01 00:25:21	...	2.5
2	2022-01-01 00:36:48	...	2.5
...	...	...	...

2	2022-12-31 23:46:00	...	null
2	2022-12-31 23:13:24	...	null
2	2022-12-31 23:00:49	...	null
1	2022-12-31 23:02:50	...	null
2	2022-12-31 23:00:15	...	null

---

**TIP**

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

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/pokeball/bulbasaur.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]]

```
[{"id": 1, "name": "Bulbasaur", "url": "http://www.serebii.net/pokemone...rass", "type": "Poison"}, {"id": 2, "name": "Ivysaur", "url": "http://www.serebii.net/pokemone...rass", "type": "Poison"}, {"id": 3, "name": "Venusaur", "url": "http://www.serebii.net/pokemone...rass", "type": "Poison"}]
```

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
---	---	---	---
164	str	list[str]	str
<hr/>			
1	Bulbasaur	["Grass", "Poison"]	0.71
2	Ivysaur	["Grass", "Poison"]	0.99
3	Venusaur	["Grass", "Poison"]	2.01
4	Charmander	["Fire"]	0.61
5	Charmeleon	["Fire"]	1.09
...	...	...	...
147	Dratini	["Dragon"]	1.80
148	Dragonair	["Dragon"]	3.99
149	Dragonite	["Dragon", "Flying"]	2.21

150	Mewtwo	["Psychic"]	2.01
151	Mew	["Psychic"]	0.41

◀ ▶

[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
<code>source</code>	Path to a file or a file-like object.
<code>schema</code>	The DataFrame schema may be declared in several ways: (1) As a dictionary of <code>{name: type}</code> pairs; if type is <code>None</code> , it will be auto-inferred. (2) As a list of column names; in this case types are automatically inferred. (3) As a list of <code>(name, type)</code> pairs; this is equivalent to the dictionary form.
<code>schema_overrides</code>	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
```

```
{"$schema":"/mediawiki/recentchange/1.0.0", "meta":  
{"$schema":"/mediawiki/recentchange/1.0.0", "meta":  
{"$schema":"/mediawiki/recentchange/1.0.0", "meta":  
{"$schema":"/mediawiki/recentchange/1.0.0", "meta":
```

```
{"$schema":"/mediawiki/recentchange/1.0.0", "meta'  
... 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  
 'id': 1659529639,  
 'length': {'new': 91166, 'old': 91108},  
 'meta': {'domain': 'en.wikipedia.org',  
          'dt': '2023-07-29T07:51:39Z',  
          'id': '0416300b-980c-45bb-b0a2-c9d7a9e2',  
          'offset': 4820784717,  
          'partition': 0,  
          'request_id': 'ea0541fb-4e72-4fc3-82f0...',  
          'stream': 'mediawiki.recentchange',  
          'topic': 'eqiad.mediawiki.recentchange',  
          'uri': 'https://en.wikipedia.org/wiki/E...'}  
}
```

```
'minor': False,  
'namespace': 0,  
'notify_url': 'https://en.wikipedia.org/w/index  
'parsedcomment': '<span dir="auto"><span class="  
' href="/wiki/EFL_Championship#I  
' title="EFL Championship">→\u2026  
' runners-up and play-off final:  
'revision': {'new': 1167689309, 'old': 116682424  
'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_  
'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:

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

```
shape: (100, 20)
```

\$schema	meta	...
---	---	
str	struct[9]	
<hr/>		
/mediawiki/recentc...	{"https://en.wikip...	...
...	...	...
/mediawiki/recentc...	{"https://en.wikip...	...

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

```
(  
    wikipedia.rename({"id": "edit_id"})  
    .unnest("meta")
```

```
.select("timestamp", "title", "user", "comment")
)
```

```
shape: (100, 4)
```

timestamp	title	user
---	---	---
i64	str	str
1690617099	EFL Championship	87.12.215.232
1690617102	Lim Sang-choon	Preferwiki
1690617104	Higher	Ss112
1690617104	International Pok...	Piotrus
1690617105	Abdul Hamid Khan ...	InternetArch...
...	...	...
1690617238	Havering Resident...	MRSC
1690617235	Olha Kharlan	2603:7000:210
1690617238	Mukim Kota Batu	Pangalau
1690617239	User:IDK1213safas...	94.101.29.27
1690617234	List of bus route...	Pedroperezhur...

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_alphabets"
pl.from_pandas(pd.read_html(url)[0])
```

shape: (62, 4)

abbreviation	Latin	translation
---	---	---
str	str	str
<hr/>		
A.D.	anno Domini	"in the year of the Lord"

A.I.	ad interim	"temporarily"
a.m.	ante meridiem	"before midday"
ca./c.	circa	"around", "about"
Cap.	capitulus	"chapter"
...	...	...
S.O.S.	si opus sit	"if there is need"
sic	sic erat scriptum	"thus it was written"
stat.	statim	"immediately"
viz.	videlicet	"namely", "in other words"
vs. v.	versus	"against"

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	rat:
---	---	---	---
i64	str	str	str
1	ACADEMY DINOSAUR	Documentary	PG
2	ACE GOLDFINGER	Horror	G
3	ADAPTATION HOLES	Documentary	NC-1
4	AFFAIR PREJUDICE	Horror	G
5	AFRICAN EGG	Family	G
6	AGENT TRUMAN	Foreign	PG

7	AIRPLANE SIERRA	Comedy	PG-13
8	AIRPORT POLLOCK	Horror	R
9	ALABAMA DEVIL	Horror	PG-13
10	ALADDIN CALENDAR	Sports	NC-17

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")
film_categories = pl.read_database_uri("SELECT * FROM category")
categories = pl.read_database_uri("SELECT * FROM category")

(
    films.join(film_categories, on="film_id", suffix="_film")
        .join(categories, on="category_id", suffix="_category")
        .select([
            "film_id",
            "title",
            pl.col("name").alias("category"),
            "rating",
            pl.col("length") / 60,
        ])
)
```

```
.limit(10)  
)
```

shape: (10, 5)

film_id	title	category	rate
---	---	---	---
i64	str	str	str
1	ACADEMY DINOSAUR	Documentary	PG
2	ACE GOLDFINGER	Horror	G
3	ADAPTATION HOLES	Documentary	NC-17
4	AFFAIR PREJUDICE	Horror	G
5	AFRICAN EGG	Family	G
6	AGENT TRUMAN	Foreign	PG
7	AIRPLANE SIERRA	Comedy	PG-13
8	AIRPORT POLLOCK	Horror	R
9	ALABAMA DEVIL	Horror	PG-13
10	ALADDIN CALENDAR	Sports	NC-17

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
<code>file</code>	File path to write the DataFrame to. If set to <code>None</code> (default), the output is returned as a string instead.
<code>has_header</code>	Whether to include a header (default: <code>True</code> ).
<code>r</code>	
<code>separator</code>	Character to separate CSV fields (default: <code>,</code> ).
<code>quote</code>	Character to use for quoting values (default: <code>"</code> ).
<code>null_value</code>	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
<code>worksheet</code>	Name of target worksheet (default: <code>Sheet1</code> ).
<code>position</code>	Table position in Excel notation (eg: “A1”), or a (row,col) integer tuple.
<code>table_style</code>	A named Excel table style, such as “Table Style Medium 4”, or a dictionary of <code>{"key": value}</code> options containing one or more of the following keys: “style”, “first_column”, “last_column”, “banded_columns”, “banded_rows”.
<code>column_widths</code>	A <code>{colname:int}</code> 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
<code>file</code>	File path to which the DataFrame should be written.
<code>compression</code>	Choose <code>zstd</code> for good compression performance. Choose <code>lz4</code> for fast compression and decompression. Choose <code>snappy</code> for more backwards compatibility guarantees when you deal with older parquet readers.
<code>compression_level</code>	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.

<sup>1</sup> We say “ideally”, because then it’s clear that you’ve not specified the correct encoding.

# 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 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](mailto: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).

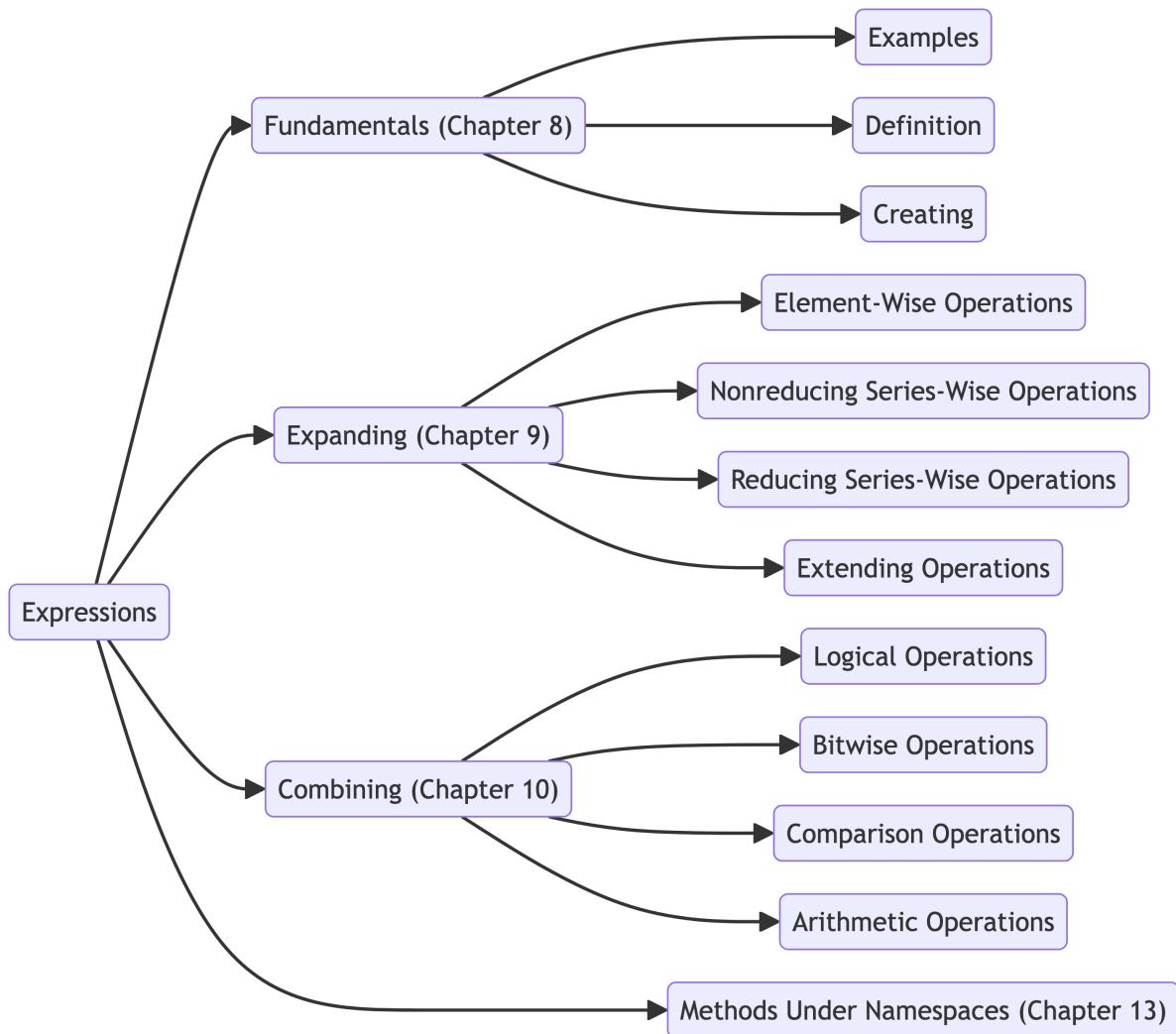


Figure 5-1. The many expression methods are organized into three chapters

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<sup>1</sup> from around the world:

```
import polars as pl

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

shape: (10, 5)

name	weight	color	is_round	orig
---	---	---	---	---
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):

```
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
---	---	---	---
str	str	str	f64
Avocado	green	South America	0.2
Banana	yellow	Asia	0.12
Blueberry	blue	North America	0.001
Cantaloupe	orange	Africa	2.5
Cranberry	red	North America	0.002
Elderberry	black	Europe	0.001
Orange	orange	Asia	0.13
Papaya	orange	South America	1.0
Peach	orange	Asia	0.15

Watermelon	green	Africa	5.0
------------	-------	--------	-----

- 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")  
)
```

shape: (10, 7)

name	weight	color	is_round	origin	is_fruit	is_berry
Avocado	200	green	false	South America	True	False
Banana	120	yellow	false	Asia	True	False
Blueberry	1	blue	false	North America	True	True
Cantaloupe	2500	orange	true	Africa	True	False
Cranberry	2	red	false	North America	True	True
Elderberry	1	black	false	Europe	True	False
Orange	130	orange	true	Asia	True	True

Papaya	1000	orange	false	South
Peach	150	orange	true	Asia
Watermelon	5000	green	true	Africa

- 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 8](#).

## 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
<hr/>				
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.groupby(  
    pl.col("origin").str.split(" ").list.last()  
) .agg(  
    pl.count(), ②  
    pl.col("weight").mean().alias("average_weight")  
)
```

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

- ➊ Each unique value of this expression (the last part of the `origin` column) leads to one group.
- ➋ The expression created by the function `pl.count()` returns the number of rows in the group.

