Joram Mutenge

# Deep Analysis *with* Polars

## Transforming and Visualizing Data for Insights

*foreword*

Arnaud Vennin, PhD

# Deep Analysis with Polars

*Transforming and Visualizing Data for Insights*

***Joram Mutenge***

**Deep Analysis with Polars**
by Joram Mutenge

# Preface

I first used Polars in 2022, while I was a graduate student at the University of Illinois Urbana-Champaign. Up to that point, I had spent years using pandas for data analysis, ever since college, really. So when I came across a faster and more modern library, I decided to give it a try. I was hooked on the first day and haven't looked back since.

Polars was introduced to the public in June 2020, though version 1.0 didn't arrive until July 2024. You could say I was an early adopter.

After graduate school, I began working as a data analyst, focusing primarily on forecast data. When I joined my company, I introduced Polars to the team, and its simplicity quickly won everyone over. Today, nearly all our analysis workflows are built in Polars.

When I first started with Polars, there weren't many resources to learn from. I read the Polars documentation[1] cover-to-cover, testing each example to see how far I could take my analysis. When the documentation didn't have the answers, I asked questions from the community in the Polars Discord channel[2]. But mostly, I learned by experimenting. I would think in pandas (the tool I knew best) and then translate that logic into Polars. Over time, that thinking flipped. Now, I think and work in Polars first.

Data professionals getting into Polars today are in a much better position. A few books have been written as the library's popularity has grown, and more tutorials and examples are available online. Still, many of these resources are introductory or surface-level. That's great for getting started, but not as helpful when you're deep into analytical work. Few, if any, focus on using Polars for advanced data analysis.

This book aims to fill that gap. My goal is to provide a practical reference for solving complex analysis problems with Polars. And, hopefully, to inspire new ways of uncovering insights from data using techniques you might not have seen before.

## Writing Conventions Used

The typographical conventions used throughout this book are as follows:

*Italic*

For new terms, dataframe column names, filenames, and file extensions and URLs.

`Constant width`

For code text, Python keywords, and Polars methods and expressions.

**Tip**
For a tip or suggestion.

**Note**
For a general note.

**Warning**
For a warning or caution.

**Expand Your Knowledge**
For additional information to enhance your knowledge.

## Polars Version

This book uses Polars 1.44.2

To ensure that your results match the output shown in the book, make sure you are running the same Polars version. Differences between versions can lead to changes in behavior, performance, or output, which may cause your results to differ from the examples presented in this book.

## Code in this Book

There is a public official GitHub repository[3] for **Deep Analysis with Polars** that includes all the code examples in this book. The code is organized by chapter for quick and easy reference. This repository serves as a practical companion for readers and allows for active collaboration through GitHub.

## Data in this Book

One of my biggest pet peeves is when examples in a book use randomly generated numbers to demonstrate data analysis. That's why the majority of datasets used in this book are real-world datasets, showing how Polars can be applied to actual problems. I've also included sources where you can download these datasets. You can download this zip file[4] to get the exact datasets used in this book.

More importantly, I've carefully selected a variety of datasets to highlight how versatile Polars is for analyzing different types of data.

## Acknowledgments

This book would not have been possible without the thoughtful feedback and encouragement I received on the first draft. I am especially grateful to Arnaud Vennin, PhD, Phil Quinn, and the many others who took the time to review the manuscript. Your valuable insights, suggestions, and constructive critiques challenged me to refine the material and ultimately produce a stronger and more polished book.

1. *https://docs.pola.rs/api/python/stable/reference/index.html*

2. *https://discord.gg/GUEXJPDBwr*

3. *https://github.com/jorammutenge/deep-analysis-with-polars-book*

4. *https://drive.google.com/file/d/1ETxGtLYz7kVgBwbk2JaXKelOvnQr-BOf/view?usp=sharing*

# Foreword

For the last decade, the data transformation landscape has been dominated by two tools at opposite ends of the spectrum, namely Apache Spark and pandas. On the one hand, Apache Spark is a distributed computing tool well-suited for gigabytes or petabytes of data, but it comes with the overhead of managing clusters. On the other hand, pandas processes data on a single machine and is limited by the available memory.

Recently, a new generation of data transformation tools has emerged, synthesizing strengths of both small- and large-scale processing by integrating innovations from across the spectrum. Among them, one has gained significant momentum: Polars.

As a data engineer, I started using Polars three years ago as a replacement for pandas on small input data pipelines. What struck me immediately was not just the processing speed, but the expressiveness of the syntax. After years of using Apache Spark, I knew right away Polars was different. It was a true game-changer.

Over the last three years, I've watched Polars evolve and become more efficient. The streaming engine, once limited to datasets slightly larger than RAM, has been redesigned. It can now handle terabytes of data. I gave it a spin and was thrilled by the results. I believe Polars is set to be a major tool in the years to come.

Since I discovered Polars, I have stayed active in the community (reporting bugs, writing an article, and giving presentations to convince coworkers to try it). That is how I met Joram. He had written the first draft of this book and was looking for reviewers. The subject looked promising, so I accepted immediately.

As a data engineer, I occasionally perform data analysis, but it isn't my main focus. Reading the manuscript, I discovered many benefits of the practice. Joram breaks down every aspect of the field using realistic, easy-to-understand use cases. This makes it easy to understand every concept. This book shows that any data practitioner can use Polars and that Polars can replace pandas even for traditional analysis tasks.

Whether you want to learn the basics of data analysis in Polars or build production-

grade pipelines, this book is for you.

I hope Joram's book gets the recognition it deserves.

*— Arnaud Vennin, PhD*
*Senior Data Engineer, Decathlon Digital*

# Table of Contents

# CHAPTER 1

# Analysis with Polars

> *Polars was needed because the current state of dataframe implementations didn't use the knowledge available in database and query engine research.*
>
> — Ritchie Vink

If you're reading this book, you're likely interested in learning how to analyze data using Polars. You might already have experience with tools such as SQL, pandas, or Spark, or you might be completely new to data analysis. Regardless of your background, this chapter establishes the foundation for the topics covered throughout the book and ensures we share a common vocabulary. I'll begin by explaining what data analysis is, then introduce Polars: what it is, why it's gaining popularity, how it compares to other tools, and how it fits into the broader process of data analysis.

## What Is Data Analysis?

Data analysis is the process of transforming raw data into meaningful insights that guide decisions, drive innovation, and improve performance across industries. Its growing importance is reshaping the job market and redefining how organizations operate.

Data analysis refers to the systematic examination of data to uncover patterns, trends, and relationships. It involves techniques such as statistical modeling, data visualization, and computational algorithms to interpret large volumes of information. Whether it's customer behavior, financial performance, or operational efficiency, data analysis helps convert complex datasets into actionable knowledge. This process is foundational in fields ranging from healthcare and finance to marketing and public policy.

The importance of data analysis has surged in recent years due to the exponential growth of digital data. With the rise of social media, e-commerce, and IoT devices, organizations now collect vast amounts of information daily. Without analysis, this data remains untapped potential. Businesses use data analytics to predict market trends, personalize customer experiences, and optimize supply chains. In a competitive landscape, the ability to make informed decisions quickly is a strategic advantage, making data analysis indispensable.

In today's job market, roles associated with data analysis are among the most in-demand. Positions such as data analyst, business intelligence analyst, data scientist, analytics engineer, and machine learning engineer are increasingly common across industries. These roles require proficiency in tools like SQL, R, pandas, and now Polars. Business Intelligence (BI) tools like Tableau and Power BI are also used. Employers value candidates who not only possess strong data manipulation skills but also excel at translating complex findings into clear, compelling insights for stakeholders. In my professional experience, I've witnessed excellent analyses fail to gain traction with senior leadership simply because they were presented ineffectively. The ability to communicate data-driven insights in a way that resonates with decision-makers is just as critical as the analysis itself. The demand for data analysis skills reflects a broader shift toward data-driven cultures in organizations.

Data analysis profoundly influences decision-making by enabling evidence-based strategies. Instead of relying on intuition or outdated assumptions, leaders can evaluate real-time data to guide their choices. For example, marketing teams use analytics to segment audiences and tailor campaigns, while operations managers rely on predictive models to forecast demand and manage inventory. This analytical approach reduces risk, enhances agility, and fosters innovation. As organizations continue to embrace digital transformation, data analysis will remain central to strategic planning and execution.

## Why Polars?

This section explores Polars as a data analysis library, highlighting its key advantages, comparing its performance and usability to pandas, and examining how Polars integrates into the broader analytical workflow.

### What Is Polars?

Polars is a blazingly fast dataframe library for manipulating structured data. It is not a programming language but a library that can be used within other languages such as Python, Rust, and R. Created by Ritchie Vink, Polars was first released in 2020

and has since evolved through contributions from a growing community of developers. I'll use Polars with the Python programming language throughout the book.

According to the Polars user guide[1], the library is built on a philosophy of delivering lightning-fast performance and efficiency in data analysis. It achieves this by leveraging all available CPU cores, optimizing queries to minimize unnecessary computation and memory usage, and handling datasets that exceed the size of your system's RAM. Polars also provides a consistent and predictable API, built around a strict schema where data types are known before execution, ensuring reliability and clarity in analytical workflows. Written in Rust, it benefits from C/C++-level performance and precise control over performance-critical aspects of its query engine, making it a powerful tool for modern data processing.

Understanding the Polars data model is essential to using the library effectively. At its core, Polars provides two main data structures: the `Series` and the `DataFrame`, which are used for working with array data and tabular data, respectively. A dataframe is a collection of series, and each series is a one-dimensional structure. Certain rules apply to the series within a dataframe: they must all have the same length, each must have a unique name, and all elements within a series must share the same data type. While dataframes and Series share some methods (functions that can be called on them), many are specific to each structure.

**Tip**
A dataframe can consist of a single column; it does not need to have many.

Reading data from a CSV file with Polars creates a dataframe. Polars automatically tries to infer the data type for each column. If it cannot determine the type, it assigns `String` as the default. In addition to CSV, Polars can read other file formats such as Parquet, Avro, and Excel. It can also load data directly from a database, Delta Lake, or even from data copied to your clipboard. Likewise, Polars can write data to all of these formats and more.

One of Polars' most distinctive features is its use of *expressions*. Expressions are composable building blocks that define operations and transformations on dataframes. Instead of manipulating raw data directly, you create expressions that describe what you want to compute. For example, the code below multiplies the *Price* and *Discount* columns in a dataframe.

```
pl.col('Price') × pl.col('Discount')
```

With expressions, you can target specific columns, apply functions, filter rows, and perform aggregations while maintaining clear and declarative syntax. These expressions can range from simple to highly complex, depending on your needs.

The power of expressions lies in their ability to chain operations together. By combining multiple expressions, you can build efficient data pipelines from simple, reusable components. For instance, you might start by selecting columns, apply a transformation such as a logarithm or normalization, filter rows based on a condition, and then group and aggregate—all within a single expression chain. Much of the code in this book follows this method-chaining style. This modular approach promotes clean, readable code and makes it easier to debug or extend data pipelines. More importantly, it leverages the Polars query engine, which analyzes your code and constructs an optimized execution plan. As a result, chaining expressions can significantly improve the execution speed of your code.

## Benefits of Polars

Compared to other dataframe libraries such as pandas, Dask, PySpark, or Ibis, Polars is relatively new. However, its growing popularity is one of the main reasons to consider it as your primary data analysis tool. I predict that soon, more companies will begin listing Polars knowledge as a desirable, if not required, skill for data professionals.

The two strongest reasons to use Polars are its speed and its simple, intuitive syntax. One of the most common sayings in the Polars community is, "I came for the speed, I stayed for the syntax." This perfectly reflects my own experience. Compared to pandas, the library I first learned, Polars code reads almost like plain English. You'll see how natural it feels when we explore its structure in Chapter 2.

In addition to its query optimizer, Polars achieves high performance through multithreading, which allows it to use all available CPU cores on your machine. The Polars Decision Support (PDS) benchmark compares Polars to other data analysis libraries in terms of execution speed, where lower times indicate better performance. Table 1 shows the latest PDS-H benchmark results at scale factor 10 (SF-10), where one unit of scale factor represents roughly 1 GB of CSV data. PDS-H supports running queries across multiple data sizes.

| Solution | Total time (sec) | Factor |
|---|---|---|
| polars[streaming]-1.30.0 | 3.89 | 1.0 |
| duckdb-1.3.0 | 5.87 | 1.5 |
| polars[in-memory]-1.30.0 | 9.68 | 2.5 |
| dask-2025.5.1 | 46.02 | 11.8 |
| pyspark-4.0.0 | 120.11 | 30.9 |

| Solution | Total time (sec) | Factor |
|---|---|---|
| pandas-2.2.3 | 365.71 | 94.0 |

*Table 1. PDS-H benchmark[2] : execution time for each query.*

Another advantage of Polars is its expression API, which makes it highly extensible. Polars encourages a method-chaining style of coding, made possible through expressions. This approach allows you to build complex data pipelines by chaining expressions together. The expression API has been so effective that pandas 3.0 adopted similar functionality by introducing the expression `pd.col`.

Polars is also easy to learn. Adopting a new technology can be challenging, especially if you're already familiar with another tool. Fortunately, Polars has a straightforward and intuitive syntax that makes the code easy to read and reason about. When I primarily wrote pandas code, it often took me half an hour or more to recall what my code from the previous week was doing. Polars code, on the other hand, reads like plain English. Now it only takes me a few minutes to understand my older Polars scripts. At first glance, the Polars documentation[3] might seem overwhelming because of the long list of methods and expressions. The good news is that you don't need to learn everything at once. Start with the general methods and expressions, then expand your knowledge over time. In my experience, learning sticks best when you encounter a real problem and then seek out the method that solves it, rather than trying to memorize a long list in advance.

Finally, Polars has an active and supportive community. The Polars team frequently updates the codebase, continually improving the library. You can raise issues or report bugs and expect quick responses. The Polars Discord community is also an excellent place to ask questions. I've asked several myself and have always been impressed by how quickly and thoughtfully people respond. I've also learned a great deal from discussions and solutions shared by other members, so even observing the conversations in the Discord channel can be an effective way to learn.

## Polars versus pandas

The de facto data analysis library in the data field is pandas. It provides many of the same functionalities as Polars and even more in some areas. Both libraries also offer unique features that set them apart. In general, almost any analysis we can perform with pandas can also be done with Polars. However, several important differences distinguish the two.

The most significant difference between Polars and pandas is the *lazy API* (lazy mode), which is Polars has but pandas does not. When we use the lazy API, Polars

does not execute each query line by line. Instead, it processes the entire query from start to finish as a single operation. Using the lazy API is recommended because:

1. It allows Polars to automatically optimize queries through its query optimizer.
2. It enables working with datasets larger than your system's memory through streaming.
3. It can detect schema errors before data processing begins.

When we read a dataset in lazy mode, Polars creates a `LazyFrame` instead of a `DataFrame`. To view the output of a `LazyFrame`, we must call the `collect` method on it.

In contrast, pandas operates only in eager mode. It lacks a lazy mode, which is ironic given that the panda is often called the laziest bear. This means pandas executes code sequentially without query optimization. Additionally, pandas is limited to datasets that fit into our system's memory.

Another major difference between the two libraries is the use of indexes. By default, pandas dataframes include an index, while Polars dataframes do not. As a result, the position of rows in a Polars dataframe may not always be consistent across runs. If you need an index in a Polars dataframe, you can create one using the `with_row_index` method, which assigns a sequential number to each row starting from zero. Adding an index may slightly impact performance. That's why it's a good idea to create the index at the stage of the code when it's actually needed.

Both Polars and pandas provide built-in plotting capabilities. In pandas, we can use the `plot` method, which relies on the **Matplotlib** library in the backend. Polars also supports the `plot` method, but it uses **Altair** as the backend plotting library. Because pandas has been around longer, more visualization libraries natively support pandas dataframes than Polars dataframes. However, support for Polars is growing, with libraries such as **Plotly** and **HvPlot** now offering native compatibility. If a visualization library supports only pandas dataframes, such as **Plotnine**, we can easily convert a Polars dataframe to a pandas dataframe using the `to_pandas` method. Conversely, we can convert a pandas dataframe to a Polars dataframe using the `from_pandas` expression. These conversion functions are unique to Polars.

