Python Polars: The Definitive Cheatsheet

Getting Started

Polars is a library for transforming, analyzing, and visualizing data with a fast and expressive DataFrame API. First released by Ritchie Vink in 2020.

Install Polars with all its dependencies in the terminal:

```
$ uv pip install "polars[all]"
```

Import Polars in Python as follows:

```
import polars as pl
pl.show_versions() # check installed versions
```

Polars queries typically read, transform, and write data:

```
fruit = pl.read_csv("fruit.csv")
fruit.filter(
    (pl.col("weight") > 1000) & pl.col("is_round")
).write_parquet("fruit.parquet")
```

Data Structures

Polars stores all of its data in a `Series` or a `DataFrame`.

structure	description
`Series`	One-dimensional. Holds sequence of values of same data type.
`DataFrame`	Two dimensional. Has rows and columns. One or more Series, all same length.
`LazyFrame`	Resembles DataFrame but holds no data. Blueprint for generating a DataFrame.

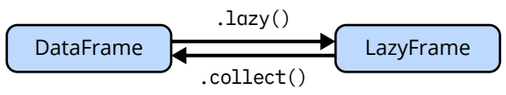
Unlike pandas, Polars DataFrames do not have a row index, and the API favors immutability and method chaining over in-place modifications.

```
series = pl.Series("sales", [150.00, 300.00, 250.00])

df = pl.DataFrame({
    "sales": series,
    "id": [41, 42, 43]
}) # or use a pl.read_*() function

df.with_row_index("id") # add explicit index
lf = df.lazy() # or use a pl.scan_*() function
```

Eager and Lazy APIs

The eager API executes immediately, whereas the lazy API builds an optimized query plan first. The optimizer automatically applies predicate pushdown (filtering early) and projection pushdown (dropping unused columns).

```
lf = df.lazy() # turn DataFrame into LazyFrame
df = lf.collect() # turn LazyFrame into DataFrame
lf.collect(engine="streaming") # process out-of-core
lf.explain() # print optimized plan
lf.show_graph() # visualize plan as graph
lf.profile() # execute and return per-node timings
```



Data Types

Polars implements most of the Apache Arrow memory specification, which is an efficient columnar format for flat and hierarchical data.

group	type	notes
Numeric	`Decimal`	128 bits, precision, scale
	`Float32`	Ranges ±3.4×10³⁸
	`Float64`	Ranges ±1.8×10³⁰⁸
	`Int8`	Ranges ±128
	`Int16`	Ranges ±32,768
	`Int32`	Ranges ±2.1×10⁹
	`Int64`	Ranges ±9.2×10¹⁸
	`Int128`	Ranges ±3.4×10³⁸
	`UInt8`	Ranges 0–255
	`UInt16`	Ranges 0–65,535
	`UInt32`	Ranges 0–4.3×10⁹
	`UInt64`	Ranges 0–1.8×10¹⁹
Temporal	`Date`	Days since Unix epoch
	`Datetime`	Microseconds since epoch
	`Duration`	Time duration / delta
	`Time`	Time of day
Nested	`Array`	Fixed-length sequence
	`List`	Variable-length sequence
	`Struct`	Multiple fields with names
String	`String`	UTF-8 text, variable length
	`Categorical`	Dict of Strings
	`Enum`	Fixed dict of Strings
Other	`Boolean`	True / False
	`Binary`	Raw bytes
	`Null`	Represents Null / None

```
df.schema # get dict of column names and data types
df.dtypes # get list of DataFrame data types
df.glimpse() # one row per column, with dtypes
df.describe() # per-column stats including nulls
df.estimated_size("mb") # in-memory size

df.select(pl.col("id").cast(pl.UInt64)) # strict cast
df.select(pl.col("id").cast(pl.Int8, strict=False))
   # overflow produces nulls
```

Reading & Writing Data

```
read_*() # read data into DataFrame
scan_*() # create LazyFrame
write_*() # write DataFrame to disk or cloud
sink_*() # stream data to disk or cloud
```

format	read	scan	write	sink
Avro	✓		✓	
Clipboard	✓		✓	
CSV	✓	✓	✓	✓
Database	✓		✓	
Delta Lake	✓	✓	✓	✓
Excel / ODS	✓		✓	
Iceberg		✓	✓	✓
IPC / Feather	✓	✓	✓	✓
JSON	✓		✓	
NDJSON	✓	✓	✓	✓
Parquet	✓	✓	✓	✓
PyArrow Dataset		✓		