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

---

#### TIP

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	orig
---	---	---	---	---
str	i64	str	bool	str

	name	count	color	is酸性	continent
	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:

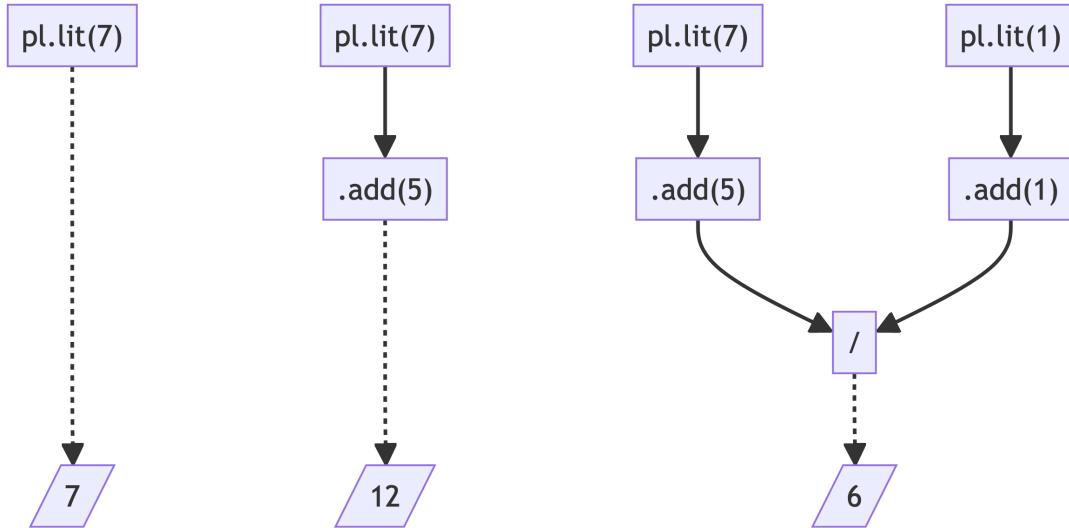
### *Series*

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 be the steps, and the functions and methods such as 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,  
        .with_columns(pl.all().mul(10).name.suff  
    )
```

shape: (3, 4)

a	b	a_times_10	b_times_10
---	---	---	---
i64	f64	i64	f64
<hr/>			
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").me
```

```
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").al
```

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

shape: (10, 6)

name	weight	color	is_round	
---	---	---	---	
str	i64	str	bool	
Avocado	200	green	false	
Banana	120	yellow	false	
Blueberry	1	blue	false	
Cantaloupe	2500	orange	true	
Cranberry	2	red	false	
Elderberry	1	black	false	
Orange	130	orange	true	
Papaya	1000	orange	false	
Peach	150	orange	true	
Watermelon	5000	green	true	

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

shape: (4, 5)

name	weight	color	is_round	
Orange	130	orange	true	

	---	---	---	---
	str	i64	str	bool
Cantaloupe	2500	orange	true	
Orange	130	orange	true	
Papaya	1000	orange	false	
Peach	150	orange	true	

```
fruit.groupby(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", "Afr  
    "latin": ["Lilium columbianum", "Iris ve  
    "color": ["orange", "purple", "orange"]  
})  
  
flowers.filter(is_orange)
```

shape: (2, 3)

name	latin	co
---	---	--
str	str	st
<hr/>		
Tiger lily	Lilium columbianum	or
African marigold	Tagetes erecta	or

*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"]; PROJE  
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 PolarType
```

---

## 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)

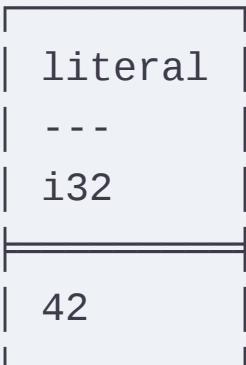
weight	weight_kg
---	---
i64	f64
<hr/>	
200	0.2
120	0.12
1	0.001
2500	2.5
2	0.002

## 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)
```



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)
```



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	orig	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])))
```

```
ShapeError: unable to add a column of length 2 to
```

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("is_round"))
```

```
shape: (10, 6)
```

name	weight	color	is_round	origin
---	---	---	---	---
str	i64	str	bool	str
Avocado	200	green	false	South America
Banana	120	yellow	false	Asia
...	...	...	...	...



Peach	150	orange	true	Asia
Watermelon	5000	green	true	Africa

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
<hr/>		
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("i"))
```

```
ShapeError: unable to add a column of length 9 to
```

## 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
<code>pl.arange(...)</code>	A range of integers (alias of <code>pl.int_range(...)</code> )
<code>pl.date_range(...)</code>	A range of dates
<code>pl.date_ranges(...)</code>	Each element is a range of dates
<code>pl.datetime_range(...)</code>	A range of datetimes
<code>pl.datetime_ranges(...)</code>	Each element is a range of datetimes
<code>pl.int_range(...)</code>	A range of integers
<code>pl.int_ranges(...)</code>	Each element is a range of integers
<code>pl.time_range(...)</code>	A range of times
<code>pl.time_ranges(...)</code>	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,
    pl.repeat(date(2021, 10, 21), 6).alias("end"
).with_columns(
    pl.datetime_ranges("start", "end", interval='1d'
)
```

shape: (6, 3)

start	end	range
---	---	---
date	date	list[datetime[μs]]
1985-10-21	2021-10-21	[1985-10-21 00:00:00, 2021-10-21 00:00:00]
1985-10-22	2021-10-21	[1985-10-22 00:00:00, 2021-10-21 00:00:00]
1985-10-23	2021-10-21	[1985-10-23 00:00:00, 2021-10-21 00:00:00]
1985-10-24	2021-10-21	[1985-10-24 00:00:00, 2021-10-21 00:00:00]
1985-10-25	2021-10-21	[1985-10-25 00:00:00, 2021-10-21 00:00:00]
1985-10-26	2021-10-21	[1985-10-26 00:00:00, 2021-10-21 00:00:00]

In [Chapter 8](#) 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 9](#).

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 8](#).

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

Method	Description
<code>Expr.name.suffix_fields(...)</code>	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})
df
```

shape: (1, 3)

text	An integer	BOOLEAN
---	---	---
str	i64	bool
value	5040	true

We can change these column names with various methods:

```
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
<hr/>		
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:

```
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:

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

```
shape: (1, 3)
```

text	an_integer	boolean
---	---	---
str	i64	bool
<hr/>		
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	orig
---	---	---	---	---
str	i64	str	bool	str
<hr/>				
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("color").str.contains("orange"))  
        .with_columns(pl.col("name").str.ends_with("ly"))  
        .collect()  
)
```

shape: (2, 6)

name	weight	color	is_round	original
---	---	---	---	---
str	i64	str	bool	str
<hr/>				
Cantaloupe	2500	orange	true	African

Watermelon	5000	green	true	Africa
------------	------	-------	------	--------

Now an example without expressions:

```
(  
    fruit  
    .lazy()  
    .filter((fruit["weight"] > 1000) & fruit["is_bitter"]  
    .with_columns(fruit["name"].str.ends_with("berry"))  
    .collect()  
)
```

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

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.

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

[OceanofPDF.com](http://OceanofPDF.com)

# 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 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](mailto: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 specifically, 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:

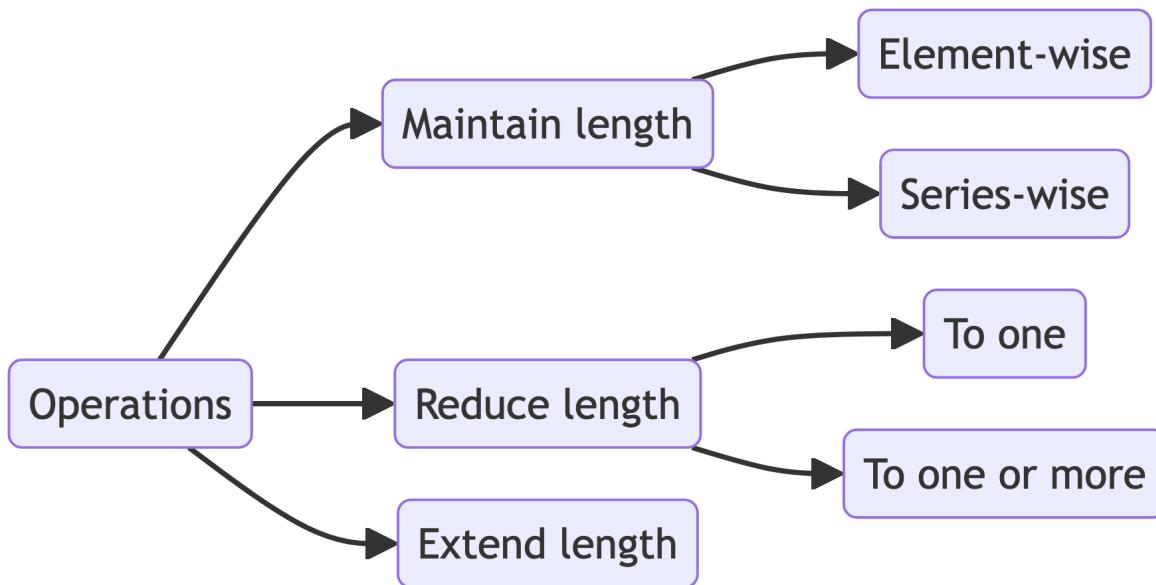
```
import math
import numpy as np

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=" ")
    .select(
        "species",
        "island",
        "sex",
        "year",
        pl.col("body_mass_g").alias("mass") / 100
    ))
penguins.with_columns(
```

```
    pl.col("mass").sqrt().alias("mass_sqrt"), ❶
    pl.col("mass").interpolate().alias("mass_fill")
)
```

```
shape: (344, 7)
```

species	island	sex	year	mass
---	---	---	---	---
str	str	str	i64	f64
Adelie	Torgersen	male	2007	3.75
Adelie	Torgersen	female	2007	3.8
Adelie	Torgersen	female	2007	3.25
Adelie	Torgersen	null	2007	null
Adelie	Torgersen	female	2007	3.45
...	...	...	...	...
Chinstrap	Dream	male	2009	4.0
Chinstrap	Dream	female	2009	3.4
Chinstrap	Dream	male	2009	3.775
Chinstrap	Dream	male	2009	4.1
Chinstrap	Dream	female	2009	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
<hr/>	
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

...	
Adelie	
Adelie	
Adelie	
Adelie	
Saiyan	

- ➊ Get the three unique values of the Series.
- ➋ Repeat each value 3,000 times. This produces a Series of three long lists.
- ➌ Use the explode method to get one long Series of 9,000 values.
- ➍ 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
<code>Expr.abs()</code>	Compute absolute values.
<code>Expr.cbrt()</code>	Compute the cube root of the elements.
<code>Expr.exp()</code>	Compute the exponential, element-wise.
<code>Expr.log(...)</code>	Compute the logarithm to a given base.
<code>Expr.log10()</code>	Compute the base 10 logarithm of the input array, element-wise.
<code>Expr.log1p()</code>	Compute the natural logarithm of each element plus one.
<code>Expr.sign()</code>	Compute the element-wise indication of the sign.
<code>Expr.sqrt()</code>	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, 10],
    .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
---	---	---	---	---
f64	f64	f64	f64	f64
-2.000	2.000	0.135	NaN	NaN
0.000	0.000	1.000	-inf	-inf
0.500	0.500	1.649	-1.000	-0.301
1.000	1.000	2.718	0.000	0.000
2.718	2.718	15.154	1.443	0.434

1000.000	1000.000	inf	9.966	3.000
----------	----------	-----	-------	-------

- 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

<b>Method</b>	<b>Description</b>
<code>Expr.arccos()</code>	Compute the element-wise value for the inverse cosine.
<code>Expr.arccosh()</code>	Compute the element-wise value for the inverse hyperbolic cosine.
<code>Expr.arcsin()</code>	Compute the element-wise value for the inverse sine.
<code>Expr.arcsinh()</code>	Compute the element-wise value for the inverse hyperbolic sine.
<code>Expr.arctan()</code>	Compute the element-wise value for the inverse tangent.
<code>Expr.arctanh()</code>	Compute the element-wise value for the inverse hyperbolic tangent.
<code>Expr.cos()</code>	Compute the element-wise value for the cosine.
<code>Expr.cosh()</code>	Compute the element-wise value for the hyperbolic cosine.
<code>Expr.degrees()</code>	Convert from radians to degrees.

Method	Description
<code>Expr.radians()</code>	Convert from degrees to radians.
<code>Expr.sin()</code>	Compute the element-wise value for the sine.
<code>Expr.sinh()</code>	Compute the element-wise value for the hyperbolic sine.
<code>Expr.tan()</code>	Compute the element-wise value for the tangent.
<code>Expr.tanh()</code>	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,
```

```
.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
---	---	---	---
f64	f64	f64	f64
-3.141593	NaN	-1.0	-180.0
0.0	1.570796	1.0	0.0
1.0	0.0	0.540302	57.29578
3.141593	NaN	-1.0	180.0
6.283185	NaN	1.0	360.0
90.0	NaN	-0.448074	5156.620156
180.0	NaN	-0.59846	10313.240312
360.0	NaN	-0.283691	20626.480625

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<sup>1</sup>.

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 / 2],  
    .with_columns(  
        ceil=pl.col("x").ceil(),  
        clip=pl.col("x").clip(-1, 1),  
        round=pl.col("x").round(2),  
        floor=pl.col("x").floor()  
    })  
)
```

```

        cut=pl.col("x").cut([-1, 1], labels=["bad", "neutral", "good"]),
        floor=pl.col("x").floor(),
        qcut=pl.col("x").qcut([0.5], labels=["below", "above"]),
        round2=pl.col("x").round(2),
        round0=pl.col("x").round(0), ②
    )
)

```

shape: (8, 8)

x	ceil	clip	cut	floor	qcut
---	---	---	---	---	---
f64	f64	f64	cat	f64	cat
-6.0	-6.0	-1.0	bad	-6.0	below
-0.5	-0.0	-0.5	neutral	-1.0	below
0.0	0.0	0.0	neutral	0.0	below
0.5	1.0	0.5	neutral	0.0	below
3.141593	4.0	1.0	good	3.0	above
9.9	10.0	1.0	good	9.0	above
9.99	10.0	1.0	good	9.0	above
9.999	10.0	1.0	good	9.0	above

- ① 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.

