

# Package: polars (via r-universe)

October 16, 2024

**Title** Lightning-Fast 'DataFrame' Library

**Version** 0.20.0

**Depends** R (>= 4.2)

**Imports** utils, codetools, methods

**Description** Lightning-fast 'DataFrame' library written in 'Rust'.  
Convert R data to 'Polars' data and vice versa. Perform fast, lazy, larger-than-memory and optimized data queries. 'Polars' is interoperable with the package 'arrow', as both are based on the 'Apache Arrow' Columnar Format.

**License** MIT + file LICENSE

**Language** en-US

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.3.2

**SystemRequirements** Cargo (rustc package manager), cmake

**URL** <https://pola-rs.github.io/r-polars/>,  
<https://github.com/pola-rs/r-polars>,  
<https://rpolars.r-universe.dev/polars>

**Suggests** arrow (>= 15.0.1), bench, bit64, callr, clock (>= 0.7.0), curl, ggplot2, jsonlite, knitr, lubridate, nanoarrow (>= 0.6.0), nycflights13, patrick, pillar, rlang, rmarkdown, testthat (>= 3.2.1), tibble, tools, vctrs, withr

**Config/Needs/website** altdoc, future.apply, here, magrittr, pkgload, yaml

**Config/Needs/dev** devtools, dplyr, RcppTOML, readr, rextendr, spelling, stringr, styler

**Config/testthat/edition** 3

**Collate** 'utils.R' 'extendr-wrappers.R' 'after-wrappers.R' 'Field.R' 'PTime.R' 'as\_polars.R' 'autocompletion.R' 'construction.R' 'dataframe\_\_frame.R' 'datatype.R' 'docs.R' 'dotdotdot.R'

'error\_\_rpolarserr.R' 'error\_\_string.R' 'error\_\_trait.R'  
 'error\_conversion.R' 'expr\_\_array.R' 'expr\_\_binary.R'  
 'expr\_\_categorical.R' 'expr\_\_datetime.R' 'expr\_\_expr.R'  
 'expr\_\_list.R' 'expr\_\_meta.R' 'expr\_\_name.R' 'expr\_\_string.R'  
 'expr\_\_struct.R' 'functions\_\_eager.R' 'functions\_\_lazy.R'  
 'functions\_\_whenthen.R' 'group\_by.R' 'group\_by\_dynamic.R'  
 'group\_by\_rolling.R' 'io\_csv.R' 'io\_ipc.R' 'io\_json.R'  
 'io\_parquet.R' 'is\_polars.R' 'lazyframe\_\_group\_by.R'  
 'lazyframe\_\_lazy.R' 'parse\_as\_duration.R' 'pkg-arrow.R'  
 'pkg-knitr.R' 'pkg-nanoarrow.R' 'polars-package.R'  
 'polars\_envvars.R' 'polars\_info.R' 'polars\_options.R'  
 'rbackground.R' 'rust\_result.R' 's3-methods.R'  
 's3-methods-operator.R' 'series\_\_series.R' 'sql.R' 'vctrs.R'  
 'zzz.R'

**Config/rextendr/version** 0.3.1

**VignetteBuilder** knitr

**Config/polars/LibVersion** 0.43.0

**Config/polars/RustToolchainVersion** nightly-2024-09-19

**Repository** <https://etiennebacher.r-universe.dev>

**RemoteUrl** <https://github.com/pola-rs/r-polars>

**RemoteRef** v0.20.0

**RemoteSha** 4bd16001d935d2e0b6e7d15d7aac12dc1cc35ec9

## Contents

as.character.RPolarsSeries . . . . .	17
as.data.frame.RPolarsDataFrame . . . . .	17
as.matrix.RPolarsDataFrame . . . . .	20
as.vector.RPolarsSeries . . . . .	20
as_arrow_table.RPolarsDataFrame . . . . .	21
as_nanoarrow_array_stream.RPolarsDataFrame . . . . .	22
as_polars_df . . . . .	23
as_polars_if . . . . .	27
as_polars_series . . . . .	27
as_record_batch_reader.RPolarsDataFrame . . . . .	29
c.RPolarsSeries . . . . .	30
DataFrame_cast . . . . .	31
DataFrame_class . . . . .	32
DataFrame_clear . . . . .	34
DataFrame_clone . . . . .	35
DataFrame_describe . . . . .	36
DataFrame_drop . . . . .	37
DataFrame_drop_in_place . . . . .	38
DataFrame_drop_nulls . . . . .	38
DataFrame_dtype_strings . . . . .	39

DataFrame_equals	39
DataFrame_estimated_size	40
DataFrame_explode	41
DataFrame_fill_nan	42
DataFrame_fill_null	42
DataFrame_filter	43
DataFrame_first	44
DataFrame_gather_every	44
DataFrame_get_column	45
DataFrame_get_columns	45
DataFrame_glimpse	46
DataFrame_group_by	47
DataFrame_group_by_dynamic	48
DataFrame_head	51
DataFrame_item	52
DataFrame_join	52
DataFrame_join_asof	54
DataFrame_join_where	57
DataFrame_last	58
DataFrame_lazy	58
DataFrame_max	59
DataFrame_mean	59
DataFrame_median	60
DataFrame_min	60
DataFrame_null_count	61
DataFrame_n_chunks	61
DataFrame_partition_by	63
DataFrame_pivot	64
DataFrame_quantile	66
DataFrame_rechunk	66
DataFrame_rename	68
DataFrame_reverse	69
DataFrame_rolling	69
DataFrame_sample	71
DataFrame_select	72
DataFrame_select_seq	73
DataFrame_shift	74
DataFrame_slice	74
DataFrame_sort	75
DataFrame_sql	76
DataFrame_std	77
DataFrame_sum	78
DataFrame_tail	78
DataFrame_to_data_frame	79
DataFrame_to_dummies	80
DataFrame_to_list	81
DataFrame_to_raw_ipc	83
DataFrame_to_series	84

DataFrame_to_struct . . . . .	85
DataFrame_transpose . . . . .	86
DataFrame_unique . . . . .	87
DataFrame_unnest . . . . .	88
DataFrame_unpivot . . . . .	88
DataFrame_var . . . . .	90
DataFrame_with_columns . . . . .	90
DataFrame_with_columns_seq . . . . .	91
DataFrame_with_row_index . . . . .	92
DataFrame_write_csv . . . . .	93
DataFrame_write_ipc . . . . .	94
DataFrame_write_json . . . . .	95
DataFrame_write_ndjson . . . . .	96
DataFrame_write_parquet . . . . .	97
DataType_Array . . . . .	99
DataType_Categorical . . . . .	99
DataType_contains_categoricals . . . . .	100
DataType_contains_views . . . . .	101
DataType_Datetime . . . . .	101
DataType_Duration . . . . .	102
DataType_Enum . . . . .	103
DataType_is_array . . . . .	104
DataType_is_binary . . . . .	104
DataType_is_bool . . . . .	105
DataType_is_categorical . . . . .	105
DataType_is_enum . . . . .	106
DataType_is_float . . . . .	106
DataType_is_integer . . . . .	107
DataType_is_known . . . . .	107
DataType_is_list . . . . .	108
DataType_is_logical . . . . .	108
DataType_is_nested . . . . .	109
DataType_is_null . . . . .	109
DataType_is_numeric . . . . .	110
DataType_is_ord . . . . .	110
DataType_is_primitive . . . . .	111
DataType_is_signed_integer . . . . .	111
DataType_is_string . . . . .	112
DataType_is_struct . . . . .	112
DataType_is_temporal . . . . .	113
DataType_is_unsigned_integer . . . . .	113
DataType_List . . . . .	114
DataType_Struct . . . . .	114
dim.RPolarsDataFrame . . . . .	115
dimnames.RPolarsDataFrame . . . . .	116
docs_translations . . . . .	116
DynamicGroupBy_agg . . . . .	118
DynamicGroupBy_class . . . . .	120

DynamicGroupBy_ungroup . . . . .	120
ExprArr_all . . . . .	121
ExprArr_any . . . . .	121
ExprArr_arg_max . . . . .	122
ExprArr_arg_min . . . . .	122
ExprArr_contains . . . . .	123
ExprArr_get . . . . .	124
ExprArr_join . . . . .	124
ExprArr_max . . . . .	125
ExprArr_median . . . . .	126
ExprArr_min . . . . .	126
ExprArr_reverse . . . . .	127
ExprArr_shift . . . . .	127
ExprArr_sort . . . . .	128
ExprArr_std . . . . .	129
ExprArr_sum . . . . .	129
ExprArr_to_list . . . . .	130
ExprArr_to_struct . . . . .	130
ExprArr_unique . . . . .	131
ExprArr_var . . . . .	132
ExprBin_contains . . . . .	132
ExprBin_decode . . . . .	133
ExprBin_encode . . . . .	134
ExprBin_ends_with . . . . .	135
ExprBin_size . . . . .	135
ExprBin_starts_with . . . . .	136
ExprCat_get_categories . . . . .	137
ExprCat_set_ordering . . . . .	137
ExprDT_cast_time_unit . . . . .	138
ExprDT_combine . . . . .	139
ExprDT_convert_time_zone . . . . .	140
ExprDT_day . . . . .	140
ExprDT_epoch . . . . .	141
ExprDT_hour . . . . .	142
ExprDT_iso_year . . . . .	142
ExprDT_is_leap_year . . . . .	143
ExprDT_microsecond . . . . .	143
ExprDT_millisecond . . . . .	144
ExprDT_minute . . . . .	145
ExprDT_month . . . . .	145
ExprDT_nanosecond . . . . .	146
ExprDT_offset_by . . . . .	147
ExprDT_ordinal_day . . . . .	148
ExprDT_quarter . . . . .	149
ExprDT_replace_time_zone . . . . .	150
ExprDT_round . . . . .	151
ExprDT_second . . . . .	152
ExprDT_strftime . . . . .	153

ExprDT_time . . . . .	154
ExprDT_timestamp . . . . .	154
ExprDT_total_days . . . . .	155
ExprDT_total_hours . . . . .	156
ExprDT_total_microseconds . . . . .	156
ExprDT_total_milliseconds . . . . .	157
ExprDT_total_minutes . . . . .	158
ExprDT_total_nanoseconds . . . . .	158
ExprDT_total_seconds . . . . .	159
ExprDT_truncate . . . . .	160
ExprDT_week . . . . .	161
ExprDT_weekday . . . . .	161
ExprDT_with_time_unit . . . . .	162
ExprDT_year . . . . .	163
ExprList_all . . . . .	163
ExprList_any . . . . .	164
ExprList_arg_max . . . . .	164
ExprList_arg_min . . . . .	165
ExprList_concat . . . . .	165
ExprList_contains . . . . .	166
ExprList_diff . . . . .	167
ExprList_eval . . . . .	167
ExprList_explode . . . . .	168
ExprList_first . . . . .	169
ExprList_gather . . . . .	169
ExprList_gather_every . . . . .	170
ExprList_get . . . . .	171
ExprList_head . . . . .	172
ExprList_join . . . . .	172
ExprList_last . . . . .	173
ExprList_len . . . . .	174
ExprList_max . . . . .	174
ExprList_mean . . . . .	175
ExprList_min . . . . .	175
ExprList_n_unique . . . . .	176
ExprList_reverse . . . . .	176
ExprList_sample . . . . .	177
ExprList_set_difference . . . . .	178
ExprList_set_intersection . . . . .	178
ExprList_set_symmetric_difference . . . . .	179
ExprList_set_union . . . . .	180
ExprList_shift . . . . .	181
ExprList_slice . . . . .	181
ExprList_sort . . . . .	182
ExprList_sum . . . . .	183
ExprList_tail . . . . .	183
ExprList_to_struct . . . . .	184
ExprList_unique . . . . .	185

ExprMeta_eq . . . . .	186
ExprMeta_has_multiple_outputs . . . . .	186
ExprMeta_is_regex_projection . . . . .	187
ExprMeta_neq . . . . .	187
ExprMeta_output_name . . . . .	188
ExprMeta_pop . . . . .	189
ExprMeta_root_names . . . . .	190
ExprMeta_tree_format . . . . .	190
ExprMeta_undo_aliases . . . . .	191
ExprName_keep . . . . .	191
ExprName_prefix . . . . .	192
ExprName_prefix_fields . . . . .	192
ExprName_suffix . . . . .	193
ExprName_suffix_fields . . . . .	194
ExprName_to_lowercase . . . . .	194
ExprName_to_uppercase . . . . .	195
ExprStruct_field . . . . .	195
ExprStruct_rename_fields . . . . .	196
ExprStruct_with_fields . . . . .	197
ExprStr_contains . . . . .	198
ExprStr_contains_any . . . . .	199
ExprStr_count_matches . . . . .	200
ExprStr_decode . . . . .	200
ExprStr_encode . . . . .	201
ExprStr_ends_with . . . . .	202
ExprStr_extract . . . . .	203
ExprStr_extract_all . . . . .	203
ExprStr_extract_groups . . . . .	204
ExprStr_extract_many . . . . .	205
ExprStr_find . . . . .	206
ExprStr_head . . . . .	207
ExprStr_join . . . . .	208
ExprStr_json_decode . . . . .	209
ExprStr_json_path_match . . . . .	209
ExprStr_len_bytes . . . . .	210
ExprStr_len_chars . . . . .	211
ExprStr_pad_end . . . . .	212
ExprStr_pad_start . . . . .	212
ExprStr_replace . . . . .	213
ExprStr_replace_all . . . . .	214
ExprStr_replace_many . . . . .	216
ExprStr_reverse . . . . .	217
ExprStr_slice . . . . .	217
ExprStr_split . . . . .	218
ExprStr_splitn . . . . .	218
ExprStr_split_exact . . . . .	219
ExprStr_starts_with . . . . .	220
ExprStr_strip_chars . . . . .	220

ExprStr_strip_chars_end . . . . .	221
ExprStr_strip_chars_start . . . . .	222
ExprStr_strptime . . . . .	222
ExprStr_tail . . . . .	225
ExprStr_to_date . . . . .	226
ExprStr_to_datetime . . . . .	227
ExprStr_to_integer . . . . .	228
ExprStr_to_lowercase . . . . .	229
ExprStr_to_time . . . . .	229
ExprStr_to_titlecase . . . . .	230
ExprStr_to_uppercase . . . . .	230
ExprStr_zfill . . . . .	231
Expr_abs . . . . .	232
Expr_add . . . . .	232
Expr_agg_groups . . . . .	233
Expr_alias . . . . .	234
Expr_all . . . . .	234
Expr_and . . . . .	235
Expr_any . . . . .	236
Expr_append . . . . .	236
Expr_approx_n_unique . . . . .	237
Expr_arccos . . . . .	238
Expr_arccosh . . . . .	238
Expr_arcsin . . . . .	239
Expr_arcsinh . . . . .	239
Expr_arctan . . . . .	240
Expr_arctanh . . . . .	240
Expr_arg_max . . . . .	241
Expr_arg_min . . . . .	241
Expr_arg_sort . . . . .	242
Expr_arg_unique . . . . .	242
Expr_backward_fill . . . . .	243
Expr_bottom_k . . . . .	243
Expr_cast . . . . .	244
Expr_ceil . . . . .	245
Expr_class . . . . .	245
Expr_clip . . . . .	246
Expr_cos . . . . .	247
Expr_cosh . . . . .	247
Expr_count . . . . .	248
Expr_cumulative_eval . . . . .	248
Expr_cum_count . . . . .	249
Expr_cum_max . . . . .	250
Expr_cum_min . . . . .	251
Expr_cum_prod . . . . .	251
Expr_cum_sum . . . . .	252
Expr_cut . . . . .	253
Expr_diff . . . . .	254



Expr_div . . . . .	254
Expr_dot . . . . .	255
Expr_drop_nans . . . . .	256
Expr_drop_nulls . . . . .	256
Expr_entropy . . . . .	257
Expr_eq . . . . .	257
Expr_eq_missing . . . . .	258
Expr_ewm_mean . . . . .	259
Expr_ewm_std . . . . .	260
Expr_ewm_var . . . . .	261
Expr_exclude . . . . .	262
Expr_exp . . . . .	263
Expr_explode . . . . .	264
Expr_extend_constant . . . . .	264
Expr_fill_nan . . . . .	265
Expr_fill_null . . . . .	265
Expr_filter . . . . .	266
Expr_first . . . . .	267
Expr_flatten . . . . .	267
Expr_floor . . . . .	268
Expr_floor_div . . . . .	268
Expr_forward_fill . . . . .	269
Expr_gather . . . . .	270
Expr_gather_every . . . . .	270
Expr_gt . . . . .	271
Expr_gt_eq . . . . .	271
Expr_hash . . . . .	272
Expr_has_nulls . . . . .	272
Expr_head . . . . .	273
Expr_implode . . . . .	273
Expr_inspect . . . . .	274
Expr_interpolate . . . . .	275
Expr_is_between . . . . .	276
Expr_is_duplicated . . . . .	277
Expr_is_finite . . . . .	277
Expr_is_first_distinct . . . . .	278
Expr_is_in . . . . .	278
Expr_is_infinite . . . . .	279
Expr_is_last_distinct . . . . .	280
Expr_is_nan . . . . .	280
Expr_is_not_nan . . . . .	281
Expr_is_not_null . . . . .	281
Expr_is_null . . . . .	282
Expr_is_unique . . . . .	282
Expr_kurtosis . . . . .	283
Expr_last . . . . .	283
Expr_limit . . . . .	284
Expr_log . . . . .	284