Common keyword arguments: `schema_overrides`, `n_rows`, `row_index_name`, `storage_options`, `compression`

```
pl.scan_parquet("s3://bucket/*.parquet",
    storage_options={"aws_region": "us-east-2"})
lf.sink_parquet(pl.PartitionBy("out/", key="x"))
```

Transforming Data

Selecting Columns

Keep columns based on their name, data type, or position.

```
df.select("a", "b") # by name
df.select(pl.col("x") * 2) # with transformation
df.select(doubled=pl.col("x") * 2) # as new column
df.select(pl.col("^.*_color$")) # by regex
df.select(pl.all()) # all columns
```

Use columns selectors for more flexibility. They can be combined using set operators (|, &, -, ^, and ~).

```
import polars.selectors as cs

df.select(cs.numeric()) # by type
df.select(cs.starts_with("val") # by prefix
# see also: cs.string(), cs.contains(), cs.first()
df.drop("a", "y", strict=False) # drop without error
```

Creating Columns

New columns are added to the right.

```
df.with_columns(new=pl.col("a") + 1) # add new column
df.with_columns(pl.col("a").fill_null(0)) # replace
df.with_columns(ones=pl.lit(1)) # literal value
df.with_row_index(name="id", offset=1) # add indices
```

Filtering Rows

Keep rows according to the values in one or more columns or expressions.

```
df.filter("valid") # based on boolean column
df.filter(pl.col("x") > 5) # single expression
df.filter(pl.col("valid"), pl.col("x") > 5) # AND
df.filter(pl.col("valid") & (pl.col("x") > 5)) # AND
df.filter(pl.col("valid") | (pl.col("x") > 5)) # OR
df.filter(valid=True, x=5) # AND using constraints

df.drop_nulls() # keep rows without missing values
df.drop_nulls("x") # only consider specific column
df.unique(subset=["x"], keep="first") # Remove dupes
```

Slicing and Sampling Rows

Keep rows based on their position.

```
df.head() # keep first five rows
df.tail(10) # keep last ten rows
df.slice(2, 5) # keep third to seventh row
df.gather_every(2) # keep every second row

df.sample(10) # keep 10 random rows
df.sample(10, with_replacement=True) # allow repeats
df.sample(fraction=0.2) # keep 20% of rows
```

Sorting Rows

Reorder rows according to the values in one or more columns or expressions.

```
df.sort("x") # by single column, ascending
df.sort("x", "y") # by multiple columns
df.sort("x", nulls_last=True) # nulls at the end
df.sort("x", descending=True) # reverse order
df.sort("x", "y", descending=[False, True]) # mixed
df.sort(pl.col("x") / pl.col("y")) # by expression
df.sort(pl.col("l").list.len()) # by list length

df.top_k(5, by="score") # keep 5 largest
df.bottom_k(5, by="score") # keep 5 smallest
```

Reshaping

Wide to long and back again.

```
df.unpivot(on=["c"], index="id") # longer
df.pivot(on="c", index="id", values="x") # wider
df.pivot(on="c", index="id", values="x",
    aggregate_function="sum") # with aggregation

df.explode("l") # one row per list element
df.unnest("s") # separate struct into columns

df.transpose(include_header=True) # Swap rows, cols

df.partition_by("group") # list of DataFrames
```

