

Introduction to Polars

A Beginners Guide to Data Analysis with Polars in Python

Nathaniel Clark

2025-04-20

Table of contents

Preface	3
1 Introduction	4
1.1 What is Polars	4
1.2 Who uses Polars	4
2 Installation	5
2.1 Basic Installation	5
2.2 Optional Dependencies	5
3 Dataframes and Series	7
3.1 Data types	7
3.2 Series	7
3.3 Dataframes	9
3.4 Inspecting Dataframes	11
Appendices	14
A Common Data Types	14
B Reading and Writing Data	15
B.1 CSV Files	15
B.2 JSON Files	16
B.3 Excel	16
B.4 Parquet Files	17
B.5 Importing Mutiple files	18

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 available 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. Throughout this guide I will mention when specific dependencies 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 dependencies for increased interoperability between different libraries.

- **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 available 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 choose
```

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 assume 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.dtype 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 do not want to perform mathematical 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=False)
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	age	city	salary
str	i64	str	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 variety 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.

```
# Import data from the sales.csv file into a Polars DataFrame
sales_df = pl.read_csv("./data/sales.csv")

# Display the last 6 rows of the sales DataFrame
sales_df.tail(6)
```

date str	region str	sales_rep str	product_category str	units_sold i64	revenue f64	cost f64	customer_s str
"2023-01-22"	"East"	"Emma Wilson"	"Furniture"	4	3599.96	2519.97	"SMB"
"2023-01-23"	"West"	"Michael Brown"	"Electronics"	11	5499.89	3299.93	"Enterprise"
"2023-01-23"	"North"	"John Smith"	"Office Supplies"	83	1660.0	996.0	"Consumer"
"2023-01-24"	"South"	"Maria Garcia"	"Software"	27	2699.73	809.92	"SMB"
"2023-01-24"	"East"	"David Johnson"	"Electronics"	6	2999.94	1799.96	"Enterprise"
"2023-01-25"	"West"	"Lisa Wong"	"Furniture"	8	7199.92	5039.94	"Consumer"

Note

For more information on reading in external data see [Appendix B](#)

4

A Common Data Types

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

B Reading and Writing Data

Polars provides robust capabilities for importing data from various sources including CSVs, JSONs, Excel spreadsheets, Parquet files, cloud storage solutions (AWS, Azure, and Google Cloud), and databases.

The importing methods follow a consistent pattern across file types, making it easy to work with different data formats.

B.1 CSV Files

The basic syntax for reading a CSV file is:

```
pl.read_csv("path/to/data.csv")
```

Alternatively, you can also read CSV files directly from the internet:

```
pl.read_csv("https://example.com/path/to/your/file.csv")
```

This capability to read files directly from URLs also works with all the file import methods we'll cover below.

This function offers numerous parameters to handle different CSV formats and configurations. For more information read the [documentation](#).

```
import polars as pl

df_csv = pl.read_csv("./data/example.csv", try_parse_dates=True)
df_csv.head(5)
```

id	first_name	last_name	email	purchase_date	product	quantity
i64	str	str	str	date	str	i64
1	"John"	"Doe"	"john.doe@example.com"	2023-01-15	"Laptop"	1
2	"Jane"	"Smith"	"jane.smith@example.com"	2023-01-16	"Smartphone"	2

id	first_name	last_name	email	purchase_date	product	quantity
i64	str	str	str	date	str	i64
3	"Robert"	"Johnson"	"rob.j@example.com"	2023-01-18	"Headphones, Wireless"	3
4	"Sarah"	"Williams"	"sarah.w@example.com"	2023-01-20	"Monitor"	1
5	"Michael"	"Brown"	"michael.b@example.com"	2023-01-22	"Keyboard"	2

B.2 JSON Files

Reading JSON files follows a similar pattern. The basic syntax is:

```
pl.read_json("docs/data/path.json")
```

JSON files have a more standardized structure than CSVs, so the reading process requires fewer configuration parameters. Polars handles JSON parsing efficiently with minimal setup. For advanced options and configurations, consult the official [documentation](#).

