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.

About the Author

I'm a Data Science student at Brigham Young University-Idaho with expertise in Python programming, data analysis, database management, and cybersecurity fundamentals. My journey in data science combines technical skills with practical problem-solving abilities.

GitHub LinkedIn Website

Introduction

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.

What you will learn

How this book is organized

Installation

Basic Installation

Polars can be installed using pip:

```
pip install polars
```

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.

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
```

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
```

Database

Cloud

Other I/O

Other

1 Dataframes and Series

1.1 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.

1.2 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=F
s3
```

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

1.3 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:

```python
# 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:

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

```
(4, 4)
```

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

```python
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.

1.4 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

1.4.1 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

Tip

Both head and tail are useful for quick data exploration without loading the entire dataset.

1.4.2 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

1.4.3 glimpse

The `glimpse` function allows you to preview your dataframe. By providing the number of rows and columns. The column names and datatypes and the first few values of each column. it can be usefull when tring to gain an intial perspective of the dataframe without requiring in depth overview

```
dataframe.glimpse(max_items_per_column)
```

You can leave the parameters blank which I would reccomend in most use cases but you can pass it a number to set the max number of items to return for each column.

```
# Reads data from a parquet file into a Polars DataFrame
# Parquet is a columnar storage file format optimized for analytics
finance_df = pl.read_parquet("./data/finance.parquet")

# Display a summary overview of the DataFrame
finance_df.glimpse()
```

```
Rows: 1310
Columns: 10
$ date           <datetime[ns]> 2024-04-15 00:00:00, 2024-04-16 00:00:00, 2024-04-17 00:00:0
$ ticker                  <str> 'AAPL', 'AAPL', 'AAPL', 'AAPL', 'AAPL', 'AAPL', 'AAPL', 'AAP
$ open                    <f64> 215.0, 215.27, 216.07, 220.02, 217.83, 216.56, 215.03, 212.2
$ high                    <f64> 215.49, 216.26, 219.1, 222.15, 219.19, 218.67, 216.66, 213.9
$ low                     <f64> 214.22, 214.32, 215.91, 218.75, 216.63, 215.46, 214.58, 212.
$ close                   <f64> 215.01, 216.14, 217.16, 220.56, 218.74, 217.13, 215.38, 212.
$ volume                  <i64> 1930612, 2166356, 2108441, 2426330, 2922733, 2282139, 234701
$ pe_ratio                <f64> 24.63, 11.55, 37.52, 31.55, 31.33, 12.62, 23.63, 14.16, 13.2
$ dividend_yield          <f64> 0.0255, 0.0005, 0.0002, 0.0161, 0.006, 0.0152, 0.0165, 0.000
$ market_cap              <f64> 2664107971.0, 5452883768.0, 6751310057.0, 4539928581.0, 5699
```

1.4.4 describe

```
finance_df.describe()
```

statistic	date	ticker	open	high	low	close
str	str	str	f64	f64	f64	f64
”count”	”1310”	”1310”	1310.0	1310.0	1310.0	1310.0
”null_count”	”0”	”0”	0.0	0.0	0.0	0.0
”mean”	”2024-10-13 21:42:35.725190”	null	309.888687	312.208565	309.056229	310.665
”std”	null	null	183.806709	185.216272	183.266391	184.279542
”min”	”2024-04-15 00:00:00”	”AAPL”	125.74	127.01	125.19	125.9
”25%”	”2024-07-15 00:00:00”	null	200.6	201.72	199.93	200.91
”50%”	”2024-10-15 00:00:00”	null	240.87	242.6	240.33	241.9
”75%”	”2025-01-14 00:00:00”	null	315.93	318.79	316.15	316.79
”max”	”2025-04-15 00:00:00”	”MSFT”	792.59	801.45	790.79	796.05

1.4.5 sample

```
finance_df.sample(3)
```

date	ticker	open	high	low	close	volume	pe_ratio	dividend_yield	ma
datetime[ns]	str	f64	f64	f64	f64	i64	f64	f64	f64
2024-07-03 00:00:00	”AMZN”	145.69	146.57	145.52	145.93	3037047	27.46	0.0017	2.8
2024-05-16 00:00:00	”AMZN”	125.74	127.01	125.19	125.9	3453751	15.58	0.0067	3.8
2024-12-19 00:00:00	”AAPL”	259.18	262.23	258.14	260.03	2213564	33.76	0.0121	6.3

1.4.6 slice

```
dataframe.slice(offset, length)
```

```
finance_df.slice(500, 6)
```

date	ticker	open	high	low	close	volume	pe_ratio	dividend_yield	ma
datetime[ns]	str	f64	f64	f64	f64	i64	f64	f64	f64
2025-03-13 00:00:00	”MSFT”	242.34	244.05	242.99	243.12	1404084	27.98	0.0271	3.2
2025-03-14 00:00:00	”MSFT”	243.49	244.84	243.64	243.76	462641	34.61	0.0248	4.0
2025-03-17 00:00:00	”MSFT”	247.86	250.29	246.92	248.3	1637110	34.78	0.0189	4.8
2025-03-18 00:00:00	”MSFT”	242.34	244.07	240.73	243.06	1112130	26.23	0.0182	2.6
2025-03-19 00:00:00	”MSFT”	243.08	245.24	243.39	243.84	1346630	11.79	0.0181	4.6

date	ticker	open	high	low	close	volume	pe_ratio	dividend_yield	ma
datetime[ns]	str	f64	f64	f64	f64	i64	f64	f64	f64
2025-03-20 00:00:00	"MSFT"	243.86	245.05	242.87	244.5	1202714	12.4	0.0133	2.4

2 The Four Contexts

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

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 i64	first_name str	last_name str	email str	purchase_date date	product str	quan 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	quan
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 str	island str	bill_length_mm f64	bill_depth_mm f64	flipper_length_mm i64	body_mass_g i64	sex 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	ticker	open	high	low	close	volume	pe_ratio	dividend_yield	ma
datetime[ns]	str	f64	f64	f64	f64	i64	f64	f64	f64
2025-03-19 00:00:00	"AAPL"	312.26	315.25	309.73	312.75	1449868	35.71	0.0035	8.1
2024-08-05 00:00:00	"AAPL"	215.53	218.05	215.34	216.47	2186172	36.04	0.0074	2.9
2024-07-31 00:00:00	"AMZN"	154.52	155.88	154.56	154.62	3927041	25.85	0.0199	3.3
2025-02-19 00:00:00	"MSFT"	222.96	225.39	223.08	223.82	1298398	30.27	0.0119	4.2

B.5 Importing Mutiple 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.

```python
# 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")
```