Another key distinction is performance. Polars is multi-threaded and uses all available CPU cores on your machine, whereas pandas is single-threaded. This makes Polars significantly faster for many operations.

The ability to have hierarchical columns in a pandas dataframe is another distinguishing feature. For example, a pandas dataframe of clothing brands could have two main columns, *Premium* with sub-columns *Gucci* and *Versace*, and *Basic* with sub-columns *Gap* and *Forever21* as illustrated in Table 2.

| | Premium | | Basic | |
|---|---|---|---|---|
| | Gucci | Versace | Forever21 | Gap |
| 0 | 1000 | 3000 | 27.95 | 23.99 |
| 1 | 2500 | 5000 | 39.98 | 54.85 |

*Table 2. Hierarchical columns in a pandas dataframe.*

This kind of hierarchical column structure is not supported in a Polars dataframe.

Finally, although both Polars and pandas are Python libraries, they're written in different programming languages: Polars in Rust, and pandas in C.

## Data Formats Polars Can Read

Polars is a versatile library that can read and process a wide range of file formats. Data analysts frequently work with datasets stored in different formats, and Polars provides native support for many of the most common ones. Some of the file formats Polars can read include:

- CSV - text-based files with extensions such as *.csv*, *.txt*, and *.tsv*.
- Excel - spreadsheet files with extensions such as *.xlsx* and *.xlx*, including macro-enabled *.xlsm* files. To read a variety of spreadsheet files, additional libraries must be installed, such as **fastexcel**, **xlsx2csv**, and **openpyxl**.
- JSON - files with the *.json* extension, commonly used for data retrieved from web APIs.
- Parquet - files with the *.parquet* extension, which use columnar storage, compression, and preserve column data types.
- Database - data stored in relational databases, which Polars can read directly through database connections.
- Avro - files with the *.avro* extension, used with the Apache Avro data serialization system. These files are common in big data and distributed systems where efficient storage and data exchange are required.
- Delta - tables stored in the Delta Lake format, which builds on top of Parquet and adds ACID transactions, schema enforcement, and time travel capabilities. This format is commonly used in modern data lakehouse architectures.
- ODS - spreadsheet files with the *.ods* extension, based on the OpenDocument Spreadsheet standard. These files are often used as an open alternative to Excel formats.
- Iceberg - tables managed using the Apache Iceberg table format, designed for

large analytic datasets. Iceberg supports schema evolution, partition evolution, and ACID transactions in distributed data environments.
- IPC - files using the Arrow IPC format, sometimes stored with *.arrow* or *.ipc* extensions. This format is part of the Apache Arrow ecosystem and enables fast, zero-copy data exchange between systems.

Polars also supports many additional file formats beyond those listed here.

**Expand Your Knowledge**

ACID is an acronym that describes four key properties that ensure reliable database transactions:

- *Atomicity* - A transaction is treated as a single unit of work. Either all changes are committed, or none are. If any part fails, the entire transaction is rolled back.
- *Consistency* - A transaction moves the system from one valid state to another, enforcing all defined rules, constraints, and data integrity requirements.
- *Isolation* - Concurrent transactions do not interfere with one another. Each transaction behaves as if it were executed independently.
- *Durability* - Once a transaction is committed, the changes are permanently stored, even in the event of a system failure.

These guarantees are essential in modern data lakehouse formats such as Delta and Iceberg, where large-scale data systems still require dependable and consistent updates.

## Conclusion

Data analysis is a dynamic field that supports a wide range of needs across companies and institutions. Polars provides a powerful toolkit for working with data, particularly when that data is organized in tables. Importing data into a project and performing initial exploration represents only one stage of a broader analytical process, and data professionals routinely work with many different file types along the way. With the foundations of data analysis, Polars, and common data formats now in place, the remainder of this book focuses on applying Polars across the full analytical workflow. Chapter 2 focuses on data preparation, starting with an introduction to data types and then moving on to profiling, cleaning, and shaping data. Chapter 3 through 6 present applications of data analysis, focusing on time series analysis, text analysis, anomaly detection, and experiment analysis. Chapter 7 covers techniques for developing complex datasets for further analysis in other tools. Chapter 8 discusses ways to report the findings of our analysis with stakeholders. Chapter 9 introduces ten questions to increase understanding of Polars concepts. Finally, Chapter 10 concludes with a list of some additional resources to support your analytics journey.

1. *https://docs.pola.rs/#philosophy*

2. *https://pola.rs/posts/benchmarks/*

3. *https://docs.pola.rs/api/python/stable/reference/index.html*

# CHAPTER 2

# Preparing Data for Analysis

> *Data cleaning is exploration, necessary for problem discovery and context-building. Infrastructure can help, but it can never remove the need entirely because there's always new data to analyze, business questions to answer, and problems to solve.*
>
> — Katie Bauer

It's a truth universally acknowledged among data professionals that 80% (if not more) of their time is spent cleaning data. Yet, ironically, many of them dislike this part of the job. They gravitate toward the more exciting aspect: analysis. I was no different during my early days in graduate school. I loved diving into the analysis but disliked the tedious process of cleaning data. Then I came across that 80% statistic and thought, "If data cleaning is going to dominate my work, I'd better learn to enjoy it, or I'll be miserable in this field." That realization shifted my mindset. I began to embrace data cleaning, seeing it as an opportunity to familiarize myself with the data. Data preparation is so common that it's earned a variety of nicknames: data wrangling, data munging, and data prep. So, is all this data preparation simply tedious groundwork, or is it the foundation of good analysis?

Analyzing data is significantly easier and more reliable when the dataset is clean. As illustrated in Figure 1, data must pass through several stages before it becomes truly analysis-ready. Skipping these foundational steps and jumping straight into analysis is not just inefficient, it's also risky. Working with messy or unvalidated data can lead to incorrect conclusions and, worse, misleading insights. When decisions are based on flawed analysis, organizations may act on false assumptions, potentially resulting in costly mistakes.

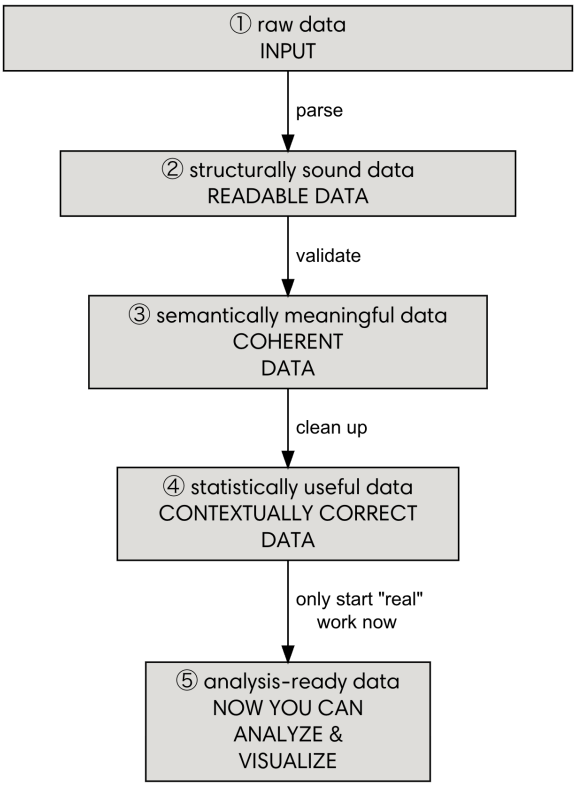



*Figure 1. Stages of data preparation by Kira Howe*[1]

Whenever I receive a new dataset, I begin by answering four essential questions before conducting any analysis:

1. What does each row represent?
2. Does the data in each column match its intended format or meaning?
3. Are there any null or missing values?
4. Are there any duplicates?

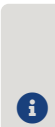
I only proceed with analysis after these questions have been addressed.

Data preparation is a key part of data exploration, and exploration is about understanding the nature of your dataset. The first question helps clarify what the dataset is actually about. For example, is it tracking students and their test scores, or is it recording blood sugar levels of diabetes patients?

The second question ensures that the data is tidy and consistent. You need to con-

firm that the values in each column are what you expect. If a column labeled *Age* contains a value like 5'7", that signals an error that needs correction.

The third question focuses on identifying null values. You cannot decide how to handle missing data until you understand the dataset. In some cases, you might remove rows with null values if doing so does not affect the analysis significantly. In other situations, you might choose to fill in missing values using the column average, the previous or next value, or the most frequent value. Whatever approach you take, it should be based on a solid understanding of the data.



**Note**
Null values in a dataset are not always represented by blank cells. Sometimes they appear as text entries such as "N/A", "Missing", empty space (""), etc. To ensure consistency, I identify all possible text representations of null values and convert them into actual `null` values signifying a blank cell. This makes it easier to handle missing data during analysis.

Question four checks for duplicate rows in the dataset. Duplicate records often indicate issues such as data entry errors, repeated data pulls, or improper joins. In most analytical settings, identical rows are not expected and can distort counts, averages, and other summary statistics. However, there are rare cases where duplicates are meaningful, such as transactional logs where repeated events are valid. Before removing any duplicates, you should understand why they exist and whether they represent true repetition or an error in data collection or processing.

A *data dictionary* simplifies the process of preparing data. It's a document (sometimes a separate dataset) that describes what each column in the main dataset represents. It may also include details about the types of values to expect in each column, how the data was collected, and how each column relates to others. Unfortunately, not every dataset comes with a data dictionary. In fact, most do not, as many companies are known for neglecting to create proper documentation for their data. Many organizations still treat documentation as a "nice-to-have" rather than a necessity.

When working with a dataset that lacks a data dictionary and the column names are not self-explanatory, it's important to consult a subject matter expert (SME) to clarify the meaning of each column. Once you have that clarity, you should create a data dictionary yourself. Your future self will thank you! Even better, you won't need to explain the dataset to new team members. A well-maintained data dictionary not only speeds up data preparation but also enhances the quality of your analysis by helping you confirm that you've used the columns correctly and applied the right filters.

It's worth noting that a data dictionary is not a substitute for data cleaning. You'll still need to clean and validate your data as part of the analysis process. In this chapter, I'll begin by reviewing the data types commonly encountered in Polars, followed by an overview of Polars code structure. Next, I'll discuss strategies for understanding your data and assessing its quality. Then, I'll introduce data-shaping techniques to help you extract the necessary columns and rows for further analysis. Finally, I'll cover useful tools for cleaning data, addressing quality issues, and reducing memory usage in your datasets.

## Types of Data

The modern world is filled with data, making the field of data analysis more important than ever. Data serves as the foundation for all forms of analysis, and every dataset consists of specific data types that fall into one or more categories. To become an effective data analyst, you need a strong understanding of the different forms data can take.

In this section, I'll begin by introducing the Polars data types most commonly encountered in analysis. Then I'll explore some conceptual groupings that help us better understand the source, quality, and potential applications of data.

### Polars Data Types

Columns in a Polars dataframe all have defined data types. You don't have to be an expert at all the different data types in Polars to be good at analysis, but later in the book, we'll encounter situations in which knowing what data type to use is crucial. This section will cover the basic data types.

The main types of data are numeric, temporal, nested, string, and other, as summarized in Table 3.

| Type | Name | Description |
|---|---|---|
| **Numeric** | `Float64` | Holds decimal numbers. |
| | `Int64` | Holds whole number integers which can be both negative and positive. |
| | `UInt64` | Holds unsigned integers that will always be positive whole numbers. |
| **Temporal** | `Datetime` | Holds calendar dates and time of day. |
| **Nested** | `List` | Holds a list of values akin to a Python list with a variable length. |

| Type | Name | Description |
|---|---|---|
| | `Struct` | Composite type that holds multiple sub-fields, each with a defined name and data type. |
| **String** | `String` (or `Utf8`) | Holds text values, including emojis. |
| | `Categorical` | Holds text values with fewer unique values (low cardinality). |
| **Other** | `Boolean` | Holds True or False values. |
| | `Null` | Holds blank values. |

*Table 3. A summary of common Polars data types.*

Numeric data types store both positive and negative numbers, including floating-point numbers (commonly known as decimals). Mathematical operations can be applied to numeric columns. Integers require less memory than decimals, and unsigned integers require even less than regular integers. The numbers in the data type names represent bits. For example, `Int64` is a 64-bit integer. Other integer types include `Int8`, `Int16`, and `Int32`. The same pattern applies to unsigned integers such as `UInt64`. In contrast, floating-point types come in only three forms: `Float64`, `Float32`, and `Float16`.

**Note**
Smaller integer types typically use less memory and can improve performance because more values fit in cache. For example, operations on `Int8` columns may be faster than on `Int16` when the data fits within the smaller range. The `Boolean` data type is even more memory efficient because Polars stores it using bit-packing, requiring only a single bit per value. In contrast, string columns are generally the least memory efficient because they must store variable-length text data.

Temporal data types include `Date`, `Datetime`, `Duration`, and `Time`. Dates and times should be stored using these types whenever possible, since Polars provides a rich set of functions for working with them. The rise of the Internet of Things has made timestamped data widely available. Such data is particularly useful for time series analysis covered in Chapter 3, which discusses date and time formatting, transformations, and calculations in detail.

Nested data types store collections of values as `List`, `Array`, `Struct`, or `Field`. The most common nested types in Polars are `List` and `Struct`. Lists in Polars do not have a fixed length, so one row might contain a list with two elements while another contains six. When all lists have the same length, it's more efficient to use an array because it offers better memory performance. The `Struct` type groups multiple fields

(columns) of different data types into a single column.

Strings represent text data and are the most versatile data type in Polars. You can even read an entire dataset as strings, including numeric values. Strings can contain letters, numbers, and special characters, including unprintable ones such as newlines, tabs, and spaces (which differ from blanks). The `Categorical` type is a specialized form of the `String` type, typically used for low-cardinality data. Categorical columns are memory-efficient because Polars stores integer codes that map to unique string values instead of storing each string repeatedly. This can greatly reduce memory usage, especially in columns with many repeated values. Polars provides several functions that make categorical data easy to analyze.

Other data types include `Boolean`, which represents logical values (`True` or `False`), and `Null`, which represents missing data. A column in Polars is assigned the `Null` type only when all its cells are blank.

Apart from Polars-specific data types, data can also be grouped in several conceptual ways. These classifications influence not only how data is stored but also how it's interpreted and analyzed. I'll explore these conceptual categories of data next.

## Structured Versus Unstructured

*Quantitative data* refers to information expressed in numerical form. It represents measurable aspects of people, objects, and activities. Anything that can be counted or measured and assigned a numerical value is quantitative in nature. Although quantitative data may include descriptive elements such as user profiles, product categories, or system settings, it's defined by numerical measures like cost, volume, duration, or frequency. This type of data is well-suited for mathematical operations such as calculating totals, averages, percentages, and trends. While much of today's quantitative data is generated automatically by digital systems, it can also be collected manually. For example, a coach entering athletes' lap times into a spreadsheet, a researcher recording rainfall amounts in a field notebook, or a lab technician noting chemical measurements.

*Qualitative data* captures the nuances of human experience through language, emotion, and subjective interpretation. Unlike quantitative data, which deals with measurable values such as speed or price, qualitative data is usually text-based and includes opinions, feelings, and descriptive statements. For instance, a speedometer showing 75 miles per hour is quantitative, but describing the drive as "fast and exhilarating" is qualitative. Similarly, knowing a customer paid $30 for a product is numerical, while their comment that it "wasn't worth the money" provides qualitative insight. This type of data often appears in survey responses, interview tran-

scripts, customer reviews, and social media posts.

In data analysis, analysts often work to make qualitative data more structured and measurable. Techniques such as keyword extraction can identify recurring phrases and count their frequency, while sentiment analysis evaluates the emotional tone of text to determine whether it's positive, negative, or neutral. These methods help convert subjective language into quantifiable insights, making it easier to summarize and interpret large amounts of text. Advances in natural language processing (NLP) have further enabled the automation and scaling of such analyses across various industries.