- ② 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. `Nan`s 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
<code>Expr.fill_nan(...)</code>	Fill floating point NaN value with a fill value.
<code>Expr.fill_null(...)</code>	Fill null values using the specified value or strategy.
<code>Expr.is_finite()</code>	Returns a Boolean Series indicating which values are finite.
<code>Expr.is_infinite()</code>	Returns a Boolean Series indicating which values are infinite.
<code>Expr.is_nan()</code>	Returns a Boolean Series indicating which values are NaN.
<code>Expr.is_not_nan()</code>	Returns a Boolean Series indicating which values are not NaN.
<code>Expr.is_not_null()</code>	Returns a Boolean Series indicating which values are not null.
<code>Expr.is_null()</code>	Returns a Boolean Series indicating which values are null.

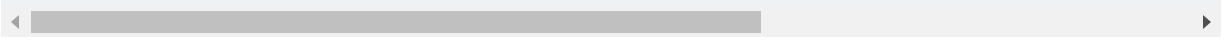
The code snippet below applies the methods `Expr.fill_nan()`, `Expr.fill_null()`, `Expr.is_finite()`, `Expr.is_infinite()`, `Expr.is_nan()`, and `Expr.is_null()` to a couple of numerical values, some of which are infinite or missing. The methods `Expr.is_not_nan()` and `Expr.is_not_null()` produce the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

```
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
42	42	42	True	False
nan	999	0	False	True

42.0	42.0	42.0	true	true
NaN	999.0	NaN	false	false
null	null	0.0	null	null
inf	inf	inf	false	false
-inf	-inf	-inf	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
<code>Expr.hash(...)</code>	Hash the elements in the selection.
<code>Expr.repeat_by(...)</code>	Repeat the elements in this Series as specified in the given expression.
<code>Expr.replace(...)</code>	Replace values in column according to remapping dictionary.

The following code snippet demonstrates these three methods:

```
(  
    pl.DataFrame({"x": ["here", "there", "their"]})  
    .with_columns(  
        hash=pl.col("x").hash(seed=1337),  
        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              |
|---------|----------------------|------------------------|
| ---     | ---                  | ---                    |
| str     | u64                  | list[str]              |
| here    | 12695211751326448172 | ["here", "here"]       |
| there   | 17329794691236705436 | ["there", "there"]     |
| their   | 2663095961041830581  | ["their", "their"]     |
| they're | 6743063676290245144  | ["they're", "they're"] |


```

- 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
<code>Expr.cum_count (...)</code>	Get an array with the cumulative count computed at every element.
<code>Expr.cum_max (...)</code>	Get an array with the cumulative max computed at every element.
<code>Expr.cum_min (...)</code>	Get an array with the cumulative min computed at every element.
<code>Expr.cum_prod (...)</code>	Get an array with the cumulative product computed at every element.
<code>Expr.cum_sum (...)</code>	Get an array with the cumulative sum computed at every element.
<code>Expr.diff(...)</code>	Calculate the n-th discrete difference.
<code>Expr.pct_change e(...)</code>	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],
                  "y": [-1, 0, 1, 2, 3, 4]}).select([
        pl.col("x").cum_count(), ❶
        pl.col("x").cum_max(),
        pl.col("x").cum_min(),
        pl.col("x").cum_prod(reverse=True),
        pl.col("x").cum_sum(),
        pl.col("x").diff(),
        pl.col("x").pct_change(),
    ])
)

```

shape: (8, 8)

x	cum_count	cum_max	cum_min	cum_prod
---	---	---	---	---
f64	u32	f64	f64	f64
0.0	1	0.0	0.0	NaN
1.0	2	1.0	0.0	NaN
2.0	3	2.0	0.0	NaN
null	3	null	null	null
2.0	4	2.0	0.0	NaN
NaN	5	2.0	0.0	NaN
-1.0	6	2.0	-1.0	-2.0

2.0	7	2.0	-1.0	2.0
-----	---	-----	------	-----

- 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
<code>Expr.backward_fill(...)</code>	Fill missing values with the next to-be-seen value.
<code>Expr.forward_fill(...)</code>	Fill missing values with the latest seen value.
<code>Expr.interpolate(...)</code>	Fill missing values using interpolation.
<code>Expr.shift(...)</code>	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,  
    .with_columns(  
        backward_fill=pl.col("x").backward_fill(),  
        forward_fill=pl.col("x").forward_fill(lin  
        interp1=pl.col("x").interpolate(method="l  
        interp2=pl.col("x").interpolate(method="i  
        shift1=pl.col("x").shift(1),  
        shift2=pl.col("x").shift(-2),
```

```
)  
)
```

```
shape: (9, 7)
```

x	backward_fill	forward_fill	interp1
---	---	---	---
f64	f64	f64	f64
-1.0	-1.0	-1.0	-1.0
0.0	0.0	0.0	0.0
1.0	1.0	1.0	1.0
null	3.0	1.0	1.666667
null	3.0	null	2.333333
3.0	3.0	3.0	3.0
4.0	4.0	4.0	4.0
NaN	NaN	NaN	NaN
6.0	6.0	6.0	6.0

- 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
<code>Expr.is_duplicate d()</code>	Get a Boolean Series that indicates which values are duplicated.
<code>Expr.is_first_distinct()</code>	Get a Boolean Series that indicates which values are first unique.
<code>Expr.is_last_distinct()</code>	Get a Boolean Series that indicates which values are last unique.
<code>Expr.is_unique()</code>	Get a Boolean Series 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
---	---	---	---
str	bool	bool	bool
A	false	true	true
C	true	true	false
D	false	true	true
C	true	false	true

- 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
<code>Expr.ewm_mean(...)</code>	Exponentially-weighted moving average.
<code>Expr.ewm_std(...)</code>	Exponentially-weighted moving standard deviation.
<code>Expr.ewm_var(...)</code>	Exponentially-weighted moving variance.
<code>Expr.rolling_apply(...)</code>	Apply a custom rolling window function.
<code>Expr.rolling_map(...)</code>	Compute a custom rolling window function.
<code>Expr.rolling_max(...)</code>	Apply a rolling max (moving max) over the values in this array.
<code>Expr.rolling_mean(...)</code>	Apply a rolling mean (moving mean) over the values in this array.
<code>Expr.rolling_median(...)</code>	Compute a rolling median.
<code>Expr.rolling_min(...)</code>	Apply a rolling min (moving min) over the values in this array.

Method	Description
<code>Expr.rolling_quan tile(...)</code>	Compute a rolling quantile.
<code>Expr.rolling_skew (...)</code>	Compute a rolling skew.
<code>Expr.rolling_std (...)</code>	Compute a rolling standard deviation.
<code>Expr.rolling_sum (...)</code>	Apply a rolling sum (moving sum) over the values in this array.
<code>Expr.rolling_var (...)</code>	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_i
    .select("date", "close")
    .with_columns(
        ewm_mean=pl.col("close").ewm_mean(com=7,
```

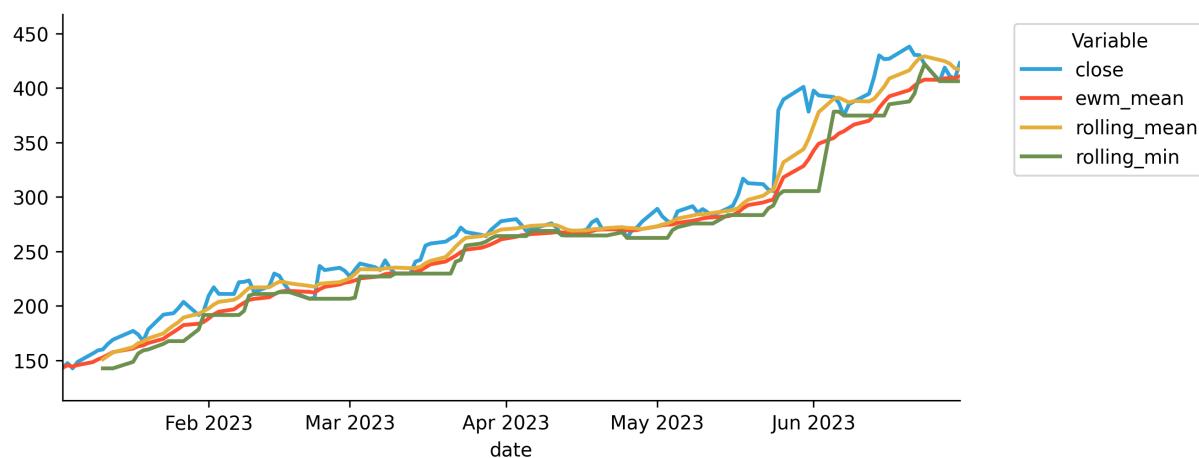
```
    rolling_mean=pl.col("close").rolling_mean(5)
    rolling_min=pl.col("close").rolling_min(5)
)
stock
```

```
shape: (124, 5)
```

date	close	ewm_mean	rolling_mean
---	---	---	---
date	f64	f64	f64
2023-01-03	143.149994	143.149994	null
2023-01-04	147.490005	145.464667	null
2023-01-05	142.649994	144.398755	null
2023-01-06	148.589996	145.664782	null
2023-01-09	156.279999	148.388917	null
...	...	...	...
2023-06-26	406.320007	407.54911	425.805
2023-06-27	418.76001	408.950473	424.695
2023-06-28	411.170013	409.227915	422.445
2023-06-29	408.220001	409.101926	418.180
2023-06-30	423.019989	410.841684	417.1185

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
<code>Expr.arg_sort(...)</code>	Get the index values that would sort this column.
<code>Expr.shuffle(...)</code>	Shuffle the contents of this expression.
<code>Expr.sort(...)</code>	Sort this column.
<code>Expr.sort_by(...)</code>	Sort this column by the ordering of other columns.
<code>Expr.reverse()</code>	Reverse the selection.
<code>Expr.rank(...)</code>	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
---	---	---	---	---	---
i64	str	u32	i64	i64	i64
1	D	2	1	1	3
3	I	0	null	3	1
null	S	1	3	3	3
3	C	3	7	7	7

7	0	4	3	null	null
---	---	---	---	------	------

## 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
<code>Expr.rle_id()</code>	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, 60, 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
<hr/>	
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
<code>Expr.all</code> (...)	Return whether all values in the column are True.
<code>Expr.any</code> (...)	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
<hr/>					
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

<b>Method</b>	<b>Description</b>
<code>Expr.entropy</code> (...)	Computes the entropy.
<code>Expr.kurtosis</code> (...)	Compute the kurtosis (Fisher or Pearson) of a dataset.
<code>Expr.max()</code>	Get maximum value.
<code>Expr.mean()</code>	Get mean value.
<code>Expr.median()</code>	Get median value using linear interpolation.
<code>Expr.min()</code>	Get minimum value.
<code>Expr.nan_max</code> ( )	Get maximum value, but propagate/poison encountered NaN values.
<code>Expr.nan_min</code> ( )	Get minimum value, but propagate/poison encountered NaN values.
<code>Expr.product</code> ( )	Compute the product of an expression.
<code>Expr.quantile</code> (...)	Get quantile value.