Expr_log10 . . . . .	285
Expr_lower_bound . . . . .	285
Expr_lt . . . . .	286
Expr_lt_eq . . . . .	286
Expr_map_batches . . . . .	287
Expr_map_elements . . . . .	289
Expr_max . . . . .	292
Expr_mean . . . . .	293
Expr_median . . . . .	293
Expr_min . . . . .	294
Expr_mod . . . . .	294
Expr_mode . . . . .	295
Expr_mul . . . . .	295
Expr_nan_max . . . . .	296
Expr_nan_min . . . . .	296
Expr_neq . . . . .	297
Expr_neq_missing . . . . .	297
Expr_not . . . . .	298
Expr_null_count . . . . .	299
Expr_n_unique . . . . .	299
Expr_or . . . . .	300
Expr_over . . . . .	300
Expr_pct_change . . . . .	302
Expr_peak_max . . . . .	303
Expr_peak_min . . . . .	303
Expr_pow . . . . .	304
Expr_product . . . . .	305
Expr_qcut . . . . .	305
Expr_quantile . . . . .	306
Expr_rank . . . . .	307
Expr_rechunk . . . . .	308
Expr_reinterpret . . . . .	309
Expr_rep . . . . .	309
Expr_repeat_by . . . . .	310
Expr_replace . . . . .	310
Expr_replace_strict . . . . .	311
Expr_reshape . . . . .	313
Expr_reverse . . . . .	314
Expr_rle . . . . .	314
Expr_rle_id . . . . .	315
Expr_rolling . . . . .	315
Expr_rolling_max . . . . .	317
Expr_rolling_max_by . . . . .	318
Expr_rolling_mean . . . . .	319
Expr_rolling_mean_by . . . . .	320
Expr_rolling_median . . . . .	321
Expr_rolling_median_by . . . . .	322
Expr_rolling_min . . . . .	324

Expr_rolling_min_by . . . . .	325
Expr_rolling_quantile . . . . .	326
Expr_rolling_quantile_by . . . . .	327
Expr_rolling_skew . . . . .	328
Expr_rolling_std . . . . .	329
Expr_rolling_std_by . . . . .	330
Expr_rolling_sum . . . . .	332
Expr_rolling_sum_by . . . . .	333
Expr_rolling_var . . . . .	334
Expr_rolling_var_by . . . . .	335
Expr_round . . . . .	336
Expr_sample . . . . .	337
Expr_search_sorted . . . . .	338
Expr_set_sorted . . . . .	339
Expr_shift . . . . .	339
Expr_shrink_dtype . . . . .	340
Expr_shuffle . . . . .	341
Expr_sign . . . . .	341
Expr_sin . . . . .	342
Expr_sinh . . . . .	342
Expr_skew . . . . .	343
Expr_slice . . . . .	343
Expr_sort . . . . .	344
Expr_sort_by . . . . .	345
Expr_sqrt . . . . .	346
Expr_std . . . . .	346
Expr_sub . . . . .	347
Expr_sum . . . . .	348
Expr_tail . . . . .	348
Expr_tan . . . . .	349
Expr_tanh . . . . .	349
Expr_top_k . . . . .	350
Expr_to_physical . . . . .	350
Expr_to_r . . . . .	351
Expr_to_series . . . . .	352
Expr_unique . . . . .	352
Expr_unique_counts . . . . .	353
Expr_upper_bound . . . . .	353
Expr_value_counts . . . . .	354
Expr_var . . . . .	354
Expr_when_then_otherwise . . . . .	355
Expr_xor . . . . .	357
global_rpool_cap . . . . .	357
GroupBy_agg . . . . .	358
GroupBy_class . . . . .	359
GroupBy_first . . . . .	359
GroupBy_last . . . . .	360
GroupBy_max . . . . .	360

GroupBy_mean . . . . .	361
GroupBy_median . . . . .	362
GroupBy_min . . . . .	362
GroupBy_null_count . . . . .	363
GroupBy_quantile . . . . .	363
GroupBy_shift . . . . .	364
GroupBy_std . . . . .	364
GroupBy_sum . . . . .	365
GroupBy_ungroup . . . . .	365
GroupBy_var . . . . .	366
head.RPolarsDataFrame . . . . .	366
infer_nanoarrow_schema.RPolarsDataFrame . . . . .	368
is_polars_df . . . . .	369
is_polars_dtype . . . . .	369
is_polars_if . . . . .	370
is_polars_series . . . . .	370
knit_print.RPolarsDataFrame . . . . .	371
LazyFrame_cast . . . . .	372
LazyFrame_class . . . . .	373
LazyFrame_clear . . . . .	375
LazyFrame_clone . . . . .	376
LazyFrame_collect . . . . .	377
LazyFrame_collect_in_background . . . . .	379
LazyFrame_drop . . . . .	380
LazyFrame_drop_nulls . . . . .	380
LazyFrame_explain . . . . .	381
LazyFrame_explode . . . . .	382
LazyFrame_fetch . . . . .	383
LazyFrame_fill_nan . . . . .	385
LazyFrame_fill_null . . . . .	386
LazyFrame_filter . . . . .	386
LazyFrame_first . . . . .	387
LazyFrame_gather_every . . . . .	387
LazyFrame_group_by . . . . .	388
LazyFrame_group_by_dynamic . . . . .	389
LazyFrame_head . . . . .	392
LazyFrame_join . . . . .	393
LazyFrame_join_asof . . . . .	395
LazyFrame_join_where . . . . .	398
LazyFrame_last . . . . .	399
LazyFrame_max . . . . .	399
LazyFrame_mean . . . . .	400
LazyFrame_median . . . . .	400
LazyFrame_min . . . . .	401
LazyFrame_print . . . . .	401
LazyFrame_profile . . . . .	402
LazyFrame_quantile . . . . .	404
LazyFrame_rename . . . . .	404

LazyFrame_reverse . . . . .	405
LazyFrame_rolling . . . . .	406
LazyFrame_select . . . . .	408
LazyFrame_select_seq . . . . .	409
LazyFrame_serialize . . . . .	409
LazyFrame_shift . . . . .	410
LazyFrame_sink_csv . . . . .	411
LazyFrame_sink_ipc . . . . .	413
LazyFrame_sink_ndjson . . . . .	414
LazyFrame_sink_parquet . . . . .	416
LazyFrame_slice . . . . .	418
LazyFrame_sort . . . . .	418
LazyFrame_sql . . . . .	419
LazyFrame_std . . . . .	421
LazyFrame_sum . . . . .	421
LazyFrame_tail . . . . .	422
LazyFrame_to_dot . . . . .	422
LazyFrame_unique . . . . .	424
LazyFrame_unnest . . . . .	425
LazyFrame_unpivot . . . . .	426
LazyFrame_var . . . . .	427
LazyFrame_with_columns . . . . .	427
LazyFrame_with_columns_seq . . . . .	428
LazyFrame_with_context . . . . .	429
LazyFrame_with_row_index . . . . .	430
LazyGroupBy_agg . . . . .	431
LazyGroupBy_class . . . . .	431
LazyGroupBy_head . . . . .	432
LazyGroupBy_print . . . . .	432
LazyGroupBy_tail . . . . .	433
LazyGroupBy_ungroup . . . . .	433
length.RPolarsDataFrame . . . . .	434
max.RPolarsDataFrame . . . . .	434
mean.RPolarsDataFrame . . . . .	435
median.RPolarsDataFrame . . . . .	435
min.RPolarsDataFrame . . . . .	436
na.omit.RPolarsLazyFrame . . . . .	436
names.RPolarsDataFrame . . . . .	437
pl_all . . . . .	437
pl_all_horizontal . . . . .	438
pl_any_horizontal . . . . .	439
pl_approx_n_unique . . . . .	439
pl_arg_sort_by . . . . .	440
pl_arg_where . . . . .	441
pl_coalesce . . . . .	442
pl_col . . . . .	443
pl_concat . . . . .	444
pl_concat_list . . . . .	445

pl_concat_str . . . . .	446
pl_corr . . . . .	447
pl_count . . . . .	448
pl_cov . . . . .	449
pl_DataFrame . . . . .	450
pl_date . . . . .	451
pl_datetime . . . . .	452
pl_datetime_range . . . . .	454
pl_datetime_ranges . . . . .	456
pl_date_range . . . . .	458
pl_date_ranges . . . . .	459
pl_deserialize_if . . . . .	461
pl_disable_string_cache . . . . .	461
pl_dtypes . . . . .	462
pl_duration . . . . .	463
pl_element . . . . .	464
pl_enable_string_cache . . . . .	465
pl_field . . . . .	465
pl_Field_class . . . . .	466
pl_first . . . . .	467
pl_fold . . . . .	468
pl_from_epoch . . . . .	469
pl_head . . . . .	470
pl_implode . . . . .	470
pl_int_range . . . . .	471
pl_int_ranges . . . . .	472
pl_is_schema . . . . .	473
pl_last . . . . .	473
pl_LazyFrame . . . . .	474
pl_len . . . . .	475
pl_lit . . . . .	476
pl_max . . . . .	477
pl_max_horizontal . . . . .	478
pl_mean . . . . .	478
pl_mean_horizontal . . . . .	479
pl_median . . . . .	480
pl_mem_address . . . . .	481
pl_min . . . . .	481
pl_min_horizontal . . . . .	482
pl_n_unique . . . . .	483
pl_pl . . . . .	483
pl_PTime . . . . .	484
pl_raw_list . . . . .	485
pl_read_csv . . . . .	487
pl_read_ipc . . . . .	489
pl_read_ndjson . . . . .	490
pl_read_parquet . . . . .	492
pl_reduce . . . . .	494

pl_rolling_corr	495
pl_rolling_cov	496
pl_scan_csv	496
pl_scan_ipc	499
pl_scan_ndjson	501
pl_scan_parquet	502
pl_select	504
pl_Series	505
pl_SQLContext	506
pl_std	507
pl_struct	508
pl_sum	509
pl_sum_horizontal	510
pl_tail	511
pl_thread_pool_size	511
pl_time	512
pl_using_string_cache	513
pl_var	514
pl_with_string_cache	515
polars_class_object	515
polars_code_completion_activate	516
polars_duration_string	517
polars_envvars	518
polars_info	520
polars_options	521
print.RPolarsSeries	522
RollingGroupBy_agg	523
RollingGroupBy_class	523
RollingGroupBy_ungroup	524
row.names.RPolarsDataFrame	525
RThreadHandle_class	525
RThreadHandle_is_finished	526
RThreadHandle_join	526
S3_arithmetic	527
Series_add	529
Series_alias	530
Series_all	530
Series_any	531
Series_append	531
Series_arg_max	532
Series_arg_min	533
Series_chunk_lengths	533
Series_class	534
Series_clear	537
Series_clone	537
Series_compare	538
Series_div	539
Series_equals	540

Series_floor_div . . . . .	541
Series_is_numeric . . . . .	541
Series_is_sorted . . . . .	542
Series_item . . . . .	542
Series_len . . . . .	543
Series_map_elements . . . . .	543
Series_max . . . . .	544
Series_mean . . . . .	545
Series_median . . . . .	545
Series_min . . . . .	546
Series_mod . . . . .	546
Series_mul . . . . .	547
Series_n_chunks . . . . .	548
Series_n_unique . . . . .	548
Series_pow . . . . .	549
Series_print . . . . .	549
Series_rename . . . . .	550
Series_rep . . . . .	550
Series_set_sorted . . . . .	551
Series_sort . . . . .	552
Series_std . . . . .	553
Series_sub . . . . .	553
Series_sum . . . . .	554
Series_to_frame . . . . .	554
Series_to_lit . . . . .	555
Series_to_r . . . . .	555
Series_value_counts . . . . .	557
Series_var . . . . .	558
show_all_public_functions . . . . .	558
show_all_public_methods . . . . .	559
SQLContext_class . . . . .	559
SQLContext_execute . . . . .	560
SQLContext_register . . . . .	560
SQLContext_register_globals . . . . .	561
SQLContext_register_many . . . . .	562
SQLContext_tables . . . . .	563
SQLContext_unregister . . . . .	563
sum.RPolarsDataFrame . . . . .	564
unique.RPolarsDataFrame . . . . .	565
[.RPolarsDataFrame . . . . .	565



---

as.character.RPolarsSeries  
*Convert to a character vector*

---

### Description

Convert to a character vector

### Usage

```
## S3 method for class 'RPolarsSeries'  
as.character(x, ..., str_length = NULL)
```

### Arguments

x	A Polars Series
...	Not used.
str_length	An integer. If specified, utf8 or categorical type Series will be formatted to a string of this length.

### Examples

```
s = as_polars_series(c("foo", "barbaz"))  
as.character(s)  
as.character(s, str_length = 3)
```

---

as.data.frame.RPolarsDataFrame  
*Convert to a data.frame*

---

### Description

Equivalent to `as_polars_df(x, ...) $to_data_frame(...)`.

### Usage

```
## S3 method for class 'RPolarsDataFrame'  
as.data.frame(x, ..., int64_conversion = polars_options()$int64_conversion)  
  
## S3 method for class 'RPolarsLazyFrame'  
as.data.frame(  
  x,  
  ...,  
  n_rows = Inf,  
  type_coercion = TRUE,
```

```

predicate_pushdown = TRUE,
projection_pushdown = TRUE,
simplify_expression = TRUE,
slice_pushdown = TRUE,
comm_subplan_elim = TRUE,
comm_subexpr_elim = TRUE,
cluster_with_columns = TRUE,
streaming = FALSE,
no_optimization = FALSE,
collect_in_background = FALSE
)

```

## Arguments

x	An object to convert to a <a href="#">data.frame</a> .
...	Additional arguments passed to methods.
int64_conversion	How should Int64 values be handled when converting a polars object to R? <ul style="list-style-type: none"> <li>• "double" (default) converts the integer values to double.</li> <li>• "bit64" uses <code>bit64::as.integer64()</code> to do the conversion (requires the package <code>bit64</code> to be attached).</li> <li>• "string" converts Int64 values to character.</li> </ul>
n_rows	Number of rows to fetch. Defaults to <code>Inf</code> , meaning all rows.
type_coercion	Logical. Coerce types such that operations succeed and run on minimal required memory.
predicate_pushdown	Logical. Applies filters as early as possible at scan level.
projection_pushdown	Logical. Select only the columns that are needed at the scan level.
simplify_expression	Logical. Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.
slice_pushdown	Logical. Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. <code>join\$head(10)</code> ).
comm_subplan_elim	Logical. Will try to cache branching subplans that occur on self-joins or unions.
comm_subexpr_elim	Logical. Common subexpressions will be cached and reused.
cluster_with_columns	Combine sequential independent calls to <a href="#">with_columns()</a> .
streaming	Logical. Run parts of the query in a streaming fashion (this is in an alpha state).
no_optimization	Logical. Sets the following parameters to FALSE: <code>predicate_pushdown</code> , <code>projection_pushdown</code> , <code>slice_pushdown</code> , <code>comm_subplan_elim</code> , <code>comm_subexpr_elim</code> , <code>cluster_with_columns</code> .

```
collect_in_background
```

Logical. Detach this query from R session. Computation will start in background. Get a handle which later can be converted into the resulting DataFrame. Useful in interactive mode to not lock R session.

### Conversion to R data types considerations

When converting Polars objects, such as [DataFrames](#) to R objects, for example via the `as.data.frame()` generic function, each type in the Polars object is converted to an R type. In some cases, an error may occur because the conversion is not appropriate. In particular, there is a high possibility of an error when converting a [Datetime](#) type without a time zone. A [Datetime](#) type without a time zone in Polars is converted to the [POSIXct](#) type in R, which takes into account the time zone in which the R session is running (which can be checked with the `Sys.timezone()` function). In this case, if ambiguous times are included, a conversion error will occur. In such cases, change the session time zone using `Sys.setenv(TZ = "UTC")` and then perform the conversion, or use the `$dt$replace_time_zone()` method on the Datetime type column to explicitly specify the time zone before conversion.

```
# Due to daylight savings, clocks were turned forward 1 hour on Sunday, March 8, 2020, 2:00:00 am
# so this particular date-time doesn't exist
non_existent_time = as_polars_series("2020-03-08 02:00:00")$str$strptime(pl$Datetime(), "%F %T")
```

```
withr::with_timezone(
  "America/New_York",
  {
    tryCatch(
      # This causes an error due to the time zone (the `TZ` env var is affected).
      as.vector(non_existent_time),
      error = function(e) e
    )
  }
)
```

```
#> <error: in to_r: ComputeError(ErrString("datetime '2020-03-08 02:00:00' is non-existent in time zone
```

```
withr::with_timezone(
  "America/New_York",
  {
    # This is safe.
    as.vector(non_existent_time$dt$replace_time_zone("UTC"))
  }
)
#> [1] "2020-03-08 02:00:00 UTC"
```

### See Also

- [as\\_polars\\_df\(\)](#)
- `<DataFrame>$to_data_frame()`

---

`as.matrix.RPolarsDataFrame`*Convert to a matrix*

---

**Description**

Equivalent to `as.data.frame(x, ...) |> as.matrix()`.

**Usage**

```
## S3 method for class 'RPolarsDataFrame'  
as.matrix(x, ...)
```

```
## S3 method for class 'RPolarsLazyFrame'  
as.matrix(x, ...)
```

**Arguments**

<code>x</code>	An object to convert to a <a href="#">matrix</a> .
<code>...</code>	Additional arguments passed to methods.

---

`as.vector.RPolarsSeries`*Convert to a vector*

---

**Description**

Convert to a vector

**Usage**

```
## S3 method for class 'RPolarsSeries'  
as.vector(x, mode)
```

**Arguments**

<code>x</code>	A Polars Series
<code>mode</code>	Not used.