## Parties of Data

*First-party data* is information collected directly by an organization from its own audience or customers. This data is gathered through interactions such as website visits, app usage, purchase history, email subscriptions, and customer surveys. Because it comes straight from the source, first-party data is considered highly reliable and relevant. Businesses use it to understand user behavior, personalize experiences, and improve marketing strategies. For example, an online retailer might track which products a customer browses and buys to recommend similar items in the future.

*Second-party data* refers to another organization's first-party data that is shared through a partnership or agreement. It's essentially someone else's direct customer data that you gain access to, often for mutual benefit. For instance, a hotel chain might share guest preferences with an airline to help tailor travel promotions. Since second-party data is obtained from a trusted source and typically comes with transparency about how it was collected, it's more accurate and targeted than third-party data, though not as direct as first-party data.

*Third-party data* is collected by entities that do not have a direct relationship with the individuals the data describes. These data aggregators gather information from various sources, such as websites, apps, and public records, then compile it into large datasets that are sold or licensed to other companies. Third-party data is often used to expand audience reach, enrich existing customer profiles, or target new segments. However, it's generally less precise and raises greater privacy concerns, particularly as regulations on data sharing and consent become stricter.

*Zero-party data*, also known as solicited data, refers to information that customers intentionally and proactively share with a brand, such as preferences, intentions, and personal context, typically through quizzes, surveys, or preference centers. Unlike first-party data, which is inferred from behavior, zero-party data is explicitly pro-

vided by customers, making it transparent, privacy-compliant, and especially valuable for personalization. Customers share this information because it offers clear benefits, including more relevant product recommendations, personalized content, and communication aligned with their preferences, creating a value exchange that benefits both the customer and the brand.

## Polars Code Structure

The first step in writing Polars code is to import the library:

```
import polars as pl
```

The alias `pl` is used so that you don't have to type the full name of the library each time you call a function. For example, to use the `from_pandas` function, you can write `pl.from_pandas` instead of `polars.from_pandas`. Using an alias is not mandatory, but it's a common convention that makes your code cleaner and easier to read. I recommend importing Polars with the alias `pl`, since most examples online and in the Polars documentation[2] follow this same pattern. Sticking to this convention will help you better understand other developers' code and make your own more consistent.

After importing the library, the next step is typically reading data from a file, most often a CSV. You can load the data eagerly with `pl.read_csv` or lazily with `pl.scan_csv` to create either a `DataFrame` or a `LazyFrame`. By convention, a variable named *df* is assigned to the dataframe containing the file's contents, though you can use any name you prefer. Some developers use more descriptive names such as *patients_data* when the dataframe contains patients' information.

**Tip**
Because Polars encourages a method-chaining style, it's easier to write and read code when you wrap it in parentheses `()`. This allows new lines to be treated as whitespace and ignored by the parser. An additional benefit is that the resulting code reads like a step-by-step recipe, which makes it easier to follow. I strongly recommend adopting this format when writing Polars code.

The `select` method specifies which columns to include in our output. We can use literal strings to select columns, as shown below:

```
df.select('Quantity', 'Price')
```

Or we can use expressions:

```
df.select(pl.col('Quantity'), pl.col('Price'))
```

Unless I'm applying an aggregation function, I typically use string literals. However,

if I want to calculate a total, I select the column as an expression and apply an aggregation, such as a sum:

```
df.select(pl.col('Quantity').sum())
```

Alternatively, we can achieve the same result using a simpler syntax, also known as syntactic sugar:

```
df.select(pl.sum('Quantity'))
```

Sometimes we might want to keep most columns and exclude only a few. Listing every column to keep can be tedious, especially with wide dataframes. In such cases, the `drop` method lets us remove specific columns. For example, to exclude *Quantity* from our output, we can write:

```
df.drop('Quantity')
```

**Note**
It's often more efficient to use the `exclude` expression instead of `drop`. When Polars detects that a column isn't used, it can skip reading it entirely, saving time and memory. In contrast, the `drop` method still reads the column before removing it. Another advantage of `exclude` is its flexibility. If the column you're trying to remove doesn't exist, your code won't fail. The `drop` method, however, will raise an error.

In most data analyses, you won't return every row in the dataframe. You'll often need to isolate specific records based on conditions in one or more columns. The `filter` method is used for this purpose. For example, to select only rows where *Gender* equals "female", you can write:

```
df.filter(pl.col('Gender') == 'female')
```

Alternatively, we can use the equivalent `eq` expression:

```
df.filter(pl.col('Gender').eq('female'))
```

To exclude rows where *Gender* is "female" use the not-equal operator:

```
df.filter(pl.col('Gender') != 'female')
```

Or its expression form:

```
df.filter(pl.col('Gender').ne('female'))
```

All common mathematical comparison operators can be used with `filter`. Table 4 lists the available operators and their corresponding Polars expressions.

| Symbol | Polars expression | Description |
|---|---|---|
| == | eq | Equal to |
| != | ne | Not equal to |
| > | gt | Greater than |
| < | lt | Less than |
| >= | ge | Greater than or equal to |
| <= | le | Less than or equal to |

*Table 4. Inequality operators and their Polars expression equivalents.*

You can also control how many rows are returned using the `head`, `tail`, or `sample` methods. For instance, to view the first five rows, use:

```
df.head()
```

To view the last five rows:

```
df.tail()
```

And to view five random rows:

```
df.sample(5)
```

I often use `sample` when I want a quick sense of the data or to check for missing values. Random sampling provides a more representative snapshot, since missing values tend to appear in the middle or end of a dataset. Relying only on `head` or `tail` might give a misleading impression—for example, that there are no missing values when they actually exist elsewhere in the file.



**Note**
To return more than five rows, include a number as a parameter. For example, `head(10)` and `tail(15)` return the first ten and last fifteen rows, respectively.

As your analysis progresses, you may need to combine data from multiple files. Polars provides two main ways to combine dataframes: concatenation using the `pl.concat` function and joins using the `join` method. Concatenation can be performed in three ways: vertically, horizontally, or diagonally. Other concatenation options include `align`, `align_full`, and `vertical_relaxed`[3].

Vertical concatenation stacks one dataframe on top of another. For this operation to succeed, the dataframes must share the same column names and, in some cases, the

same data types for corresponding columns. The number of rows does not need to match. To perform this operation, use the `how='vertical'` option.

If the data types of a column differ between the two dataframes, for example, `Int32` and `Int64`, use `how='vertical_relaxed'`. This option allows the operation to proceed by safely reconciling compatible data type differences.

Horizontal concatenation places dataframes side by side. In this case, the dataframes must have the same number of rows, and column names must be unique. Use the `how='horizontal'` option.

Diagonal concatenation merges dataframes by taking the union of their column schemas, filling in missing values with `null`. As long as the data types are consistent, column names can differ. Use the `how='diagonal'` option.

You can also combine dataframes using the `join` method. Except for a `cross` join, every join requires a matching key column or set of columns. For example, if *df1* and *df2* each contain student test scores, you can join them on *Student_Name* with an inner join:

```
(df1
 .join(df2, on='Student_Name', how='inner')
)
```

Polars supports several `join` types. Table 5 summarizes the five most commonly used in data analysis.

| Type | Description |
|---|---|
| `inner` | Computes set intersection. Keeps matching rows from both dataframes. |
| `left` | Keeps all rows from the left and matching rows from the right. Non-matching left rows have right-side columns filled with `null`. |
| `anti` | Computes set difference. Keeps rows from the left that do not have matches in the right. |
| `full` | Keeps all rows from both dataframes, filling non-matching columns with null. |
| `cross` | Computes the Cartesian product[4] of the two dataframes. |

*Table 5. Five common join types in Polars.*

After selecting and filtering data, you'll often want to compute summaries or aggregations. This is where the `group_by` method becomes essential. To use it, you must have at least one non-aggregated column—the column or columns you want to group by. The columns listed in `group_by` are not aggregated; instead, the columns

specified in `agg` are combined using statistical functions. In Polars, `group_by` is a reducing operation, meaning it returns either a single value or fewer rows than the original dataframe.

## Profiling: Distributions

Part of understanding your dataset involves performing *exploratory data analysis* (EDA). Originally developed by John Tukey, EDA is the process of analyzing datasets to summarize their main characteristics, often through statistical visualizations. It's a crucial first step in data analysis and data science for uncovering patterns, spotting anomalies, and verifying assumptions before formal modeling. The primary goals of EDA are to identify errors, discover trends, detect outliers, and generate hypotheses for further investigation.

Profiling is an essential part of EDA. It focuses on analyzing a dataset to understand its structure, content, quality, and relationships. During profiling, you look for issues such as missing values, inconsistencies, and anomalies to determine whether the data is suitable for use in applications like data warehousing, business intelligence, or data integration. In other words, data profiling ensures that the data is accurate, complete, and reliable.

It's in the profiling stage that I aim to answer four essential questions before performing any analysis. First, I determine what each row in the dataset represents. Second, I verify the accuracy and consistency of the data by checking that each column contains valid values. Third, I identify whether the dataset contains any missing values. Lastly, I check if there are any duplicate values. If a data dictionary is available, I examine it to understand what each column means. Additionally, when column names are long or contain spaces, I shorten them and remove the spaces to make them easier to work with.

**Tip**

It's easier to work with a dataset when column names are short and free of spaces or unusual characters such as `&`, `$`, `@`, `#`, and `!`. Underscores are acceptable, especially for replacing spaces in column names. For example, *Customer Account* can become *Customer_Account*.

Once you know what each row represents, you can often infer the domain your dataset belongs to. Is it about patient records or e-commerce transactions? Does it come from a survey for a new product launch or from an annual technology study like the Python Developer Survey? Profiling helps reveal the dataset's domain and, more importantly, equips you with relevant questions to ask the people who col-

lected the data. Profiling is too valuable a step to skip—even if you collected the data yourself.

After gaining a solid understanding of the dataset, I move on to visualizations. Humans are not built to process thousands or millions of rows in a table. It's nearly impossible to draw insights from scrolling through that much data. This is where visualizations become indispensable.

I usually begin by examining visualizations that show data distributions, such as histograms. These reveal the range of values in the data and how frequently they occur. They also help identify the presence of `null` or negative values. Distributions can be visualized for both continuous and categorical data.

In this section, we'll explore how to create histograms, how binning can help us better understand the distribution of continuous data, and how to use n-tiles for more precise insights into data distributions.

## Histograms and Frequencies

Checking the frequency of values in each column is one of the best ways to familiarize yourself with a dataset. Frequency checks answer questions such as whether certain values are valid or how often a suspicious value appears. You can perform frequency checks on any data type, including text, numerical, date, and boolean values.

The number of occurrences of each value in a column can be calculated in two ways: with `value_counts` or with `group_by`. That there are multiple ways to achieve the same result should not be surprising. As you write more Polars code, you'll find that many tasks can be expressed in different ways.

Let's start with a dataset containing retail sales for a clothing store. Polars can read directly from a CSV file stored on the web, so there's no need to download it. I'll use the **Hvplot** library for visualizations.

```
import polars as pl
import hvplot.polars

data_url = 'https://raw.githubusercontent.com/jorammutenge/learn-
rust/refs/heads/main/sample_sales.csv'

clothing_store_sales = (pl.read_csv(data_url, try_parse_dates=True)
      .select(pl.exclude('Account Name'))
      .rename(lambda c: c.title().replace(' ','_'))
      .rename(dict(Account_Number='Customer_ID'))
      )
clothing_store_sales
```

```
shape: (1_000, 7)
+-------------+----------+----------+----------+------------+-----------+----------------+
| Customer_ID | Sku      | Category | Quantity | Unit_Price | Ext_Price | Date           |
| ---         | ---      | ---      | ---      | ---        | ---       | ---            |
| i64         | str      | str      | i64      | f64        | f64       | datetime[μs]   |
+=============================================================================================+
| 803666      | HX-24728 | Hat      | 1        | 98.98      | 98.98     | 2014-09-28 ... |
| 64898       | LK-02338 | Sweater  | 9        | 34.8       | 313.2     | 2014-04-24 ... |
| 423621      | ZC-07383 | Sweater  | 12       | 60.24      | 722.88    | 2014-09-17 ... |
| 137865      | QS-76400 | Sweater  | 5        | 15.25      | 76.25     | 2014-01-30 ... |
| 435433      | RU-25060 | Sweater  | 19       | 51.83      | 984.77    | 2014-08-24 ... |
| ...         | ...      | ...      | ...      | ...        | ...       | ...            |
| 29068       | FT-50146 | Sweater  | 2        | 46.48      | 92.96     | 2014-08-10 ... |
| 77116       | IC-59308 | Socks    | 19       | 29.25      | 555.75    | 2013-11-20 ... |
| 23749       | IC-59308 | Socks    | 18       | 54.79      | 986.22    | 2014-03-10 ... |
| 172519      | RU-25060 | Sweater  | 15       | 62.53      | 937.95    | 2014-04-11 ... |
| 914594      | LK-02338 | Sweater  | 11       | 86.4       | 950.4     | 2014-02-14 ... |
+-------------+----------+----------+----------+------------+-----------+----------------+
```

As mentioned earlier, we can find the frequency of each clothing item in *Category* in two ways. The first approach uses `value_counts`.

```
(clothing_store_sales
 .get_column('Category')
 .value_counts(name='Count')
 )
```

---

```
shape: (3, 2)
+----------+-------+
| Category | Count |
| ---      | ---   |
| str      | u32   |
+==================+
| Socks    | 297   |
| Sweater  | 526   |
| Hat      | 177   |
+----------+-------+
```

The second approach, which uses `group_by`, is often the most common.

```
(clothing_store_sales
 .group_by('Category')
 .agg(Count=pl.len())
 )
```

---

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

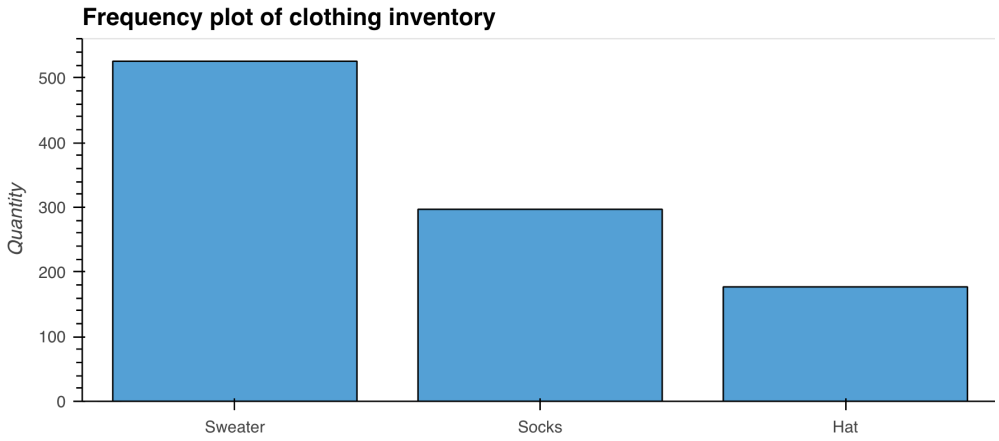
```
+----------+-------+
| Category | Count |
| ---      | ---   |
| str      | u32   |
+==================+
| Sweater  | 526   |
| Socks    | 297   |
| Hat      | 177   |
+----------+-------+
```

A *frequency plot* is a graphical representation showing how often values occur in a dataset. It helps visualize distributions and patterns within the data. The profiled column is usually plotted on the x-axis, while the count of observations appears on the y-axis. The chart below illustrates a frequency plot created from the code above. Since *Category* is categorical, we can use a bar chart to represent the frequency of each clothing category. When category names are long, a horizontal bar chart often works best.