Method	Description
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),
        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 |
+-----+-----+-----+-----+-----+
| --- | --- | --- | --- | --- |
| f64 | f64 | f64 | f64 | f64 |
+-----+-----+-----+-----+-----+
| 20.752443 | 4.994978 | 9.931565 | 0.003245 | 2
+-----+-----+-----+-----+-----+

```

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
<code>Expr.approx_n_unique()</code>	Approximate count of unique values.
<code>Expr.count()</code>	Count the number of values in this expression.
<code>Expr.len()</code>	Count the number of values in this expression.
<code>Expr.n_unique()</code>	Count unique values.
<code>Expr.null_count()</code>	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))
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
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
<code>Expr.arg_max()</code>	Get the index of the maximal value.
<code>Expr.arg_min()</code>	Get the index of the minimal value.
<code>Expr.first()</code>	Get the first value.
<code>Expr.get(...)</code>	Return a single value by index.
<code>Expr.implode()</code>	Aggregate values into a list.
<code>Expr.last()</code>	Get the last value.
<code>Expr.lower_bound()</code>	Calculate the lower bound.
<code>Expr.upper_bound()</code>	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
---	---	---	---	-
u32	i64	i64	list[i64]	i64
0	0	null	[0, 7245, ... 3723]	3723

- 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
<code>Expr.arg_unique()</code>	Get index of first unique value.
<code>Expr.unique(...)</code>	Get unique values of this expression.
<code>Expr.unique_counts()</code>	Return a count of the unique values in the order of appearance.
<code>Expr.value_counts(...)</code>	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),  
    unique_counts=pl.col("x").unique_counts(),  
    value_counts=pl.col("x").value_counts(),  
)  
)
```

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}

- ➊ Maintaining the order of the values is computationally more intensive.
- ➋ 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
<code>Expr.bottom_k(...)</code>	Return the k smallest elements.
<code>Expr.head(...)</code>	Get the first n rows.
<code>Expr.limit(...)</code>	Get the first n rows (alias for <code>Expr.head()</code> ).
<code>Expr.sample(...)</code>	Sample from this expression.
<code>Expr.slice(...)</code>	Get a slice of this expression.
<code>Expr.tail(...)</code>	Get the last n rows.
<code>Expr.gather(...)</code>	Take values by index.
<code>Expr.gather_every(...)</code>	Take every nth value in the Series and return as a new Series.
<code>Expr.top_k(...)</code>	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],  
    gather_every=pl.col("x").gather_every(247),  
    top_k=pl.col("x").top_k(7),  
)
```

```
shape: (7, 7)
```

bottom_k	head	sample	slice	gather	ga
---	---	---	---	---	---
i64	i64	i64	i64	i64	i64
null	0	6871	807	7245	0
0	7245	2202	8634	7245	868
1	5227	7328	2109	5227	848
6	2747	1648	null	2747	835
7	9816	5761	1740	2657	180
10	2657	9315	3333	5393	363

21	4578	8370	788	8203	584
----	------	------	-----	------	-----

- 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
<code>Expr.drop_nans()</code>	Drop floating point NaN values.
<code>Expr.drop_nulls()</code>	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
<hr/>	
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
<code>Expr.arg_true()</code>	Return indices where expression evaluates True.
<code>Expr.flatten()</code>	Flatten a list or string column.
<code>Expr.mode()</code>	Compute the most occurring value(s).
<code>Expr.reshape(...)</code>	Reshape this Expr to a flat Series or a Series of Lists.
<code>Expr.rle()</code>	Get the lengths of runs of identical values.
<code>Expr.search_sorted(...)</code>	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(),
    )
)
```

```
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(),  
    )  
)
```

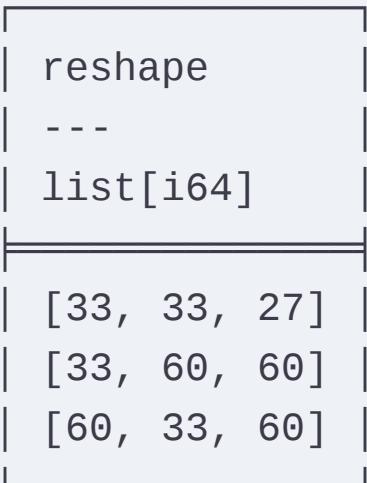
shape: (2, 1)

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)
```



- 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(), ❷  
    )  
)
```

```
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),  
    )  
)
```

shape: (1, 1)



- 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](#)).

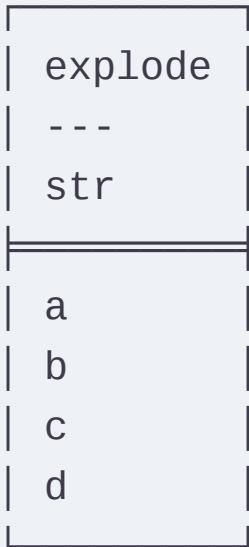
Table 6-20. Two Series-wise operations that extend

Method	Description
<code>Expr.explode()</code>	Explode a list expression.
<code>Expr.extend_constant</code> <code>(...)</code>	Extend the Series with a constant 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)



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.

- 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()`.

# 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 10th 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](mailto: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
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](#).

---

#### NOTE

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()`.

---

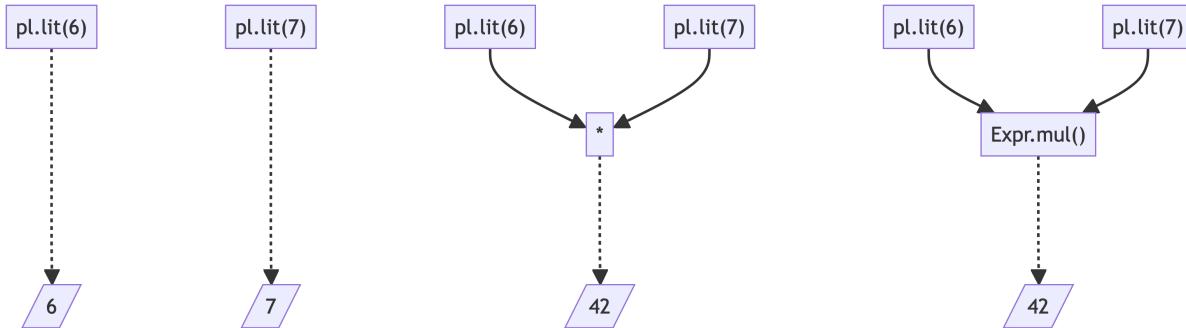


Figure 7-1. Combining expressions using inline operators or method chaining produce the same result

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")  
    )
```

```
)  
)
```

```
shape: (4, 4)
```

i	j	*	Expr.mul()
---	---	---	---
f64	i64	f64	f64
<hr/>			
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
<hr/>	
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
+	<code>Expr.add(...)</code>	Addition
-	<code>Expr.sub(...)</code>	Subtraction
*	<code>Expr.mul(...)</code>	Multiplication
/	<code>Expr.truediv(...)</code>	Division
//	<code>Expr.floordiv(...)</code>	Floor division
**	<code>Expr.pow(...)</code>	Power
%	<code>Expr.mod(...)</code>	Modulus
N/A	<code>Expr.dot(...)</code>	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_align="right")  
    .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"),  
        pl.col("i").dot(pl.col("j")).alias("i · j")  
    )  
)
```

shape: (5, 10)

i	j	i + j	i - j	i * j	i / j	i % j
---	---	---	---	---	---	---
i64	i64	i64	i64	i64	f64	
0	1	1	-1	0	0.00	
2	2	4	0	4	1.00	
2	3	5	-1	6	0.67	
-2	4	2	-6	-8	-0.50	
-2	-5	-7	3	10	0.40	

- 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.

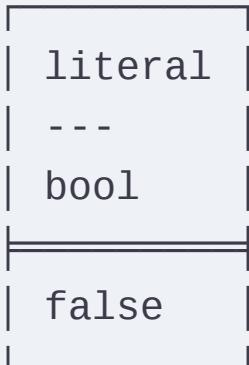
# Comparison Operations

*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)
```



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
<hr/>	
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
<	<code>Expr.lt(...)</code>	Less than
<=	<code>Expr.le(...)</code>	Less than or equal
==	<code>Expr.eq(...)</code>	Equal
>=	<code>Expr.ge(...)</code>	Greater than or equal
>	<code>Expr.gt(...)</code>	Greater than
!=	<code>Expr.ne(...)</code>	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 of the native expressions API.

Here are some things you might want to try:

- instead of `pl.col('a')` and `pl.col('b')`, use `'a'` and `'b'`
- instead of `pl.col('a') in [y, z]`, use `pl.col('a') == y | pl.col('a') == z`
- instead of `max(pl.col('a'), pl.col('b'))`, use `pl.col('a').max() | pl.col('b').max()`

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) <
```

```
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() < 1).alias("b between [-1, 1]"),
    ((1 < pl.col("b")) & (pl.col("b") < 3)).alias("b between [1, 3]")
)

```

shape: (4, 8)

a	b	a == b	a <= b	a > 0	b	c	d
---	---	---	---	---	---	---	---
f64	f64	bool	bool	bool	bool	bool	bool
-273.15	1.4142	false	true	false	true	false	true
0.0	2.7183	false	true	false	true	false	true
42.0	42.0	true	true	true	true	true	true
100.0	3.1415	false	false	true	true	false	true

- 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)  
    str_num=pl.lit("5") < pl.lit(3).cast(pl.String)  
    datetime_time=pl.datetime(1999, 1, 1).dt.time  
) .transpose(include_header=True,  
            header_name="comparison",  
            column_names=["allowed"])
```

shape: (5, 2)

comparison	allowed
---	---
str	bool
<hr/>	
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* (`~`).<sup>1</sup> 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	<code>&amp;</code>	<code>Expr.and_(...)</code>	Logical AND
Disjunction	<code> </code>	<code>Expr.or_(...)</code>	Logical OR
Negation	<code>~</code>	<code>Expr.not_()</code>	Logical NOT
Exclusive OR	<code>^</code>	<code>Expr.xor(...)</code>	Logical XOR

---

#### 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 conjunction operation results in `True` only if both expressions are `True`. The following code snippet applies six Boolean operations (the

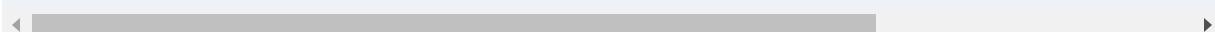
four listed in [Table 7-3](#) and two bonus operations<sup>2</sup> NAND and NOR) to all possible combinations of `p` and `q`. The output is known as a *truth table*.

```
(  
    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 NAND q"),  
        ((pl.col("p")) | (pl.col("q"))).not_().alias("p NOR q")  
    )  
)
```

shape: (4, 8)

p	q	p & q	p   q	~p	p ^ q
---	---	---	---	---	---
bool	bool	bool	bool	bool	bool
true	true	true	true	false	false
true	false	false	true	false	true

false	true	false	true	true	true
false	false	false	false	true	false



- 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.

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<sup>3</sup> operations.

Here's an example that applies the bitwise OR operator (`|`) to the values `10` and `34`, which yields, logically<sup>4</sup>, `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
&	<code>Expr.and_(...)</code>	Bitwise AND
	<code>Expr.or_(...)</code>	Bitwise OR
~	<code>Expr.not_(...)</code>	Bitwise NOT
^	<code>Expr.xor(...)</code>	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]
    }, 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	$\sim x$	$x \wedge y$
---	---	---	---	---	---
u8	u8	u8	u8	u8	u8
<hr/>					
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.

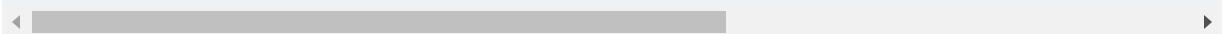
Let's take a look at the binary string representations of these integers to understand how each operator works:

```
bits.select(pl.all()).map_elements("{0:08b}").format("binary")
```

```
MapWithoutReturnDtypeWarning: Calling `map_element` without specifying `dtype` can lead to unpredictable results. Specify `dtype` or silence this warning.
```