### Conversion to R data types considerations

When converting Polars objects, such as [DataFrames](#) to R objects, for example via the `as.data.frame()` generic function, each type in the Polars object is converted to an R type. In some cases, an error may occur because the conversion is not appropriate. In particular, there is a high possibility of an error when converting a [Datetime](#) type without a time zone. A [Datetime](#) type without a time zone in Polars is converted to the [POSIXct](#) type in R, which takes into account the time zone in which the R session is running (which can be checked with the `Sys.timezone()` function). In this case, if ambiguous times are included, a conversion error will occur. In such cases, change the session time zone using `Sys.setenv(TZ = "UTC")` and then perform the conversion, or use the `$dt$replace_time_zone()` method on the Datetime type column to explicitly specify the time zone before conversion.

```
# Due to daylight savings, clocks were turned forward 1 hour on Sunday, March 8, 2020, 2:00:00 am
# so this particular date-time doesn't exist
non_existent_time = as_polars_series("2020-03-08 02:00:00")$str$strptime(pl$Datetime(), "%F %T")
```

```
withr::with_timezone(
  "America/New_York",
  {
    tryCatch(
      # This causes an error due to the time zone (the `TZ` env var is affected).
      as.vector(non_existent_time),
      error = function(e) e
    )
  }
)
```

```
#> <error: in to_r: ComputeError(ErrString("datetime '2020-03-08 02:00:00' is non-existent in time zone
```

```
withr::with_timezone(
  "America/New_York",
  {
    # This is safe.
    as.vector(non_existent_time$dt$replace_time_zone("UTC"))
  }
)
#> [1] "2020-03-08 02:00:00 UTC"
```

---

```
as_arrow_table.RPolarsDataFrame
```

*Create a arrow Table from a Polars object*

---

### Description

Create a arrow Table from a Polars object

**Usage**

```
## S3 method for class 'RPolarsDataFrame'
as_arrow_table(x, ..., compat_level = FALSE)
```

**Arguments**

x	A Polars DataFrame
...	Ignored
compat_level	Use a specific compatibility level when exporting Polars' internal data structures. This can be: <ul style="list-style-type: none"> <li>an integer indicating the compatibility version (currently only 0 for oldest and 1 for newest);</li> <li>a logical value with TRUE for the newest version and FALSE for the oldest version.</li> </ul>

**Examples**

```
library(arrow)

pl_df = as_polars_df(mtcars)
as_arrow_table(pl_df)
```

---

```
as_nanoarrow_array_stream.RPolarsDataFrame
```

*Create a nanoarrow\_array\_stream from a Polars object*

---

**Description**

Create a nanoarrow\_array\_stream from a Polars object

**Usage**

```
## S3 method for class 'RPolarsDataFrame'
as_nanoarrow_array_stream(x, ..., schema = NULL, compat_level = FALSE)
```

```
## S3 method for class 'RPolarsSeries'
as_nanoarrow_array_stream(x, ..., schema = NULL, compat_level = FALSE)
```

**Arguments**

x	A polars object
...	Ignored
schema	must stay at default value NULL
compat_level	Use a specific compatibility level when exporting Polars' internal data structures. This can be:

- an integer indicating the compatibility version (currently only 0 for oldest and 1 for newest);
- a logical value with TRUE for the newest version and FALSE for the oldest version.

### Examples

```
library(nanoarrow)

pl_df = as_polars_df(mtcars)$head(5)
pl_s = as_polars_series(letters[1:5])

as.data.frame(as_nanoarrow_array_stream(pl_df))
as.vector(as_nanoarrow_array_stream(pl_s))
```

---

as_polars_df	<i>To polars DataFrame</i>
--------------	----------------------------

---

### Description

`as_polars_df()` is a generic function that converts an R object to a [polars DataFrame](#).

### Usage

```
as_polars_df(x, ...)
```

## Default S3 method:  
as\_polars\_df(x, ...)

## S3 method for class 'data.frame'  
as\_polars\_df(  
 x,  
 ...,  
 rownames = NULL,  
 make\_names\_unique = TRUE,  
 schema = NULL,  
 schema\_overrides = NULL  
)

## S3 method for class 'RPolarsDataFrame'  
as\_polars\_df(x, ...)

## S3 method for class 'RPolarsGroupBy'  
as\_polars\_df(x, ...)

## S3 method for class 'RPolarsRollingGroupBy'  
as\_polars\_df(x, ...)

```
## S3 method for class 'RPolarsDynamicGroupBy'  
as_polars_df(x, ...)  
  
## S3 method for class 'RPolarsSeries'  
as_polars_df(x, ...)  
  
## S3 method for class 'RPolarsLazyFrame'  
as_polars_df(  
  x,  
  n_rows = Inf,  
  ...,  
  type_coercion = TRUE,  
  predicate_pushdown = TRUE,  
  projection_pushdown = TRUE,  
  simplify_expression = TRUE,  
  slice_pushdown = TRUE,  
  comm_subplan_elim = TRUE,  
  comm_subexpr_elim = TRUE,  
  cluster_with_columns = TRUE,  
  streaming = FALSE,  
  no_optimization = FALSE,  
  collect_in_background = FALSE  
)  
  
## S3 method for class 'RPolarsLazyGroupBy'  
as_polars_df(x, ...)  
  
## S3 method for class 'ArrowTabular'  
as_polars_df(  
  x,  
  ...,  
  rechunk = TRUE,  
  schema = NULL,  
  schema_overrides = NULL,  
  experimental = FALSE  
)  
  
## S3 method for class 'RecordBatchReader'  
as_polars_df(x, ..., experimental = FALSE)  
  
## S3 method for class 'nanoarrow_array'  
as_polars_df(x, ...)  
  
## S3 method for class 'nanoarrow_array_stream'  
as_polars_df(x, ..., experimental = FALSE)
```



**Arguments**

x	Object to convert to a polars DataFrame.
...	Additional arguments passed to methods.
rownames	How to treat existing row names of a data frame: <ul style="list-style-type: none"> <li>• NULL: Remove row names. This is the default.</li> <li>• A string: The name of a new column, which will contain the row names. If x already has a column with that name, an error is thrown.</li> </ul>
make_names_unique	A logical flag to replace duplicated column names with unique names. If FALSE and there are duplicated column names, an error is thrown.
schema	named list of DataTypes, or character vector of column names. Should match the number of columns in x and correspond to each column in x by position. If a column in x does not match the name or type at the same position, it will be renamed/recast. If NULL (default), convert columns as is.
schema_overrides	named list of DataTypes. Cast some columns to the DataType.
n_rows	Number of rows to fetch. Defaults to Inf, meaning all rows.
type_coercion	Logical. Coerce types such that operations succeed and run on minimal required memory.
predicate_pushdown	Logical. Applies filters as early as possible at scan level.
projection_pushdown	Logical. Select only the columns that are needed at the scan level.
simplify_expression	Logical. Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.
slice_pushdown	Logical. Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. <code>join\$head(10)</code> ).
comm_subplan_elim	Logical. Will try to cache branching subplans that occur on self-joins or unions.
comm_subexpr_elim	Logical. Common subexpressions will be cached and reused.
cluster_with_columns	Combine sequential independent calls to <code>with_columns()</code> .
streaming	Logical. Run parts of the query in a streaming fashion (this is in an alpha state).
no_optimization	Logical. Sets the following parameters to FALSE: <code>predicate_pushdown</code> , <code>projection_pushdown</code> , <code>slice_pushdown</code> , <code>comm_subplan_elim</code> , <code>comm_subexpr_elim</code> , <code>cluster_with_columns</code> .
collect_in_background	Logical. Detach this query from R session. Computation will start in background. Get a handle which later can be converted into the resulting DataFrame. Useful in interactive mode to not lock R session.
rechunk	A logical flag (default TRUE). Make sure that all data of each column is in contiguous memory.

`experimental` If TRUE, use experimental Arrow C stream interface inside the function. This argument is experimental and may be removed in the future.

## Details

For [LazyFrame](#) objects, this function is a shortcut for `$collect()` or `$fetch()`, depending on whether the number of rows to fetch is infinite or not.

## Value

a [DataFrame](#)

## Examples

```
# Convert the row names of a data frame to a column
as_polars_df(mtcars, rownames = "car")

# Convert a data frame, with renaming all columns
as_polars_df(
  data.frame(x = 1, y = 2),
  schema = c("a", "b")
)

# Convert a data frame, with renaming and casting all columns
as_polars_df(
  data.frame(x = 1, y = 2),
  schema = list(b = pl$Int64, a = pl$String)
)

# Convert a data frame, with casting some columns
as_polars_df(
  data.frame(x = 1, y = 2),
  schema_overrides = list(y = pl$String) # cast some columns
)

# Convert an arrow Table to a polars DataFrame
at = arrow::arrow_table(x = 1:5, y = 6:10)
as_polars_df(at)

# Create a polars DataFrame from a data.frame
lf = as_polars_df(mtcars)$lazy()

# Collect all rows from the LazyFrame
as_polars_df(lf)

# Fetch 5 rows from the LazyFrame
as_polars_df(lf, 5)
```

---

as\_polars\_lf                    *To polars LazyFrame*

---

### Description

[as\\_polars\\_lf\(\)](#) is a generic function that converts an R object to a polars LazyFrame. It is basically a shortcut for [as\\_polars\\_df\(x, ...\)](#) with the [\\$lazy\(\)](#) method.

### Usage

```
as_polars_lf(x, ...)  
  
## Default S3 method:  
as_polars_lf(x, ...)  
  
## S3 method for class 'RPolarsLazyFrame'  
as_polars_lf(x, ...)  
  
## S3 method for class 'RPolarsLazyGroupBy'  
as_polars_lf(x, ...)
```

### Arguments

x                    Object to convert to a polars DataFrame.  
...                  Additional arguments passed to methods.

### Value

a [LazyFrame](#)

### Examples

```
as_polars_lf(mtcars)
```

---

as\_polars\_series                *To polars Series*

---

### Description

[as\\_polars\\_series\(\)](#) is a generic function that converts an R object to a [polars Series](#).

**Usage**

```
as_polars_series(x, name = NULL, ...)  
  
## Default S3 method:  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'RPolarsSeries'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'RPolarsExpr'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'RPolarsThen'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'RPolarsChainedThen'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'POSIXlt'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'data.frame'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'vctrs_rcrd'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'Array'  
as_polars_series(x, name = NULL, ..., rechunk = TRUE)  
  
## S3 method for class 'ChunkedArray'  
as_polars_series(x, name = NULL, ..., rechunk = TRUE)  
  
## S3 method for class 'RecordBatchReader'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'nanoarrow_array'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'nanoarrow_array_stream'  
as_polars_series(x, name = NULL, ..., experimental = FALSE)  
  
## S3 method for class 'clock_time_point'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'clock_sys_time'  
as_polars_series(x, name = NULL, ...)
```

```
## S3 method for class 'clock_zoned_time'
as_polars_series(x, name = NULL, ...)
```

```
## S3 method for class 'list'
as_polars_series(x, name = NULL, ...)
```

### Arguments

x	Object to convert into <a href="#">a polars Series</a> .
name	A character to use as the name of the <a href="#">Series</a> . If NULL (default), the name of x is used or an empty character "" will be used if x has no name.
...	Additional arguments passed to methods.
rechunk	A logical flag (default TRUE). Make sure that all data is in contiguous memory.
experimental	If TRUE, use experimental Arrow C stream interface inside the function. This argument is experimental and may be removed in the future.

### Value

a [Series](#)

### Examples

```
as_polars_series(1:4)

as_polars_series(list(1:4))

as_polars_series(data.frame(a = 1:4))

as_polars_series(as_polars_series(1:4, name = "foo"))

as_polars_series(pl$lit(1:4))

# Nested type support
as_polars_series(list(data.frame(a = I(list(1:4)))))
```

---

```
as_record_batch_reader.RPolarsDataFrame
  Create a arrow RecordBatchReader from a Polars object
```

---

### Description

Create a arrow RecordBatchReader from a Polars object

### Usage

```
## S3 method for class 'RPolarsDataFrame'
as_record_batch_reader(x, ..., compat_level = FALSE)
```

**Arguments**

x	<a href="#">A Polars DataFrame</a>
...	Ignored
compat_level	Use a specific compatibility level when exporting Polars' internal data structures. This can be: <ul style="list-style-type: none"> <li>an integer indicating the compatibility version (currently only 0 for oldest and 1 for newest);</li> <li>a logical value with TRUE for the newest version and FALSE for the oldest version.</li> </ul>

**Examples**

```
library(arrow)

pl_df = as_polars_df(mtcars)
as_record_batch_reader(pl_df)
```

---

c.RPolarsSeries

*Combine to a Series*


---

**Description**

Combine to a Series

**Usage**

```
## S3 method for class 'RPolarsSeries'
c(x, ...)
```

**Arguments**

x	A Polars Series
...	Series(s) or any object that can be converted to a Series.

**Details**

All objects must have the same datatype. Combining does not rechunk. Read more about R vectors, Series and chunks in [docs\\_translations](#):

**Value**

a combined Series

**Examples**

```
s = c(as_polars_series(1:5), 3:1, NA_integer_)
s$chunk_lengths() # the series contain three unmerged chunks
```

---

DataFrame_cast	<i>Cast DataFrame column(s) to the specified dtype</i>
----------------	--

---

### Description

This allows to convert all columns to a datatype or to convert only specific columns. Contrarily to the Python implementation, it is not possible to convert all columns of a specific datatype to another datatype.

### Usage

```
DataFrame_cast(dtypes, ..., strict = TRUE)
```

### Arguments

<code>dtypes</code>	Either a datatype or a list where the names are column names and the values are the datatypes to convert to.
<code>...</code>	Ignored.
<code>strict</code>	If TRUE (default), throw an error if a cast could not be done (for instance, due to an overflow). Otherwise, return null.

### Value

A DataFrame

### Examples

```
df = pl$DataFrame(  
  foo = 1:3,  
  bar = c(6, 7, 8),  
  ham = as.Date(c("2020-01-02", "2020-03-04", "2020-05-06"))  
)  
  
# Cast only some columns  
df$cast(list(foo = pl$Float32, bar = pl$UInt8))  
  
# Cast all columns to the same type  
df$cast(pl$String)
```

## Description

The DataFrame-class is simply two environments of respectively the public and private methods/function calls to the polars Rust side. The instantiated DataFrame-object is an externalptr to a low-level Rust polars DataFrame object.

The S3 method `.DollarNames.RPolarsDataFrame` exposes all public `$foobar()`-methods which are callable onto the object. Most methods return another DataFrame- class instance or similar which allows for method chaining. This class system could be called "environment classes" (in lack of a better name) and is the same class system `extendr` provides, except here there are both a public and private set of methods. For implementation reasons, the private methods are external and must be called from `.pr$DataFrame$methodname()`. Also, all private methods must take any `self` as an argument, thus they are pure functions. Having the private methods as pure functions solved/simplified self-referential complications.

## Details

Check out the source code in [R/dataframe\\_frame.R](#) to see how public methods are derived from private methods. Check out [extendr-wrappers.R](#) to see the `extendr`-auto-generated methods. These are moved to `.pr` and converted into pure external functions in [after-wrappers.R](#). In [zzz.R](#) (named `zzz` to be last file sourced) the `extendr`-methods are removed and replaced by any function prefixed `DataFrame_`.

## Active bindings

### columns:

`$columns` returns a character vector with the column names.

### dtypes:

`$dtypes` returns a unnamed list with the [data type](#) of each column.

### flags:

`$flags` returns a nested list with column names at the top level and column flags in each sublist.

Flags are used internally to avoid doing unnecessary computations, such as sorting a variable that we know is already sorted. The number of flags varies depending on the column type: columns of type `array` and `list` have the flags `SORTED_ASC`, `SORTED_DESC`, and `FAST_EXPLODE`, while other column types only have the former two.

- `SORTED_ASC` is set to `TRUE` when we sort a column in increasing order, so that we can use this information later on to avoid re-sorting it.
- `SORTED_DESC` is similar but applies to sort in decreasing order.

### height:

`$height` returns the number of rows in the DataFrame.



**schema:**

\$schema returns a named list with the [data type](#) of each column.

**shape:**

\$shape returns a numeric vector of length two with the number of rows and the number of columns.

**width:**

\$width returns the number of columns in the DataFrame.

**Conversion to R data types considerations**

When converting Polars objects, such as [DataFrames](#) to R objects, for example via the `as.data.frame()` generic function, each type in the Polars object is converted to an R type. In some cases, an error may occur because the conversion is not appropriate. In particular, there is a high possibility of an error when converting a [Datetime](#) type without a time zone. A [Datetime](#) type without a time zone in Polars is converted to the [POSIXct](#) type in R, which takes into account the time zone in which the R session is running (which can be checked with the `Sys.timezone()` function). In this case, if ambiguous times are included, a conversion error will occur. In such cases, change the session time zone using `Sys.setenv(TZ = "UTC")` and then perform the conversion, or use the `$dt$replace_time_zone()` method on the [Datetime](#) type column to explicitly specify the time zone before conversion.

```
# Due to daylight savings, clocks were turned forward 1 hour on Sunday, March 8, 2020, 2:00:00 am
# so this particular date-time doesn't exist
non_existent_time = as_polars_series("2020-03-08 02:00:00")$str$strptime(pl$Datetime(), "%F %T")

withr::with_timezone(
  "America/New_York",
  {
    tryCatch(
      # This causes an error due to the time zone (the `TZ` env var is affected).
      as.vector(non_existent_time),
      error = function(e) e
    )
  }
)
#> <error: in to_r: ComputeError(ErrString("datetime '2020-03-08 02:00:00' is non-existent in time zone

withr::with_timezone(
  "America/New_York",
  {
    # This is safe.
    as.vector(non_existent_time$dt$replace_time_zone("UTC"))
  }
)
#> [1] "2020-03-08 02:00:00 UTC"
```

**Examples**

```

# see all public exported method names (normally accessed via a class
# instance with $)
ls(.pr$env$RPolarsDataFrame)

# see all private methods (not intended for regular use)
ls(.pr$DataFrame)

# make an object
df = as_polars_df(iris)

# call an active binding
df$shape

# use a private method, which has mutability
result = .pr$DataFrame$set_column_from_robj(df, 150:1, "some_ints")

# Column exists in both dataframes-objects now, as they are just pointers to
# the same object
# There are no public methods with mutability.
df2 = df

df$columns
df2$columns

# Show flags
df$sort("Sepal.Length")$flags

# set_column_from_robj-method is fallible and returned a result which could
# be "ok" or an error.
# No public method or function will ever return a result.
# The `result` is very close to the same as output from functions decorated
# with purrr::safely.
# To use results on the R side, these must be unwrapped first such that
# potentially errors can be thrown. `unwrap(result)` is a way to communicate
# errors happening on the Rust side to the R side. `Extendr` default behavior
# is to use `panic!`(s) which would cause some unnecessarily confusing and
# some very verbose error messages on the inner workings of rust.
# `unwrap(result)` in this case no error, just a NULL because this mutable
# method does not return any ok-value.

# Try unwrapping an error from polars due to unmatched column lengths
err_result = .pr$DataFrame$set_column_from_robj(df, 1:10000, "wrong_length")
tryCatch(unwrap(err_result, call = NULL), error = \(e) cat(as.character(e)))

```

**Description**

Returns a n-row null-filled DataFrame with an identical schema. n can be greater than the current number of rows in the DataFrame.

**Usage**

```
DataFrame_clear(n = 0)
```

**Arguments**

n                      Number of (null-filled) rows to return in the cleared frame.