**Tip**

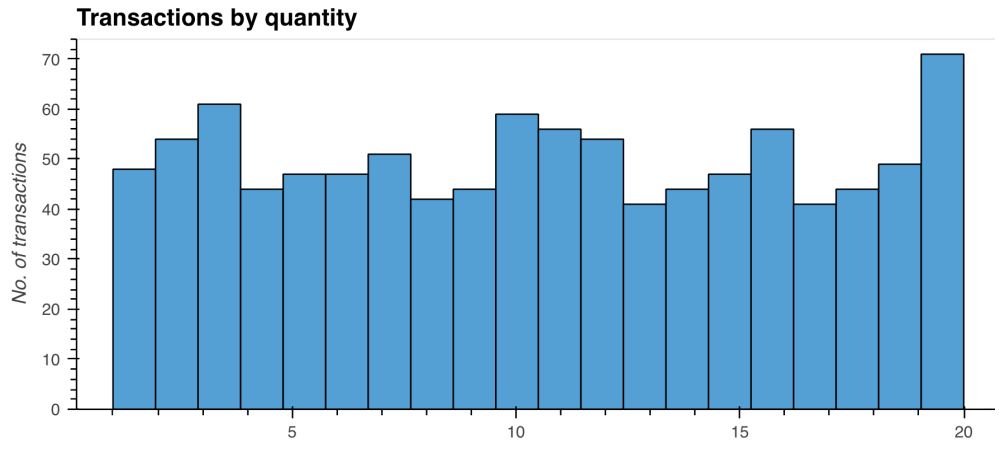
When creating a frequency plot as a bar chart, I prefer to sort the bars by count, starting with the largest. This makes the chart easier to interpret and more visually appealing. The only exception is when working with ordinal categorical values, such as months.

```
(clothing_store_sales
 .group_by('Category')
 .agg(Count=pl.len())
 .sort('Count', descending=True)
 .hvplot.bar(x='Category', y='Count',
             title='Clothing type',
             xlabel='', ylabel='Quantity')
 .opts(title='Frequency plot of clothing inventory')
 )
```




A *histogram* is another graphical representation that uses bars to show how numerical data is distributed. Each bar represents the frequency of data points within a specific range or *bin*. This allows you to visualize patterns such as frequency distribution, outliers, and skewness. Using the *clothing_store_sales* dataset, you can create a histogram to visualize the distribution of values in *Quantity* as shown below.

```
(clothing_store_sales
 .select('Quantity')
 .hvplot.hist(title='Quantity distribution',
              xlabel='', ylabel='No. of transactions')
 .opts(title='Transactions by quantity')
 )
```



The histogram above helps you determine whether customers tend to buy items in large or small quantities.

Frequency counts can become more complex than the examples shown so far. Consider this as a potential interview question: using *clothing_store_sales*, write Polars code that returns the distribution of orders per customer. This is trickier than it might first appear. Many people who rush into writing code without careful thought get it wrong. If you want to test your Polars skills, try it and compare your results with the example below.

To solve this, first identify a column that can serve as an order ID. If each row in *clothing_store_sales* represents a transaction by a customer, you can infer that the *Date* value can be used as a transaction ID. It's worth noting that this example assumes a small store with one checkout counter, where multiple transactions cannot occur at the same time. For larger stores with multiple checkouts, even timestamps with seconds would not be sufficient for unique transaction IDs.

Next, perform two aggregations: the first counts the number of transactions made by each customer, and the second counts how many customers fall into each transaction count category.

```
(clothing_store_sales
 .group_by('Customer_ID')
 .agg(Transactions=pl.count('Date'))
 .group_by('Transactions')
 .agg(Customers=pl.len())
 )
```

---

```
shape: (5, 2)
+--------------+-----------+
| Transactions | Customers |
| ---          | ---       |
| u32          | u32       |
+==========================+
| 5            | 2         |
| 4            | 3         |
| 1            | 508       |
| 2            | 166       |
| 3            | 46        |
+--------------+-----------+
```

This type of profiling is useful whenever you need to understand how frequently certain values appear in your data. While we've focused on `count` and `len`, other aggregation functions like `mean`, `sum`, `min`, or `max` can also be used to create histograms. For

instance, you might want to profile customers by the `sum` of their transactions, the `mean` (average) transaction size, or the `max` (most recent) transaction date.

## Binning

Binning, or discretization, is the process of turning continuous data into categorical data. For example, you might have a *Height* column with continuous values and convert it into *Height_Range*, containing bins such as short, medium, and tall. Each bin represents a range of height values, and the number of records that fall into each range is then counted. These bins can be equal in size or vary in width. The choice depends on whether you prefer bins with similar ranges or bins that contain roughly the same number of records. Bins can be created with `when-then-otherwise` expressions, rounding, and logarithms.

**Warning**
Binning is a destructive process. Each time you bin data, you move from the specific to the general, therefore losing information. It's difficult to return to continuous values once the data has been binned. It's therefore better to create a new column for the binned data and keep the original column with continuous values.

A `when-then-otherwise` expression allows conditional logic to be evaluated. These expressions are flexible, and I will return to them throughout the book as we apply them to data profiling, cleaning, text analysis, and more. The code below shows the basic structure of a `when-then-otherwise` expression, using the height example introduced earlier.

```
(df
 .with_columns(pl.when(pl.col('Height') < some_number)
               .then(pl.lit('Short'))
               .when(pl.col('Height').is_between(some_number, another_number))
               .then(pl.lit('Medium'))
               .otherwise(pl.lit('Tall'))
               .alias('Height_Range'))
 )
```

A `when` condition can be an equality, inequality, or any other logical condition. A `then` value can be a number, a text value, or a column in the dataframe. You can include any number of conditions, and Polars assigns the corresponding value to each row that meets a condition. Any rows that do not satisfy a stated condition receive the value in `otherwise`, which serves as the default. As with `then`, the `otherwise` value can be a number, a text value, or a column.

The `when-then-otherwise` expression makes it easy to control the number of bins, the

ranges that define each bin, and the labels assigned to them. This approach is especially useful when a dataset contains a long tail of very small or very large values that you want to group rather than leave isolated in sparsely populated bins. It also helps when certain ranges have business meaning that must be reproduced in the data. For example, many retailers classify products as "low-margin," "mid-margin," or "high-margin" based on profit percentage, because pricing and promotional strategies differ across these categories. Returning to the clothing store example, imagine that you want to send discount coupons to customers based on their total spend and need to know how many customers fall into each group. You can bin *Ext_Price* into three ranges with the following code:

```
(clothing_store_sales
 .with_columns(pl.when(pl.col('Ext_Price') <= 100)
               .then(pl.lit('up to 100'))
               .when(pl.col('Ext_Price') <= 500)
               .then(pl.lit('100 to 500'))
               .otherwise(pl.lit('500+'))
               .alias('Spend_Bin'))
 .with_columns(pl.when(pl.col('Ext_Price') <= 100)
               .then(pl.lit('Small'))
               .when(pl.col('Ext_Price') <= 500)
               .then(pl.lit('Medium'))
               .otherwise(pl.lit('Large'))
               .alias('Spend_Category'))
 .group_by('Spend_Bin','Spend_Category')
 .agg(Customers=pl.len())
 )
```

---

```
shape: (3, 3)
+------------+---------------+-----------+
| Spend_Bin  | Spend_Category | Customers |
| ---        | ---            | ---       |
| str        | str            | u32       |
+=========================================+
| 500+       | Large          | 466       |
| 100 to 500 | Medium         | 420       |
| up to 100  | Small          | 114       |
+------------+---------------+-----------+
```

Bins of varying size may be useful in certain situations, but fixed-size bins often work better for specific types of analysis. You can create fixed-size bins using n-tiles, rounding, or logarithms. Rounding reduces the precision of the original values. To round to the nearest ten, for instance, you divide the number by ten, round the result, then multiply by ten. The `round` function accepts a positive integer argument,

so `round(2)` rounds to two decimal places and `round()` rounds to the nearest whole number. The divide-then-multiply technique makes it possible to round whole numbers to the nearest tens, hundreds, or thousands, similar to rounding with a negative value in PostgreSQL. Table 6 shows how rounding affects a single value across several levels of precision.

| Formula | Meaning | Result |
|---|---|---|
| `round(2)` | Two decimal places | 123456.79 |
| `round(1)` | One decimal place | 123456.8 |
| `round()` or `round(0)` | Nearest whole number | 123457 |
| `truediv(10).round().mul(10)` | Nearest tens | 123460 |
| `truediv(100).round().mul(100)` | Nearest hundreds | 123500 |
| `truediv(1000).round().mul(1000)` | Nearest thousands | 123000 |

*Table 6. : The number 123456.789 rounded to various levels of precision.*

When working with values where the largest are orders of magnitude greater than the smallest, logarithms are an effective tool for creating bins. The brightness of stars, the population sizes of cities, and the frequency of words in natural language are all examples of phenomena with this property. Logarithms do not create bins of equal width, but the bin sizes increase in a useful pattern. One helpful way to understand logarithms is to remember that they answer the question: How many times do I multiply 10 by itself to get this number?

$$\log(number) = exponent$$

In this expression, 10 is the base, and the exponent is sometimes called the power. Although 10 is the most common base, other numbers can be used as the base. Table 7 shows the logarithms of several powers of 10.

| Formula | Result |
|---|---|
| log(1) | 0.0 |
| log(10) | 1.0 |
| log(100) | 2.0 |
| log(1000) | 3.0 |

| Formula | Result |
|---|---|
| log(10000) | 4.0 |

*Table 7. Results of the `log` function on powers of 10.*

In Polars, the `log` function returns the logarithm of values in the specified column. The following example uses logarithms to create bins based on *Unit_Price* in the *clothing_store_sales* dataset, and then counts the number of customers in each created bin.

```
(clothing_store_sales
 .with_columns(pl.col('Unit_Price').log().alias('Price_Bin'))
 .group_by('Price_Bin')
 .agg(pl.col('Customer_ID').count().alias('Customers'))
 )
```

---

```
shape: (939, 2)
+-----------+-----------+
| Price_Bin | Customers |
| ---       | ---       |
| f64       | u32       |
+=======================+
| 4.506123  | 1         |
| 3.959288  | 1         |
| 3.487069  | 1         |
| 4.255613  | 1         |
| 3.032064  | 1         |
| ...       | ...       |
| 4.592693  | 1         |
| 3.697839  | 1         |
| 3.873282  | 1         |
| 4.247924  | 2         |
| 2.502255  | 1         |
+-----------+-----------+
```

**Tip**
In Polars, `log(0)` returns `-inf`, and the `log` of any negative number is `NaN` (Not a Number). The `log` function works on all positive numbers, not just powers of 10.

## n-Tiles

Polars does not have a built-in function to compute n-tiles, but the flexibility of its expressions makes it easy to construct n-tile calculations. You may already be familiar with the *median*, which is the middle value of a dataset and represents the 50$^{th}$

percentile. Half of the observations fall below the median and half above it. Quartiles extend this idea by dividing a dataset into four equal groups, each containing 25 percent of the observations.

1. **First quartile (Q1)**: The 25th percentile, meaning that one quarter of the observations fall below this value. It's also known as the lower quartile.
2. **Second quartile (Q2)**: The median, representing the 50th percentile and splitting the dataset into two equal halves.
3. **Third quartile (Q3)**: The 75th percentile, meaning that three quarters of the observations fall below this value. It's also called the upper quartile.

Percentiles divide a dataset into one hundred equal parts. Deciles divide it into ten equal parts. More generally, n-tiles divide the data into *n* equal-sized groups and allow you to compute any percentile you need, such as the 17th or 63.5th percentile.

To compute n-tiles with expressions, it helps to understand the underlying formula.

#### Formula for ntile

Suppose you have:

$N$ = total number of rows
$k$ = number of bins (tiles)
$i$ = row index (starting at 1 after sorting)

The bin assignment is:

$$\text{ntile}(i) = \lfloor \frac{(i-1) \cdot k}{N} \rfloor + 1$$

#### Explanation

- Sort the data by the chosen column.
- For each row *i*:
  - Multiply its position (*i*-1) by the number of bins *k*.
  - Divide by total rows *N*.
  - Take the floor (round down).
  - Add 1 to make bins start at 1 instead of 0.

#### Using the formula

- $N$ = 12 rows

- $k = 10$ bins

For row 1 (smallest value, 29.94):

$$ntile(1) = \lfloor \frac{(1-1) \cdot 10}{12} \rfloor + 1 = \lfloor 0 \rfloor + 1 = 1$$

For row 12 (largest value, 599.94):

$$ntile(12) = \lfloor \frac{(12-1) \cdot 10}{12} \rfloor + 1 = \lfloor \frac{110}{12} \rfloor + 1 = \lfloor 9.17 \rfloor + 1 = 10$$

This can be summarized more simply as:

$$ntile = floor((row\_index - 1) \times bins \div total\_rows) + 1$$

Now that the mechanics of n-tile calculations are clear, you can apply them in Polars. The example below uses 12 transactions showing the total spend.

```
shape: (12, 1)
+-------------+
| Total_Spend |
| ---         |
| f64         |
+=============+
| 29.94       |
| 59.94       |
| 71.94       |
| 77.94       |
| 89.94       |
| 119.94      |
| 140.94      |
| 209.94      |
| 335.88      |
| 359.94      |
| 539.94      |
| 599.94      |
+-------------+
```

To compute 10 n-tiles, sort the values in *Total_Spend* and assign a bin from 1 to 10. Below is the Polars code that performs this calculation.

```
(df
 .sort('Total_Spend')
 .with_row_index()
 .with_columns(N_Tile=(pl.col('index') * 10).floordiv(pl.len()).add(1))
 .select('Total_Spend','N_Tile')
)
```

```
shape: (12, 2)
+-------------+--------+
| Total_Spend | N_Tile |
| ---         | ---    |
| f64         | u32    |
+======================+
| 29.94       | 1      |
| 59.94       | 1      |
| 71.94       | 2      |
| 77.94       | 3      |
| 89.94       | 4      |
| 119.94      | 5      |
| 140.94      | 6      |
| 209.94      | 6      |
| 335.88      | 7      |
| 359.94      | 8      |
| 539.94      | 9      |
| 599.94      | 10     |
+-------------+--------+
```

You can extend this approach to a more meaningful example using the *clothing_store_sales* dataset. Suppose you want to divide orders into 10 bins based on *Ext_Price* and then identify the lowest and highest total spend within each bin. The following code produces that summary:

```
(clothing_store_sales
 .sort('Ext_Price')
 .with_row_index()
 .with_columns(N_Tile=(pl.col('index') * 10).floordiv(pl.len()).add(1))
 .group_by('N_Tile')
 .agg(Lower_Bound=pl.min('Ext_Price'),
      Upper_Bound=pl.max('Ext_Price'),
      Orders=pl.col('Date').len())
 .sort('N_Tile')
 )
```

---

```
shape: (10, 4)
+--------+-------------+-------------+--------+
| N_Tile | Lower_Bound | Upper_Bound | Orders |
| ---    | ---         | ---         | ---    |
| u32    | f64         | f64         | u32    |
+=============================================+
| 1      | 11.13       | 95.05       | 100    |
| 2      | 95.13       | 172.65      | 100    |
| 3      | 172.85      | 246.84      | 100    |
| 4      | 247.92      | 349.12      | 100    |
| 5      | 350.0       | 456.04      | 100    |
```

```
| 6      | 456.64      | 572.04      | 100    |
| 7      | 575.82      | 749.2       | 100    |
| 8      | 750.0       | 964.32      | 100    |
| 9      | 968.16      | 1224.0      | 100    |
| 10     | 1227.0      | 1958.6      | 100    |
+--------+-------------+-------------+--------+
```

The output shows, for example, that bin 1 contains 100 orders, with total spend ranging from $11.13 to $95.05.

You can also compute percentiles directly. In Postgres SQL, this is done with `percent_rank`. Polars does not have a direct equivalent, but you can match the Postgres behavior by combining expressions, including `rank`. Before implementing this in Polars, it's helpful to review how `percent_rank` is defined.

#### The formula

$$\text{percent\_rank} = \frac{\text{rank} - 1}{\text{total\_rows} - 1}$$



**Note**
In Postgres, `percent_rank` is a window function based on `rank`, not `row_number`. All tied values receive the same rank, and the next rank is skipped. To mimic this behavior in Polars, use `rank(method='min')`.

