Introduction to Polars

A Beginners Guide to Data Analysis with Polars in Python

Nathaniel Clark

2025-04-20

Table of contents

Preface

Welcome to "Introduction to Polars." This book emerged from my journey as a data science student who chose to explore Polars rather than pandas—the standard library taught in my course. When I approached my instructor about using this alternative technology, he supported my decision while honestly acknowledging that course materials wouldn't cover my chosen path.

As I navigated through the course, I discovered a significant gap in beginner-friendly Polars resources for data science newcomers. While the official documentation proved valuable, it often assumed a level of familiarity that beginners might not possess. Nevertheless, through persistence and experimentation, I gained proficiency and successfully completed my coursework.

This book aims to bridge that gap by offering an accessible introduction to Polars for those new to data manipulation libraries. I've designed it especially for readers with limited prior experience in data science, incorporating the insights and solutions I discovered along my learning journey.

1 Introduction

1.1 What is Polars

Polars is a modern data manipulation library avalible for Python, R, NodeJs and Rust. It is designed as a high-performance alternative to pandas, especially for large datasets. It features syntax that's both human-readable and similar to R's data manipulation paradigms. Polars stands out for three main reasons:

- **Performance**: Built in Rust, Polars delivers exceptional speed through parallel processing by default and a sophisticated query optimizer that analyzes and improves execution plans.
- **Memory efficiency**: Using a columnar memory format rather than row-based storage, Polars efficiently handles larger-than-memory datasets and performs operations with minimal memory overhead.
- **Lazy evaluation**: Polars supports both eager and lazy execution modes. The lazy API builds optimized query plans before execution, similar to database query planners, resulting in more efficient data processing pipelines.

1.2 Who uses Polars

2 Installation

2.1 Basic Installation

Polars can be installed using pip:

```
pip install polars
```

2.2 Optional Dependencies

Polars offers various optional dependencies for specific use cases, which are omitted to reduce the footprint of the library. Throught this guide I will mention when specific dependancts are required/used.

To install all optional dependencies:

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

> **Note**
>
> I recommend installing all optional dependencies due to convenience. And the fact that the relative footprint is still not excessive.

2.2.1 Interoperability

Polars offers the following dependencys for increased interoperability between different librarys.

- **pandas**: allows conversion to and from pandas dataframes/series
- **numpy**: allows conversion between numpy arrays
- **pyarrow**:allows for data conversion between PyArrow tables and arrays
- **pydantic**: allows for conversion from Pydantic models to polars

```
pip install 'polars[pandas, numpy, pyarrow, pydantic]' # remove the unused dependencies
```

2.2.2 Excel

Polars has a few options for different engines used to convert xlsx files to a format more readable by polars.

The different engines avalible are:

- **calamine**
- **openpyxl**
- **xlsx2csv**

> Tip
>
> There are some differences in the engines performance and behaviour to learn more see the official documentation.

Additionally Polars support one other optional dependency related to Excel: - **xlsxwriter**: which allows you to write to xlsx files

```
pip install 'polars[excel]' # if you want to install all Excel dependencies
pip install 'polars[calamine, openpyxl, xlsx2csv, xlsxwriter]' # if you want to pick and cho
```

2.2.3 Database

2.2.4 Cloud

2.2.5 Other I/O

2.2.6 Other

3 Dataframes and Series

3.1 Data types

Polars allows you to store data in a variety of formats called data types. These data types fall generally into the following categories:

- **Numeric**: Signed integers, unsigned integers, floating point numbers, and decimals
- **Nested**: Lists, structs, and arrays for handling complex data
- **Temporal**: Dates, datetimes, and times for working with time-based data
- **Miscellaneous**: Strings, binary data, Booleans, categoricals, enums, and objects

The most common data types you will be working with are generally: Strings, signed and unsigned integers, floating point numbers or floats, decimals, dates or datetimes and booleans. For more information on each of these data types see Appendix A.

3.2 Series

The two most common data structures in Polars are DataFrames and Series. Series are one-dimensional data structures where

Creating a Series is straightforward with the following syntax:

`pl.Series(name, values_list)`

Where "name" is the label for your Series and "values_list" contains the data. Here's a simple example:

```
import polars as pl
s = pl.Series("example", [1, 2, 3, 4, 5])
s
```

example
i64
1
2

example
i64
3
4
5

When you create a series Polars will infer the data type for the values you provide. So in the above example I gave it [1, 2, 3, 4, 5] and it set the datatype to Int64 if instead gave it [1, 2, 3, 4.0, 5] it would asume it is Float64.

```
s2 = pl.Series("payment", [132.50, 120, 116, 98.75 ,42])
s2
```

payment
f64
132.5
120.0
116.0
98.75
42.0

```
s3 = pl.Series("mixed", [1, "text", True, 3.14], strict=False)
# series.dytpe outputs a the data type of the series
print(f"Mixed series type: {s3.dtype}")
s3
```

```
Mixed series type: String
```

mixed
str
”1”
”text”
”true”
”3.14”

You can set the data type of the series as well by using the `dtype` parameter. A example use case is when storing a id number the id number should be stored as a string not a int due

to the fact that we we do not want to perform mathmatical operations on the identification number therefore it is best stored as a string.

```
# strict=False allows automatic conversion from different data types
s3 = pl.Series("id number", [143823, 194203, 553420, 234325, 236532], dtype=pl.Utf8, strict=
s3
```

id number str
”143823”
”194203”
”553420”
”234325”
”236532”

3.3 Dataframes

DataFrames are tabular data structures (rows and columns) composed of multiple Series, with each column representing a single Series. The design of a dataframe is called schema. A schema is a mapping of column to the data types.