Summarizing and Aggregating

Split. Apply. Combine.

```
dfg = df.group_by("x") # split into groups
dfg = df.group_by("x", "y") # based on multiple cols

dfg.len() # number of rows in each group
dfg.head(2) # get the first 2 rows of each group
dfg.mean() # compute means of all columns per group
dfg.map_groups(…) # apply UDF to each group

dfg.agg(…) # compute aggregations for each group
dfg.agg(pl.col("y")) # aggregate values to a list
dfg.agg(avg=pl.col("y").mean()) # compute means of y

# add aggregation as new column to original dataframe
df.with_columns(avg=pl.col("y").mean().over("x"))

# group based on a time value or index
df.group_by_dynamic("timestamp",
    every="1h", group_by="store")

# compute 7 day rolling sum of sales per store
df.rolling(index_column="date", period="7d",
    group_by="store").agg(pl.col("sales").sum())

# create missing rows based on temporal column
df.upsample(time_column="date", every="1d",
    group_by="store", maintain_order=True)

# Aggregate across columns (row-wise)
df.select(pl.sum_horizontal(cs.numeric()))
df.select(pl.any_horizontal(cs.boolean()))
```

Joining and Concatenating

Combine multiple DataFrames into one.

```
df.join(o, on="key") # inner join with DataFrame o
df.join(o, on="key", how="left") # left join
df.join(o, left_on="a", right_on="b") # different key
df.join(o, on="key", how="full", coalesce=True)
    # full outer, merge key cols
df.join(o, on="key", how="semi") # keep matching rows
df.join(o, on="key", how="anti") # drop matching rows
df.join(o, how="cross") # cartesian product
df.join_asof(o, on="ts", by="i") # nearest-match join
df.join_where(o, pl.col("a") >= pl.col("b"))
    # inequality / non-equi join on predicates
```

Common keyword arguments for `df.join()`:

```
left_on, right_on, coalesce, join_nulls, suffix,
validate("m:m", "m:1", "1:m", "1:1")
```

```
pl.concat([df, o]) # stack rows vertically
pl.concat([df, o], how="horizontal") # side-by-side
pl.concat([df, o], how="diagonal") # union of cols
pl.concat([df, o], how="vertical_relaxed")
    # coerce mismatched dtypes

# Update df values with non-null values from new_df
df.update(o, on="id", how="left")
```

Expressions

definition of an expression

An expression is a tree of operations that describe how to construct one or more Series.

- **Series:** Same-type array; column or standalone
- **Tree of operations:** Single, linear, or branched
- **Describe:** Passive recipe; needs function to execute
- **Construct:** Output may be internal, not a new column
- **One or more:** One expr can make multiple series

Beginning Expressions

```
pl.col("name") # expression based on existing column
pl.col("*") # expression based on all columns
pl.all() # expression based on all columns
pl.lit("ok") # expression based on literal value

pl.arange(0, 5) # [0, 1, 2, 3, 4]
pl.date_range(…) # a range of dates
pl.date_ranges(…) # each element is a range of dates
# int, time, datetime also have *_range(s) functions
```

Combining Expressions with Arithmetic

You can perform arithmetic with both expressions and Python values.

operator	method	description
+	e.add(…)	Addition
-	e.sub(…)	Subtraction
*	e.mul(…)	Multiplication
/	e.truediv(…)	Division
//	e.floordiv(…)	Floor division
**	e.pow(…)	Power
%	e.mod(…)	Modulus
N/A	e.dot(…)	Dot product

Combining Expressions by Comparing

Unlike in Python, you cannot chain multiple comparisons.

operator	method	description
<	e.lt(…)	Less than
<=	e.le(…)	Less than or equal to
==	e.eq(…)	Equal
>=	e.ge(…)	Greater than or equal to
>	e.gt(…)	Greater than
!=	e.ne(…)	Not equal

Combining Expressions with Boolean Logic

Note that `and`, `or`, and `not` are reserved keywords in Python, hence the underscores in the method names.

operator	method	description
&	e.and_(…)	Logical AND
|	e.or_(…)	Logical OR
~	e.not_()	Logical NOT
^	e.xor(…)	Logical XOR

Conditional Expression

```
df.with_columns(
    pl.when(pl.col("age") < 18).then(pl.lit("minor"))
      .when(pl.col("age") < 65).then(pl.lit("adult"))
      .otherwise(pl.lit("senior")) # first match wins
      .alias("group")
)
```