**Value**

A n-row null-filled DataFrame with an identical schema

**Examples**

```
df = pl$DataFrame(  
  a = c(NA, 2, 3, 4),  
  b = c(0.5, NA, 2.5, 13),  
  c = c(TRUE, TRUE, FALSE, NA)  
)  
  
df$clear()  
  
df$clear(n = 5)
```

---

DataFrame_clone	<i>Clone a DataFrame</i>
-----------------	--------------------------

---

**Description**

This makes a very cheap deep copy/clone of an existing [DataFrame](#). Rarely useful as DataFrames are nearly 100% immutable. Any modification of a DataFrame should lead to a clone anyways, but this can be useful when dealing with attributes (see examples).

**Usage**

```
DataFrame_clone()
```

**Value**

A DataFrame

**Examples**

```
df1 = pl$DataFrame(iris)

# Make a function to take a DataFrame, add an attribute, and return a DataFrame
give_attr = function(data) {
  attr(data, "created_on") = "2024-01-29"
  data
}
df2 = give_attr(df1)

# Problem: the original DataFrame also gets the attribute while it shouldn't!
attributes(df1)

# Use $clone() inside the function to avoid that
give_attr = function(data) {
  data = data$clone()
  attr(data, "created_on") = "2024-01-29"
  data
}
df1 = pl$DataFrame(iris)
df2 = give_attr(df1)

# now, the original DataFrame doesn't get this attribute
attributes(df1)
```

---

DataFrame\_describe      *Summary statistics for a DataFrame*

---

**Description**

This returns the total number of rows, the number of missing values, the mean, standard deviation, min, max, median and the percentiles specified in the argument percentiles.

**Usage**

```
DataFrame_describe(percentiles = c(0.25, 0.75), interpolation = "nearest")
```

**Arguments**

percentiles	One or more percentiles to include in the summary statistics. All values must be in the range [0; 1].
interpolation	Interpolation method for computing quantiles. One of "nearest", "higher", "lower", "midpoint", or "linear".

**Value**

DataFrame

## Examples

```
pl$DataFrame(iris)$describe()

# string, date, boolean columns are also supported:
df = pl$DataFrame(
  int = 1:3,
  string = c(letters[1:2], NA),
  date = c(as.Date("2024-01-20"), as.Date("2024-01-21"), NA),
  cat = factor(c(letters[1:2], NA)),
  bool = c(TRUE, FALSE, NA)
)
df

df$describe()
```

---

DataFrame_drop	<i>Drop columns of a DataFrame</i>
----------------	------------------------------------

---

## Description

Drop columns of a DataFrame

## Usage

```
DataFrame_drop(..., strict = TRUE)
```

## Arguments

...	Characters of column names to drop. Passed to <code>pl\$col()</code> .
strict	Validate that all column names exist in the schema and throw an exception if a column name does not exist in the schema.

## Value

DataFrame

## Examples

```
pl$DataFrame(mtcars)$drop(c("mpg", "hp"))

# equivalent
pl$DataFrame(mtcars)$drop("mpg", "hp")
```

---

DataFrame\_drop\_in\_place  
*Drop in place*

---

**Description**

Drop a single column in-place and return the dropped column.

**Usage**

```
DataFrame_drop_in_place(name)
```

**Arguments**

name                    string Name of the column to drop.

**Value**

Series

**Examples**

```
dat = pl$DataFrame(iris)
x = dat$drop_in_place("Species")
x
dat$columns
```

---

DataFrame\_drop\_nulls    *Drop nulls (missing values)*

---

**Description**

Drop all rows that contain nulls (which correspond to NA in R).

**Usage**

```
DataFrame_drop_nulls(subset = NULL)
```

**Arguments**

subset                    A character vector with the names of the column(s) for which nulls are considered. If NULL (default), use all columns.

**Value**

DataFrame

**Examples**

```
tmp = mtcars
tmp[1:3, "mpg"] = NA
tmp[4, "hp"] = NA
tmp = pl$DataFrame(tmp)

# number of rows in `tmp` before dropping nulls
tmp$height

tmp$drop_nulls()$height
tmp$drop_nulls("mpg")$height
tmp$drop_nulls(c("mpg", "hp"))$height
```

---

DataFrame\_dtype\_strings

*Data types information*

---

**Description**

Get the data type of all columns as strings. You can see all available types with `names(pl$dtypes)`. The data type of each column is also shown when printing the DataFrame.

**Usage**

```
DataFrame_dtype_strings()
```

**Value**

A character vector with the data type of each column

**Examples**

```
pl$DataFrame(iris)$dtype_strings()
```

---

DataFrame\_equals

*Compare two DataFrames*

---

**Description**

Check if two DataFrames are equal.

**Usage**

```
DataFrame_equals(other)
```

**Arguments**

other            DataFrame to compare with.

**Value**

A logical value

**Examples**

```
dat1 = pl$DataFrame(iris)
dat2 = pl$DataFrame(iris)
dat3 = pl$DataFrame(mtcars)
dat1$equals(dat2)
dat1$equals(dat3)
```

---

DataFrame\_estimated\_size

*Estimated size*

---

**Description**

Return an estimation of the total (heap) allocated size of the DataFrame.

**Usage**

```
DataFrame_estimated_size()
```

**Format**

function

**Value**

Estimated size in bytes

**Examples**

```
pl$DataFrame(mtcars)$estimated_size()
```



---

DataFrame_explode	<i>Explode columns containing a list of values</i>
-------------------	--

---

## Description

Explode columns containing a list of values

## Usage

```
DataFrame_explode(...)
```

## Arguments

... Column(s) to be exploded as individual Into<Expr> or list/vector of Into<Expr>. In a handful of places in rust-polars, only the plain variant Expr::Column is accepted. This is currently one of such places. Therefore pl\$col("name") and pl\$all() is allowed, not pl\$col("name")\$alias("newname"). "name" is implicitly converted to pl\$col("name").

## Value

DataFrame

## Examples

```
df = pl$DataFrame(  
  letters = letters[1:4],  
  numbers = list(1, c(2, 3), c(4, 5), c(6, 7, 8)),  
  numbers_2 = list(0, c(1, 2), c(3, 4), c(5, 6, 7)) # same structure as numbers  
)  
df  
  
# explode a single column, append others  
df$explode("numbers")  
  
# explode two columns of same nesting structure, by names or the common dtype  
# "List(Float64)"  
df$explode("numbers", "numbers_2")  
df$explode(pl$col(pl$List(pl$Float64)))
```

---

DataFrame\_fill\_nan      *Fill floating point NaN value with a fill value*

---

**Description**

Fill floating point NaN value with a fill value

**Usage**

```
DataFrame_fill_nan(value)
```

**Arguments**

value                  Value used to fill NaN values.

**Value**

DataFrame

**Examples**

```
df = pl$DataFrame(  
  a = c(1.5, 2, NaN, 4),  
  b = c(1.5, NaN, NaN, 4)  
)  
df$fill_nan(99)
```

---

DataFrame\_fill\_null      *Fill nulls*

---

**Description**

Fill null values (which correspond to NA in R) using the specified value or strategy.

**Usage**

```
DataFrame_fill_null(fill_value)
```

**Arguments**

fill\_value            Value to fill nulls with.

**Value**

DataFrame

**Examples**

```
df = pl$DataFrame(  
  a = c(1.5, 2, NA, 4),  
  b = c(1.5, NA, NA, 4)  
)  
  
df$fill_null(99)  
  
df$fill_null(pl$col("a")$mean())
```

---

DataFrame_filter	<i>Filter rows of a DataFrame</i>
------------------	-----------------------------------

---

**Description**

Filter rows with an Expression defining a boolean column. Multiple expressions are combined with & (AND). This is equivalent to `dplyr::filter()`.

**Usage**

```
DataFrame_filter(...)
```

**Arguments**

... Polars expressions which will evaluate to a boolean.

**Details**

Rows where the condition returns NA are dropped.

**Value**

A DataFrame with only the rows where the conditions are TRUE.

**Examples**

```
df = pl$DataFrame(iris)  
  
df$filter(pl$col("Sepal.Length") > 5)  
  
# This is equivalent to  
# df$filter(pl$col("Sepal.Length") > 5 & pl$col("Petal.Width") < 1)  
df$filter(pl$col("Sepal.Length") > 5, pl$col("Petal.Width") < 1)  
  
# rows where condition is NA are dropped  
iris2 = iris  
iris2[c(1, 3, 5), "Species"] = NA  
df = pl$DataFrame(iris2)  
  
df$filter(pl$col("Species") == "setosa")
```

DataFrame\_first      *Get the first row of the DataFrame.*

---

**Description**

Get the first row of the DataFrame.

**Usage**

```
DataFrame_first()
```

**Value**

A DataFrame with one row.

**Examples**

```
pl$DataFrame(mtcars)$first()
```

---

DataFrame\_gather\_every      *Take every nth row in the DataFrame*

---

**Description**

Take every nth row in the DataFrame

**Usage**

```
DataFrame_gather_every(n, offset = 0)
```

**Arguments**

n	Gather every n-th row.
offset	Starting index.

**Value**

A DataFrame

**Examples**

```
df = pl$DataFrame(a = 1:4, b = 5:8)
df$gather_every(2)

df$gather_every(2, offset = 1)
```

---

DataFrame\_get\_column *Get column (as one Series)*

---

### Description

Extract a DataFrame column as a Polars series.

### Usage

```
DataFrame_get_column(name)
```

### Arguments

name                      Name of the column to extract.

### Value

Series

### Examples

```
df = pl.DataFrame(iris[1:2, ])
df.get_column("Species")
```

---

DataFrame\_get\_columns *Get the DataFrame as a List of Series*

---

### Description

Get the DataFrame as a List of Series

### Usage

```
DataFrame_get_columns()
```

### Value

A list of [Series](#)

### See Also

- [DataFrame.to\\_list\(\)](#): Similar to this method but returns a list of vectors instead of [Series](#).

**Examples**

```
df = pl$DataFrame(foo = 1L:3L, bar = 4L:6L)
df$get_columns()
```

```
df = pl$DataFrame(
  a = 1:4,
  b = c(0.5, 4, 10, 13),
  c = c(TRUE, TRUE, FALSE, TRUE)
)
df$get_columns()
```

---

DataFrame_glimpse	<i>Show a dense preview of the DataFrame</i>
-------------------	--

---

**Description**

The formatting shows one line per column so that wide DataFrames display cleanly. Each line shows the column name, the data type, and the first few values.

**Usage**

```
DataFrame_glimpse(
  ...,
  max_items_per_column = 10,
  max_colname_length = 50,
  return_as_string = FALSE
)
```

**Arguments**

... Ignored.

max\_items\_per\_column  
Maximum number of items to show per column.

max\_colname\_length  
Maximum length of the displayed column names. Values that exceed this value are truncated with a trailing ellipsis.

return\_as\_string  
Logical (default FALSE). If TRUE, return the output as a string.

**Value**

DataFrame

**Examples**

```
pl$DataFrame(iris)$glimpse()
```

---

DataFrame_group_by	<i>Group a DataFrame</i>
--------------------	--------------------------

---

### Description

This doesn't modify the data but only stores information about the group structure. This structure can then be used by several functions (`$agg()`, `$filter()`, etc.).

### Usage

```
DataFrame_group_by(..., maintain_order = polars_options()$maintain_order)
```

### Arguments

...	Column(s) to group by. Accepts <a href="#">expression</a> input. Characters are parsed as column names.
maintain_order	Ensure that the order of the groups is consistent with the input data. This is slower than a default group by. Setting this to TRUE blocks the possibility to run on the streaming engine. The default value can be changed with <code>options(polars.maintain_order = TRUE)</code> .

### Details

Within each group, the order of the rows is always preserved, regardless of the `maintain_order` argument.

### Value

[GroupBy](#) (a DataFrame with special groupby methods like `$agg()`)

### See Also

- [<DataFrame>\\$partition\\_by\(\)](#)

### Examples

```
df = pl$DataFrame(
  a = c("a", "b", "a", "b", "c"),
  b = c(1, 2, 1, 3, 3),
  c = c(5, 4, 3, 2, 1)
)

df$group_by("a")$agg(pl$col("b")$sum())

# Set `maintain_order = TRUE` to ensure the order of the groups is consistent with the input.
df$group_by("a", maintain_order = TRUE)$agg(pl$col("c"))

# Group by multiple columns by passing a list of column names.
```

```
df$group_by(c("a", "b"))$agg(pl$max("c"))

# Or pass some arguments to group by multiple columns in the same way.
# Expressions are also accepted.
df$group_by("a", pl$col("b") %% 2)$agg(
  pl$col("c")$mean()
)

# The columns will be renamed to the argument names.
df$group_by(d = "a", e = pl$col("b") %% 2)$agg(
  pl$col("c")$mean()
)
```

---

DataFrame\_group\_by\_dynamic

*Group based on a date/time or integer column*

---

### Description

If you have a time series  $\langle t_0, t_1, \dots, t_n \rangle$ , then by default the windows created will be:

- $(t_0 - \text{period}, t_0]$
- $(t_1 - \text{period}, t_1]$
- ...
- $(t_n - \text{period}, t_n]$

whereas if you pass a non-default offset, then the windows will be:

- $(t_0 + \text{offset}, t_0 + \text{offset} + \text{period}]$
- $(t_1 + \text{offset}, t_1 + \text{offset} + \text{period}]$
- ...
- $(t_n + \text{offset}, t_n + \text{offset} + \text{period}]$

### Usage

```
DataFrame_group_by_dynamic(
  index_column,
  ...,
  every,
  period = NULL,
  offset = NULL,
  include_boundaries = FALSE,
  closed = "left",
  label = "left",
  group_by = NULL,
  start_by = "window"
)
```



**Arguments**

<code>index_column</code>	Column used to group based on the time window. Often of type Date/Datetime. This column must be sorted in ascending order (or, if by is specified, then it must be sorted in ascending order within each group). In case of a rolling group by on indices, dtype needs to be either Int32 or Int64. Note that Int32 gets temporarily cast to Int64, so if performance matters use an Int64 column.
<code>...</code>	Ignored.
<code>every</code>	Interval of the window.
<code>period</code>	A character representing the length of the window, must be non-negative. See the Polars <code>duration string language</code> section for details.
<code>offset</code>	A character representing the offset of the window, or NULL (default). If NULL, <code>-period</code> is used. See the Polars <code>duration string language</code> section for details.
<code>include_boundaries</code>	Add two columns <code>"_lower_boundary"</code> and <code>"_upper_boundary"</code> columns that show the boundaries of the window. This will impact performance because it's harder to parallelize.
<code>closed</code>	Define which sides of the temporal interval are closed (inclusive). This can be either <code>"left"</code> , <code>"right"</code> , <code>"both"</code> or <code>"none"</code> .
<code>label</code>	Define which label to use for the window: <ul style="list-style-type: none"> <li><code>"left"</code>: lower boundary of the window</li> <li><code>"right"</code>: upper boundary of the window</li> <li><code>"datapoint"</code>: the first value of the index column in the given window. If you don't need the label to be at one of the boundaries, choose this option for maximum performance.</li> </ul>
<code>group_by</code>	Also group by this column/these columns.
<code>start_by</code>	The strategy to determine the start of the first window by: <ul style="list-style-type: none"> <li><code>"window"</code>: start by taking the earliest timestamp, truncating it with <code>every</code>, and then adding <code>offset</code>. Note that weekly windows start on Monday.</li> <li><code>"datapoint"</code>: start from the first encountered data point.</li> <li>a day of the week (only takes effect if <code>every</code> contains <code>"w"</code>): <code>"monday"</code> starts the window on the Monday before the first data point, etc.</li> </ul>

**Details**

In case of a rolling operation on an integer column, the windows are defined by:

- `"1i" # length 1`
- `"10i" # length 10`

**Value**

A [GroupBy](#) object

**See Also**

- [<DataFrame>\\$rolling\(\)](#)

**Examples**

```
df = pl$DataFrame(
  time = pl$datetime_range(
    start = strptime("2021-12-16 00:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC"),
    end = strptime("2021-12-16 03:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC"),
    interval = "30m"
  ),
  n = 0:6
)

# get the sum in the following hour relative to the "time" column
df$group_by_dynamic("time", every = "1h")$agg(
  vals = pl$col("n"),
  sum = pl$col("n")$sum()
)

# using "include_boundaries = TRUE" is helpful to see the period considered
df$group_by_dynamic("time", every = "1h", include_boundaries = TRUE)$agg(
  vals = pl$col("n")
)

# in the example above, the values didn't include the one *exactly* 1h after
# the start because "closed = 'left'" by default.
# Changing it to "right" includes values that are exactly 1h after. Note that
# the value at 00:00:00 now becomes included in the interval [23:00:00 - 00:00:00],
# even if this interval wasn't there originally
df$group_by_dynamic("time", every = "1h", closed = "right")$agg(
  vals = pl$col("n")
)

# To keep both boundaries, we use "closed = 'both'". Some values now belong to
# several groups:
df$group_by_dynamic("time", every = "1h", closed = "both")$agg(
  vals = pl$col("n")
)

# Dynamic group bys can also be combined with grouping on normal keys
df = df$with_columns(
  groups = as_polars_series(c("a", "a", "a", "b", "b", "a", "a"))
)
df

df$group_by_dynamic(
  "time",
  every = "1h",
  closed = "both",
  group_by = "groups",
  include_boundaries = TRUE
)$agg(pl$col("n"))
```

```
# We can also create a dynamic group by based on an index column
df = pl$LazyFrame(
  idx = 0:5,
  A = c("A", "A", "B", "B", "B", "C")
)$with_columns(pl$col("idx")$set_sorted())
df

df$group_by_dynamic(
  "idx",
  every = "2i",
  period = "3i",
  include_boundaries = TRUE,
  closed = "right"
)$agg(A_agg_list = pl$col("A"))
```

---

DataFrame_head	<i>Get the first n rows.</i>
----------------	------------------------------

---

## Description

Get the first n rows.

## Usage

```
DataFrame_head(n = 5L)
```

```
DataFrame_limit(n = 5L)
```

## Arguments

n                      Number of rows to return. If a negative value is passed, return all rows except the last `abs(n)`.

## Details

`$limit()` is an alias for `$head()`.