```
shape: (6, 6)
```

x	y	x & y	x   y	~x
---	---	---	---	---
str	str	str	str	str
00000001	00000001	00000001	00000001	11111110
00000001	00000000	00000000	00000001	11111111
00000000	00000001	00000000	00000001	11111111
00000000	00000000	00000000	00000000	11111111
00000111	00000010	00000010	00000111	11111110
00001010	00100010	00000010	00101010	11111110



---

## 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
<code>pl.all_horizontal(...)</code>	Compute the bitwise AND horizontally across columns.
<code>pl.any_horizontal(...)</code>	Compute the bitwise OR horizontally across columns.
<code>pl.arctan2(...)</code>	Compute two argument arctan in radians.
<code>pl.arctan2d(...)</code>	Compute two argument arctan in degrees.
<code>pl.arg_sort_by(...)</code>	Return the row indices that would sort the columns.
<code>pl.arg_where(...)</code>	Return indices where condition evaluates True.
<code>pl.coalesce(...)</code>	Folds the columns from left to right, keeping the first non-null value.
<code>pl.concat_list(...)</code>	Horizontally concatenate columns into a single list column.
<code>pl.concat_str(...)</code>	Horizontally concatenate columns into a single string column.

Function	Description
<code>pl.corr(...)</code>	Compute the Pearson's or Spearman rank correlation between two columns.
<code>pl.cov(...)</code>	Compute the covariance between two columns/ expressions.
<code>pl.cum_fold(...)</code>	Cumulatively fold horizontally across columns with a left fold.
<code>pl.cum_reduce(...)</code>	Cumulatively reduce horizontally across columns with a left fold.
<code>pl.cum_sum_horizontal(...)</code>	Cumulatively sum all values horizontally across columns.
<code>pl.fold(...)</code>	Accumulate over multiple columns horizontally / row wise with a left fold.
<code>pl.format(...)</code>	Format expressions as a string.
<code>pl.map_batches(...)</code>	Map a custom function over multiple columns/expressions.
<code>pl.max_horizontal(...)</code>	Get the maximum value horizontally across columns.

Function	Description
<code>pl.min_horizontal(...)</code>	Get the minimum value horizontally across columns.
<code>pl.reduce(...)</code>	Accumulate over multiple columns horizontally/ row wise with a left fold.
<code>pl.rolling_corr(...)</code>	Compute the rolling correlation between two columns/ expressions.
<code>pl.rolling_cov(...)</code>	Compute the rolling covariance between two columns/ expressions.
<code>pl.struct(...)</code>	Collect columns into a struct column.
<code>pl.sum_horizontal(...)</code>	Sum all values horizontally across columns.
<code>pl.when(...)</code>	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',
    'last_name': ['Boole', 'Hopper', 'Tukey', 'Gödel'],
    '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("combined"),
    pl.struct(pl.all()).alias("struct")
)
```

```
shape: (5, 2)
```

concat_list	struct
---	---
list[str]	struct[3]
<hr/>	
["George", "Boole"]	{"George", "Boole", "England"}
["Grace", "Hopper"]	{"Grace", "Hopper", "United States"}
["John", "Tukey"]	{"John", "Tukey", "United States"}
["Kurt", "Gödel"]	{"Kurt", "Gödel", "Austria"}
["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("full_name"),  
    pl.format("{} , {} from {}",  
        "last_name", "first_name", "country").alias("full_name"))
```

```
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	...
---	---	---	---	..
i64	bool	bool	bool	..
1	true	false	true	1
7	false	false	false	1
42	true	false	true	1
101	false	false	false	1
999	true	true	true	1

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 8](#).) 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
<hr/>	
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.

- Because negation (`~`) operates on a single expression, it's not combining expressions, but we're still discussing it here. It's only logical.
- | NAND stands for NOT AND. NOR stands for NOT OR.
- | Bitwise operations are perhaps a bit niche, but this is *The Definitive Guide* after all.
- | See Douglas Adams' *The Hitchhiker's Guide to the Galaxy* for a comprehensive explanation.

[OceanofPDF.com](http://OceanofPDF.com)

# Chapter 8. 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 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](mailto: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 operations.

[Figure 8-1](#) 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).

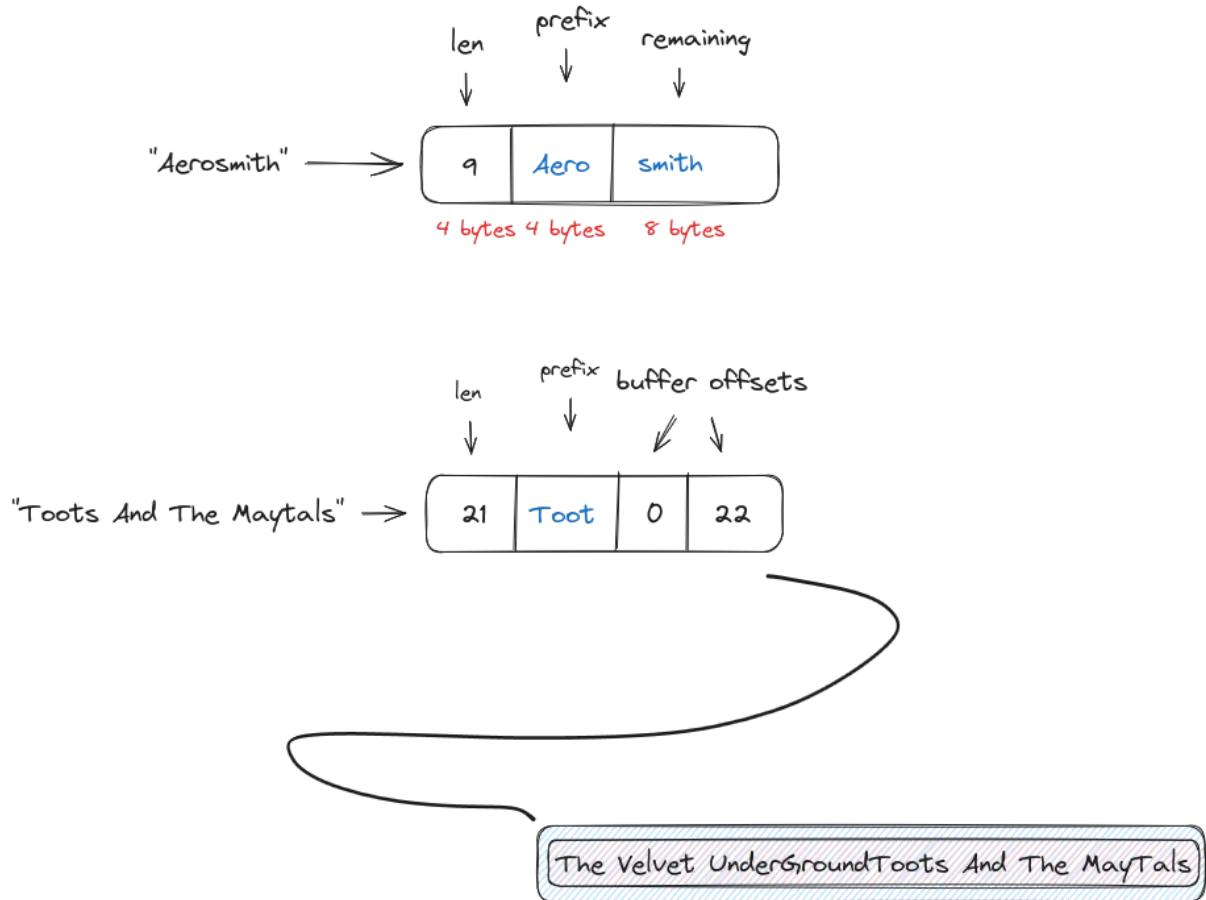


Figure 8-1. How short and long strings are stored in memory

## 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 8-1](#) allow you to convert strings to and from different data types or formats.

Table 8-1. Conversion methods for the String data type

Function or method	Description
<code>pl.Expr.str.decode(...)</code>	Decode a value using the provided encoding.
<code>pl.Expr.str.encode(...)</code>	Encode a value using the provided encoding.
<code>pl.Expr.str.json_decode(...)</code>	Parse string values as JSON.
<code>pl.Expr.str.json_extract(...)</code>	Parse string values as JSON.
<code>pl.Expr.str.json_path_match(...)</code>	Extract the first match of a JSON string with the provided JSONPath expression.
<code>pl.Expr.str.strptime(...)</code>	Convert a <code>String</code> column into a <code>Date / Datetime / Time</code> column.
<code>pl.Expr.str.to_date(...)</code>	Convert a <code>String</code> column into a <code>Date</code> column.
<code>pl.Expr.str.to_datetime(...)</code>	Convert a <code>String</code> column into a <code>Datetime</code> column.

Function or method	Description
<code>pl.Expr.str.to_decimal(...)</code>	Convert a <code>String</code> column into a <code>Decimal</code> column.
<code>pl.Expr.str.to_integer(...)</code>	Convert an <code>String</code> column into an <code>Int64</code> column with base radix <sup>a</sup> .
<code>pl.Expr.str.to_time(...)</code>	Convert a <code>String</code> column into a <code>Time</code> column.
<code>pl.Expr.str.parse_int(...)</code>	Parse integers with base radix from strings.

<sup>a</sup> 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 8-2](#) can return attributes of the string values in a column or allow you to query for certain patterns.

Table 8-2. Descriptive methods for the String data type

Function or method	Description
<code>pl.Expr.str.contains(...)</code>	Check if strings in <code>Series</code> contain a substring that matches a regex.
<code>pl.Expr.str.find(...)</code>	Return the index of the first substring in <code>Series</code> strings matching a pattern.
<code>pl.Expr.str.len_bytes()</code>	Return the length of each string as the number of bytes.
<code>pl.Expr.str.len_chars()</code>	Return the length of each string as the number of characters.
<code>pl.Expr.str.lengths()</code>	Return the number of bytes in each string.
<code>pl.Expr.str.n_chars()</code>	Return the length of each string as the number of characters.
<code>pl.Expr.str.startsWith(...)</code>	Check if string values start with a substring.

## **Manipulation**

The methods in [Table 8-3](#) allow you to manipulate the string values in a column.

Table 8-3. Manipulation methods for the String data type

Function or method	Description
<code>pl.Expr.str.concat(...)</code>	Vertically concatenate the string values in the column to a single string value.
<code>pl.Expr.str.contains_any(...)</code>	Use the <code>aho-corasick</code> algorithm to find matches.
<code>pl.Expr.str.count_matches(...)</code>	Count all successive non-overlapping regex matches.
<code>pl.Expr.str.count_matches(...)</code>	Count all successive non-overlapping regex matches.
<code>pl.Expr.str.ends_with(...)</code>	Check if string values end with a substring.
<code>pl.Expr.str.explode()</code>	Returns a column with a separate row for every string character.
<code>pl.Expr.str.extract(...)</code>	Extract the target capture group from provided patterns.
<code>pl.Expr.str.extract_all(...)</code>	Extract all matches for the given regex pattern.

Function or method	Description
<code>pl.Expr.str.extract_groups(...)</code>	Extract all capture groups for the given regex pattern.
<code>pl.Expr.str.ljust(...)</code>	Return the string left-justified in a string of length <code>length</code> .
<code>pl.Expr.str.lstrip(...)</code>	Remove leading characters.
<code>pl.Expr.str.pad_end(...)</code>	Pad the end of the string until it reaches the given <code>length</code> .
<code>pl.Expr.str.pad_start(...)</code>	Pad the start of the string until it reaches the given <code>length</code> .
<code>pl.Expr.str.replace(...)</code>	Replace first matching regex/literal substring with a new string value.
<code>pl.Expr.str.replace_all(...)</code>	Replace all matching regex/literal substrings with a new string value.
<code>pl.Expr.str.replace_many(...)</code>	Use the <code>aho-corasick</code> algorithm to replace many matches.
<code>pl.Expr.str.reverse</code>	Returns string values in reversed order.

Function or method	Description
<code>()</code>	
<code>pl.Expr.str.rjust(...)</code>	Return the string right justified in a string of length length.
<code>pl.Expr.str.rstrip(...)</code>	Remove trailing characters.
<code>pl.Expr.str.slice(...)</code>	Create subslices of the string values of a string series.
<code>pl.Expr.str.split(...)</code>	Split the string by a substring.
<code>pl.Expr.str.split_exact(...)</code>	Split the string by a substring using n splits.
<code>pl.Expr.str.splitn(...)</code>	Split the string by a substring, restricted to returning at most n items.
<code>pl.Expr.str.strip_chars(...)</code>	Remove leading and trailing characters.
<code>pl.Expr.str.strip_chars_start(...)</code>	Remove leading characters.

Function or method	Description
<code>pl.Expr.str.strip_chars_end(...)</code>	Remove trailing characters.
<code>pl.Expr.str.strip_prefix(...)</code>	Remove prefix.
<code>pl.Expr.str.strip_suffix(...)</code>	Remove suffix.
<code>pl.Expr.str.to_lowercase()</code>	Modify the strings to their lowercase equivalent.
<code>pl.Expr.str.to_titlecase()</code>	Modify the strings to their titlecase equivalent.
<code>pl.Expr.str.to_uppercase()</code>	Modify the strings to their uppercase equivalent.
<code>pl.Expr.str.zfill(...)</code>	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:

```
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() ①  
        .str.to_lowercase() ②  
        .str.replace_all("_", " ") ③  
        .alias("processed_text") ④  
)  
print(df)
```

shape: (3, 2)

raw_text	processed_text
---	---
str	str
Data Science is amazing	data science is ar
Data_analysis > Data entry	data analysis > da
Python&Polars; Fast	python&polars; fas

- 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.