Math, Trigonometry, and Rounding

```
e.abs(), e.sign(), e.exp() # abs, sign, exp
e.cbrt(), e.sqrt() # cube root, square root
e.log(…), e.log10(), e.log1p() # logs
e.cos(), e.sin(), e.tan() # trig
e.cosh(), e.sinh(), e.tanh() # hyperbolic
e.arccos(), e.arcsin(), e.arctan() # inv trig
e.arccosh(), e.arcsinh(), e.arctanh() # inv hyper
e.degrees(), e.radians() # rad/deg convert
e.ceil(), e.floor(), e.round(…) # rounding
e.clip(…), e.cut(…), e.qcut(…) # clip/bin
```

Missing Values and Shapes

null means missing; NaN is a float from undefined math.

```
e.fill_nan(…), e.fill_null(…) # fill values
e.is_finite(), e.is_infinite() # finite/inf
e.is_nan(), e.is_not_nan() # nan checks
e.is_null(), e.is_not_null() # null checks
e.drop_nans(), e.drop_nulls() # drop missing
e.flatten(), e.reshape(…) # reshape list/col
e.explode(), e.implode() # list explode/implode
```

Shifts, Cumulative, and Rolling

```
e.backward_fill(…), e.forward_fill(…) # fill
e.interpolate(…), e.shift(…) # interpolate/shift
e.cum_count(…), e.cum_sum(…) # count, sum
e.cum_max(…), e.cum_min(…) # max, min
e.diff(…), e.pct_change(…) # diff, % change
e.ewm_mean(…), e.ewm_std(…), e.ewm_var(…) # ewm
e.rolling_max(…), e.rolling_min(…) # roll max/min
e.rolling_mean(…), e.rolling_median(…) # mean/med
e.rolling_std(…), e.rolling_var(…) # roll std/var
e.rolling_map(…) # custom roll
```

Sorting, Ranking, and Boolean

```
e.sort(…), e.sort_by(…) # sort column(s)
e.arg_sort(…) # row indices to sort
e.shuffle(…), e.reverse() # shuffle/reverse
e.rank(…) # assign ranks to data
e.is_duplicated(), e.is_unique() # dups/uniq
e.is_first_distinct() # first unique
e.is_last_distinct() # last unique
```

Summaries and Statistics

```
e.all(…), e.any(…) # true if all/any true
e.max(), e.min(), e.mean() # max, min, mean
e.nan_max(), e.nan_min() # max/min with nan
e.median(), e.std(), e.var(…) # median, std, var
e.entropy(…), e.kurtosis(…), e.skew(…) # stats
e.product(), e.quantile(…), e.sum() # math sum
e.arg_max(), e.arg_min() # index of max/min
e.first(), e.last(), e.get(…) # get by position
e.mode() # most occurring values
```

Counting, Unique, and Selection

```
e.len() # count total rows
e.count() # count non-null values
e.null_count() # count null values
e.n_unique(), e.approx_n_unique() # unique
e.arg_unique(), e.unique(…) # unique indices/vals
e.unique_counts(), e.value_counts(…) # counts
e.head(…), e.tail(…), e.limit(…) # row selection
e.bottom_k(…), e.top_k(…) # k smallest/largest
e.gather(…), e.gather_every(…) # take by index
e.sample(…), e.slice(…) # sample/slice expr
e.arg_true() # indices where true
e.replace(…) # replace via dict
e.search_sorted(…) # search sorted index
```

Arrays and Lists

Arrays have a fixed length; lists do not.

```
e.cast(pl.Array(pl.Int8, 3)) # cast to fixed array
e.arr.max() # maximum value of array
e.arr.sort() # sort inner arrays

pl.list("a", "b") # combine cols to list
e.list.len() # get list lengths
e.list.get(0) # get element by index
e.list.sort() # sort inner lists
e.list.join("-") # join list elements into String
e.list.contains(5) # check if list contains value
```

Categoricals and Enums

Categoricals infer categories and sort lexically; Enums are fixed and sort in declaration order.

```
e.cast(pl.Categorical) # cast String to Categorical
e.cast(pl.Enum(["Good", "Bad"])) # cast to exact Enum
e.cat.get_categories() # get categories
```