Percentile calculations in Polars often use aggregations with the `over` expression, similar to SQL's `partition by`. The example below computes total sales per SKU for 2014, then ranks the SKUs by their total sales.

```
(clothing_store_sales
 .with_columns(Year=pl.col('Date').dt.year())
 .filter(pl.col('Year') == 2014)
 .group_by('Sku','Year')
 .agg(pl.sum('Ext_Price'))
 .with_columns(Pct_Rank=(pl.col('Ext_Price').rank('min') - 1) / (pl.len() - 1))
 .sort('Pct_Rank')
 )
```

---

```
shape: (10, 4)
+----------+------+-----------+----------+
| Sku      | Year | Ext_Price | Pct_Rank |
| ---      | ---  | ---       | ---      |
| str      | i32  | f64       | f64      |
```

```
+=======================================+
| HX-24728 | 2014 | 30086.75  | 0.0      |
| LK-02338 | 2014 | 33828.48  | 0.111111 |
| GS-95639 | 2014 | 35141.11  | 0.222222 |
| XX-25746 | 2014 | 38738.46  | 0.333333 |
| FT-50146 | 2014 | 39826.58  | 0.444444 |
| AE-95093 | 2014 | 44515.29  | 0.555556 |
| RU-25060 | 2014 | 48418.29  | 0.666667 |
| IC-59308 | 2014 | 48471.12  | 0.777778 |
| QS-76400 | 2014 | 51783.84  | 0.888889 |
| ZC-07383 | 2014 | 54538.08  | 1.0      |
+----------+------+-----------+----------+
```

You can also perform this ranking per year by keeping both years in the data and ranking within each year:

```
(clothing_store_sales
 .with_columns(Year=pl.col('Date').dt.year())
 .group_by('Sku','Year')
 .agg(pl.sum('Ext_Price'))
 .with_columns(((pl.col('Ext_Price').rank('min').over('Year') - 1)
                .truediv(pl.len().over('Year') - 1))
                .alias('Pct_Rank'))
 .sort('Year','Pct_Rank')
 )
```

---

```
shape: (20, 4)
+----------+------+-----------+----------+
| Sku      | Year | Ext_Price | Pct_Rank |
| ---      | ---  | ---       | ---      |
| str      | i32  | f64       | f64      |
+=======================================+
| ZC-07383 | 2013 | 6945.68   | 0.0      |
| GS-95639 | 2013 | 9569.33   | 0.111111 |
| HX-24728 | 2013 | 9822.69   | 0.222222 |
| AE-95093 | 2013 | 12631.68  | 0.333333 |
| FT-50146 | 2013 | 13732.11  | 0.444444 |
| ...      | ...  | ...       | ...      |
| AE-95093 | 2014 | 44515.29  | 0.555556 |
| RU-25060 | 2014 | 48418.29  | 0.666667 |
| IC-59308 | 2014 | 48471.12  | 0.777778 |
| QS-76400 | 2014 | 51783.84  | 0.888889 |
| ZC-07383 | 2014 | 54538.08  | 1.0      |
+----------+------+-----------+----------+
```

The `over` expression divides the data by year (2013 and 2014) and computes a percent rank within each group.

Percent rank is less suited to binning than n-tiles, but it can be used to create continuous distributions or serve directly as an output, or even used as input for further analysis. Both n-tile and percent rank calculations require sorting and may be expensive with large datasets. Filtering early to reduce the number of rows helps improve performance.

There is no single correct way to examine data distributions. Analysts have flexibility in how they apply these techniques to understand data and communicate findings. At the same time, data scientists must use judgment and follow data ethics guidelines when working with sensitive information.

## Profiling: Data Quality

If the quality of your data is poor, the quality of your analysis will suffer as well. This may seem obvious, yet for many data professionals, myself included, it has been a lesson learned through experience. It's easy to focus on the mechanics of processing data, perfecting Polars code to make it more idiomatic, or crafting the right visualization, only to have stakeholders overlook all of that and point out a single data inconsistency. Ensuring data quality can be one of the most challenging and frustrating parts of analysis. The issue is not limited to the idea of "garbage in, garbage out." You may have strong inputs, but incorrect assumptions can still produce poor results.

Ideally, you would compare your data against a ground truth, or what is otherwise known to be correct, but this is not always possible. For example, when processing a CSV export from a production system alongside a replica file, you can compare the row counts in each dataframe to verify that all records were captured. In other situations, you might know the expected sales totals for a given month and compute aggregates, such as summing a *Sales* column or counting rows, to confirm the results match those expectations. Often, discrepancies between your computed values and the expected ground truth can be traced to whether you applied the appropriate filters, such as excluding cancelled orders or test accounts, how you handled nulls or inconsistent spellings in categorical columns, and whether you joined dataframes correctly on the intended columns.

Profiling provides a proactive way to surface data quality issues before they distort our analysis. It exposes inconsistent datetime formats, reveals the presence of nulls, and identifies categorical encodings that need interpretation. It can also show when columns contain different kinds of values packed into a single column. In addition, profiling highlights gaps or sudden shifts in the data that may result from logging outages, schema changes, or incomplete exports. While data is rarely perfect, system-

atic profiling ensures that problems are detected early, reduces unexpected issues during analysis, and makes downstream transformations more reliable.

## Detecting Duplicates

A *duplicate* is a row that appears more than once in the dataframe with the same values. Duplicates can occur for many reasons, such as:

- Someone entering the same record twice or copying and pasting incorrectly.
- APIs or logging systems resending a payload after a timeout, which creates repeated entries.
- Multiple snapshots of the same record saved without unique identifiers.
- A batch job or ETL pipeline running more than once and appending identical data.

Duplicates, however they arise, are like sand in the gears of your analysis. They can significantly distort your results. Early in my career, I analyzed financial data for the accounting department without checking for duplicates. As a result, my figures were almost doubled. The accountant I was assisting noticed the issue immediately. When she told me that my values were nearly twice what they should have been, I knew the cause right away. My middle school math teacher, Mr. Chipo, often told us, "You should have a feeling for numbers." He meant that you must train your intuition well enough to detect when calculations are far outside a reasonable range. I should have followed that advice during that analysis. It was embarrassing to discover that the hours I had invested were wasted because the results were clearly incorrect. Since then, checking for duplicates has become a habit I hardly need to think about.

The good news is that checking for duplicates in Polars is straightforward. When you load data and display the dataframe, Polars shows the total number of rows. You can then use the `is_duplicated` method to identify repeated rows. The example below creates a dataframe named *student_grades* with one duplicate.

```
shape: (7, 2)
+---------+-------+
| Student | Grade |
| ---     | ---   |
| str     | i64   |
+=================+
| Spencer | 93    |
| Emily   | 88    |
| Hannah  | 73    |
| Aria    | 69    |
| Mona    | 55    |
| Paige   | 89    |
| Aria    | 69    |
```

```
+---------+-------+
```

If you call `is_duplicated` on *student_grades*, the output may not look very useful at first.

```
student_grades.is_duplicated()
```

---

```
shape: (7,)
Series: '' [bool]
[
    false
    false
    false
    true
    false
    false
    true
]
```

The result is a boolean mask. Most values are false, which means they are not duplicates, while the two true values indicate duplicated rows. This output alone is not very informative. To view the actual duplicate rows, use the boolean mask to filter the dataframe.

```
(student_grades
 .filter(student_grades.is_duplicated())
 )
```

---

```
shape: (2, 2)
+---------+-------+
| Student | Grade |
| ---     | ---   |
| str     | i64   |
+=================+
| Aria    | 69    |
| Aria    | 69    |
+---------+-------+
```

Another way to detect duplicates is by using `len` together with `over`. In the code below, `over` groups by all columns, and `len` counts the number of occurrences for each row. The counts are added to a new column called *Len*. Filtering for rows with a count greater than 1 reveals the duplicates.

```
(student_grades
 .with_columns(Len=pl.len().over('Student','Grade'))
```

```
 .filter(pl.col('Len') > 1)
)
```

```
shape: (2, 3)
+---------+-------+-----+
| Student | Grade | Len |
| ---     | ---   | --- |
| str     | i64   | u32 |
+=======================+
| Aria    | 69    | 2   |
| Aria    | 69    | 2   |
+---------+-------+-----+
```

**Warning**

`over` is a special type of group by that does not reduce the height of the dataframe. It's, however, an expensive operation, so use it sparingly.

Detecting duplicates is only part of the process. It's even more important to understand why duplicates appear and, if possible, address the underlying cause. Some helpful questions to consider are:

1. Can the data collection process be improved to prevent duplication?
2. Are multiple data sources being combined without proper deduplication rules?
3. Have you overlooked a one-to-many relationship in a `join`?

## Deduplication with Group By and Unique

Duplicates are not always a sign of poor data quality. Consider a scenario where you want to create a list of employees who have contributed to a project so they can be recognized in a company newsletter. You might `join` the employees table with the projects table so that only employees who appear in the projects table are included.

Below is the *employees* dataframe.

```
shape: (8, 2)
+----------------+-------------+
| Name           | Employee_ID |
| ---            | ---         |
| str            | str         |
+==============================+
| Joram Mutenge  | E0001       |
| Marion Daigle  | E0002       |
| Ashwin Bhakre  | E0003       |
| Eleonora Kiziv | E0004       |
| Sasha Crespi   | E0005       |
```

```
| Paul Lee       | E0006       |
| Oliver Haynes  | E0007       |
| David Nesbitt  | E0008       |
+----------------+-------------+
```

And here's the *projects* dataframe.

```
shape: (10, 2)
+-------------+-------------+
| Employee_ID | Contributed |
| ---         | ---         |
| str         | str         |
+=============================+
| E0004       | Yes         |
| E0002       | Yes         |
| E0004       | Yes         |
| E0003       | Yes         |
| E0004       | Yes         |
| E0005       | Yes         |
| E0003       | Yes         |
| E0004       | Yes         |
| E0001       | Yes         |
| E0006       | Yes         |
+-------------+-------------+
```

Now let's join the two dataframes to see the employees who contributed to at least one project.

```
(employees
 .join(projects, on='Employee_ID', how='inner')
 )
```

---

```
shape: (10, 3)
+----------------+-------------+-------------+
| Name           | Employee_ID | Contributed |
| ---            | ---         | ---         |
| str            | str         | str         |
+=============================================+
| Eleonora Kiziv | E0004       | Yes         |
| Marion Daigle  | E0002       | Yes         |
| Eleonora Kiziv | E0004       | Yes         |
| Ashwin Bhakre  | E0003       | Yes         |
| Eleonora Kiziv | E0004       | Yes         |
| Sasha Crespi   | E0005       | Yes         |
| Ashwin Bhakre  | E0003       | Yes         |
| Eleonora Kiziv | E0004       | Yes         |
| Joram Mutenge  | E0001       | Yes         |
| Paul Lee       | E0006       | Yes         |
```

```
+----------------+-------------+-------------+
```

This produces a row for each employee for each project. Many employees work on multiple projects, so duplicates appear simply because of how the join is structured. There is no data quality issue. You just need to remove repeated rows. One way to do this in Polars is to use the `unique` method.

```
(employees
 .join(projects, on='Employee_ID', how='inner')
 .unique()
 )
```

---

```
shape: (6, 3)
+----------------+-------------+-------------+
| Name           | Employee_ID | Contributed |
| ---            | ---         | ---         |
| str            | str         | str         |
+=============================================+
| Paul Lee       | E0006       | Yes         |
| Joram Mutenge  | E0001       | Yes         |
| Ashwin Bhakre  | E0003       | Yes         |
| Eleonora Kiziv | E0004       | Yes         |
| Sasha Crespi   | E0005       | Yes         |
| Marion Daigle  | E0002       | Yes         |
+----------------+-------------+-------------+
```

When you call `unique`, Polars compares entire rows and keeps each distinct row only once. A row duplicated two or six times will appear only once. If the dataframe shrinks after using `unique`, duplicate rows were present. If it remains the same size, no duplicates exist. To deduplicate based on a single column, include the column name:

```
df.unique('Col_1')
```

To deduplicate on multiple columns, provide a list of column names:

```
df.unique(['Col_1','Col_2'])
```

Another deduplication method is to use `group_by` with `first` as the aggregation. While this may feel less intuitive, it functions similarly to `unique`.

```
(employees
 .join(projects, on='Employee_ID', how='inner')
 .group_by(pl.all()).first() ❶
 )
```

❶ `pl.all()` is Polars shorthand for selecting every column. The SQL-like syntax

`pl.col('*')` produces the same result. The `first` aggregation keeps the first occurrence of each repeated row and removes the rest.

---

```
shape: (6, 3)
+---------------+-------------+-------------+
| Name          | Employee_ID | Contributed |
| ---           | ---         | ---         |
| str           | str         | str         |
+=============================================+
| Paul Lee      | E0006       | Yes         |
| Ashwin Bhakre | E0003       | Yes         |
| Joram Mutenge | E0001       | Yes         |
| Eleonora Kiziv| E0004       | Yes         |
| Sasha Crespi  | E0005       | Yes         |
| Marion Daigle | E0002       | Yes         |
+---------------+-------------+-------------+
```

Duplicate data, or data containing multiple records per entity, even when they are not literal duplicates, is one of the most common sources of incorrect results. If your number of customers or total sales unexpectedly multiplies, duplicates may be the cause. Fortunately, Polars provides several tools to prevent this.

Next, I'll focus on missing data, which is another common problem.

## Preparing: Data Cleaning

Profiling often highlights where targeted changes can make data more useful for analysis. Many of these improvements involve `when-then-otherwise` transformations, handling null values, or adjusting data types.

### Cleaning Data with When-Then-Otherwise Transformations

The `when-then-otherwise` expression is a versatile tool for data preparation. It supports cleaning, enrichment, and summarization directly within a dataframe. Even when values are technically correct, they may not be consistent or standardized, which can complicate analysis. By applying `when-then-otherwise`, you can transform values into a uniform format or group them into categories that are easier to interpret. The structure of this expression was introduced earlier in the chapter in the section on binning.

Inconsistent values appear for several reasons:

- Different source systems may use slightly different labels.

- Application code can change over time.
- Options may be presented in multiple languages.
- Users might be allowed to enter free-text values instead of choosing from a predefined list.

For example, a column containing country information may include entries such as "USA," "United States," and "U.S." With `when-then-otherwise`, you can standardize these variations into a single representation to ensure that downstream analysis operates on clean and reliable values.

Below is the *countries* dataframe with untidy values.

```
shape: (5, 1)
+---------------+
| Country       |
| ---           |
| str           |
+===============+
| United States |
| USA           |
| U.S.          |
| CAN           |
| JPN           |
+---------------+
```

You can clean *countries* using the `when-then-otherwise` expression as shown below.

```
(countries
 .with_columns(pl.when(pl.col('Country') == 'United States')
               .then(pl.lit('USA'))
               .when(pl.col('Country') == 'U.S.')
               .then(pl.lit('USA'))
               .otherwise(pl.col('Country'))
               .alias('Country_Cleaned'))
 )
```

---

```
shape: (5, 2)
+---------------+-----------------+
| Country       | Country_Cleaned |
| ---           | ---             |
| str           | str             |
+=================================+
| United States | USA             |
| USA           | USA             |
| U.S.          | USA             |
| CAN           | CAN             |
| JPN           | JPN             |
```

```
+---------------+-----------------+
```

The `when-then-otherwise` expression is also useful for adding categorization or enrichment that is missing from raw data. A common example is the Net Promoter Score (NPS), which organizations use to measure customer sentiment. NPS surveys ask respondents to rate, on a scale from 0 to 10, how likely they are to recommend a company or product. Ratings from 0 to 6 are detractors, 7 and 8 are passive, and 9 and 10 are promoters. The final score is calculated by subtracting the percentage of detractors from the percentage of promoters.[5]

Survey datasets often include optional comments and may be enriched with additional information about the respondent. When working with this type of data in Polars, the first step is usually to assign each response to one of the three categories using `when-then-otherwise`.