```
df_json = pl.read_json("./data/basketball.json")

df_json
```

B.3 Excel

Polars doesn't include a native Excel reader. Instead, it leverages external libraries like `fastexcel`, `xlsx2csv`, or `openpyxl` to parse Excel files into Polars-compatible formats. Among these options, Polars recommends `fastexcel` for optimal performance.

While it's generally better to avoid using Excel files where possible (you can usually export as CSV directly from Excel), reading Excel files is straightforward with the right dependencies installed.

Before attempting to read Excel files, make sure you have at least one of these libraries installed:

```
$ pip install fastexcel xlsx2csv openpyxl
```

The basic syntax for reading an Excel file with Polars is:


```
pl.read_excel("path/to/data.xlsx")
```

If your Excel file contains multiple sheets, you can specify which one to read using the `sheet_name` parameter:

```
df = pl.read_excel("path/to/data.xlsx", sheet_name="example")
```

For additional Excel reading options and parameters, refer to the [Polars Excel documentation](#), which covers sheet selection, range specification, and handling of complex Excel files.

```
df_xlsx = pl.read_excel("./data/penguins.xlsx", sheet_name="Dream Island")  
  
df_xlsx.tail(5)
```

species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
str	str	f64	f64	i64	i64	str
"Chinstrap"	"Dream"	55.8	19.8	207	4000	"male"
"Chinstrap"	"Dream"	43.5	18.1	202	3400	"female"
"Chinstrap"	"Dream"	49.6	18.2	193	3775	"male"
"Chinstrap"	"Dream"	50.8	19.0	210	4100	"male"
"Chinstrap"	"Dream"	50.2	18.7	198	3775	"female"

This example spreadsheet can be accessed via [this Google Sheets link](#).

B.4 Parquet Files

Parquet is a columnar storage format designed for efficient data analytics. It provides excellent compression and fast query performance, making it a popular choice for data science workflows. Polars includes native, high-performance support for reading Parquet files.

The basic syntax for reading a Parquet file is:

```
pl.read_parquet("path/to/data.parquet")
```

```
df_par = pl.read_parquet("./data/finance.parquet")  
df_par.sample(4)
```

date datetime[ns]	ticker str	open f64	high f64	low f64	close f64	volume i64	pe_ratio f64	dividend_yield f64	ma f64
2024-10-15 00:00:00	"GOOG"	259.76	260.74	257.93	260.35	5219388	23.76	0.0054	9.0
2024-11-26 00:00:00	"META"	753.4	759.93	749.62	756.83	4864442	35.96	0.0089	3.0
2024-05-02 00:00:00	"META"	469.75	472.91	468.96	471.98	8265830	34.52	0.0168	2.0
2024-09-25 00:00:00	"MSFT"	253.9	256.62	251.8	254.26	1590665	27.14	0.0043	3.0

B.5 Importing Multiple files

For situations where you need to combine data from multiple files into a single DataFrame, Polars offers straightforward approaches. While the syntax is relatively simple, the implementation may vary depending on your specific file organization.

When working with multiple files of the same type and similar naming patterns in a single directory, Polars supports glob pattern matching:

```
pl.read_filetype("path/to/data/my_many_files_*.filetype")
```

For files with different names but the same format, placing them in a single directory allows you to use wildcard patterns to import them all at once:

```
pl.read_filetype("path/to/data/import/*.filetype")
```

Alternatively, for files located in different directories or even on different servers, you can provide a list of filepaths or URLs:

```
pl.read_filetype([
    "path/to/first/file.filetype",
    "path/to/second/file.filetype",
    "another/location/file.filetype"
])
```

If you're working with different file types that share the same schema (identical columns and datatypes) and want to combine them into a single DataFrame, you'll need to read each file individually and then concatenate them. Polars makes this process straightforward with its `concat` function, which can merge DataFrames regardless of their original file formats.

```
# Read files of different formats
df1 = pl.read_csv("path/to/file.csv")
df2 = pl.read_parquet("path/to/file.parquet")
df3 = pl.read_json("path/to/file.json")

# Concatenate into a single DataFrame
combined_df = pl.concat([df1, df2, df3], how="vertical")
```