## Value

A [DataFrame](#)

## Examples

```
df = pl$DataFrame(foo = 1:5, bar = 6:10, ham = letters[1:5])

df$head(3)

# Pass a negative value to get all rows except the last `abs(n)`.
df$head(-3)
```

---

DataFrame_item	<i>Return the element at the given row/column.</i>
----------------	--

---

### Description

If row and column location are not specified, the [DataFrame](#) must have dimensions (1, 1).

### Usage

```
DataFrame_item(row = NULL, column = NULL)
```

### Arguments

row	Optional row index (0-indexed).
column	Optional column index (0-indexed) or name.

### Value

A value of length 1

### Examples

```
df = pl$DataFrame(a = c(1, 2, 3), b = c(4, 5, 6))
df$select((pl$col("a") * pl$col("b"))$sum())$item()
df$item(1, 1)
df$item(2, "b")
```

---

DataFrame_join	<i>Join DataFrames</i>
----------------	------------------------

---

### Description

This function can do both mutating joins (adding columns based on matching observations, for example with `how = "left"`) and filtering joins (keeping observations based on matching observations, for example with `how = "inner"`).

**Usage**

```
DataFrame_join(
  other,
  on = NULL,
  how = "inner",
  ...,
  left_on = NULL,
  right_on = NULL,
  suffix = "_right",
  validate = "m:m",
  join_nulls = FALSE,
  allow_parallel = TRUE,
  force_parallel = FALSE,
  coalesce = NULL
)
```

**Arguments**

<code>other</code>	DataFrame to join with.
<code>on</code>	Either a vector of column names or a list of expressions and/or strings. Use <code>left_on</code> and <code>right_on</code> if the column names to match on are different between the two DataFrames.
<code>how</code>	One of the following methods: "inner", "left", "right", "full", "semi", "anti", "cross".
<code>...</code>	Ignored.
<code>left_on, right_on</code>	Same as <code>on</code> but only for the left or the right DataFrame. They must have the same length.
<code>suffix</code>	Suffix to add to duplicated column names.
<code>validate</code>	Checks if join is of specified type: <ul style="list-style-type: none"> <li>• "m:m" (default): many-to-many, doesn't perform any checks;</li> <li>• "1:1": one-to-one, check if join keys are unique in both left and right datasets;</li> <li>• "1:m": one-to-many, check if join keys are unique in left dataset</li> <li>• "m:1": many-to-one, check if join keys are unique in right dataset</li> </ul> <p>Note that this is currently not supported by the streaming engine, and is only supported when joining by single columns.</p>
<code>join_nulls</code>	Join on null values. By default null values will never produce matches.
<code>allow_parallel</code>	Allow the physical plan to optionally evaluate the computation of both DataFrames up to the join in parallel.
<code>force_parallel</code>	Force the physical plan to evaluate the computation of both DataFrames up to the join in parallel.
<code>coalesce</code>	Coalescing behavior (merging of join columns). <ul style="list-style-type: none"> <li>• NULL: join specific.</li> <li>• TRUE: Always coalesce join columns.</li> <li>• FALSE: Never coalesce join columns.</li> </ul>

**Value**

DataFrame

**Examples**

```
# inner join by default
df1 = pl$DataFrame(list(key = 1:3, payload = c("f", "i", NA)))
df2 = pl$DataFrame(list(key = c(3L, 4L, 5L, NA_integer_)))
df1$join(other = df2, on = "key")

# cross join
df1 = pl$DataFrame(x = letters[1:3])
df2 = pl$DataFrame(y = 1:4)
df1$join(other = df2, how = "cross")
```

---

DataFrame\_join\_asof    *Perform joins on nearest keys*

---

**Description**

This is similar to a left-join except that we match on nearest key rather than equal keys.

**Usage**

```
DataFrame_join_asof(
  other,
  ...,
  left_on = NULL,
  right_on = NULL,
  on = NULL,
  by_left = NULL,
  by_right = NULL,
  by = NULL,
  strategy = c("backward", "forward", "nearest"),
  suffix = "_right",
  tolerance = NULL,
  allow_parallel = TRUE,
  force_parallel = FALSE,
  coalesce = TRUE
)
```

**Arguments**

other	DataFrame or LazyFrame
...	Not used, blocks use of further positional arguments
left_on, right_on	Same as on but only for the left or the right DataFrame. They must have the same length.

<code>on</code>	Either a vector of column names or a list of expressions and/or strings. Use <code>left_on</code> and <code>right_on</code> if the column names to match on are different between the two DataFrames.
<code>by_left, by_right</code>	Same as <code>by</code> but only for the left or the right table. They must have the same length.
<code>by</code>	Join on these columns before performing asof join. Either a vector of column names or a list of expressions and/or strings. Use <code>left_by</code> and <code>right_by</code> if the column names to match on are different between the two tables.
<code>strategy</code>	Strategy for where to find match: <ul style="list-style-type: none"> <li>• "backward" (default): search for the last row in the right table whose on key is less than or equal to the left key.</li> <li>• "forward": search for the first row in the right table whose on key is greater than or equal to the left key.</li> <li>• "nearest": search for the last row in the right table whose value is nearest to the left key. String keys are not currently supported for a nearest search.</li> </ul>
<code>suffix</code>	Suffix to add to duplicated column names.
<code>tolerance</code>	Numeric tolerance. By setting this the join will only be done if the near keys are within this distance. If an asof join is done on columns of dtype "Date", "Datetime", "Duration" or "Time", use the Polars duration string language. About the language, see the <code>Polars duration string language</code> section for details. There may be a circumstance where R types are not sufficient to express a numeric tolerance. In that case, you can use the expression syntax like <code>tolerance = pl\$lit(42)\$cast(pl\$UInt64)</code>
<code>allow_parallel</code>	Allow the physical plan to optionally evaluate the computation of both DataFrames up to the join in parallel.
<code>force_parallel</code>	Force the physical plan to evaluate the computation of both DataFrames up to the join in parallel.
<code>coalesce</code>	Coalescing behavior (merging of <code>on</code> / <code>left_on</code> / <code>right_on</code> columns): <ul style="list-style-type: none"> <li>• TRUE: Always coalesce join columns;</li> <li>• FALSE: Never coalesce join columns. Note that joining on any other expressions than <code>col</code> will turn off coalescing.</li> </ul>

### Details

Both tables (DataFrames or LazyFrames) must be sorted by the `asof_join` key.

### Value

New joined DataFrame

### Polars duration string language

Polars duration string language is a simple representation of durations. It is used in many Polars functions that accept durations.

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)
- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds

By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".

## Examples

```
# create two DataFrames to join asof
gdp = pl$DataFrame(
  date = as.Date(c("2015-1-1", "2016-1-1", "2017-5-1", "2018-1-1", "2019-1-1")),
  gdp = c(4321, 4164, 4411, 4566, 4696),
  group = c("b", "a", "a", "b", "b")
)

pop = pl$DataFrame(
  date = as.Date(c("2016-5-12", "2017-5-12", "2018-5-12", "2019-5-12")),
  population = c(82.19, 82.66, 83.12, 83.52),
  group = c("b", "b", "a", "a")
)

# optional make sure tables are already sorted with "on" join-key
gdp = gdp$sort("date")
pop = pop$sort("date")

# Left-join_asof DataFrame pop with gdp on "date"
# Look backward in gdp to find closest matching date
pop$join_asof(gdp, on = "date", strategy = "backward")

# ... and forward
pop$join_asof(gdp, on = "date", strategy = "forward")

# join by a group: "only look within within group"
pop$join_asof(gdp, on = "date", by = "group", strategy = "backward")

# only look 2 weeks and 2 days back
pop$join_asof(gdp, on = "date", strategy = "backward", tolerance = "2w2d")
```



```
# only look 11 days back (numeric tolerance depends on polars type, <date> is in days)
pop$join_asof(gdp, on = "date", strategy = "backward", tolerance = 11)
```

---

DataFrame\_join\_where *Perform a join based on one or multiple (in)equality predicates*

---

### Description

This performs an inner join, so only rows where all predicates are true are included in the result, and a row from either DataFrame may be included multiple times in the result.

Note that the row order of the input DataFrames is not preserved.

### Usage

```
DataFrame_join_where(other, ..., suffix = "_right")
```

### Arguments

other	DataFrame to join with.
...	(In)Equality condition to join the two tables on. When a column name occurs in both tables, the proper suffix must be applied in the predicate. For example, if both tables have a column "x" that you want to use in the conditions, you must refer to the column of the right table as "x<suffix>".
suffix	Suffix to append to columns with a duplicate name.

### Value

A DataFrame

### Examples

```
east = pl$DataFrame(
  id = c(100, 101, 102),
  dur = c(120, 140, 160),
  rev = c(12, 14, 16),
  cores = c(2, 8, 4)
)

west = pl$DataFrame(
  t_id = c(404, 498, 676, 742),
  time = c(90, 130, 150, 170),
  cost = c(9, 13, 15, 16),
  cores = c(4, 2, 1, 4)
)

east$join_where(
  west,
```

```
pl$col("dur") < pl$col("time"),  
pl$col("rev") < pl$col("cost")  
)
```

---

DataFrame_last	<i>Get the last row of the DataFrame.</i>
----------------	---

---

**Description**

Get the last row of the DataFrame.

**Usage**

```
DataFrame_last()
```

**Value**

A DataFrame with one row.

**Examples**

```
pl$DataFrame(mtcars)$last()
```

---

DataFrame_lazy	<i>Convert an existing DataFrame to a LazyFrame</i>
----------------	---

---

**Description**

Start a new lazy query from a DataFrame.

**Usage**

```
DataFrame_lazy()
```

**Value**

A LazyFrame

**Examples**

```
pl$DataFrame(iris)$lazy()
```

---

DataFrame_max	<i>Max</i>
---------------	------------

---

**Description**

Aggregate the columns in the DataFrame to their maximum value.

**Usage**

```
DataFrame_max()
```

**Value**

A DataFrame with one row.

**Examples**

```
p1$DataFrame(mtcars)$max()
```

---

DataFrame_mean	<i>Mean</i>
----------------	-------------

---

**Description**

Aggregate the columns in the DataFrame to their mean value.

**Usage**

```
DataFrame_mean()
```

**Value**

A DataFrame with one row.

**Examples**

```
p1$DataFrame(mtcars)$mean()
```

---

DataFrame_median	<i>Median</i>
------------------	---------------

---

**Description**

Aggregate the columns in the DataFrame to their median value.

**Usage**

```
DataFrame_median()
```

**Value**

A DataFrame with one row.

**Examples**

```
p1$DataFrame(mtcars)$median()
```

---

DataFrame_min	<i>Min</i>
---------------	------------

---

**Description**

Aggregate the columns in the DataFrame to their minimum value.

**Usage**

```
DataFrame_min()
```

**Value**

A DataFrame with one row.

**Examples**

```
p1$DataFrame(mtcars)$min()
```

---

DataFrame\_null\_count    *Count null values*

---

**Description**

Create a new DataFrame that shows the null (which correspond to NA in R) counts per column.

**Usage**

```
DataFrame_null_count()
```

**Format**

function

**Value**

DataFrame

**Examples**

```
x = mtcars
x[1, 2:3] = NA
pl$DataFrame(x)$null_count()
```

---

DataFrame\_n\_chunks    *Number of chunks of the Series in a DataFrame*

---

**Description**

Number of chunks (memory allocations) for all or first Series in a DataFrame.

**Usage**

```
DataFrame_n_chunks(strategy = "first")
```

**Arguments**

strategy    Either "all" or "first". "first" only returns chunks for the first Series.

## Details

A DataFrame is a vector of Series. Each Series in rust-polars is a wrapper around a ChunkedArray, which is like a virtual contiguous vector physically backed by an ordered set of chunks. Each chunk of values has a contiguous memory layout and is an arrow array. Arrow arrays are a fast, thread-safe and cross-platform memory layout.

In R, combining with `c()` or `rbind()` requires immediate vector re-allocation to place vector values in contiguous memory. This is slow and memory consuming, and it is why repeatedly appending to a vector in R is discouraged.

In polars, when we concatenate or append to Series or DataFrame, the re-allocation can be avoided or delayed by simply appending chunks to each individual Series. However, if chunks become many and small or are misaligned across Series, this can hurt the performance of subsequent operations.

Most places in the polars api where chunking could occur, the user have to typically actively opt-out by setting an argument `rechunk = FALSE`.

## Value

A real vector of chunk counts per Series.

## See Also

[<DataFrame>\\$rechunk\(\)](#)

## Examples

```
# create DataFrame with misaligned chunks
df = pl$concat(
  1:10, # single chunk
  pl$concat(1:5, 1:5, rechunk = FALSE, how = "vertical")$rename("b"), # two chunks
  how = "horizontal"
)
df
df$n_chunks()

# rechunk a chunked DataFrame
df$rechunk()$n_chunks()

# rechunk is not an in-place operation
df$n_chunks()

# The following toy example emulates the Series "chunkyness" in R. Here it a
# S3-classed list with same type of vectors and where have all relevant S3
# generics implemented to make behave as if it was a regular vector.
"+.chunked_vector" = \(x, y) structure(list(unlist(x) + unlist(y)), class = "chunked_vector")
print.chunked_vector = \(x, ...) print(unlist(x), ...)
c.chunked_vector = \(...) {
  structure(do.call(c, lapply(list(...), unclass)), class = "chunked_vector")
}
rechunk = \(x) structure(unlist(x), class = "chunked_vector")
x = structure(list(1:4, 5L), class = "chunked_vector")
x
```

```
x + 5:1
lapply(x, tracemem) # trace chunks to verify no re-allocation
z = c(x, x)
z # looks like a plain vector
lapply(z, tracemem) # mem allocation in z are the same from x
str(z)
z = rechunk(z)
str(z)
```

---

DataFrame\_partition\_by

*Split a DataFrame into multiple DataFrames*


---

## Description

Similar to [\\$group\\_by\(\)](#). Group by the given columns and return the groups as separate [DataFrames](#). It is useful to use this in combination with functions like [lapply\(\)](#) or `purrr::map()`.

## Usage

```
DataFrame_partition_by(
  ...,
  maintain_order = TRUE,
  include_key = TRUE,
  as_nested_list = FALSE
)
```

## Arguments

...	Characters of column names to group by. Passed to <a href="#">pl\$col()</a> .
maintain_order	If TRUE, ensure that the order of the groups is consistent with the input data. This is slower than a default partition by operation.
include_key	If TRUE, include the columns used to partition the DataFrame in the output.
as_nested_list	This affects the format of the output. If FALSE (default), the output is a flat <a href="#">list</a> of <a href="#">DataFrames</a> . IF TRUE and one of the <code>maintain_order</code> or <code>include_key</code> argument is TRUE, then each element of the output has two children: key and data. See the examples for more details.

## Value

A list of [DataFrames](#). See the examples for details.

## See Also

- [<DataFrame>\\$group\\_by\(\)](#)

**Examples**

```

df = pl$DataFrame(
  a = c("a", "b", "a", "b", "c"),
  b = c(1, 2, 1, 3, 3),
  c = c(5, 4, 3, 2, 1)
)
df

# Pass a single column name to partition by that column.
df$partition_by("a")

# Partition by multiple columns.
df$partition_by("a", "b")

# Partition by column data type
df$partition_by(pl$String)

# If `as_nested_list = TRUE`, the output is a list whose elements have a `key` and a `data` field.
# The `key` is a named list of the key values, and the `data` is the DataFrame.
df$partition_by("a", "b", as_nested_list = TRUE)

# `as_nested_list = TRUE` should be used with `maintain_order = TRUE` or `include_key = TRUE`.
tryCatch(
  df$partition_by("a", "b", maintain_order = FALSE, include_key = FALSE, as_nested_list = TRUE),
  warning = function(w) w
)

# Example of using with lapply(), and printing the key and the data summary
df$partition_by("a", "b", maintain_order = FALSE, as_nested_list = TRUE) |>
  lapply(\(x) {
    sprintf("\nThe key value of `a` is %s and the key value of `b` is %s\n", x$key$a, x$key$b) |>
      cat()
    x$data$drop(names(x$key))$describe() |>
      print()
    invisible(NULL)
  }) |>
  invisible()

```

---

 DataFrame\_pivot

*Pivot data from long to wide*


---

**Description**

Pivot data from long to wide

**Usage**

```

DataFrame_pivot(
  on,

```



```

    ...,
    index,
    values,
    aggregate_function = NULL,
    maintain_order = TRUE,
    sort_columns = FALSE,
    separator = "_"
  )

```

### Arguments

on	Name of the column(s) whose values will be used as the header of the output DataFrame.
...	Not used.
index	One or multiple keys to group by.
values	Column values to aggregate. Can be multiple columns if the on arguments contains multiple columns as well.
aggregate_function	One of: <ul style="list-style-type: none"> <li>• string indicating the expressions to aggregate with, such as 'first', 'sum', 'max', 'min', 'mean', 'median', 'last', 'count'),</li> <li>• an Expr e.g. pl\$element()\$sum()</li> </ul>
maintain_order	Sort the grouped keys so that the output order is predictable.
sort_columns	Sort the transposed columns by name. Default is by order of discovery.
separator	Used as separator/delimiter in generated column names.

### Value

DataFrame

### Examples

```

df = pl$DataFrame(
  foo = c("one", "one", "one", "two", "two", "two"),
  bar = c("A", "B", "C", "A", "B", "C"),
  baz = c(1, 2, 3, 4, 5, 6)
)
df

df$pivot(
  values = "baz", index = "foo", on = "bar"
)

# Run an expression as aggregation function
df = pl$DataFrame(
  col1 = c("a", "a", "a", "b", "b", "b"),
  col2 = c("x", "x", "x", "x", "y", "y"),
  col3 = c(6, 7, 3, 2, 5, 7)
)

```

```
)
df

df$pivot(
  index = "col1",
  on = "col2",
  values = "col3",
  aggregate_function = pl$element()$tanh()$mean()
)
```

---

DataFrame\_quantile      *Quantile*

---

### Description

Aggregate the columns in the DataFrame to a unique quantile value. Use `$describe()` to specify several quantiles.

### Usage

```
DataFrame_quantile(quantile, interpolation = "nearest")
```

### Arguments

`quantile`      Numeric of length 1 between 0 and 1.  
`interpolation`      One of "nearest", "higher", "lower", "midpoint", or "linear".