Dataframes are the workhorses of data analysis and what you’ll use most frequently.

With DataFrames, you can write powerful queries to filter, transform, aggregate, and reshape your data efficiently.

DataFrames can be created in several ways:

1. From a dictionary of sequences (lists, arrays)
2. With explicit schema specification
3. From a sequence of (name, dtype) pairs
4. From NumPy arrays
5. From a list of lists (row-oriented data)
6. By converting pandas DataFrames
7. By importing existing tabular data from CSVs, JSON, SQL, Parquet files, etc.

In real-world environments, you’ll typically work with preexisting data, though understanding various creation methods is valuable. We’ll cover data import techniques later, but for now, here’s an example of a DataFrame created from a dictionary of lists:

```
# Create a DataFrame from a dictionary of lists
df = pl.DataFrame({
    "name": ["Alice", "Bob", "Charlie", "David"],
    "age": [25, 30, 35, 40],
    "city": ["New York", "Boston", "Chicago", "Seattle"],
    "salary": [75000, 85000, 90000, 95000]
})

df
```

name str	age i64	city str	salary i64
”Alice”	25	”New York”	75000
”Bob”	30	”Boston”	85000
”Charlie”	35	”Chicago”	90000
”David”	40	”Seattle”	95000

every data frame has a shape. the shape is the number of rows and columns in a dataframe `shape(rows,columns)`

the shape for the above dataframe is:

```
print(df.shape)
```

```
(4, 4)
```

you can view the schema of any dataframe with the following command

```
print(df.schema)
```

```
Schema({'name': String, 'age': Int64, 'city': String, 'salary': Int64})
```

We see here that the schema is returned as a dictionary. In the above example the column name has the string datatype. Though you can view the data type already when displaying the dataframe.

3.4 Inspecting Dataframes

In polars there are a varity of ways to inspect a dataframe, all of which have different use cases. The ones that we will be covering right now are:

- head
- tail
- glimpse
- sample
- describe
- slice

3.4.1 head

the `head` functions allows you to view the first x rows of the dataframe. By default the number of rows it shows is 5, though you can specify the number of rows to view.

```
dataframe.head(n)
```

Where n is the number of rows to return if you give it a negative number it will turn all rows except the last n rows.

```
import numpy as np

# Create NumPy arrays for sandwich data
sandwich_names = np.array(['BLT', 'Club', 'Tuna', 'Ham & Cheese', 'Veggie'])
prices = np.array([8.99, 10.50, 7.50, 6.99, 6.50])
calories = np.array([550, 720, 480, 520, 320])
vegetarian = np.array([False, False, False, False, True])

# Create DataFrame from NumPy arrays
sandwich_df = pl.DataFrame({
    "sandwich": sandwich_names,
    "price": prices,
    "calories": calories,
    "vegetarian": vegetarian
})


sandwich_df.head(3)
```

sandwich str	price f64	calories i64	vegetarian bool
”BLT”	8.99	550	false
”Club”	10.5	720	false
”Tuna”	7.5	480	false

3.4.2 tail

The `tail` function is essentially the inverse of head. It allows you to view the last n rows of the dataframe. The default for tail is also five rows.

```
dataframe.tail(n)
```

Where n is the number of rows to return if you give it a negative number it will turn all rows except the first n rows.

4

A Common Data Types

Data Type	Polars Type	Description	Example
Strings	`pl.Utf8`	Text data	`"hello"`
Signed Integers	`pl.Int8`, `pl.Int16`, `pl.Int32`, `pl.Int64`	Whole numbers that can be positive or negative	`-42`
Unsigned Integers	`pl.UInt8`, `pl.UInt16`, `pl.UInt32`, `pl.UInt64`	Whole numbers that can only be positive	`42`
Floating Point	`pl.Float32`, `pl.Float64`	Real numbers with decimal points	`3.14159`
Decimals	`pl.Decimal`	Fixed-precision numbers, useful for financial calculations	`Decimal("10.99")`
Dates/DateTimes	`pl.Date`, `pl.Datetime`	Calendar dates and time values	`2023-01-01`, `2023-01-01T12:30:00`
Booleans	`pl.Boolean`	Logical values: true or false	`True`, `False`
Time	`pl.Time`	Time of day without date	`12:30:45`
Duration	`pl.Duration`	Time spans or intervals	`3d 12h 30m 45s`
Categorical	`pl.Categorical`	Efficient storage for repeated string values	`pl.Series(["a", "b", "a"]).cast(pl.Categorical)`
List	`pl.List`	Lists of values of any type	`[1, 2, 3]`
Struct	`pl.Struct`	Composite type with named fields	`{"field1": 1, "field2": "a"}`
Null	`pl.Null`	Missing or undefined values	`None` or `null`

B Reading and Writing Data