Below is the *survey_responses* dataframe:

```
shape: (10, 5)
+-------------+--------+---------+---------+------------+
| Response_ID | Gender | Country | Premium | Likelihood |
| ---         | ---    | ---     | ---     | ---        |
| i64         | str    | str     | str     | i64        |
+=====================================================+
| 10001       | M      | USA     | Yes     | 5          |
| 10002       | M      | JPN     | No      | 3          |
| 10003       | F      | CAN     | Yes     | 10         |
| 10004       | M      | USA     | No      | 7          |
| 10005       | F      | GBR     | No      | 9          |
| 10006       | M      | AUS     | Yes     | 8          |
| 10007       | F      | IND     | No      | 2          |
| 10008       | M      | DEU     | Yes     | 9          |
| 10009       | F      | BRA     | No      | 5          |
| 10010       | M      | FRA     | Yes     | 10         |
+-------------+--------+---------+---------+------------+
```

You can assign the appropriate categories to each response as shown below:

```
(survey_responses
 .select('Response_ID','Likelihood')
 .with_columns(pl.when(pl.col('Likelihood') <= 6)
               .then(pl.lit('Detractor'))
               .when(pl.col('Likelihood') <= 8)
               .then(pl.lit('Passive'))
               .otherwise(pl.lit('Promoter'))
               .alias('Response_Type'))
 )
```

```
shape: (10, 3)
+-------------+------------+---------------+
| Response_ID | Likelihood | Response_Type |
| ---         | ---        | ---           |
| i64         | i64        | str           |
+=========================================+
| 10001       | 5          | Detractor     |
| 10002       | 3          | Detractor     |
| 10003       | 10         | Promoter      |
| 10004       | 7          | Passive       |
| 10005       | 9          | Promoter      |
| 10006       | 8          | Passive       |
| 10007       | 2          | Detractor     |
| 10008       | 9          | Promoter      |
| 10009       | 5          | Detractor     |
| 10010       | 10         | Promoter      |
+-------------+------------+---------------+
```

The data type of the evaluated column does not need to match the type returned by the expression. In this example, the condition checks an integer and returns a string. You can also list specific values with `is_in`, which is helpful when the input is not continuous or when values should not be grouped by order.

```
(survey_responses
 .select('Response_ID','Likelihood')
 .with_columns(pl.when(pl.col('Likelihood').is_in([0,1,2,3,4,5,6])) ❶
               .then(pl.lit('Detractor'))
               .when(pl.col('Likelihood').is_in([7,8]))
               .then(pl.lit('Passive'))
               .otherwise(pl.lit('Promoter'))
               .alias('Response_Type'))
 )
```

❶ The list of numbers `[0,1,2,3,4,5,6]` can be replaced with `range(0,7)`.

---

```
shape: (10, 3)
+-------------+------------+---------------+
| Response_ID | Likelihood | Response_Type |
| ---         | ---        | ---           |
| i64         | i64        | str           |
+=========================================+
| 10001       | 5          | Detractor     |
| 10002       | 3          | Detractor     |
| 10003       | 10         | Promoter      |
| 10004       | 7          | Passive       |
| 10005       | 9          | Promoter      |
```

```
| 10006       | 8          | Passive       |
| 10007       | 2          | Detractor     |
| 10008       | 9          | Promoter      |
| 10009       | 5          | Detractor     |
| 10010       | 10         | Promoter      |
+-------------+------------+---------------+
```

The `when-then-otherwise` expression can evaluate multiple columns and can include `and`/`or` logic pipe operator (`|`). Nested conditions are also possible, though `and`/`or` logic often removes the need for nesting.

```
(survey_responses
 .with_columns(pl.when((pl.col('Likelihood') <= 6) &
                       (pl.col('Country') == 'USA') &
                       (pl.col('Premium') == 'Yes'))
               .then(pl.lit('USA premium detractor'))
               .when((pl.col('Likelihood') >= 9) &
                     (pl.col('Country').is_in(['AUS','DEU']) |
                     (pl.col('Premium') == 'No')))
               .then(pl.lit('Some desired label'))
               .otherwise(pl.lit('Promoter'))
               .alias('Response_Type'))
 )
```

---

```
shape: (10, 6)
+-------------+--------+---------+---------+------------+----------------------+
| Response_ID | Gender | Country | Premium | Likelihood | Response_Type        |
| ---         | ---    | ---     | ---     | ---        | ---                  |
| i64         | str    | str     | str     | i64        | str                  |
+=================================================================================+
| 10001       | M      | USA     | Yes     | 5          | USA premium detractor |
| 10002       | M      | JPN     | No      | 3          | Promoter             |
| 10003       | F      | CAN     | Yes     | 10         | Promoter             |
| 10004       | M      | USA     | No      | 7          | Promoter             |
| 10005       | F      | GBR     | No      | 9          | Some desired label   |
| 10006       | M      | AUS     | Yes     | 8          | Promoter             |
| 10007       | F      | IND     | No      | 2          | Promoter             |
| 10008       | M      | DEU     | Yes     | 9          | Some desired label   |
| 10009       | F      | BRA     | No      | 5          | Promoter             |
| 10010       | M      | FRA     | Yes     | 10         | Promoter             |
+-------------+--------+---------+---------+------------+----------------------+
```

**Tip**

When using the `&` operator, it's always safe to wrap conditions in parentheses. Without them, the code often fails. You can replace `&` with comma `,` removing the need for parentheses. I think this produces cleaner expressions.

The previous code can therefore be written as:

```
(survey_responses
 .with_columns(pl.when(pl.col('Likelihood') <= 6,
                       pl.col('Country') == 'USA',
                       pl.col('Premium') == 'Yes')
               .then(pl.lit('USA premium detractor'))
               .when(pl.col('Likelihood') >= 9,
                     (pl.col('Country').is_in(['AUS','DEU']) |
                     (pl.col('Premium') == 'No')))
               .then(pl.lit('Some desired label'))
               .otherwise(pl.lit('Promoter'))
               .alias('Response_Type'))
 )
```

---

```
shape: (10, 6)
+-------------+--------+---------+---------+------------+----------------------+
| Response_ID | Gender | Country | Premium | Likelihood | Response_Type        |
| ---         | ---    | ---     | ---     | ---        | ---                  |
| i64         | str    | str     | str     | i64        | str                  |
+=================================================================================+
| 10001       | M      | USA     | Yes     | 5          | USA premium detractor |
| 10002       | M      | JPN     | No      | 3          | Promoter             |
| 10003       | F      | CAN     | Yes     | 10         | Promoter             |
| 10004       | M      | USA     | No      | 7          | Promoter             |
| 10005       | F      | GBR     | No      | 9          | Some desired label   |
| 10006       | M      | AUS     | Yes     | 8          | Promoter             |
| 10007       | F      | IND     | No      | 2          | Promoter             |
| 10008       | M      | DEU     | Yes     | 9          | Some desired label   |
| 10009       | F      | BRA     | No      | 5          | Promoter             |
| 10010       | M      | FRA     | Yes     | 10         | Promoter             |
+-------------+--------+---------+---------+------------+----------------------+
```

This version is often easier to read because it uses fewer parentheses.



**Expand Your Knowledge**

*ALTERNATIVES FOR CLEANING DATA*

Cleaning or enriching data with a `when-then-otherwise` expression works well when the list of variations is short, easy to enumerate, and unlikely to change often. This approach is straightforward and keeps the logic contained within your transformation pipeline.

For longer or frequently changing lists, a separate lookup dataframe can be a better option. Instead of chaining many conditions, you maintain a mapping table that can be joined with your main dataset. This allows cleaned values to be updated independently of the transformation logic and reused across multiple workflows. For example, a lookup dataframe might map product category codes to their full descriptive names.

Many people begin with `when-then-otherwise` because it's quick and simple. Once the list becomes difficult to manage, or the same cleaning step is needed in multiple places, moving to a lookup dataframe makes maintenance easier and ensures consistency across projects.

It's also worth considering whether the data can be cleaned upstream. A small set of conditions can grow into dozens or even hundreds, which increases the complexity of your transformations. In some cases, the insights gained during cleaning justify asking engineers to modify the source system so meaningful categorizations appear in the data stream directly. This reduces the need for extensive downstream cleaning.

Another common use of the `when-then-otherwise` expression is to build indicator columns (flags) that mark whether a condition is met rather than returning a specific value. These indicators are especially helpful during data profiling because they make it easy to see how often a particular attribute occurs. They are also widely used when preparing datasets for statistical modeling, where they are often referred to as dummy variables. A dummy variable takes on values of 0 or 1 to indicate the absence or presence of a qualitative feature. For instance, you might create *Is_Female* and *Is_Promoter* flags based on the *Gender* and *Likelihood* (to recommend) columns.

```
(survey_responses
 .with_columns(pl.when(pl.col('Gender') == 'F')
               .then(1)
               .otherwise(0)
               .alias('Is_Female'))
 .with_columns(pl.when(pl.col('Likelihood').is_in([9,10]))
               .then(1)
               .otherwise(0)
               .alias('Is_Promoter'))
 )
```

```
shape: (10, 7)
+-------------+--------+---------+---------+------------+-----------+-------------+
| Response_ID | Gender | Country | Premium | Likelihood | Is_Female | Is_Promoter |
| ---         | ---    | ---     | ---     | ---        | ---       | ---         |
| i64         | str    | str     | str     | i64        | i32       | i32         |
+=================================================================================+
| 10001       | M      | USA     | Yes     | 5          | 0         | 0           |
| 10002       | M      | JPN     | No      | 3          | 0         | 0           |
| 10003       | F      | CAN     | Yes     | 10         | 1         | 1           |
| 10004       | M      | USA     | No      | 7          | 0         | 0           |
| 10005       | F      | GBR     | No      | 9          | 1         | 1           |
| 10006       | M      | AUS     | Yes     | 8          | 0         | 0           |
| 10007       | F      | IND     | No      | 2          | 1         | 0           |
```

```
| 10008       | M      | DEU     | Yes     | 9          | 0         | 1           |
| 10009       | F      | BRA     | No      | 5          | 1         | 0           |
| 10010       | M      | FRA     | Yes     | 10         | 0         | 1           |
+-------------+--------+---------+---------+------------+-----------+-------------+
```

Using `when-then-otherwise` to transform categorical values into dummy variables works, but an even more idiomatic approach is to use `to_dummies`. The code below transforms *Gender* into a dummy variable column.

```
(survey_responses
 .select('Gender')
 .to_dummies('Gender')
 )
```

---

```
shape: (10, 2)
+----------+----------+
| Gender_F | Gender_M |
| ---      | ---      |
| u8       | u8       |
+=====================+
| 0        | 1        |
| 0        | 1        |
| 1        | 0        |
| 0        | 1        |
| 1        | 0        |
| 0        | 1        |
| 1        | 0        |
| 0        | 1        |
| 1        | 0        |
| 0        | 1        |
+----------+----------+
```

When working with datasets that contain multiple rows per customer, such as subscription records across different streaming services, you can simplify the data by creating flags with `when-then-otherwise` expressions combined with an aggregation. In Polars, flags are often Boolean fields that indicate the presence or absence of an attribute. Sometimes these Booleans are expressed as 1 and 0 so that aggregate functions can be applied easily.

For example, imagine each row represents a customer's subscription to a streaming service. You can create a flag that marks whether a customer has ever subscribed to Netflix. The logic returns 1 for any row where the subscription type is Netflix and 0 otherwise. By applying a `max` aggregation across all rows for each customer, you capture whether Netflix appears at least once. As long as a customer subscribed to Netflix at least once, the flag will be 1; otherwise, it will be 0. The same approach can be

used to create a flag for Hulu subscriptions.

Below is the *streaming_subs* dataframe.

```
streaming_subs = pl.DataFrame({
    'Customer_ID': [1001, 1002, 1003, 1004, 1005,
                    1003, 1006, 1007, 1008, 1009],
    'Subscription': ['Netflix','Hulu','Hulu',
                     'AMC','Paramount','Netflix',
                     'Hulu','Netflix','Netflix','Netflix'],
    'Months_Subscribed': [8, 7, 4, 9, 2, 5, 1, 6, 10, 2]})

streaming_subs
```

---

```
shape: (10, 3)
+-------------+--------------+-------------------+
| Customer_ID | Subscription | Months_Subscribed |
| ---         | ---          | ---               |
| i64         | str          | i64               |
+=================================================+
| 1001        | Netflix      | 8                 |
| 1002        | Hulu         | 7                 |
| 1003        | Hulu         | 4                 |
| 1004        | AMC          | 9                 |
| 1005        | Paramount    | 2                 |
| 1003        | Netflix      | 5                 |
| 1006        | Hulu         | 1                 |
| 1007        | Netflix      | 6                 |
| 1008        | Netflix      | 10                |
| 1009        | Netflix      | 2                 |
+-------------+--------------+-------------------+
```

And here's the code that uses a `when-then-otherwise` expression in an aggregation.

```
(streaming_subs
 .group_by('Customer_ID')
 .agg(pl.when(pl.col('Subscription') == 'Netflix')
      .then(1)
      .otherwise(0)
      .max()
      .alias('Netflix_Subscriber'),
     pl.when(pl.col('Subscription') == 'Hulu')
      .then(1)
      .otherwise(0)
      .max()
      .alias('Hulu_Subscriber')
      )
 )
```

You can also construct more complex flag conditions, such as requiring a threshold or minimum duration of subscription before assigning a value of 1:

```
(streaming_subs
 .group_by('Customer_ID')
 .agg(pl.when(pl.col('Subscription') == 'Netflix', pl.col('Months_Subscribed') > 5)
      .then(1)
      .otherwise(0)
      .max()
      .alias('Loyal_Netflix'),
      pl.when(pl.col('Subscription') == 'Hulu', pl.col('Months_Subscribed') > 5)
      .then(1)
      .otherwise(0)
      .max()
      .alias('Loyal_Hulu')
      )
 )
```

---

```
shape: (9, 3)
```

| Customer_ID | Loyal_Netflix | Loyal_Hulu |
|---|---|---|
| i64 | i32 | i32 |
| 1005 | 0 | 0 |
| 1004 | 0 | 0 |
| 1001 | 1 | 0 |
| 1009 | 0 | 0 |
| 1002 | 0 | 1 |
| 1006 | 0 | 0 |
| 1008 | 1 | 0 |
| 1003 | 0 | 0 |
| 1007 | 1 | 0 |

The `when-then-otherwise` expression is a powerful tool. As shown above, it can clean messy fields, enrich datasets with derived values, and create flags or dummy variables that make qualitative attributes easier to analyze. In the next section, I will explore several of Polars' specialized functions for handling null values, which extend this same logic to situations where data may be missing or incomplete.

## Type Conversions and Casting

Unlike pandas, which is lenient about column data types, Polars enforces strict type consistency. Each column in a Polars dataframe must use a single, consistent data

type. When you load or create data, Polars attempts to infer each column's type and usually gets it right. If Polars cannot determine a column's type, it defaults to `String`. Mixed data types within the same column are not allowed. For example, you cannot store strings in an integer column or place booleans in a date column. This strictness on types supports optimized performance and efficient memory use.

Having the correct data types in your columns lets you take advantage of functions that apply specifically to those types. String functions like `to_uppercase` and `strip_chars` work on string columns, and datetime functions like `weekday` and `total_hours` apply to datetime columns. However, you may sometimes need to override the data type and treat a column as a different type. This is where type casting becomes essential. Polars lets you convert a column's type so you can use the appropriate operations on it.

*Type conversion functions* allow properly formatted values to be converted from one data type to another. In Polars, the keyword used for type conversion is `cast`. The dataframe below shows a column with `String` data type.