- ③ You may want to replace all underscores with spaces when working with filenames or URLs.
- ④ 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) ①  
            .alias("first_5_chars"),  
        pl.col("processed_text")  
            .str.split(" ") ②  
            .list.get(0) ③  
            .alias("first_word"),  
        pl.col("processed_text")  
            .str.split(" ")  
            .list.get(1) ④  
            .alias("second_word"),
```

```
)  
)
```

shape: (3, 5)

raw_text	processed_text	first_5_chars
---	t	---
str	---	str
	str	
<hr/>		
Data Science	data science	data
is am...	is amaz...	
Data_analysis	data analysis	data
> Data...	> data...	
Python&Polars;	python&polars	pytho
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.

- ③ You get the first element from that list of strings, which will be the first word.
- ④ 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_ch
---	---	s
str	str	---
		u32

Data Science is	data science	23
amazing	is amazing	
Data_analysis >	data analysis	26
Data entry	> data entry	
Python&Polars;	python&polars;	19
Fast	fast	

- 
- ① Calculates the amount of characters in the string.
  - ② Calculates the amount of bytes that the string takes in memory.
  - ③ Counts the times the letter “a” occurs in the string.



---

**TIP**

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 bo
})
```

```
hashtag_regex = r"#(\w+)" ❶

df.with_columns(
    pl.col("post").str.extract_all(hashtag_regex)
)
```

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", "
```

```
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 8-4. Methods for the Categorical data type

Function or method	Description
<code>Expr.cat.get_categories()</code>	Get the categories stored in this data type.
<code>Expr.cat.set_ordering(...)</code>	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", "value1"],  
     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
<hr/>	
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 re-encoding is done to perform this merge operation. Cache or an Enum type if the categories are known.
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:

```
with pl.StringCache():
    df1 = pl.DataFrame(
        {
            "categorical_column": ["value3", "value2"],
            "other": ["a", "b", "c"],
```

```

        },
        schema={"categorical_column": pl.Categorical(
    )
df2 = pl.DataFrame(
{
    "categorical_column": ["value2", "value3"],
    "other": ["d", "e", "f"],
},
schema={"categorical_column": pl.Categorical(
)
# Even outside the global string cache's scope, you
# 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_c
```

```
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  
    )  
    .sort(by="categorical_lexical")  
)
```

shape: (3, 2)

	categorical_physical
categorical_lexical	---
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"]
)

cat_series = pl.Series(
    ["Polar", "Panda", "Brown", "Brown", "Polar"]
)
```

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 8-5](#).

Table 8-5. Temporal data types

Data type	Description	Example	Storage
Date	Represents a calendar date without a time of day.	Birthdays	<code>int32</code> 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	<code>int64</code> 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 <code>timedelta</code> in Python.	Elapsed time between two events	<code>int64</code> that is created when subtracting Date/Datetime.

Data type	Description	Example	Storage
Time	Focuses only on time of day.	Scheduling of daily tasks.	<code>int64</code> 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 8-6. Methods for conversion to and from other data types.

Function or method	Description
<code>Expr.dt.cast_timeunit(...)</code>	Cast the underlying data to another time unit.
<code>Expr.dt.strftime(...)</code>	Convert a Date / Time / Datetime column into a String column with the given format.
<code>Expr.dt.to_string(...)</code>	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 8-7. Methods for describing temporal data.

Function or method	Description
<code>Expr.dt.base_utc_offset()</code>	Base offset from UTC.
<code>Expr.dt.date()</code>	Extract date from <code>date(time)</code> .
<code>Expr.dt.datetime()</code>	Return <code>Datetime</code> .
<code>Expr.dt.day()</code>	Extract day from underlying <code>Date</code> representation.
<code>Expr.dt.days()</code>	Extract the total days from a <code>Duration</code> type.
<code>Expr.dt.dst_offset()</code>	Additional offset currently in effect (typically due to daylight saving time).
<code>Expr.dt.epoch(...)</code>	Get the time passed since the Unix epoch in the given time unit.
<code>Expr.dt.hour()</code>	Extract the hour from underlying <code>Datetime</code> representation.
<code>Expr.dt.hours()</code>	Extract the total hours from a <code>Duration</code> type.

Function or method	Description
<code>Expr.dt.is_leap_year()</code>	Determine whether the year of the underlying date is a leap year.
<code>Expr.dt.iso_year()</code>	Extract ISO year from underlying <code>Date</code> representation.
<code>Expr.dt.microsecond()</code>	Extract microseconds from underlying <code>DateTime</code> representation.
<code>Expr.dt.microseconds()</code>	Extract the total microseconds from a <code>Duration</code> type.
<code>Expr.dt.millisecond()</code>	Extract milliseconds from underlying <code>DateTime</code> representation.
<code>Expr.dt.milliseconds()</code>	Extract the total milliseconds from a <code>Duration</code> type.
<code>Expr.dt.minute()</code>	Extract minutes from underlying <code>DateTime</code> representation.
<code>Expr.dt.minutes()</code>	Extract the total minutes from a <code>Duration</code> type.
<code>Expr.dt.month()</code>	Extract the month from underlying <code>Date</code>

Function or method	Description
	representation.
<code>Expr.dt.nanosecond()</code>	Extract nanoseconds from underlying <code>DateTime</code> representation.
<code>Expr.dt.nanoseconds()</code>	Extract the total nanoseconds from a <code>Duration</code> type.
<code>Expr.dt.ordinal_day()</code>	Extract ordinal day from underlying <code>Date</code> representation.
<code>Expr.dt.quarter()</code>	Extract quarter from underlying <code>Date</code> representation.
<code>Expr.dt.second(...)</code>	Extract seconds from underlying <code>DateTime</code> representation.
<code>Expr.dt.seconds()</code>	Extract the total seconds from a <code>Duration</code> type.
<code>Expr.dt.time()</code>	Extract time.
<code>Expr.dt.timestamp(...)</code>	Return a timestamp in the given time unit.

Function or method	Description
<code>Expr.dt.total_days()</code>	Extract the total days from a <code>Duration</code> type.
<code>Expr.dt.total_hours()</code>	Extract the total hours from a <code>Duration</code> type.
<code>Expr.dt.total_microseconds()</code>	Extract the total microseconds from a <code>Duration</code> type.
<code>Expr.dt.total_milliseconds()</code>	Extract the total milliseconds from a <code>Duration</code> type.
<code>Expr.dt.total_minutes()</code>	Extract the total minutes from a <code>Duration</code> type.
<code>Expr.dt.total_nanoseconds()</code>	Extract the total nanoseconds from a <code>Duration</code> type.
<code>Expr.dt.total_seconds()</code>	Extract the total seconds from a <code>Duration</code> type.
<code>Expr.dt.year()</code>	Extract year from underlying <code>Date</code> representation.

## **Manipulation**

The following methods allow you to manipulate temporal data.

Table 8-8. Methods for manipulating temporal data.

Function or method	Description
<code>Expr.dt.replace_time_zone(...)</code>	Replace time zone for an expression of type <code>Datetime</code> .
<code>Expr.dt.combine(...)</code>	Create a naive <code>Datetime</code> from an existing <code>Date / Datetime</code> expression and a <code>Time</code> .
<code>Expr.dt.month_start()</code>	Roll backward to the first day of the month.
<code>Expr.dt.month_end()</code>	Roll forward to the last day of the month.
<code>Expr.dt.offset_by(...)</code>	Offset this date by a relative time offset.
<code>Expr.dt.round(...)</code>	Divide the <code>Date / DateTime</code> range into buckets.
<code>Expr.dt.truncate(...)</code>	Divide the <code>Date / DateTime</code> range into buckets.
<code>Expr.dt.week()</code>	Extract the week from the underlying <code>Date</code> representation.

Function or method	Description
<code>Expr.dt.weekday()</code>	Extract the week day from the underlying <code>Date</code> representation.
<code>Expr.dt.with_time_unit(...)</code>	Set time unit of an expression of type <code>Date</code> , <code>Time</code> or <code>Duration</code> .
<code>Expr.dt.convert_time_zone(...)</code>	Convert to given time zone for an expression of type <code>DateTime</code> .

## 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_date=True)
```

```
shape: (18_476, 8)
```

symbol	date	open	...	close
---	---	---		---
str	date	f64		f64
ASML	1999-01-04	11.765625	...	12.140625
ASML	1999-01-05	11.859375	...	13.96875
ASML	1999-01-06	14.25	...	16.875
ASML	1999-01-07	14.742188	...	16.851562
ASML	1999-01-08	16.078125	...	15.796875
...	...	...	...	...
TSM	2023-06-26	102.019997	...	100.110002
TSM	2023-06-27	101.150002	...	102.080002
TSM	2023-06-28	100.5	...	100.919997
TSM	2023-06-29	101.339996	...	100.639997
TSM	2023-06-30	101.400002	...	100.919997

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")
)
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
<hr/>		
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(  
    {  
        "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")
        .alias("parsed_data")
    ).with_columns(
        pl.col("parsed_data")
        .dt.convert_time_zone("Europe/Amsterdam")
        .alias("converted_data")
    )
)
print(df)

```

shape: (4, 3)

utc_mixed_offset_data	parsed_data
---	---
str	datetime[μs, UTC]
2021-03-27T00:00:00+0100	2021-03-26 23:00:00

2021-03-28T00:00:00+0100	2021-03-27 23:00:00
2021-03-29T00:00:00+0200	2021-03-28 22:00:00
2021-03-30T00:00:00+0200	2021-03-29 22:00:00

◀ ▶

- 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 9](#) 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 8-9. Methods for the List type

Function or method	Description
<code>Expr.list.all()</code>	Evaluate whether all Boolean values in a list are true.
<code>Expr.list.any()</code>	Evaluate whether any Boolean value in a list is true.
<code>Expr.list.drop_nuls()</code>	Drop all null values in the list.
<code>Expr.list.arg_max()</code>	Retrieve the index of the maximum value in every sublist.
<code>Expr.list.arg_min()</code>	Retrieve the index of the minimal value in every sublist.
<code>Expr.list.concat(...)</code>	Concatenate the arrays in a Series in linear time.
<code>Expr.list.contains(...)</code>	Check if sublists contain the given item.

Function or method	Description
<code>Expr.list.count_matches(...)</code>	Count how often the value produced by element occurs.
<code>Expr.list.count_matches(...)</code>	Count how often the value produced by element occurs.
<code>Expr.list.diff(...)</code>	Calculate the first discrete difference between shifted items of every sublist.
<code>Expr.list.eval(...)</code>	Run any Polars expression against the list's elements.
<code>Expr.list.explode()</code>	Return a column with a separate row for every list element.
<code>Expr.list.first()</code>	Get the first value of the sublists.
<code>Expr.list.gather(...)</code>	Take sublists by multiple indices.
<code>Expr.list.get(...)</code>	Get the value by index in the sublists.
<code>Expr.list.head(...)</code>	Slice the first n values of every sublist.
<code>Expr.list.join(...)</code>	Join all string items in a sublist and place a separator between them.

Function or method	Description
<code>Expr.list.last()</code>	Get the last value of the sublists.
<code>Expr.list.len()</code>	Return the number of elements in each list.
<code>Expr.list.lengths()</code>	Return the number of elements in each list.
<code>Expr.list.max()</code>	Compute the max value of the lists in the array.
<code>Expr.list.mean()</code>	Compute the mean value of the lists in the array.
<code>Expr.list.min()</code>	Compute the min value of the lists in the array.
<code>Expr.list.reverse()</code>	Reverse the arrays in the list.
<code>Expr.list.sample(...)</code>	Sample from this list.
<code>Expr.list.set_difference(...)</code>	Compute the set difference between the elements in this list and the elements of

Function or method	Description
	other.
<code>Expr.list.set_intersection(...)</code>	Compute the set intersection between the elements in this list and the elements of other.
<code>Expr.list.set_symmetric_difference(...)</code>	Compute the set symmetric difference between the elements in this list and the elements of other.
<code>Expr.list.set_union(...)</code>	Compute the set union between the elements in this list and the elements of other.
<code>Expr.list.shift(...)</code>	Shift list values by the given number of indices.
<code>Expr.list.slice(...)</code>	Slice every sublist.
<code>Expr.list.sort(...)</code>	Sort the lists in this column.
<code>Expr.list.sum()</code>	Sum all the lists in the array.
<code>Expr.list.tail(...)</code>	Slice the last n values of every sublist.
<code>Expr.list.take(...)</code>	Take sublists by multiple indices.

Function or method	Description
<code>Expr.list.to_array (...)</code>	Convert a <code>List</code> column into an <code>Arra y</code> column with the same inner data type.
<code>Expr.list.to_struct (...)</code>	Convert the <code>Series</code> of type <code>List</code> to a <code>Series</code> of type <code>Struct</code> .
<code>Expr.list.unique (...)</code>	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]]
})
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
<hr/>		
[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

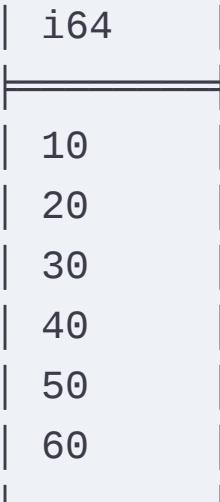
[60]	[true]	true
------	--------	------

- 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)		
<table border="1"><tr><td>values</td></tr><tr><td>---</td></tr></table>	values	---
values		
---		



# 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 makes 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 8-10. Methods for the Array type

Function or method	Description
<code>Expr.arr.max()</code>	Compute the max values of the sub-arrays.
<code>Expr.arr.min()</code>	Compute the min values of the sub-arrays.
<code>Expr.arr.median()</code>	Compute the median of the values of the sub-arrays.
<code>Expr.arr.sum()</code>	Compute the sum values of the sub-arrays.
<code>Expr.arr.std(...)</code>	Compute the std of the values of the sub-arrays.
<code>Expr.arr.to_list()</code>	Convert an <code>Array</code> column into a <code>List</code> column with the same inner data type.
<code>Expr.arr.unique(...)</code>	Get the unique/distinct values in the array.
<code>Expr.arr.var(...)</code>	Compute the variance of the values of the sub-arrays.
<code>Expr.arr.all()</code>	Evaluate whether all Boolean values are true for every subarray.

Function or method	Description
<code>Expr.arr.any()</code>	Evaluate whether any Boolean value is true for every subarray.
<code>Expr.arr.sort(...)</code>	Sort the arrays in this column.
<code>Expr.arr.reverse()</code>	Reverse the arrays in this column.
<code>Expr.arr.arg_min()</code>	Retrieve the index of the minimal value in every sub-array.
<code>Expr.arr.arg_max()</code>	Retrieve the index of the maximum value in every sub-array.
<code>Expr.arr.get(...)</code>	Get the value by index in the sub-arrays.
<code>Expr.arr.first()</code>	Get the first value of the sub-arrays.
<code>Expr.arr.last()</code>	Get the last value of the sub-arrays.
<code>Expr.arr.join(...)</code>	Join all string items in a sub-array and place a separator between them.
<code>Expr.arr.explode()</code>	Returns a column with a separate row for every array element.

Function or method	Description
<code>Expr.arr.contains(s(...))</code>	Check if sub-arrays contain the given item.
<code>Expr.arr.count_matches(...)</code>	Count how often the value produced by element occurs.
<code>Expr.arr.to_struct(...)</code>	Convert the <code>Series</code> of type <code>Array</code> to a <code>Series</code> of type <code>Struct</code> .
<code>Expr.arr.shift(...)</code>	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.argmax()  
        .alias("warmest_weekday")  
    )  
)
```