### Value

DataFrame

### Examples

```
pl$DataFrame(mtcars)$quantile(.4)
```

---

DataFrame\_rechunk      *Rechunk a DataFrame*

---

### Description

Rechunking re-allocates any "chunked" memory allocations to speed-up e.g. vectorized operations.

### Usage

```
DataFrame_rechunk()
```

## Details

A DataFrame is a vector of Series. Each Series in rust-polars is a wrapper around a ChunkedArray, which is like a virtual contiguous vector physically backed by an ordered set of chunks. Each chunk of values has a contiguous memory layout and is an arrow array. Arrow arrays are a fast, thread-safe and cross-platform memory layout.

In R, combining with `c()` or `rbind()` requires immediate vector re-allocation to place vector values in contiguous memory. This is slow and memory consuming, and it is why repeatedly appending to a vector in R is discouraged.

In polars, when we concatenate or append to Series or DataFrame, the re-allocation can be avoided or delayed by simply appending chunks to each individual Series. However, if chunks become many and small or are misaligned across Series, this can hurt the performance of subsequent operations.

Most places in the polars api where chunking could occur, the user have to typically actively opt-out by setting an argument `rechunk = FALSE`.

## Value

A DataFrame

## See Also

[<DataFrame>\\$n\\_chunks\(\)](#)

## Examples

```
# create DataFrame with misaligned chunks
df = pl$concat(
  1:10, # single chunk
  pl$concat(1:5, 1:5, rechunk = FALSE, how = "vertical")$rename("b"), # two chunks
  how = "horizontal"
)
df
df$n_chunks()

# rechunk a chunked DataFrame
df$rechunk()$n_chunks()

# rechunk is not an in-place operation
df$n_chunks()

# The following toy example emulates the Series "chunkyness" in R. Here it a
# S3-classed list with same type of vectors and where have all relevant S3
# generics implemented to make behave as if it was a regular vector.
"+.chunked_vector" = \(x, y) structure(list(unlist(x) + unlist(y)), class = "chunked_vector")
print.chunked_vector = \(x, ...) print(unlist(x), ...)
c.chunked_vector = \(...) {
  structure(do.call(c, lapply(list(...), unclass)), class = "chunked_vector")
}
rechunk = \(x) structure(unlist(x), class = "chunked_vector")
x = structure(list(1:4, 5L), class = "chunked_vector")
x
```

```
x + 5:1
lapply(x, tracemem) # trace chunks to verify no re-allocation
z = c(x, x)
z # looks like a plain vector
lapply(z, tracemem) # mem allocation in z are the same from x
str(z)
z = rechunk(z)
str(z)
```

---

DataFrame_rename	<i>Rename column names of a DataFrame</i>
------------------	---

---

## Description

Rename column names of a DataFrame

## Usage

```
DataFrame_rename(...)
```

## Arguments

... One of the following:

- Key value pairs that map from old name to new name, like `old_name = "new_name"`.
- As above but with params wrapped in a list
- An R function that takes the old names character vector as input and returns the new names character vector.

## Details

If existing names are swapped (e.g. A points to B and B points to A), polars will block projection and predicate pushdowns at this node.

## Value

[DataFrame](#)

## Examples

```
df = pl$DataFrame(
  foo = 1:3,
  bar = 6:8,
  ham = letters[1:3]
)

df$rename(foo = "apple")
```

```
df$rename(
  \(column_name) paste0("c", substr(column_name, 2, 100))
)
```

---

DataFrame\_reverse      *Reverse*

---

### Description

Reverse the DataFrame (the last row becomes the first one, etc.).

### Usage

```
DataFrame_reverse()
```

### Value

DataFrame

### Examples

```
pl$DataFrame(mtcars)$reverse()
```

---

DataFrame\_rolling      *Create rolling groups based on a date/time or integer column*

---

### Description

If you have a time series  $\langle t_0, t_1, \dots, t_n \rangle$ , then by default the windows created will be:

- $(t_0 - \text{period}, t_0]$
- $(t_1 - \text{period}, t_1]$
- ...
- $(t_n - \text{period}, t_n]$

whereas if you pass a non-default offset, then the windows will be:

- $(t_0 + \text{offset}, t_0 + \text{offset} + \text{period}]$
- $(t_1 + \text{offset}, t_1 + \text{offset} + \text{period}]$
- ...
- $(t_n + \text{offset}, t_n + \text{offset} + \text{period}]$

**Usage**

```

DataFrame_rolling(
    index_column,
    ...,
    period,
    offset = NULL,
    closed = "right",
    group_by = NULL
)

```

**Arguments**

<code>index_column</code>	Column used to group based on the time window. Often of type Date/Datetime. This column must be sorted in ascending order (or, if <code>by</code> is specified, then it must be sorted in ascending order within each group). In case of a rolling group by on indices, <code>dtype</code> needs to be either <code>Int32</code> or <code>Int64</code> . Note that <code>Int32</code> gets temporarily cast to <code>Int64</code> , so if performance matters use an <code>Int64</code> column.
<code>...</code>	Ignored.
<code>period</code>	A character representing the length of the window, must be non-negative. See the Polars duration string language section for details.
<code>offset</code>	A character representing the offset of the window, or <code>NULL</code> (default). If <code>NULL</code> , <code>-period</code> is used. See the Polars duration string language section for details.
<code>closed</code>	Define which sides of the temporal interval are closed (inclusive). This can be either <code>"left"</code> , <code>"right"</code> , <code>"both"</code> or <code>"none"</code> .
<code>group_by</code>	Also group by this column/these columns.

**Details**

In case of a rolling operation on an integer column, the windows are defined by:

- `"1i"` # length 1
- `"10i"` # length 10

**Value**

A [RollingGroupBy](#) object

**Polars duration string language**

Polars duration string language is a simple representation of durations. It is used in many Polars functions that accept durations.

It has the following format:

- `1ns` (1 nanosecond)
- `1us` (1 microsecond)
- `1ms` (1 millisecond)

- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)
- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds

By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".

### See Also

- `<DataFrame>$group_by_dynamic()`

### Examples

```
date = c(
  "2020-01-01 13:45:48",
  "2020-01-01 16:42:13",
  "2020-01-01 16:45:09",
  "2020-01-02 18:12:48",
  "2020-01-03 19:45:32",
  "2020-01-08 23:16:43"
)
df = pl$DataFrame(dt = date, a = c(3, 7, 5, 9, 2, 1))$with_columns(
  pl$col("dt")$str$strptime(pl$Datetime())$set_sorted()
)

df$rolling(index_column = "dt", period = "2d")$agg(
  sum_a = pl$sum("a"),
  min_a = pl$min("a"),
  max_a = pl$max("a")
)
```

---

DataFrame\_sample

*Take a sample of rows from a DataFrame*

---

### Description

Take a sample of rows from a DataFrame

**Usage**

```

DataFrame_sample(
  n = NULL,
  ...,
  fraction = NULL,
  with_replacement = FALSE,
  shuffle = FALSE,
  seed = NULL
)

```

**Arguments**

n	Number of rows to return. Cannot be used with fraction.
...	Ignored.
fraction	Fraction of rows to return. Cannot be used with n. Can be larger than 1 if with_replacement is TRUE.
with_replacement	Allow values to be sampled more than once.
shuffle	If TRUE, the order of the sampled rows will be shuffled. If FALSE (default), the order of the returned rows will be neither stable nor fully random.
seed	Seed for the random number generator. If set to NULL (default), a random seed is generated for each sample operation.

**Value**

DataFrame

**Examples**

```

df = pl$DataFrame(iris)
df$sample(n = 20)
df$sample(fraction = 0.1)

```

---

DataFrame_select	<i>Select and modify columns of a DataFrame</i>
------------------	---

---

**Description**

Similar to `dplyr::mutate()`. However, it discards unmentioned columns (like `.` in `data.table`).

**Usage**

```

DataFrame_select(...)

```



**Arguments**

... Columns to keep. Those can be expressions (e.g `pl$col("a")`), column names (e.g `"a"`), or list containing expressions or column names (e.g `list(pl$col("a"))`).

**Value**

DataFrame

**Examples**

```
pl$DataFrame(iris)$select(  
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),  
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")  
)
```

---

DataFrame\_select\_seq *Select and modify columns of a DataFrame*

---

**Description**

Similar to `dplyr::mutate()`. However, it discards unmentioned columns (like `.` in `data.table`).

This will run all expression sequentially instead of in parallel. Use this when the work per expression is cheap. Otherwise, `$select()` should be preferred.

**Usage**

```
DataFrame_select_seq(...)
```

**Arguments**

... Columns to keep. Those can be expressions (e.g `pl$col("a")`), column names (e.g `"a"`), or list containing expressions or column names (e.g `list(pl$col("a"))`).

**Value**

DataFrame

**Examples**

```
pl$DataFrame(iris)$select_seq(  
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),  
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")  
)
```

---

DataFrame_shift	<i>Shift a DataFrame</i>
-----------------	--------------------------

---

**Description**

Shift the values by a given period. If the period (n) is positive, then n rows will be inserted at the top of the DataFrame and the last n rows will be discarded. Vice-versa if the period is negative. In the end, the total number of rows of the DataFrame doesn't change.

**Usage**

```
DataFrame_shift(n = 1, fill_value = NULL)
```

**Arguments**

n	Number of indices to shift forward. If a negative value is passed, values are shifted in the opposite direction instead.
fill_value	Fill the resulting null values with this value. Accepts expression input. Non-expression inputs are parsed as literals.

**Value**

DataFrame

**Examples**

```
df = pl$DataFrame(a = 1:4, b = 5:8)

df$shift(2)

df$shift(-2)

df$shift(-2, fill_value = 100)
```

---

DataFrame_slice	<i>Slice</i>
-----------------	--------------

---

**Description**

Get a slice of the DataFrame.

**Usage**

```
DataFrame_slice(offset, length = NULL)
```

**Arguments**

offset	Start index, can be a negative value. This is 0-indexed, so offset = 1 doesn't include the first row.
length	Length of the slice. If NULL (default), all rows starting at the offset will be selected.

**Value**

DataFrame

**Examples**

```
# skip the first 2 rows and take the 4 following rows
p1$DataFrame(mtcars)$slice(2, 4)

# this is equivalent to:
mtcars[3:6, ]
```

---

DataFrame_sort	<i>Sort a DataFrame</i>
----------------	-------------------------

---

**Description**

Sort a DataFrame

**Usage**

```
DataFrame_sort(
  by,
  ...,
  descending = FALSE,
  nulls_last = FALSE,
  maintain_order = FALSE
)
```

**Arguments**

by	Column(s) to sort by. Can be character vector of column names, a list of Expr(s) or a list with a mix of Expr(s) and column names.
...	More columns to sort by as above but provided one Expr per argument.
descending	Logical. Sort in descending order (default is FALSE). This must be either of length 1 or a logical vector of the same length as the number of Expr(s) specified in by and ...
nulls_last	A logical or logical vector of the same length as the number of columns. If TRUE, place null values last instead of first.
maintain_order	Whether the order should be maintained if elements are equal. If TRUE, streaming is not possible and performance might be worse since this requires a stable search.

**Value**

DataFrame

**Examples**

```
df = mtcars
df$mpg[1] = NA
df = pl$DataFrame(df)
df$sort("mpg")
df$sort("mpg", nulls_last = TRUE)
df$sort("cyl", "mpg")
df$sort(c("cyl", "mpg"))
df$sort(c("cyl", "mpg"), descending = TRUE)
df$sort(c("cyl", "mpg"), descending = c(TRUE, FALSE))
df$sort(pl$col("cyl"), pl$col("mpg"))
```

DataFrame\_sql

*Execute a SQL query against the DataFrame***Description**

The calling frame is automatically registered as a table in the SQL context under the name "self". All [DataFrames](#) and [LazyFrames](#) found in the enviro are also registered, using their variable name. More control over registration and execution behaviour is available by the [SQLContext](#) object.

**Usage**

```
DataFrame_sql(query, ..., table_name = NULL, enviro = parent.frame())
```

**Arguments**

query	A character of the SQL query to execute.
...	Ignored.
table_name	NULL (default) or a character of an explicit name for the table that represents the calling frame (the alias "self" will always be registered/available).
enviro	The environment to search for polars <a href="#">DataFrames/LazyFrames</a> .

**Details**

This functionality is considered **unstable**, although it is close to being considered stable. It may be changed at any point without it being considered a breaking change.

**Value**[DataFrame](#)

**See Also**

- [SQLContext](#)

**Examples**

```
df1 = pl$DataFrame(
  a = 1:3,
  b = c("zz", "yy", "xx"),
  c = as.Date(c("1999-12-31", "2010-10-10", "2077-08-08"))
)

# Query the DataFrame using SQL:
df1$sql("SELECT c, b FROM self WHERE a > 1")

# Join two DataFrames using SQL.
df2 = pl$DataFrame(a = 3:1, d = c(125, -654, 888))
df1$sql(
  "
SELECT self.*, d
FROM self
INNER JOIN df2 USING (a)
WHERE a > 1 AND EXTRACT(year FROM c) < 2050
"
)

# Apply transformations to a DataFrame using SQL, aliasing "self" to "frame".
df1$sql(
  query = r"(
SELECT
a,
MOD(a, 2) == 0 AS a_is_even,
CONCAT_WS(':', b, b) AS b_b,
EXTRACT(year FROM c) AS year,
0::float AS 'zero'
FROM frame
)",
  table_name = "frame"
)
```

---

DataFrame\_std

*Std*


---

**Description**

Aggregate the columns of this DataFrame to their standard deviation values.

**Usage**

```
DataFrame_std(ddof = 1)
```

**Arguments**

ddof                   Delta Degrees of Freedom: the divisor used in the calculation is  $N - \text{ddof}$ , where  $N$  represents the number of elements. By default ddof is 1.

**Value**

A DataFrame with one row.

**Examples**

```
p1$DataFrame(mtcars)$std()
```

---

DataFrame_sum	<i>Sum</i>
---------------	------------

---

**Description**

Aggregate the columns of this DataFrame to their sum values.

**Usage**

```
DataFrame_sum()
```

**Value**

A DataFrame with one row.

**Examples**

```
p1$DataFrame(mtcars)$sum()
```

---

DataFrame_tail	<i>Get the last n rows.</i>
----------------	-----------------------------

---

**Description**

Get the last n rows.

**Usage**

```
DataFrame_tail(n = 5L)
```

**Arguments**

n                   Number of rows to return. If a negative value is passed, return all rows except the first `abs(n)`.

**Value**

A [DataFrame](#)

**Examples**

```
df = pl$DataFrame(foo = 1:5, bar = 6:10, ham = letters[1:5])

df$tail(3)

# Pass a negative value to get all rows except the first `abs(n)`.
df$tail(-3)
```

---

DataFrame\_to\_data\_frame

*Return Polars DataFrame as R data.frame*

---

**Description**

Return Polars DataFrame as R data.frame

**Usage**

```
DataFrame_to_data_frame(  
  ...,  
  int64_conversion = polars_options()$int64_conversion  
)
```

**Arguments**

... Any args passed to `as.data.frame()`.

int64\_conversion How should Int64 values be handled when converting a polars object to R?

- "double" (default) converts the integer values to double.
- "bit64" uses `bit64::as.integer64()` to do the conversion (requires the package `bit64` to be attached).
- "string" converts Int64 values to character.

**Value**

An R data.frame

### Conversion to R data types considerations

When converting Polars objects, such as `DataFrames` to R objects, for example via the `as.data.frame()` generic function, each type in the Polars object is converted to an R type. In some cases, an error may occur because the conversion is not appropriate. In particular, there is a high possibility of an error when converting a `Datetime` type without a time zone. A `Datetime` type without a time zone in Polars is converted to the `POSIXct` type in R, which takes into account the time zone in which the R session is running (which can be checked with the `Sys.timezone()` function). In this case, if ambiguous times are included, a conversion error will occur. In such cases, change the session time zone using `Sys.setenv(TZ = "UTC")` and then perform the conversion, or use the `$dt$replace_time_zone()` method on the `Datetime` type column to explicitly specify the time zone before conversion.

```
# Due to daylight savings, clocks were turned forward 1 hour on Sunday, March 8, 2020, 2:00:00 am
# so this particular date-time doesn't exist
non_existent_time = as_polars_series("2020-03-08 02:00:00")$str$strptime(pl$Datetime(), "%F %T")

withr::with_timezone(
  "America/New_York",
  {
    tryCatch(
      # This causes an error due to the time zone (the `TZ` env var is affected).
      as.vector(non_existent_time),
      error = function(e) e
    )
  }
)
#> <error: in to_r: ComputeError(ErrString("datetime '2020-03-08 02:00:00' is non-existent in time zone

withr::with_timezone(
  "America/New_York",
  {
    # This is safe.
    as.vector(non_existent_time$dt$replace_time_zone("UTC"))
  }
)
#> [1] "2020-03-08 02:00:00 UTC"
```

### Examples

```
df = pl$DataFrame(iris[1:3, ])
df$to_data_frame()
```



**Description**

Convert variables into dummy/indicator variables

**Usage**

```
DataFrame_to_dummies(columns = NULL, ..., separator = "_", drop_first = FALSE)
```

**Arguments**

columns	Column name(s) or selector(s) that should be converted to dummy variables. If NULL (default), convert all columns.
...	Ignored.
separator	Separator/delimiter used when generating column names.
drop_first	Remove the first category from the variables being encoded.

**Value**

A DataFrame

**Examples**

```
df = pl$DataFrame(foo = 1:2, bar = 3:4, ham = c("a", "b"))
df$to_dummies()
df$to_dummies(drop_first = TRUE)
df$to_dummies(c("foo", "bar"), separator = "::<")
```

---

DataFrame\_to\_list      *Return Polars DataFrame as a list of vectors*

---

**Description**

Return Polars DataFrame as a list of vectors

**Usage**

```
DataFrame_to_list(
  unnest_structs = TRUE,
  ...,
  int64_conversion = polars_options()$int64_conversion
)
```

**Arguments**

`unnest_structs` Logical. If TRUE (default), then `$unnest()` is applied on any struct column.

... Any args passed to `as.data.frame()`.

`int64_conversion` How should Int64 values be handled when converting a polars object to R?