Dates, Datetimes, Times, and Durations

Dates track days; Datetimes track microseconds.

```
pl.date(2026, 12, 31) # create Date
pl.datetime(2026, 6, 30, 23, 59, 0) # create Datetime
pl.duration(days=1) # create a time duration
e.dt.month() # extract month
e.dt.replace(…) # replace time units
e.dt.strftime(…) # format datetime
e.dt.convert_time_zone("UTC") # convert timezone
e.dt.total_seconds() # duration to seconds
```

Strings

Strings are UTF-8, so lengths & slices count chars, not bytes.

```
e.str.contains(…) # check regex match
e.str.split(…) # split by separator
e.str.to_uppercase() # make all-caps
e.str.to_datetime() # convert to Datetime
e.str.extract(r"(\d+)") # extract regex group
e.str.strip_chars(…) # trim whitespace or other chars
```

Structs

Group multiple columns into a single row element.

```
pl.struct("a", "b") # combine columns into a Struct
e.struct.field(…) # extract field
e.struct.rename_fields(…) # rename fields
e.struct.with_fields(…) # add or adjust fields
```

Binaries

Use for raw byte data and base64/hex conversions.

```
e.bin.base64_decode() # decode base64
e.bin.hex_encode() # encode to hex string
```

Output Names

Control the final column names of your expressions.

```
e.name.prefix(…) # add prefix to name
e.name.to_lowercase() # lowercase expression name
```

Meta

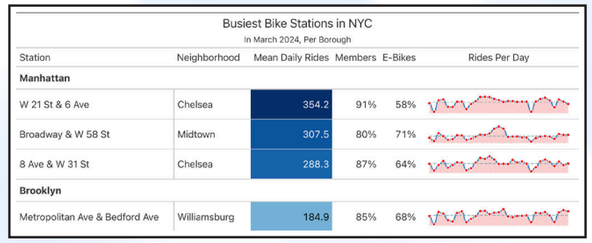
Introspection methods primarily used for plugins.

```
e.meta.output_name() # get expression output name
e.meta.is_regex() # check if expression is a regex
e.meta.has_multiple_outputs() # multiple outputs
```

Styling Data

Use Great Tables to style DataFrames.

```
from great_tables import GT

(
    GT(df)
    .tab_stub(rowname_col="…")
    .cols_label(…).tab_header(title="…")
    .fmt_number(…).fmt_nanoplot(…)
    .data_color(columns="…", palette="…")
)
```



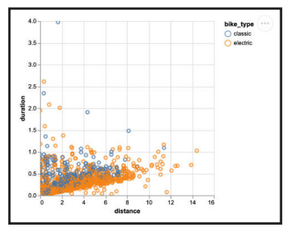
Visualizing Data

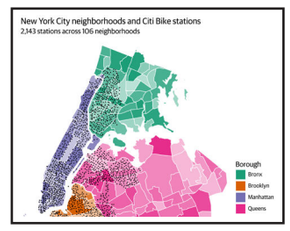
The built-in plotting methods use Altair under the hood:

```
df.plot.scatter(x="…", y="…", color="…")
```

See example on the left. Many other packages can work with Polars DataFrames, including Plotnine (see example on the right), Plotly, hvPlot, Seaborn, and Matplotlib. Otherwise, convert to pandas using `df.to_pandas()` first.

```
from plotnine import *

ggplot(df, aes(x="", y="", color="")) + geom_point()
```





Polars Cloud

Execute a query on a cluster of instances in your own environment.

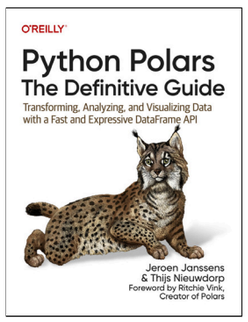
```
import polars_cloud as pc

ctx = pc.ComputeContext(
    workspace="workspace_name",
    cpus=4, memory=16, cluster_size=32
)

lf.remote(ctx).execute().await_result()
```

Book

This cheatsheet is based on the book *Python Polars: The Definitive Guide* by Jeroen Janssens and Thijs Nieuwdorp, published by O'Reilly.

The book is available in both print and ebook formats at your favorite bookstore.

Visit polarsguide.com for details.



 • More cheatsheets at opensource.posit.co/resources/cheatsheets • polars 1.43.0 • Updated: 2026-08