CHEAT SHEET

POLARS

HIGH-PERFORMANCE DATA PROCESSING

SERIES: PYTHON | LEVEL: INTERMEDIATE | UPDATED: 2026

Polars is a DataFrame library written in Rust on the Apache Arrow memory format. You describe work as expressions and the query engine plans them, runs them across all cores, and skips columns it does not need. Two approaches: eager for exploring (`read_*` → DataFrame); lazy for anything that has to be fast (`scan_*` → LazyFrame).

read_csv · scan_csv

Loads a file. `read_` pulls it into memory now; `scan_` only reads the header and waits for a plan. Also: `{read|scan}_parquet`, `{read|scan}_json`, `{read|scan}_database`, `{read|scan}_excel`.

```
import polars as pl

df = pl.read_csv("sales.csv")
lf = pl.scan_csv("sales.csv")
```

glimpse · describe

First look at a frame. `glimpse` prints one row per column with its type; `describe` gives summary statistics.

```
df.glimpse()
df.describe()
df.schema
```

```
Rows: 5000
Columns: 4
$ region <str> 'East', 'West'
$ units  <i64> 12, 40
$ price  <f64> 9.99, 4.50
```

select

Chooses and computes columns. Anything you pass is an expression, so you can rename and calculate in one pass.

```
df.select(
    "region",
    (pl.col("units") * pl.col("price"))
        .alias("revenue"),
)
```

filter

Keeps rows where an expression is true. Combine conditions with `&` and `|`, and bracket each one.

```
df.filter(
    (pl.col("units") > 10)
    & (pl.col("region") == "East")
)

df.filter(pl.col("region").is_in(["East", "West"]))
```

with_columns

Adds or overwrites columns and keeps everything else. Keyword arguments name the result.

```
df.with_columns(
    revenue = pl.col("units")
            * pl.col("price"),
    units = pl.col("units").cast(pl.Int32),
)
```

when · then · otherwise

Branches inside an expression. Chain extra `when` clauses for more bands; the first match wins.

```
df.with_columns(
    tier = pl.when(pl.col("revenue") > 500)
             .then(pl.lit("high"))
             .when(pl.col("revenue") > 100)
             .then(pl.lit("mid"))
             .otherwise(pl.lit("low"))
)
```

group_by · agg

Splits rows into groups, then reduces each group with one expression per output column.

```
df.group_by("region").agg(
    pl.col("revenue").sum().alias("total"),
    pl.col("price").mean().round(2),
    pl.len().alias("orders"),
)
```

```
shape: (2, 4)
region  total     price  orders
East    18420.5   7.22   1204
West     9310.0   6.85    811
```

over

Runs an aggregation per group but returns a value for every row. Operates as a window function, no join required.

```
df.with_columns(
    share = pl.col("revenue")
          / pl.col("revenue").sum()
              .over("region"),
    rank = pl.col("revenue").rank("dense")
              .over("region"),
)
```

sort · top_k

Orders rows. Pass a list of columns with a matching list of directions; `top_k` skips the full sort when you only want the head.

```
df.sort("revenue", descending=True)

df.sort(["region", "revenue"],
        descending=[False, True])

df.top_k(10, by="revenue")
```

join

Matches rows between frames. `semi` and `anti` filter the left frame by what exists on the right without adding columns. How options: `inner | left | right | full | semi | anti | cross`.

```
df.join(targets, on="region", how="left")
```

.str · .dt

Type-specific method namespaces. Reach through them for string, date, list and struct operations.

```
df.with_columns(
    pl.col("region").str.to_uppercase(),
    pl.col("date").str.to_date("%Y-%m-%d")
        .dt.month().alias("month"),
)
```

fill_null · drop_nulls

`null` is missing and `NaN` is a float value; different states, different methods. A strategy fills from the column itself, so you do not have to invent a literal for it.

```
df.null_count()
df.drop_nulls(subset=["price"])

df.with_columns(
    pl.col("price").fill_null(strategy="forward"),
    pl.col("units").fill_null(0),
)
```

```
shape: (1, 4)
region  units  price  date
0       0      37     0
```

scan · collect

The lazy pipeline. Nothing runs until `collect`, so the optimizer can push filters down to the file and read only the columns used. Use `.collect(engine="streaming")` if larger than memory. `.explain()` shows the plan.

```
(pl.scan_csv("sales.csv")
   .filter(pl.col("units") > 10)
   .group_by("region")
   .agg(pl.col("price").mean())
   .collect())
```

write · sink

Saves results. `sink_` writes straight from a LazyFrame without materialising the whole frame first.

```
df.write_parquet("out.parquet")
lf.sink_parquet("out.parquet")
```

convert

Converts a Polars DataFrame to another format. `to_pandas()` hands it off to Pandas, `to_arrow` is native no-copy.

```
df.to_pandas()
df.to_arrow()
```

pivot · unpivot

Turns long into wide and back, aggregates as it reshapes; eager.

```
wide = df.pivot(on="region", index="month",
                values="revenue",
                aggregate_function="sum")

wide.unpivot(index="month",
             variable_name="region",
             value_name="revenue")
```