shape: (3, 5)

location	temperatures	median	max	warmest_weekday
---	---	---	---	---
str	array[i64, 7]	f64	i64	u32
<hr/>				
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 8-11. Methods for the struct type

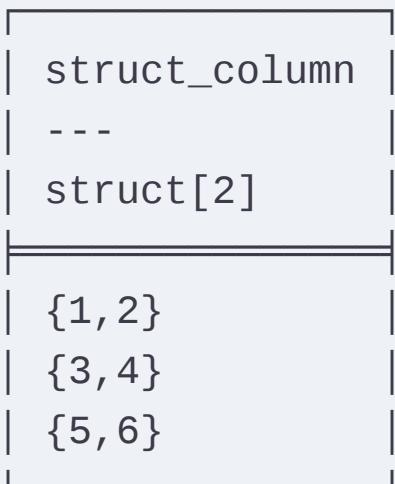
Function or method	Description
<code>Expr.struct.field(...)</code>	Retrieve a Struct field as a new Series.
<code>Expr.struct.json_encode()</code>	Convert this struct to a string column with json values.
<code>Expr.struct.rename_fields(...)</code>	Rename the fields of the struct.

## 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)
```

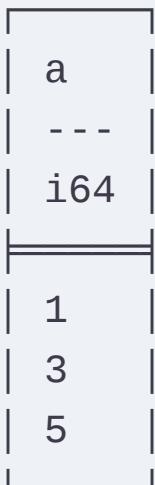
```
shape: (3, 1)
```



You can retrieve values from a struct using the `field` method:

```
df.select(pl.col("struct_column").struct.field("a"))
```

```
shape: (3, 1)
```



To return multiple columns, you can use the `unnest` method. Note that the `unnest` method is not part of the struct namespace and should be called on the DataFrame/LazyFrame/Series level.

```
df = df.unnest("struct_column")
print(df)
```

shape: (3, 2)

a	b
---	---
i64	i64
<hr/>	
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", "apple", "orange", "orange"]})
```

```
})  
print(df)
```

```
shape: (6, 1)
```

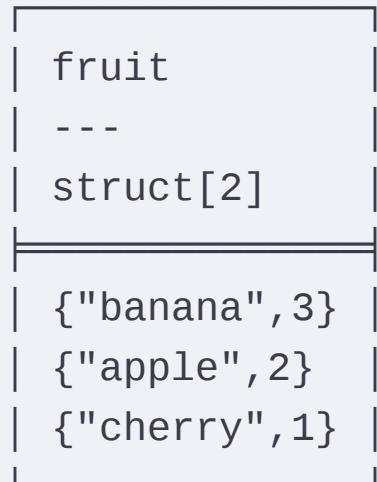
fruit
---
str
cherry
apple
banana
banana
apple
banana

You can count the number of occurrences per unique element in the `fruit` column using `value_counts()`:

```
print(  
    df  
    .select(  
        pl.col("fruit")  
        .value_counts(sort=True))
```

```
)  
)
```

```
shape: (3, 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:

```
print(  
    df.select(  
        pl.col("fruit")  
        .value_counts(sort=True)  
)
```

```
.unnest("fruit")  
)
```

shape: (3, 2)

fruit	count
---	---
str	u32
<hr/>	
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.

[OceanofPDF.com](http://OceanofPDF.com)

# Chapter 9. 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 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](mailto: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 9-1](#) lists the methods you can use out of the box to analyze data.

Table 9-1. The available methods in the Group By context

Method	Description
<code>pl.DataFrame.groupby().__iter__()</code>	Allows iteration over the groups of the group by operation.
<code>pl.DataFrame.groupby().agg(...)</code>	Compute aggregations for each group of a group by operation.
<code>pl.DataFrame.groupby().all()</code>	Aggregate the groups into Series.
<code>pl.DataFrame.groupby().apply(...)</code>	Apply a custom/user-defined function (UDF) over the groups as a sub-DataFrame.
<code>pl.DataFrame.groupby().first()</code>	Aggregate the first values in the group.
<code>pl.DataFrame.groupby().head(...)</code>	Get the first n rows of each group.
<code>pl.DataFrame.groupby().last()</code>	Aggregate the last values in the group.
<code>pl.DataFrame.groupby().len()</code>	Return the number of rows in each group.

Method	Description
<code>pl.DataFrame.groupby().map_groups(...)</code>	Apply a custom/user-defined function (UDF) over the groups as a sub-DataFrame.
<code>pl.DataFrame.groupby().max()</code>	Reduce the groups to the maximal value.
<code>pl.DataFrame.groupby().mean()</code>	Reduce the groups to the mean values.
<code>pl.DataFrame.groupby().median()</code>	Return the median per group.
<code>pl.DataFrame.groupby().min()</code>	Reduce the groups to the minimal value.
<code>pl.DataFrame.groupby().n_unique()</code>	Count the unique values per group.
<code>pl.DataFrame.groupby().quantile(...)</code>	Compute the quantile per group.
<code>pl.DataFrame.groupby().sum()</code>	Reduce the groups to the sum.
<code>pl.DataFrame.groupby().tail(n)</code>	Get the last n rows of each group.

Method	Description
( ... )	

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"]
2021	["Goldband - Noodgeval", "Bankzitters -"]

2020	[ "DI-RECT - Soldier On", "Miss Montreal
2019	[ "Danny Vera - Roller Coaster", "Floor
2018	[ "Lady Gaga & Bradley Cooper - Shallow",
...	...
1960	[ "Etta James - At Last", "Shadows - Apac
1959	[ "Jacques Brel - Ne Me Quitte Pas", "Elv
1958	[ "Chuck Berry - Johnny B. Goode", "Ella
1957	[ "Johnny Cash - I Walk The Line", "Elvis
1956	[ "Elvis Presley - Love Me Tender", "Elv:

- 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
<hr/>			
2022	179	Multicolor	Son M...
2022	370	Je Blik Richting Mij	Bankz...
2022	395	L'enfer	Strom...
2021	55	Noodgeval	Goldb...
2021	149	Stapelgek	Bankz...
2021	210	Dat Heb Jij Gedaan	Meau...
2020	19	Soldier On	DI-REC...
2020	38	Door De Wind	Miss N...
2020	77	Impossible (Orchestr...	Nothing...

- 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.

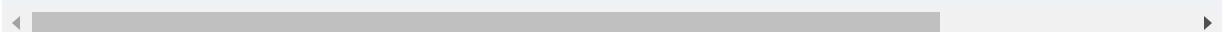
In the same vein, you can get the lowest 3 positions per year of release for the 3 most recent years of release, using the `tail()` method.

```
(  
    top2000  
    .group_by("jaar", maintain_order=True)  
    .tail(3)  
    .sort("jaar", descending=True)  
    .head(9)  
)
```

shape: (9, 4)

jaar	positie	titel	artiest
---	---	---	---
i64	i64	str	str

2022	1391	De Diepte	S10
2022	1688	Zeit	Rammstein
2022	1716	THE LONELIEST	Måneskin
2021	1865	Bon Gepakt	Donnie & Re
2021	1978	Hold On	Armin van Bu
2021	2000	Drivers License	Olivia Rodríg
2020	1824	Smoorverliefd	Snelle
2020	1879	The Business	Tiesto
2020	1902	Levitating	Dua Lipa ft



#### 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)
```

	len
artiest	len
---	---
str	u32
<hr/>	
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", "
```

```
.group_by("Product_Category", "Sub_Category")
.max()
.sort("Unit_Price", descending=True) ❸
.head(10)
)
```

shape: (10, 3)

Product_Category	Sub_Category	Unit_Price
---	---	---
str	str	i64
<hr/>		
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
Seniors (64+)	13.530137
Youth (<25)	12.124018
Adults (35-64)	12.045303
Young Adults (25-34)	11.560899

---	---
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, ... 950]
Germany	[160, 53, ... 746]	[295, 98, ... 980]
United Kingdom	[1053, 1053, ... 112]	[1728, 1728, ... 112]
Australia	[1366, 1188, ... 655]	[2401, 2401, ... 655]
United States	[524, 407, ... 542]	[929, 729, ... 542]
France	[427, 427, ... 655]	[787, 787, ... 655]

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 Tran...")  
    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,
Germany	[160, 53, ... 746]	[295,
Canada	[590, 590, ... 630]	[950,
United Kingdom	[1053, 1053, ... 112]	[1728,
France	[427, 427, ... 655]	[787,
Australia	[1366, 1188, ... 655]	[2401,

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")  
        pl.col("Revenue").sum().alias("Total Revenue")  
        pl.col("Revenue").mean().alias("Average Revenue")  
    )  
)
```

shape: (6, 5)

Country	Total Profit	Average Profit per T...
---	---	---
str	i64	f64
Germany	3359995	302.756803
Canada	3717296	262.187615
United States	11073644	282.447687
United Kingdom	4413853	324.071439
Australia	6776030	283.089489

France	2880282	261.891435
--------	---------	------------

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	Total Profit	Total Revenue
---	---	---
str	i64	i64
Australia	6776030	21302059

France	2880282	8432872
United Kingdom	4413853	10646196
Germany	3359995	8978596
Canada	3717296	7935738
United States	11073644	27975547

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")
)
)
```

```
shape: (6, 3)
```

Country	Profit > 1000	Number of Transactions with Profit
---	---	---
str	list[bool]	u32
Canada	[false, false, ... false]	868
Australia	[true, true, ... false]	1233
United States	[false, false, ... false]	2623
Germany	[false, false, ... false]	659
United Kingdom	[true, true, ... false]	788
France	[false, false, ... false]	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")  
  
(  
    df  
        .select("Country", "Profit")  
        .group_by("Country")  
        .agg(  
            custom_agg(pl.col("Profit"))  
        )  
)
```

shape: (6, 3)

Country	Profit > 1000	Number
---	---	---
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:

```
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
<hr/>	
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 expression.  
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 |
|---|-------|
| 1 | 2     |
| 2 | 3     |
| 3 | 4     |
| 4 | 5     |


```

These suggestions are usually a good starting point to find a more efficient way to implement your logic!

---

---

### @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:

```
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],
    "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:

```
from sklearn.preprocessing import StandardScaler

def scale_temperature(group):
    scaler = StandardScaler()
    scaled_values = scaler.fit_transform(group[['temperature']])
    return group.with_columns(pl.Series(values=scaled_values, name='scaled_feature'))

df = pl.DataFrame({
    "group": ["USA", "USA", "USA", "USA", "NL", "NL", "NL"],
    "temperature": [32, 50, 70, 65, 0, 10, 15]
})

result = df.groupby("group").map_groups(scale_temperature)
result
```

shape: (7, 3)

group	temperature	scaled_feature
---	---	---
str	i64	f64
<hr/>		
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.