---

```
shape: (5, 1)
+-------------+
| Customer_ID |
| ---         |
| str         |
+=============+
| 2367        |
| 5498        |
| 9975        |
| 4326        |
| 3302        |
+-------------+
```

To change a string column to an integer column, write:

```
(customer_nums
 .with_columns(pl.col('Customer_ID').cast(pl.Int32))
 )
```

---

```
shape: (5, 1)
+-------------+
| Customer_ID |
| ---         |
| i32         |
+=============+
```

```
| 2367        |
| 5498        |
| 9975        |
| 4326        |
| 3302        |
+-------------+
```

Alternatively, you can use the `to_integer` expression:

```
(customer_nums
 .with_columns(pl.col('Customer_ID').str.to_integer())
 )
```

---

```
shape: (5, 1)
+-------------+
| Customer_ID |
| ---         |
| i32         |
+=============+
| 2367        |
| 5498        |
| 9975        |
| 4326        |
| 3302        |
+-------------+
```

Although numeric values stored as strings may seem strange, it's a likely scenario when working with price data that includes a currency symbol such as the pound sign (£). Polars assigns a `String` type to such columns, which prevents mathematical operations. You can remove the £ character with the `replace` expression, which will be discussed further when we look at text analysis in Chapter 5.

Below is a dataframe showing price values in pounds:

---

```
shape: (5, 2)
+---------------------------+-------+
| Item                      | Price |
| ---                       | ---   |
| str                       | str   |
+===================================+
| Free Range Eggs (12 pack) | £2.85 |
| Wholemeal Bread           | £1.20 |
| Orange Juice (1L)         | £2.00 |
| Greek Yogurt (500g)       | £1.75 |
| Porridge Oats (1kg)       | £1.50 |
```

```
+--------------------------+-------+
```

The following code removes the £ symbol from the *Price* column, which allows casting the values to `Float32`:

```
(breakfast_foods
 .with_columns(pl.col('Price').str.replace('£','').cast(pl.Float32))
 )
```

---

```
shape: (5, 2)
+--------------------------+-------+
| Item                     | Price |
| ---                      | ---   |
| str                      | f32   |
+==================================+
| Free Range Eggs (12 pack) | 2.85  |
| Wholemeal Bread          | 1.2   |
| Orange Juice (1L)        | 2.0   |
| Greek Yogurt (500g)      | 1.75  |
| Porridge Oats (1kg)      | 1.5   |
+--------------------------+-------+
```

**Note**
Directly casting to a numeric data type such as `Float32` does not work unless the £ character is removed first. Since £ is not a numeric character, conversion only succeeds when the column contains only numeric characters.

Dates and datetimes appear in many formats, so you need to know how to cast them correctly in Polars. This section introduces a few examples, and Chapter 3 will explore date and datetime operations in more depth.

Imagine you are working with transaction or event data recorded at the minute level, but want to summarize values by day. If you group directly by the full datetime, you will end up with far more rows than necessary. By casting a `Datetime` column to a `Date`, you reduce the granularity, shrink the output, and obtain the daily summary you need:

```
(clothing_store_sales
 .group_by(pl.col('Date').dt.date()).len()
 )
```

---

```
shape: (352, 2)
+------------+-----+
| Date       | len |
```

```
| ---        | --- |
| date       | u32 |
+===================+
| 2014-09-15 | 2   |
| 2014-07-04 | 3   |
| 2013-12-21 | 2   |
| 2014-02-17 | 5   |
| 2014-08-03 | 4   |
| ...        | ... |
| 2013-12-07 | 2   |
| 2014-06-24 | 4   |
| 2014-05-13 | 2   |
| 2013-11-24 | 2   |
| 2013-12-29 | 2   |
+------------+-----+
```

Sometimes the year, month, and day values are stored in separate columns because they were parsed from a longer string. Polars lets you combine these values back into a date using `pl.date`, placing each component in the proper order `(year, month, day)`.

The code below creates a new column, *Date_Bought*, with September 29, 2009 as the date:

```
(clothing_store_sales
 .select('Category')
 .unique()
 .with_columns(Date_Bought=pl.date(2009,7,29))
 )
```

---

```
shape: (3, 2)
+----------+-------------+
| Category | Date_Bought |
| ---      | ---         |
| str      | date        |
+========================+
| Hat      | 2009-07-29  |
| Socks    | 2009-07-29  |
| Sweater  | 2009-07-29  |
+----------+-------------+
```

You can also create the date as a string, separated by dashes `-` or forward slashes `/`, and convert it to a `Date` using the `to_date` expression:

```
(clothing_store_sales
 .select('Category')
 .unique()
```

```
 .with_columns(Date_Bought=pl.lit('2012-10-18').str.to_date())
 )
```

```
shape: (3, 2)
+----------+-------------+
| Category | Date_Bought |
| ---      | ---         |
| str      | date        |
+========================+
| Sweater  | 2012-10-18  |
| Socks    | 2012-10-18  |
| Hat      | 2012-10-18  |
+----------+-------------+
```

**Tip**

For `to_date` to function correctly, the date components must be arranged in one of two ways: `(day, month, year)` or `(year, month, day)`.

Some data types take precedence when used together. For example, mathematical operations between integers and floats produce a float.

## Reducing Memory Usage by Using Appropriate Data Types

Polars is an in-memory data manipulation library. When you read a dataset, the data is stored in your machine's memory (RAM). Keeping data in memory results in faster operations because Polars does not need to reread the data each time you run a new operation. This differs from SQL, where each query retrieves data from the database. However, some datasets may be too large to fit entirely in memory. Lazy mode helps because it allows us to execute our query without loading the full dataset. Only the necessary subset is returned when the `collect` method is called on the `LazyFrame`. It's also possible to load the entire dataset with lazy mode if no filtering is applied on rows or columns.

Assigning appropriate data types can further reduce the amount of memory used to store a dataset. Lower memory usage generally improves performance. A smaller dataset is almost always faster to query than a larger one.

To demonstrate how data types affect memory usage, I load a dataset containing sales transactions from a UK-based online store. I then show the initial size of the data in memory, apply the appropriate data types, and compute the percentage reduction to see how much memory was saved.

Below is the *online_retail_sales* dataframe:

```
online_retail_sales = pl.read_excel('data/uk_online_retail_sales.xlsx')
online_retail_sales
```

```
shape: (541_909, 8)
+-----------+-----------+------------------------+-----+------------+----------------+
| InvoiceNo | StockCode | Description            | ... | CustomerID | Country        |
| ---       | ---       | ---                    |     | ---        | ---            |
| i64       | str       | str                    |     | i64        | str            |
+=============================================================================================+
| 536365    | 85123A    | WHITE HANGING HEART ... | ... | 17850      | United Kingdom |
| 536365    | 71053     | WHITE METAL LANTERN    | ... | 17850      | United Kingdom |
| 536365    | 84406B    | CREAM CUPID HEARTS C... | ... | 17850      | United Kingdom |
| 536365    | 84029G    | KNITTED UNION FLAG H... | ... | 17850      | United Kingdom |
| 536365    | 84029E    | RED WOOLLY HOTTIE WH... | ... | 17850      | United Kingdom |
| ...       | ...       | ...                    | ... | ...        | ...            |
| 581587    | 22613     | PACK OF 20 SPACEBOY ... | ... | 12680      | France         |
| 581587    | 22899     | CHILDREN'S APRON DOL... | ... | 12680      | France         |
| 581587    | 23254     | CHILDRENS CUTLERY DO... | ... | 12680      | France         |
| 581587    | 23255     | CHILDRENS CUTLERY CI... | ... | 12680      | France         |
| 581587    | 22138     | BAKING SET 9 PIECE R... | ... | 12680      | France         |
+-----------+-----------+------------------------+-----+------------+----------------+
```

To check the size of the loaded dataframe in megabytes (MB), run:

```
initial_memory_usage = online_retail_sales.estimated_size('mb')
initial_memory_usage
```

```
43.96474361419678
```

The following code assigns more efficient data types to columns that benefit from conversion:

```
final_memory_usage = (online_retail_sales
 .with_columns(pl.col('CustomerID').cast(pl.Int16),
               pl.col('Quantity').cast(pl.Int32),
               pl.col('InvoiceNo').cast(pl.Int32),
               pl.col('Country').cast(pl.Categorical),
               pl.col('UnitPrice').cast(pl.Float16))
 .estimated_size('mb')
 )
final_memory_usage
```

```
28.782983779907227
```

To calculate the percentage reduction in memory usage, run:

```
((final_memory_usage / initial_memory_usage) - 1) * 100
```

---

```
-34.53166921093375
```

The size of the loaded dataset is reduced by 35 percent. Although this change may seem small for a dataset of this size, the impact becomes significant when working with gigabytes (GB) of data.

You may wonder how to determine which data types are appropriate. For example, how do you decide between `Int16` and `Int32`? Each data type has a lower bound (minimum value) and an upper bound (maximum value). If all values in a column fall within those bounds, the data type is a suitable choice. Polars can tell us exactly what these limits are.

Before checking bounds, start by generating summary statistics for the dataframe:

```
online_retail_sales.describe()
```

---

```
shape: (9, 9)
+------------+--------------+-----------+-----+-----------+-------------+-------------+
| statistic  | InvoiceNo    | StockCode | ... | UnitPrice | CustomerID  | Country     |
| ---        | ---          | ---       |     | ---       | ---         | ---         |
| str        | f64          | str       |     | f64       | f64         | str         |
+=================================================================================+
| count      | 532618.0     | 541909    | ... | 541909.0  | 406829.0    | 541909      |
| null_count | 9291.0       | 0         | ... | 0.0       | 135080.0    | 0           |
| mean       | 559965.752027 | null      | ... | 4.611114  | 15287.69057 | null        |
| std        | 13428.417281 | null      | ... | 96.759853 | 1713.600303 | null        |
| min        | 536365.0     | 10002     | ... | -11062.06 | 12346.0     | Australia   |
| 25%        | 547906.0     | null      | ... | 1.25      | 13953.0     | null        |
| 50%        | 560688.0     | null      | ... | 2.08      | 15152.0     | null        |
| 75%        | 571841.0     | null      | ... | 4.13      | 16791.0     | null        |
| max        | 581587.0     | m         | ... | 38970.0   | 18287.0     | Unspecified |
+------------+--------------+-----------+-----+-----------+-------------+-------------+
```

Focus on the numerical columns and review their minimum and maximum values. If the maximum value in a column exceeds the upper bound of the data type you are considering, that data type is not appropriate. Both the minimum and maximum values should fall within the allowable range.

For example, to inspect the bounds of `Int16`, use:

```
pl.select(Min=pl.Int16.min(), Max=pl.Int16.max())
```

```
shape: (1, 2)
+--------+-------+
| Min    | Max   |
| ---    | ---   |
| i16    | i16   |
+================+
| -32768 | 32767 |
+--------+-------+
```

We can replace 16 with 8, 32, or 64 to check other integer types. For float types, Polars has bounds `-inf` and `inf`. We can get slightly precise bounds with the **NumPy** library using `finfo`:

```
import numpy as np

np.finfo('float32')
```

---

```
finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)
```

I also converted *Country* from `String` to `Categorical` because it has low cardinality, meaning it contains a relatively small number of unique values (only 38).



**Note**
The threshold between low and high cardinality depends on your use case. As a general guideline, if a column has fewer than 100 unique values, I treat it as low cardinality. In some cases, converting from `String` to `Categorical` when the number of unique values is around 120 does not reduce memory usage and may even increase it.

## Dealing with Nulls and Empty Strings

In Polars, as in many other in-process analytical query engines, `null` and the empty string (`""`) represent different concepts. A `null` value indicates that the data is missing, unknown, or not applicable. More importantly, `null` can occur in any data type, not just strings. When a value is `null`, it means the cell contains no value at all.

An empty string (`""`) is a valid string with a length of zero. It shows that a string value is present, but it contains no characters.

Operations on `null` generally return `null`, except with horizontal addition. The dataframe below shows two prices paid by each customer and the two items each customer purchased.

```
shape: (3, 5)
```

```
+----------+---------+---------+---------+---------+
| Customer | Price_1 | Price_2 | Item_1  | Item_2  |
| ---      | ---     | ---     | ---     | ---     |
| str      | i64     | i64     | str     | str     |
+=============================================+
| David    | 3       | 9       | Mango   | null    |
| Ashwin   | 12      | null    | Guava   |         |
| Aja      | 14      | 7       | Cassava | Peanuts |
+----------+---------+---------+---------+---------+
```

Next, I'll examine the results of mathematical operations on the two price columns.

```
(df
 .select('Customer','Price_1','Price_2')
 .with_columns(Horizontal_Add=pl.sum_horizontal('Price_1','Price_2'),
               Add=pl.col('Price_1') + pl.col('Price_2'),
               Subtract=pl.col('Price_1') - pl.col('Price_2'),
               Multiply=pl.col('Price_1') * pl.col('Price_2'),
               Divide=pl.col('Price_1') / pl.col('Price_2'),)
 )
```

---

```
shape: (3, 8)
+----------+---------+---------+----------------+-----+----------+----------+----------+
| Customer | Price_1 | Price_2 | Horizontal_Add | ... | Subtract | Multiply | Divide   |
| ---      | ---     | ---     | ---            |     | ---      | ---      | ---      |
| str      | i64     | i64     | i64            |     | i64      | i64      | f64      |
+=============================================================================================+
| David    | 3       | 9       | 12             | ... | -6       | 27       | 0.333333 |
| Ashwin   | 12      | null    | 12             | ... | null     | null     | null     |
| Aja      | 14      | 7       | 21             | ... | 7        | 98       | 2.0      |
+----------+---------+---------+----------------+-----+----------+----------+----------+
```

**Warning**

It's unexpected that `sum_horizontal` and adding two columns directly do not produce the same result. This behavior is consistent for string values as well.

For string values, addition is the only supported operation.

```
(df
 .select('Customer','Item_1','Item_2')
 .with_columns(Horizontal_Add=pl.sum_horizontal('Item_1','Item_2'),
               Add=pl.col('Item_1') + pl.col('Item_2'))
 )
```

---

```
shape: (3, 5)
```

```
+----------+---------+---------+----------------+----------------+
| Customer | Item_1  | Item_2  | Horizontal_Add | Add            |
| ---      | ---     | ---     | ---            | ---            |
| str      | str     | str     | str            | str            |
+==========+=========+=========+================+================+
| David    | Mango   | null    | Mango          | null           |
| Ashwin   | Guava   |         | Guava          | Guava          |
| Aja      | Cassava | Peanuts | CassavaPeanuts | CassavaPeanuts |
+----------+---------+---------+----------------+----------------+
```

Nulls are often inconvenient for analysis and can make results harder to interpret. They may also confuse business users who are unfamiliar with what a `null` value represents or who might assume it signals a data quality issue.

Fortunately, filtering `null` values is straightforward. For example, you can remove rows where *Item_2* is `null`.

```
(df
 .filter(pl.col('Item_2').is_not_null())
 )
```

---

```
shape: (2, 5)
+----------+---------+---------+---------+---------+
| Customer | Price_1 | Price_2 | Item_1  | Item_2  |
| ---      | ---     | ---     | ---     | ---     |
| str      | i64     | i64     | str     | str     |
+=================================================+
| Ashwin   | 12      | null    | Guava   |         |
| Aja      | 14      | 7       | Cassava | Peanuts |
+----------+---------+---------+---------+---------+
```

Or you can keep only the rows where *Item_2* is `null`.

```
(df
 .filter(pl.col('Item_2').is_null()) ❶
 )
```

❶ `is_null` does not match rows where *Item_2* is the empty string (`""`).