- "double" (default) converts the integer values to double.
- "bit64" uses `bit64::as.integer64()` to do the conversion (requires the package `bit64` to be attached).
- "string" converts Int64 values to character.

**Details**

For simplicity reasons, this implementation relies on unnesting all structs before exporting to R. If `unnest_structs = FALSE`, then struct columns will be returned as nested lists, where each row is a list of values. Such a structure is not very typical or efficient in R.

**Value**

R list of vectors

**Conversion to R data types considerations**

When converting Polars objects, such as [DataFrames](#) to R objects, for example via the `as.data.frame()` generic function, each type in the Polars object is converted to an R type. In some cases, an error may occur because the conversion is not appropriate. In particular, there is a high possibility of an error when converting a [Datetime](#) type without a time zone. A [Datetime](#) type without a time zone in Polars is converted to the [POSIXct](#) type in R, which takes into account the time zone in which the R session is running (which can be checked with the `Sys.timezone()` function). In this case, if ambiguous times are included, a conversion error will occur. In such cases, change the session time zone using `Sys.setenv(TZ = "UTC")` and then perform the conversion, or use the `$dt$replace_time_zone()` method on the Datetime type column to explicitly specify the time zone before conversion.

```
# Due to daylight savings, clocks were turned forward 1 hour on Sunday, March 8, 2020, 2:00:00 am
# so this particular date-time doesn't exist
non_existent_time = as_polars_series("2020-03-08 02:00:00")$str$strptime(pl$Datetime(), "%F %T")

withr::with_timezone(
  "America/New_York",
  {
    tryCatch(
      # This causes an error due to the time zone (the `TZ` env var is affected).
      as.vector(non_existent_time),
      error = function(e) e
    )
  }
)
```

```
#> <error: in to_r: ComputeError(ErrString("datetime '2020-03-08 02:00:00' is non-existent in time zone

withr::with_timezone(
  "America/New_York",
  {
    # This is safe.
    as.vector(non_existent_time$dt$replace_time_zone("UTC"))
  }
)
#> [1] "2020-03-08 02:00:00 UTC"
```

**See Also**

- `<DataFrame>$get_columns()`: Similar to this method but returns a list of [Series](#) instead of vectors.

**Examples**

```
pl$DataFrame(iris)$to_list()
```

---

DataFrame\_to\_raw\_ipc *Write Arrow IPC data to a raw vector*

---

**Description**

Write Arrow IPC data to a raw vector

**Usage**

```
DataFrame_to_raw_ipc(
  compression = c("uncompressed", "zstd", "lz4"),
  ...,
  compat_level = FALSE
)
```

**Arguments**

<code>compression</code>	NULL or a character of the compression method, "uncompressed" or "lz4" or "zstd". NULL is equivalent to "uncompressed". Choose "zstd" for good compression performance. Choose "lz4" for fast compression/decompression.
<code>...</code>	Ignored.
<code>compat_level</code>	Use a specific compatibility level when exporting Polars' internal data structures. This can be: <ul style="list-style-type: none"> <li>• an integer indicating the compatibility version (currently only 0 for oldest and 1 for newest);</li> <li>• a logical value with TRUE for the newest version and FALSE for the oldest version.</li> </ul>

**Value**

A raw vector

**See Also**

- `<DataFrame>$write_ipc()`

**Examples**

```
df = pl$DataFrame(  
  foo = 1:5,  
  bar = 6:10,  
  ham = letters[1:5]  
)  
  
raw_ipc = df$to_raw_ipc()  
  
pl$read_ipc(raw_ipc)  
  
if (require("arrow", quietly = TRUE)) {  
  arrow::read_ipc_file(raw_ipc, as_data_frame = FALSE)  
}
```

---

DataFrame\_to\_series    *Get column by index*

---

**Description**

Extract a DataFrame column (by index) as a Polars series. Unlike `get_column()`, this method will not fail but will return a NULL if the index doesn't exist in the DataFrame. Keep in mind that Polars is 0-indexed so "0" is the first column.

**Usage**

```
DataFrame_to_series(idx = 0)
```

**Arguments**

`idx`                    Index of the column to return as Series. Defaults to 0, which is the first column.

**Value**

Series or NULL

**Examples**

```
df = pl$DataFrame(iris[1:10, ])  
  
# default is to extract the first column  
df$to_series()  
  
# Polars is 0-indexed, so we use idx = 1 to extract the *2nd* column  
df$to_series(idx = 1)  
  
# doesn't error if the column isn't there  
df$to_series(idx = 8)
```

---

DataFrame\_to\_struct    *Convert DataFrame to a Series of type "struct"*

---

**Description**

Convert DataFrame to a Series of type "struct"

**Usage**

```
DataFrame_to_struct(name = "")
```

**Arguments**

name	Name given to the new Series
------	------------------------------

**Value**

A Series of type "struct"

**Examples**

```
# round-trip conversion from DataFrame with two columns  
df = pl$DataFrame(a = 1:5, b = c("one", "two", "three", "four", "five"))  
s = df$to_struct()  
s  
  
# convert to an R list  
s$to_r()  
  
# Convert back to a DataFrame  
df_s = s$to_frame()  
df_s
```

---

DataFrame\_transpose    *Transpose a DataFrame over the diagonal.*

---

### Description

Transpose a DataFrame over the diagonal.

### Usage

```
DataFrame_transpose(  
  include_header = FALSE,  
  header_name = "column",  
  column_names = NULL  
)
```

### Arguments

`include_header` If TRUE, the column names will be added as first column.

`header_name` If `include_header` is TRUE, this determines the name of the column that will be inserted.

`column_names` Character vector indicating the new column names. If NULL (default), the columns will be named as "column\_1", "column\_2", etc. The length of this vector must match the number of rows of the original input.

### Details

This is a very expensive operation.

Transpose may be the fastest option to perform non foldable (see `fold()` or `reduce()`) row operations like median.

Polars transpose is currently eager only, likely because it is not trivial to deduce the schema.

### Value

DataFrame

### Examples

```
# simple use-case  
pl$DataFrame(mtcars)$transpose(include_header = TRUE, column_names = rownames(mtcars))  
  
# All rows must have one shared supertype, recast Categorical to String which is a supertype  
# of f64, and then dataset "Iris" can be transposed  
pl$DataFrame(iris)$with_columns(pl$col("Species")$cast(pl$String))$transpose()
```

---

DataFrame_unique	<i>Drop duplicated rows</i>
------------------	-----------------------------

---

**Description**

Drop duplicated rows

**Usage**

```
DataFrame_unique(subset = NULL, ..., keep = "any", maintain_order = FALSE)
```

**Arguments**

subset	A character vector with the names of the column(s) to use to identify duplicates. If NULL (default), use all columns.
...	Not used.
keep	Which of the duplicate rows to keep: <ul style="list-style-type: none"><li>• "any" (default): Does not give any guarantee of which row is kept. This allows more optimizations.</li><li>• "first": Keep first unique row.</li><li>• "last": Keep last unique row.</li><li>• "none": Don't keep duplicate rows.</li></ul>
maintain_order	Keep the same order as the original data. Setting this to TRUE makes it more expensive to compute and blocks the possibility to run on the streaming engine.

**Value**

DataFrame

**Examples**

```
df = pl$DataFrame(  
  x = c(1:3, 1:3, 3:1, 1L),  
  y = c(1:3, 1:3, 1:3, 1L)  
)  
df$height  
  
df$unique()$height  
  
# subset to define unique, keep only last or first  
df$unique(subset = "x", keep = "last")  
df$unique(subset = "x", keep = "first")  
  
# only keep unique rows  
df$unique(keep = "none")
```

---

DataFrame_unnest	<i>Unnest the Struct columns of a DataFrame</i>
------------------	---

---

**Description**

Unnest the Struct columns of a DataFrame

**Usage**

```
DataFrame_unnest(...)
```

**Arguments**

... Names of the struct columns to unnest. This doesn't accept Expr. If nothing is provided, then all columns of datatype [Struct](#) are unnested.

**Value**

A DataFrame where some or all columns of datatype Struct are unnested.

**Examples**

```
df = pl$DataFrame(
  a = 1:5,
  b = c("one", "two", "three", "four", "five"),
  c = 6:10
)$
  select(
    pl$struct("b"),
    pl$struct(c("a", "c"))$alias("a_and_c")
  )
df

# by default, all struct columns are unnested
df$unnest()

# we can specify specific columns to unnest
df$unnest("a_and_c")
```

---

DataFrame_unpivot	<i>Unpivot a Frame from wide to long format</i>
-------------------	---

---

**Description**

Unpivot a Frame from wide to long format



**Usage**

```
DataFrame_unpivot(  
  on = NULL,  
  ...,  
  index = NULL,  
  variable_name = NULL,  
  value_name = NULL  
)
```

**Arguments**

on	Values to use as identifier variables. If value_vars is empty all columns that are not in id_vars will be used.
...	Not used.
index	Columns to use as identifier variables.
variable_name	Name to give to the new column containing the names of the melted columns. Defaults to "variable".
value_name	Name to give to the new column containing the values of the melted columns. Defaults to "value".

**Details**

Optionally leaves identifiers set.

This function is useful to massage a Frame into a format where one or more columns are identifier variables (id\_vars), while all other columns, considered measured variables (value\_vars), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

**Value**

A new DataFrame

**Examples**

```
df = pl$DataFrame(  
  a = c("x", "y", "z"),  
  b = c(1, 3, 5),  
  c = c(2, 4, 6),  
  d = c(7, 8, 9)  
)  
df$unpivot(index = "a", on = c("b", "c", "d"))
```

---

DataFrame_var	<i>Var</i>
---------------	------------

---

**Description**

Aggregate the columns of this DataFrame to their variance values.

**Usage**

```
DataFrame_var(ddof = 1)
```

**Arguments**

ddof                   Delta Degrees of Freedom: the divisor used in the calculation is  $N - \text{ddof}$ , where  $N$  represents the number of elements. By default ddof is 1.

**Value**

A DataFrame with one row.

**Examples**

```
p1$DataFrame(mtcars)$var()
```

---

DataFrame_with_columns	<i>Modify/append column(s)</i>
------------------------	--------------------------------

---

**Description**

Add columns or modify existing ones with expressions. This is the equivalent of `dplyr::mutate()` as it keeps unmentioned columns (unlike `$select()`).

**Usage**

```
DataFrame_with_columns(...)
```

**Arguments**

...                   Any expressions or string column name, or same wrapped in a list. If first and only element is a list, it is unwrapped as a list of args.

**Value**

A DataFrame

**Examples**

```

pl$DataFrame(iris)$with_columns(
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")
)

# same query
l_expr = list(
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")
)
pl$DataFrame(iris)$with_columns(l_expr)

pl$DataFrame(iris)$with_columns(
  pl$col("Sepal.Length")$abs(), # not named expr will keep name "Sepal.Length"
  SW_add_2 = (pl$col("Sepal.Width") + 2)
)

```

---

DataFrame\_with\_columns\_seq

*Modify/append column(s)*


---

**Description**

Add columns or modify existing ones with expressions. This is the equivalent of `dplyr::mutate()` as it keeps unmentioned columns (unlike `$select()`).

This will run all expression sequentially instead of in parallel. Use this when the work per expression is cheap. Otherwise, `$with_columns()` should be preferred.

**Usage**

```
DataFrame_with_columns_seq(...)
```

**Arguments**

... Any expressions or string column name, or same wrapped in a list. If first and only element is a list, it is unwrapped as a list of args.

**Value**

A DataFrame

**Examples**

```

pl$DataFrame(iris)$with_columns_seq(
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")
)

```

```
# same query
l_expr = list(
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")
)
pl$DataFrame(iris)$with_columns_seq(l_expr)

pl$DataFrame(iris)$with_columns_seq(
  pl$col("Sepal.Length")$abs(), # not named expr will keep name "Sepal.Length"
  SW_add_2 = (pl$col("Sepal.Width") + 2)
)
```

---

DataFrame\_with\_row\_index

*Add a column for row indices*

---

### Description

Add a new column at index 0 that counts the rows

### Usage

```
DataFrame_with_row_index(name, offset = NULL)
```

### Arguments

name	string name of the created column
offset	positive integer offset for the start of the counter

### Value

A new DataFrame object with a counter column in front

### Examples

```
df = pl$DataFrame(mtcars)

# by default, the index starts at 0 (to mimic the behavior of Python Polars)
df$with_row_index("idx")

# but in R, we use a 1-index
df$with_row_index("idx", offset = 1)
```

---

DataFrame\_write\_csv    *Write to comma-separated values (CSV) file*

---

## Description

Write to comma-separated values (CSV) file

## Usage

```
DataFrame_write_csv(
  file,
  ...,
  include_bom = FALSE,
  include_header = TRUE,
  separator = ",",
  line_terminator = "\n",
  quote_char = "\"",
  batch_size = 1024,
  datetime_format = NULL,
  date_format = NULL,
  time_format = NULL,
  float_precision = NULL,
  null_values = "",
  quote_style = "necessary"
)
```

## Arguments

file	File path to which the result should be written.
...	Ignored.
include_bom	Whether to include UTF-8 BOM (byte order mark) in the CSV output.
include_header	Whether to include header in the CSV output.
separator	Separate CSV fields with this symbol.
line_terminator	String used to end each row.
quote_char	Byte to use as quoting character.
batch_size	Number of rows that will be processed per thread.
datetime_format	A format string, with the specifiers defined by the chrono Rust crate. If no format specified, the default fractional-second precision is inferred from the maximum timeunit found in the frame's Datetime cols (if any).
date_format	A format string, with the specifiers defined by the chrono Rust crate.
time_format	A format string, with the specifiers defined by the chrono Rust crate.

float_precision	Number of decimal places to write, applied to both Float32 and Float64 datatypes.
null_values	A string representing null values (defaulting to the empty string).
quote_style	Determines the quoting strategy used. <ul style="list-style-type: none"> <li>• "necessary" (default): This puts quotes around fields only when necessary. They are necessary when fields contain a quote, delimiter or record terminator. Quotes are also necessary when writing an empty record (which is indistinguishable from a record with one empty field). This is the default.</li> <li>• "always": This puts quotes around every field.</li> <li>• "non_numeric": This puts quotes around all fields that are non-numeric. Namely, when writing a field that does not parse as a valid float or integer, then quotes will be used even if they aren't strictly necessary.</li> <li>• "never": This never puts quotes around fields, even if that results in invalid CSV data (e.g. by not quoting strings containing the separator).</li> </ul>

**Value**

Invisibly returns the input DataFrame.

**Examples**

```
dat = pl$DataFrame(mtcars)

destination = tempfile(fileext = ".csv")
dat$select(pl$col("drat", "mpg"))$write_csv(destination)

pl$read_csv(destination)
```

---

DataFrame\_write\_ipc    *Write to Arrow IPC file (a.k.a Feather file)*

---

**Description**

Write to Arrow IPC file (a.k.a Feather file)

**Usage**

```
DataFrame_write_ipc(
  file,
  compression = c("uncompressed", "zstd", "lz4"),
  ...,
  compat_level = TRUE
)
```

**Arguments**

file	File path to which the result should be written.
compression	NULL or a character of the compression method, "uncompressed" or "lz4" or "zstd". NULL is equivalent to "uncompressed". Choose "zstd" for good compression performance. Choose "lz4" for fast compression/decompression.
...	Ignored.
compat_level	Use a specific compatibility level when exporting Polars' internal data structures. This can be: <ul style="list-style-type: none"><li>• an integer indicating the compatibility version (currently only 0 for oldest and 1 for newest);</li><li>• a logical value with TRUE for the newest version and FALSE for the oldest version.</li></ul>

**Value**

Invisibly returns the input DataFrame.

**See Also**

- [<DataFrame>\\$to\\_raw\\_ipc\(\)](#)

**Examples**

```
dat = pl$DataFrame(mtcars)

destination = tempfile(fileext = ".arrow")
dat$write_ipc(destination)

if (require("arrow", quietly = TRUE)) {
  arrow::read_ipc_file(destination, as_data_frame = FALSE)
}
```

---

DataFrame\_write\_json *Write to JSON file*

---

**Description**

Write to JSON file

**Usage**

```
DataFrame_write_json(file, ..., pretty = FALSE, row_oriented = FALSE)
```

**Arguments**

file	File path to which the result should be written.
...	Ignored.
pretty	Pretty serialize JSON.
row_oriented	Write to row-oriented JSON. This is slower, but more common.

**Value**

Invisibly returns the input DataFrame.

**Examples**

```
if (require("jsonlite", quiet = TRUE)) {
  dat = pl$DataFrame(head(mtcars))
  destination = tempfile()

  dat$select(pl$col("drat", "mpg"))$write_json(destination)
  jsonlite::fromJSON(destination)

  dat$select(pl$col("drat", "mpg"))$write_json(destination, row_oriented = TRUE)
  jsonlite::fromJSON(destination)
}
```

---

DataFrame\_write\_ndjson

*Write to NDJSON file*

---

**Description**

Write to NDJSON file

**Usage**

```
DataFrame_write_ndjson(file)
```

**Arguments**

file	File path to which the result should be written.
------	--

**Value**

Invisibly returns the input DataFrame.