```
df = pl.DataFrame({
    "group": ["USA", "USA", "USA", "USA", "NL", "NL"],
    "temperature": [32, 50, 70, 65, 0, 10, 15]
})

for group in df.groupby(["group"]):
    print(group)
```

```
(('NL',), shape: (3, 2)
```

group	temperature
NL	0
NL	10
NL	15

```

| --- | --- |
| 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), ①
        function=lambda acc, x: acc + x, ②
        exprs=pl.col("*)") ③
    ).alias("sum")
)
```

shape: (1, 4)

col1	col2	col3	sum
---	---	---	---
i64	i64	i64	i64
<hr/>			
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:

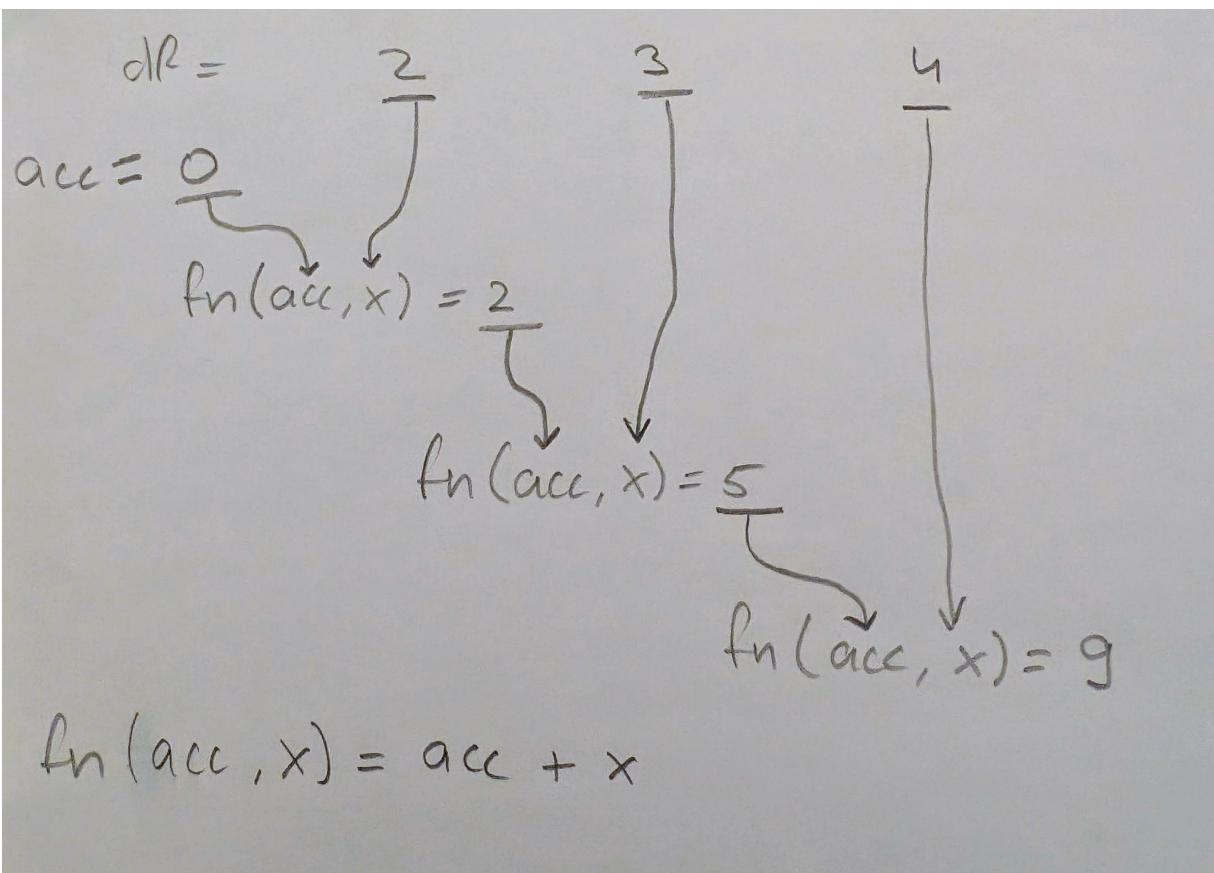


Figure 9-1. How a fold function is executed.

```

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
<hr/>			
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:

```
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_su
---	---	---	---
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
---	---	---
i64	str	str
<hr/>		
2013	Stromae	Papaoutai
1969	John Denver	Leaving On A Jet Plane
1971	Led Zeppelin	Immigrant Song
2009	Anouk	For Bitter Or Worse
2015	Snollebollekes	Links Rechts
1984	Alphaville	Forever Young
1977	ABBA	Take A Chance On Me
1975	Rod Stewart	Sailing
1986	Metallica	Master Of Puppets
2005	Alderliefste & Ramse...	Laat Me/Vivre

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.

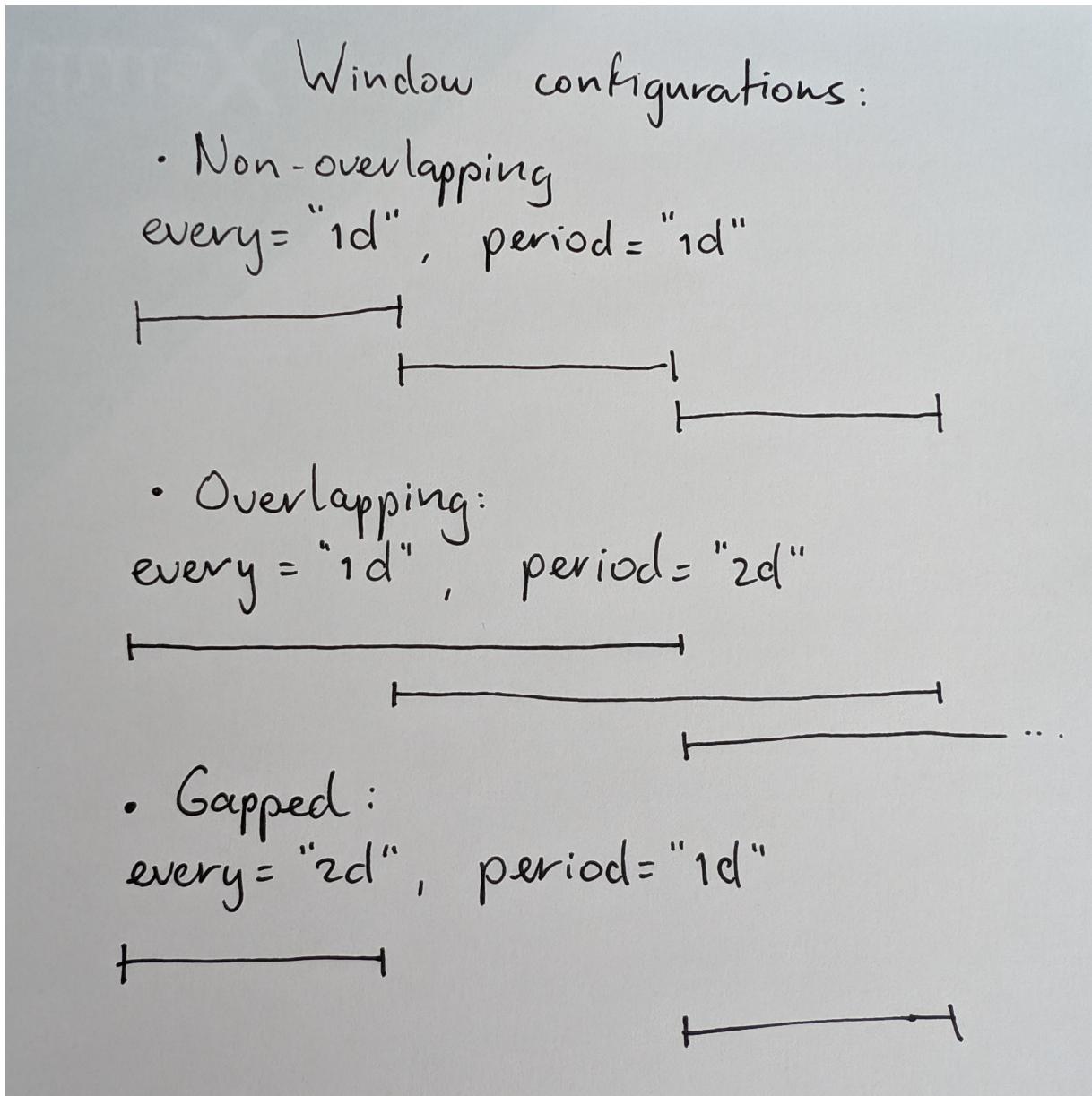
The `every`, `period` and `offset` arguments can be specified using the following strings:

Table 9-2. 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 9-2](#).



*Figure 9-2. 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 9-3](#).

Table 9-3. Closed interval options

Parameter	Description	Interval	Contains a	Contains b
<b>left</b>	The lower bound is inclusive, the upper bound is exclusive.	$[a, b)$	✓	✗
<b>right</b>	The lower bound is exclusive, the upper bound is inclusive.	$(a, b]$	✗	✓
<b>both</b>	Both the lower and upper bounds are inclusive.	$[a, b]$	✓	✓
<b>none</b>	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.

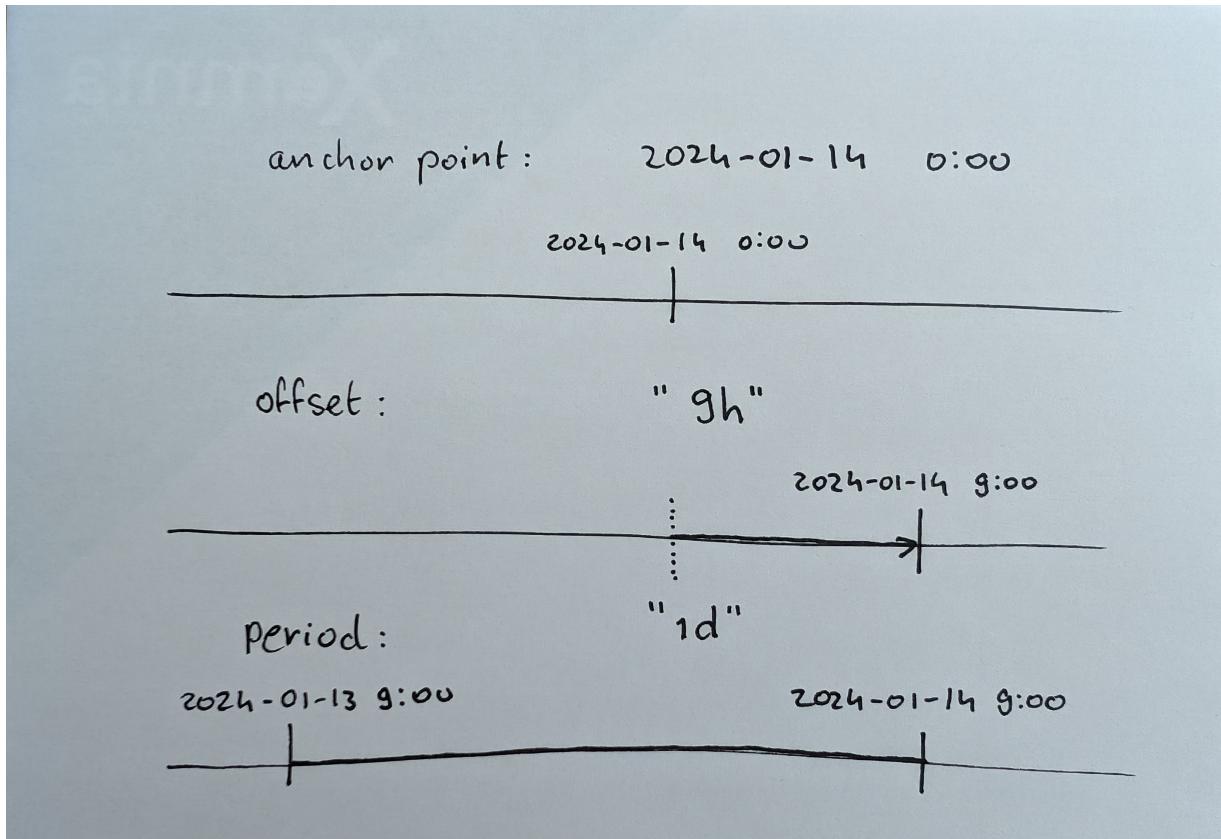
### *closed*

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 9-3](#).



*Figure 9-3. 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()

df = pl.DataFrame({
    "date": dates_repeated,
    "store": ["Store A", "Store B"] * dates.len(),
    "sales": [
        200, 150, 220, 160, 250, 180, 270, 190, 210,
        170, 220, 180, 240, 190, 250, 200, 230
    ]
}).set_sorted("date", "store") ❹
```

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.

- ③ Filter out weekend days.
- ④ Repeat the dates for the two weeks and sort them.
- ⑤ 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.

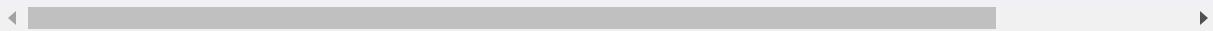
```
result = (
    df.rolling( ❶
        index_column="date",
        period="7d",
        group_by="store",
        check_sorted=False, ❷
    ).agg( ❸
        pl.sum("sales").alias("sum_of_last_7_days"
    )
)

final_df = df.join(result, on=["date", "store"])

final_df
```

shape: (20, 4)

date	store	sales	sum_of_last_7_days_sales
---	---	---	---
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.

# 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.

[OceanofPDF.com](http://OceanofPDF.com)