---

```
shape: (1, 5)
+----------+---------+---------+--------+--------+
| Customer | Price_1 | Price_2 | Item_1 | Item_2 |
| ---      | ---     | ---     | ---    | ---    |
| str      | i64     | i64     | str    | str    |
+=============================================+
| David    | 3       | 9       | Mango  | null   |
```

```
+----------+---------+---------+--------+--------+
```

Instead of removing `null` values, you may want to replace them. The `fill_null` expression replaces `null` values with a constant (such as 10 or 20) or with a strategy (such as `min`, `max`, or `mean`) for numeric columns.

```
(df
 .select('Customer','Price_2')
 .with_columns(Fill_Constant=pl.col('Price_2').fill_null(10),
               Fill_Avg=pl.col('Price_2').fill_null(strategy='mean'))
 )
```

---

```
shape: (3, 4)
┌──────────┬─────────┬───────────────┬──────────┐
│ Customer ┆ Price_2 ┆ Fill_Constant ┆ Fill_Avg │
│ ---      ┆ ---     ┆ ---           ┆ ---      │
│ str      ┆ i64     ┆ i64           ┆ i64      │
╞══════════╪═════════╪═══════════════╪══════════╡
│ David    ┆ 9       ┆ 9             ┆ 9        │
│ Ashwin   ┆ null    ┆ 10            ┆ 8        │
│ Aja      ┆ 7       ┆ 7             ┆ 7        │
└──────────┴─────────┴───────────────┴──────────┘
```

Only two strategies, `min` and `max`, are available for columns with the `String` data type. `min` uses the shortest string in the column as the replacement, while `max` uses the longest. When you use a numeric constant like 10, Polars coerces it to a string.

```
(df
 .select('Customer','Item_2')
 .with_columns(Fill_Numeric=pl.col('Item_2').fill_null(10),
               Fill_Shortest=pl.col('Item_2').fill_null(strategy='min'),
               Fill_Longest=pl.col('Item_2').fill_null(strategy='max'),
               Fill_String=pl.col('Item_2').fill_null('Watermelon'))
 )
```

---

```
shape: (3, 6)
+----------+---------+--------------+---------------+--------------+-------------+
| Customer | Item_2  | Fill_Numeric | Fill_Shortest | Fill_Longest | Fill_String |
| ---      | ---     | ---          | ---           | ---          | ---         |
| str      | str     | str          | str           | str          | str         |
+=================================================================================+
| David    | null    | 10           |               | Peanuts      | Watermelon  |
| Ashwin   |         |              |               |              |             |
| Aja      | Peanuts | Peanuts      | Peanuts       | Peanuts      | Peanuts     |
+----------+---------+--------------+---------------+--------------+-------------+
```

To replace empty strings, use the `replace` expression.

```
(df
 .select('Customer','Item_2')
 .with_columns(pl.col('Item_2').str.replace('','Papaya'))
 )
```

---

```
shape: (3, 2)
+----------+---------------+
| Customer | Item_2        |
| ---      | ---           |
| str      | str           |
+=========================+
| David    | null          |
| Ashwin   | Papaya        |
| Aja      | PapayaPeanuts |
+----------+---------------+
```

Often you may want to treat empty strings and `null` values the same. In that case, convert empty strings to `null`. You might expect `replace('', None)` to work, but it does not. Instead, use a `when-then-otherwise` expression.

```
(df
 .select('Customer','Item_2')
 .with_columns(pl.when(pl.col('Item_2').eq(''))
               .then(None)
               .otherwise(pl.col('Item_2'))
               .alias('Item_2'))
 )
```

---

```
shape: (3, 2)
┌──────────┬──────────┐
│ Customer ┆ Item_2   │
│ ---      ┆ ---      │
│ str      ┆ str      │
╞══════════╪══════════╡
│ David    ┆ null     │
│ Ashwin   ┆ null     │
│ Aja      ┆ Peanuts  │
└──────────┴──────────┘
```

Data can be missing for many reasons, and understanding the root cause is essential when deciding how to address it. Several approaches are available for finding and replacing missing values. These include using `when-then-otherwise` expressions to set defaults, and filling `null` values with a constant or a strategy.

Data cleaning is an important part of data preparation. Cleaning may be required to correct poor data quality, such as inconsistent or missing values, or to make later analysis easier and more meaningful. The flexibility of Polars allows us to carry out these tasks in multiple ways.

After the data is cleaned, the next step in the preparation process is often shaping the dataset.

## Preparing: Shaping Data

In data analysis, the structure of a dataset when read into memory isn't fixed. It's malleable, so we can present it in many forms by adjusting columns, rows, or both. This process is known as *shaping data*. Every dataframe has a shape, and the output of each Polars operation has one as well. Although the idea of shaping data may seem abstract at first, its importance becomes clear as you work with more datasets. Like any skill, it can be learned, practiced, and eventually mastered.

One of the most important aspects of shaping data is determining the level of granularity you need. Just as photographs range from wide panoramic shots to close-up portraits and finally to the individual pixels, data can also exist at different levels of detail. If a company's annual revenue represents the panoramic view, then a department's revenue is the portrait, and the sales from one employee are the pixels. Data at an even more granular level might include individual transactions, customer orders, or the timestamp of every click on a website.

Another key idea in shaping data is *flattening data*. This involves condensing the rows that describe an entity, sometimes to just one. You can flatten data by merging multiple tables into a single result set or by summarizing values through aggregation.

In this section, I will begin by discussing factors to consider when deciding on data shapes. I will then explore common use cases such as pivoting and unpivoting. Throughout the upcoming chapters, you will see practical examples of how data can be reshaped for different types of analysis. Chapter 8 looks more closely at strategies for keeping complex Polars queries manageable when building datasets for advanced analysis.

### Choosing Output: BI, Visualization, Statistics, ML

Shaping your data in Polars depends heavily on what you plan to do with it later. A common best practice is to prepare a dataframe that is as compact as possible while still keeping the level of detail you need. Doing the heavy lifting inside Polars, and taking advantage of its fast query engine and lazy evaluation reduces the amount of

work required in downstream tools. This approach minimizes both the cost of moving data and the overhead of additional transformations. Your output might feed into a BI platform for dashboards, a spreadsheet for business review, a statistical environment like R or Stata, or a machine learning pipeline in Python. It may also go directly into a visualization library.

### Business Intelligence Outputs

When preparing data for BI dashboards, context is important. Some situations require highly detailed datasets that allow users to slice and drill down interactively. Others benefit from small, aggregated tables with precomputed metrics at the appropriate grain, such as daily totals by region or product category, so dashboards load quickly and remain responsive. In these cases, the heavy transformations and calculations are performed upstream, reducing the computational burden on the BI tool at query time. Different BI tools vary in how well they handle raw detail compared with aggregated inputs, so understanding their strengths is essential. There is no universal rule. Your shaping strategy should match how the data will be consumed.

### Visualization Outputs

For visualizations, lean and aggregated datasets generally work best. Whether you use commercial visualization software or programming libraries in Python, R, or JavaScript, think carefully about the filters and views your audience will need. In some cases, this means preparing multiple layers of data: one at a detailed level and another at a broader, all-inclusive level. In Polars, you can create these layers efficiently by combining dataframes through joins or concatenations.

### Statistics and Machine Learning Outputs

For statistical analysis or machine learning, a clear structure is essential. You need to identify the main entity under study, establish the appropriate level of aggregation, and select the features that matter. A model may require one row per customer with all relevant attributes or one row per transaction enriched with customer-level data. Polars makes it easy to reshape data into the "tidy data" structure described by Hadley Wickham[6], where:

- Each variable is a column
- Each observation is a row
- Each value is a cell

This tidy structure ensures compatibility with most statistical and machine learning frameworks.

#### Transformations in Polars

Polars excels at restructuring data, whether you are pivoting, unpivoting, grouping, or aggregating. Its lazy API allows you to chain transformations while Polars optimizes execution behind the scenes. This makes it easy to move from raw source data to the precise shape required for BI, visualization, statistical analysis, or machine learning workflows.

## Pivoting from Long Format to Wide Format

A *pivot table* reorganizes a dataset so that one column defines the rows, another defines the columns, and the cells at their intersections contain aggregated results such as sums (`sum`), counts (`len`), or averages (`mean`). This transformation condenses raw data into a compact and structured view that is easier to interpret, particularly for business reporting or dashboards. Pivot tables convert long format data, which has fewer columns and many rows, into wide format data, which has more columns and fewer rows.

Although pivot tables are often associated with Excel's drag-and-drop interface, Polars provides a programmatic way to achieve the same result. The `pivot` method, combined with aggregation functions, is used to create a pivot table. For example, using the *clothing_store_sales* dataset, you can `group_by` the *Year* derived from the *Date* column and by *Category*, then apply the sum aggregation to compute the total amount for each clothing category. You can then create a pivot table from the resulting dataframe by pivoting on the *Category* column.

```
(clothing_store_sales
 .with_columns(pl.col('Date').dt.year())
 .pivot(index='Date', on='Category',
        values='Ext_Price', aggregate_function='sum') ❶
 .rename(lambda c: f'{c}_Amount' if c != 'Date' else 'Year') ❷
 )
```

❶ You can omit `aggregate_function` to return the raw values.

❷ The *lambda* function appends *Amount* to all columns except *Date*, which is renamed to *Year*.

---

```
shape: (2, 4)
+------+------------+---------------+--------------+
```

```
| Year | Hat_Amount | Sweater_Amount | Socks_Amount |
| ---  | ---        | ---            | ---          |
| i32  | f64        | f64            | f64          |
+=================================================+
| 2014 | 68825.21   | 228395.27      | 128127.52    |
| 2013 | 30468.8    | 73320.73       | 41042.41     |
+------+------------+----------------+--------------+
```

**Tip**
The column whose values you want to become new columns in the final dataframe is the one you pivot on.

A *pivot operation* is especially useful when working with datasets stored in long format rather than wide format. Since Polars is columnar-based, this design is often more efficient for handling sparse data, because adding new columns is computationally expensive while adding rows is not. Instead of storing many attributes in separate columns, you can store multiple records per entity, with each attribute represented as a row containing columns such as *Attribute_Name* and *Attribute_Value*.



**Expand Your Knowledge**

*WHAT IS SPARSE DATA?*
Sparse data is a dataset where a large percentage of values are zero or empty. This differs from missing data, which is unknown. Sparse data has known values that are simply zero.

With Polars, you can reshape long format data into wide format whenever needed by using the `pivot` method. For example, consider a customer dataset where each row contains a single attribute of a customer. By pivoting on the *Attribute_Name* column and aggregating the *Attribute_Value*, you can construct a compact table where each customer's attributes appear as columns. This approach makes it easy to assemble complete customer records when required while keeping the underlying storage efficient, especially when many attributes are sparse.

## Unpivoting from Wide Format to Long Format

Most managers and business executives prefer viewing data in wide format. They often want months shown across the screen rather than down the screen. Throughout my data career, I have received many Excel files in wide format, which I convert to long format before performing any analysis. Polars provides the `unpivot` method to perform this transformation. It's the opposite of `pivot`. I follow a simple rule: data that needs processing should be in long format, and data that is ready for reporting should be in wide format. Table 8 shows the gross domestic product (GDP) of Zam-

bia and its two neighboring countries over five years.

| Country_Name | Year_2020 | Year_2021 | Year_2022 | Year_2023 | Year_2024 |
|---|---|---|---|---|---|
| Angola | 48502 | 66505 | 104400 | 84875 | 80397 |
| Namibia | 10584 | 12402 | 12569 | 12408 | 13372 |
| Zambia | 18138 | 22096 | 29164 | 27578 | 26326 |

*Table 8. Country GDP by year (in millions)*[7]

You can turn Table 8 into a result set with one row per country per year by using the `unpivot` method.

```
(zambia_namibia_angola_gdp
 .unpivot(index='Country_Name', value_name='GDP', variable_name='Year')
 .with_columns(pl.col('Year').str.replace('Year_','').cast(pl.Int32))
 )
```

---

```
shape: (15, 3)
+--------------+------+-------+
| Country_Name | Year | GDP   |
| ---          | ---  | ---   |
| str          | i32  | i32   |
+============================+
| Angola       | 2020 | 48502 |
| Namibia      | 2020 | 10584 |
| Zambia       | 2020 | 18138 |
| Angola       | 2021 | 66505 |
| Namibia      | 2021 | 12402 |
| ...          | ...  | ...   |
| Namibia      | 2023 | 12408 |
| Zambia       | 2023 | 27578 |
| Angola       | 2024 | 80397 |
| Namibia      | 2024 | 13372 |
| Zambia       | 2024 | 26326 |
+--------------+------+-------+
```

As a data professional, you'll encounter datasets in countless formats and structures, and many of them won't be ready for direct use in your final outputs. Polars provides a range of tools to reshape and reorganize data, from combining datasets to pivoting or unpivoting tables to applying conditional logic with `when-then-otherwise` expressions. Mastering these techniques allows you to mold your data into the form you need, giving you more control over both your analysis and how you communicate your findings.

## Conclusion

The process of preparing data may seem like a preliminary step before the "real" analysis begins, but it's actually a cornerstone of meaningful work. Remember, 80% of all data work is data cleaning. Taking time to understand the different data types you'll encounter is essential, and examining each dataset closely helps ensure quality. Data profiling is a powerful way to uncover what's inside a dataset and spot potential issues, and it's something I revisit often as projects grow in complexity. Since challenges with data quality are ongoing, we've explored strategies for cleaning and enriching datasets to make them more reliable. Equally important is learning how to reshape data into the right format for your goals. These themes will continue to appear throughout the book, reinforcing their importance. Up next, we'll dive into time series analysis, beginning our exploration of specialized analytical methods.

1. *https://www.linkedin.com/feed/update/urn:li:activity:7381507318145974272/*

2. *https://docs.pola.rs/api/python/stable/reference/index.html*

3. *https://docs.pola.rs/api/python/dev/reference/api/polars.concat.html*

4. *https://en.wikipedia.org/wiki/Cartesian_product*

5. Frederick F Reichheld, "The One Number You Need to Grow," *Harvard Business Review* 81, no. 12 (2003): 46-55.

6. Hadley Wickham, "Tidy Data," *Journal of Statistical Software* 59, no. 10 (2014): 1-23, doi:10.18637/jss.v059.i10.

7. World Bank Group, "Countries GDP," 2024, *https://data.worldbank.org/indicator/NY.GDP.MKTP.CD?end=2024&start=1960*

## About the Author

**Joram Mutenge** trained as a data scientist at the University of Illinois Urbana-Champaign, where he earned his master's degree. He is passionate about bringing harmony to the chaos in numbers and helping stakeholders and executives clearly understand the insights within their data. Joram has been analyzing data with Polars since 2022 across multiple industries, including manufacturing, retail, and cybersecurity. In addition to Polars, he has worked with several dataframe technologies, including pandas, Ibis, and Narwhals. He is also the instructor of the Udemy course *Analyzing Data With Polars in Python*[1] and a *2026 PyCon speaker*[2] where he presented a talk on writing idiomatic Polars.

## One More Thing

Thank you for reading this book. If it helped you better understand Polars or improved the way you work with data, I would greatly appreciate your support. As an independent author, visibility depends heavily on reader feedback rather than large marketing campaigns.

If you have a moment, please consider leaving a short, honest review on Amazon or sharing your thoughts on social media. Even a few sentences about what you learned or how the book helped you can make a meaningful impact. Your feedback helps other readers discover the book and decide if it is right for them.

Thank you again for your support.

1. *https://www.udemy.com/course/analyzing-data-with-polars-in-python/*

2. *https://www.youtube.com/watch?v=5-4xTrN-M98*

# Deep Analysis *with* Polars

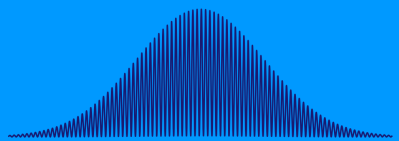
As data grows in size and complexity, efficient tools are essential. Polars is a fast, modern DataFrame library designed for performance and clarity. This book shows you how to use it to analyze data effectively and at scale.

Ideal for programmers, data analysts, data scientists, data engineers, and analytics engineers, this guide combines real-world datasets with practical best practices to help you build reliable, high-performance analytical workflows.

With clear explanations, diagrams, and tables, you'll quickly learn how to work with:

- **Series and DataFrames**
- **Expressions and chaining**
- **Grouping and aggregation**
- **Joining and pivoting**
- **Plotting and visualization**
- **And more**

**Joram Mutenge** has analyzed data across industries, including manufacturing, cybersecurity, and consumer services. Known for translating complex data into clear insights, he has helped organizations make confident, data-driven decisions. He has used Polars in production since 2022.



conterval.com