## Examples

```
dat = pl$DataFrame(head(mtcars))

destination = tempfile()
dat$select(pl$col("drat", "mpg"))$write_ndjson(destination)

pl$read_ndjson(destination)
```

---

DataFrame\_write\_parquet

*Write to parquet file*

---

## Description

Write to parquet file

## Usage

```
DataFrame_write_parquet(
  file,
  ...,
  compression = "zstd",
  compression_level = 3,
  statistics = TRUE,
  row_group_size = NULL,
  data_page_size = NULL,
  partition_by = NULL,
  partition_chunk_size_bytes = 4294967296
)
```

## Arguments

file	File path to which the result should be written. This should be a path to a directory if writing a partitioned dataset.
...	Ignored.
compression	String. The compression method. One of: <ul style="list-style-type: none"><li>• "lz4": fast compression/decompression.</li><li>• "uncompressed"</li><li>• "snappy": this guarantees that the parquet file will be compatible with older parquet readers.</li><li>• "gzip"</li><li>• "lzo"</li><li>• "brotli"</li><li>• "zstd": good compression performance.</li></ul>

<code>compression_level</code>	<p>NULL or Integer. The level of compression to use. Only used if method is one of 'gzip', 'brotli', or 'zstd'. Higher compression means smaller files on disk:</p> <ul style="list-style-type: none"> <li>• "gzip": min-level: 0, max-level: 10.</li> <li>• "brotli": min-level: 0, max-level: 11.</li> <li>• "zstd": min-level: 1, max-level: 22.</li> </ul>
<code>statistics</code>	<p>Whether statistics should be written to the Parquet headers. Possible values:</p> <ul style="list-style-type: none"> <li>• TRUE: enable default set of statistics (default)</li> <li>• FALSE: disable all statistics</li> <li>• "full": calculate and write all available statistics.</li> <li>• A named list where all values must be TRUE or FALSE, e.g. <code>list(min = TRUE, max = FALSE)</code>. Statistics available are "min", "max", "distinct_count", "null_count".</li> </ul>
<code>row_group_size</code>	<p>NULL or Integer. Size of the row groups in number of rows. If NULL (default), the chunks of the DataFrame are used. Writing in smaller chunks may reduce memory pressure and improve writing speeds.</p>
<code>data_page_size</code>	<p>Size of the data page in bytes. If NULL (default), it is set to <math>1024^2</math> bytes. will be ~1MB.</p>
<code>partition_by</code>	<p>Column(s) to partition by. A partitioned dataset will be written if this is specified.</p>
<code>partition_chunk_size_bytes</code>	<p>Approximate size to split DataFrames within a single partition when writing. Note this is calculated using the size of the DataFrame in memory - the size of the output file may differ depending on the file format / compression.</p>

### Value

Invisibly returns the input DataFrame.

### Examples

```
dat = pl$DataFrame(mtcars)

# write data to a single parquet file
destination = withr::local_tempfile(fileext = ".parquet")
dat$write_parquet(destination)

# write data to folder with a hive-partitioned structure
dest_folder = withr::local_tempdir()
dat$write_parquet(dest_folder, partition_by = c("gear", "cyl"))
list.files(dest_folder, recursive = TRUE)
```

*DataType\_Array      Create Array DataType*

**Description**

The Array and List datatypes are very similar. The only difference is that sub-arrays all have the same length while sublists can have different lengths. Array methods can be accessed via the \$arr subnamespace.

**Usage**

```
DataType_Array(datatype = "unknown", width)
```

**Arguments**

datatype	An inner DataType. The default is "Unknown" and is only a placeholder for when inner DataType does not matter, e.g. as used in example.
width	The length of the arrays.

**Value**

An array DataType with an inner DataType

**Examples**

```
# basic Array
pl$Array(pl$Int32, 4)
# some nested Array
pl$Array(pl$Array(pl$Boolean, 3), 2)
```

*DataType\_Categorical      Create Categorical DataType*

**Description**

Create Categorical DataType

**Usage**

```
DataType_Categorical(ordering = "physical")
```

**Arguments**

ordering	Either "physical" (default) or "lexical".
----------	---

**Details**

When a categorical variable is created, its string values (or "lexical" values) are stored and encoded as integers ("physical" values) by order of appearance. Therefore, sorting a categorical value can be done either on the lexical or on the physical values. See Examples.

**Value**

A Categorical DataType

**Examples**

```
# default is to order by physical values
df = pl$DataFrame(x = c("z", "z", "k", "a", "z"), schema = list(x = pl$Categorical()))
df$sort("x")

# when setting ordering = "lexical", sorting will be based on the strings
df_lex = pl$DataFrame(
  x = c("z", "z", "k", "a", "z"),
  schema = list(x = pl$Categorical("lexical"))
)
df_lex$sort("x")
```

---

DataType\_contains\_categoricals

*Check whether the data type contains categoricals*

---

**Description**

Check whether the data type contains categoricals

**Usage**

```
DataType_contains_categoricals()
```

**Value**

A logical value

**Examples**

```
pl$List(pl$Categorical())$contains_categoricals()
pl$List(pl$Enum(c("a", "b")))$contains_categoricals()
pl$List(pl$Float32)$contains_categoricals()
pl$List(pl$List(pl$Categorical()))$contains_categoricals()
```

`DataType_contains_views`

*Check whether the data type contains views*

**Description**

Check whether the data type contains views

**Usage**

`DataType_contains_views()`

**Value**

A logical value

**Examples**

```
pl$List(pl$string)$contains_views()
pl$List(pl$Binary)$contains_views()
pl$List(pl$Float32)$contains_views()
pl$List(pl$List(pl$Binary))$contains_views()
```

`DataType_Datetime`

*Data type representing a calendar date and time of day.*

**Description**

The underlying representation of this type is a 64-bit signed integer. The integer indicates the number of time units since the Unix epoch (1970-01-01 00:00:00). The number can be negative to indicate datetimes before the epoch.

**Usage**

`DataType_Datetime(time_unit = "us", time_zone = NULL)`

**Arguments**

<code>time_unit</code>	Unit of time. One of "ms", "us" (default) or "ns".
<code>time_zone</code>	Time zone string, as defined in <a href="#">OlsonNames()</a> . Setting <code>timezone = "*" </code> will match any timezone, which can be useful to select all <code>Datetime</code> columns containing a timezone.

**Value**

Datetime `DataType`

**Examples**

```
pl$Datetime("ns", "Pacific/Samoa")

df = pl$DataFrame(
  naive_time = as.POSIXct("1900-01-01"),
  zoned_time = as.POSIXct("1900-01-01", "UTC")
)
df

df$select(pl$col(pl$Datetime("us", "*")))
```

---

DataType_Duration	<i>Data type representing a time duration</i>
-------------------	---

---

**Description**

Data type representing a time duration

**Usage**

```
DataType_Duration(time_unit = "us")
```

**Arguments**

`time_unit` Unit of time. One of "ms", "us" (default) or "ns".

**Value**

Duration DataType

**Examples**

```
test = pl$DataFrame(
  a = 1:2,
  b = c("a", "b"),
  c = pl$duration(weeks = c(1, 2), days = c(0, 2))
)

# select all columns of type "duration"
test$select(pl$col(pl$Duration()))
```

DataType\_Enum

*Create Enum DataType*

### Description

An Enum is a fixed set categorical encoding of a set of strings. It is similar to the [Categorical](#) data type, but the categories are explicitly provided by the user and cannot be modified.

### Usage

```
DataType_Enum(categories)
```

### Arguments

`categories`      A character vector specifying the categories of the variable.

### Details

This functionality is **unstable**. It is a work-in-progress feature and may not always work as expected. It may be changed at any point without it being considered a breaking change.

### Value

An Enum DataType

### Examples

```
pl$DataFrame(
  x = c("Polar", "Panda", "Brown", "Brown", "Polar"),
  schema = list(x = pl$Enum(c("Polar", "Panda", "Brown")))
)

# All values of the variable have to be in the categories
dtype = pl$Enum(c("Polar", "Panda", "Brown"))
tryCatch(
  pl$DataFrame(
    x = c("Polar", "Panda", "Brown", "Brown", "Polar", "Black"),
    schema = list(x = dtype)
  ),
  error = function(e) e
)

# Comparing two Enum is only valid if they have the same categories
df = pl$DataFrame(
  x = c("Polar", "Panda", "Brown", "Brown", "Polar"),
  y = c("Polar", "Polar", "Polar", "Brown", "Brown"),
  z = c("Polar", "Polar", "Polar", "Brown", "Brown"),
  schema = list(
    x = pl$Enum(c("Polar", "Panda", "Brown")),
```

```

    y = pl$Enum(c("Polar", "Panda", "Brown")),
    z = pl$Enum(c("Polar", "Black", "Brown"))
  )
)

# Same categories
df$with_columns(x_eq_y = pl$col("x") == pl$col("y"))

# Different categories
tryCatch(
  df$with_columns(x_eq_z = pl$col("x") == pl$col("z")),
  error = function(e) e
)

```

---

DataType\_is\_array      *Check whether the data type is an array type*

---

### Description

Check whether the data type is an array type

### Usage

```
DataType_is_array()
```

### Value

A logical value

### Examples

```
pl$Array(width = 2)$is_array()
pl$Float32$is_array()
```

---

DataType\_is\_binary      *Check whether the data type is a binary type*

---

### Description

Check whether the data type is a binary type

### Usage

```
DataType_is_binary()
```

### Value

A logical value



**Examples**

```
pl$Binary$is_binary()
pl$Float32$is_binary()
```

`DataType_is_bool`      *Check whether the data type is a boolean type*

**Description**

Check whether the data type is a boolean type

**Usage**

```
DataType_is_bool()
```

**Value**

A logical value

**Examples**

```
pl$Boolean$is_bool()
pl$Float32$is_bool()
```

`DataType_is_categorical`  
*Check whether the data type is a Categorical type*

**Description**

Check whether the data type is a Categorical type

**Usage**

```
DataType_is_categorical()
```

**Value**

A logical value

**Examples**

```
pl$Categorical()$is_categorical()
pl$Enum(c("a", "b"))$is_categorical()
```

---

DataType\_is\_enum      *Check whether the data type is an Enum type*

---

**Description**

Check whether the data type is an Enum type

**Usage**

DataType\_is\_enum()

**Value**

A logical value

**Examples**

```
p1$Enum(c("a", "b"))$is_enum()  
p1$Categorical()$is_enum()
```

---

DataType\_is\_float      *Check whether the data type is a float type*

---

**Description**

Check whether the data type is a float type

**Usage**

DataType\_is\_float()

**Value**

A logical value

**Examples**

```
p1$Float32$is_float()  
p1$Int32$is_float()
```

---

DataType\_is\_integer     *Check whether the data type is an integer type*

---

**Description**

Check whether the data type is an integer type

**Usage**

DataType\_is\_integer()

**Value**

A logical value

**Examples**

```
p1$Int32$is_integer()  
p1$Float32$is_integer()
```

---

DataType\_is\_known     *Check whether the data type is known*

---

**Description**

Check whether the data type is known

**Usage**

DataType\_is\_known()

**Value**

A logical value

**Examples**

```
p1$string$is_known()  
p1$unknown$is_known()
```

---

DataType\_is\_list      *Check whether the data type is a list type*

---

**Description**

Check whether the data type is a list type

**Usage**

```
DataType_is_list()
```

**Value**

A logical value

**Examples**

```
p1$list()$is_list()  
p1$Float32$is_list()
```

---

DataType\_is\_logical      *Check whether the data type is a logical type*

---

**Description**

Check whether the data type is a logical type

**Usage**

```
DataType_is_logical()
```

**Value**

A logical value

---

DataType\_is\_nested      *Check whether the data type is a nested type*

---

**Description**

Check whether the data type is a nested type

**Usage**

```
DataType_is_nested()
```

**Value**

A logical value

**Examples**

```
pl$List()$is_nested()  
pl$Array(width = 2)$is_nested()  
pl$Float32$is_nested()
```

---

DataType\_is\_null      *Check whether the data type is a null type*

---

**Description**

Check whether the data type is a null type

**Usage**

```
DataType_is_null()
```

**Value**

A logical value

**Examples**

```
pl$Null$is_null()  
pl$Float32$is_null()
```

---

DataType\_is\_numeric    *Check whether the data type is a numeric type*

---

**Description**

Check whether the data type is a numeric type

**Usage**

```
DataType_is_numeric()
```

**Value**

A logical value

**Examples**

```
pl$Float32$is_numeric()  
pl$Int32$is_numeric()  
pl$string$is_numeric()
```

---

DataType\_is\_ord    *Check whether the data type is an ordinal type*

---

**Description**

Check whether the data type is an ordinal type

**Usage**

```
DataType_is_ord()
```

**Value**

A logical value

**Examples**

```
pl$string$is_ord()  
pl$categorical$is_ord()
```

---

DataType\_is\_primitive *Check whether the data type is a primitive type*

---

**Description**

Check whether the data type is a primitive type

**Usage**

```
DataType_is_primitive()
```

**Value**

A logical value

**Examples**

```
pl$Float32$is_primitive()  
pl$List()$is_primitive()
```

---

DataType\_is\_signed\_integer  
*Check whether the data type is a signed integer type*

---

**Description**

Check whether the data type is a signed integer type

**Usage**

```
DataType_is_signed_integer()
```

**Value**

A logical value

**Examples**

```
pl$Int32$is_signed_integer()  
pl$UInt32$is_signed_integer()
```

---

DataType\_is\_string      *Check whether the data type is a String type*

---

**Description**

Check whether the data type is a String type

**Usage**

```
DataType_is_string()
```

**Value**

A logical value

**Examples**

```
p1$string$is_string()  
p1$float32$is_string()
```

---

DataType\_is\_struct      *Check whether the data type is a temporal type*

---

**Description**

Check whether the data type is a temporal type

**Usage**

```
DataType_is_struct()
```

**Value**

A logical value

**Examples**

```
p1$struct()$is_struct()  
p1$float32$is_struct()
```



---

DataType\_is\_temporal *Check whether the data type is a temporal type*

---

**Description**

Check whether the data type is a temporal type

**Usage**

```
DataType_is_temporal()
```

**Value**

A logical value

**Examples**

```
pl$Date$is_temporal()  
pl$Float32$is_temporal()
```

---

DataType\_is\_unsigned\_integer  
*Check whether the data type is an unsigned integer type*

---

**Description**

Check whether the data type is an unsigned integer type

**Usage**

```
DataType_is_unsigned_integer()
```

**Value**

A logical value

**Examples**

```
pl$UInt32$is_unsigned_integer()  
pl$Int32$is_unsigned_integer()
```

---

DataType_List	<i>Create List DataType</i>
---------------	-----------------------------

---

**Description**

Create List DataType

**Usage**

```
DataType_List(datatype = "unknown")
```

**Arguments**

datatype      The inner DataType.

**Value**

A list DataType with an inner DataType

**Examples**

```
# some nested List
pl$List(pl$List(pl$Boolean))

# check if some maybe_list is a List DataType
maybe_List = pl$List(pl$UInt64)
pl$same_outer_dt(maybe_List, pl$List())
```

---

DataType_Struct	<i>Create Struct DataType</i>
-----------------	-------------------------------

---

**Description**

One can create a Struct data type with `pl$Struct()`. There are also multiple ways to create columns of data type Struct in a DataFrame or a Series, see the examples.

**Usage**

```
DataType_Struct(...)
```

**Arguments**

...      Either named inputs of the form `field_name = datatype` or objects of class `RPolarsField` created by `pl$Field()`.

**Value**

A Struct DataType containing a list of Fields

**Examples**

```
# create a Struct-DataType
pl$Struct(foo = pl$Int32, pl$field("bar", pl$Boolean))

# check if an element is any kind of Struct()
test = pl$Struct(a = pl$UInt64)
pl$same_outer_dt(test, pl$Struct())

# `test` is a type of Struct, but it doesn't mean it is equal to an empty Struct
test == pl$Struct()

# The way to create a `Series` of type `Struct` is a bit convoluted as it involves
# `data.frame()`, `list()`, and `I()`:
as_polars_series(
  data.frame(a = 1:2, b = I(list(c("x", "y"), "z")))
)

# A slightly simpler way would be via `tibble::tibble()` or
# `data.table::data.table()`:
if (requireNamespace("tibble", quietly = TRUE)) {
  as_polars_series(
    tibble::tibble(a = 1:2, b = list(c("x", "y"), "z"))
  )
}

# Finally, one can use `pl$struct()` to convert existing columns or `Series`
# to a `Struct`:
x = pl$DataFrame(
  a = 1:2,
  b = list(c("x", "y"), "z")
)

out = x$select(pl$struct(c("a", "b")))
out

out$schema
```

---

dim.RPolarsDataFrame *Get the dimensions*

---

**Description**

Get the dimensions

**Usage**

```
## S3 method for class 'RPolarsDataFrame'
dim(x)

## S3 method for class 'RPolarsLazyFrame'
dim(x)
```

**Arguments**

x                    A [DataFrame](#) or [LazyFrame](#)

---

```
dimnames.RPolarsDataFrame
                          Get the row and column names
```

---

**Description**

Get the row and column names

**Usage**

```
## S3 method for class 'RPolarsDataFrame'
dimnames(x)

## S3 method for class 'RPolarsLazyFrame'
dimnames(x)
```

**Arguments**

x                    A [DataFrame](#) or [LazyFrame](#)

---

```
docs_translations        Translation definitions across python, R and polars.
```

---

**Description**

#Comments for how the R and python world translates into polars:

R and python are both high-level glue languages great for Data Science. Rust is a pedantic low-level language with similar use cases as C and C++. Polars is written in ~100k lines of rust and has a rust API. Py-polars the python API for polars, is implemented as an interface with the rust API. r-polars is very parallel to py-polars except it interfaces with R. The performance and behavior are unexpectedly quite similar as the 'engine' is the exact same rust code and data structures.

**Format**

info

**Value**

Not applicable

**Translation details****R and the integerish:**

R only has a native Int32 type, no Uint32, Int64, UInt64 , ... types. These days Int32 is getting a bit small, to refer to more rows than  $\sim 2^{31}-1$ . There are packages which provide int64, but the most normal hack' is to just use floats as 'integerish'. There is an unique float64 value for every integer up to about  $2^{52}$  which is plenty for all practical concerns. Some polars methods may accept or return a floats even though an integer ideally would be more accurate. Most R functions intermix Int32 (integer) and Float64 (double) seamlessly.

**Missingness:**

R has allocated a value in every vector type to signal missingness, these are collectively called NAs. Polars uses a bool bitmask to signal NA-like missing value and it is called Null and Nulls in plural. Not to confuse with R NULL (see paragraph below). Polars supports missingness for any possible type as it kept separately in the bitmask. In python lists the symbol None can carry a similar meaning. R NA ~ polars Null ~ py-polars [None] (in a py list)

**Sorting and comparisons:**

From writing a lot of tests for all implementations, it appears polars does not have a fully consistent nor well documented behavior, when it comes to comparisons and sorting of floats. Though some general thumb rules do apply: Polars have chosen to define in sorting that Null is a value lower than  $-\text{Inf}$  as in `Expr.arg_min()` However except when Null is ignored `Expr.min()`, there is a `Expr.nan_min()` but no `Expr.nan_min()`. NaN is sometimes a value higher than Inf and sometimes regarded as a Null. Polars conventions  $\text{NaN} > \text{Inf} > 99 > -99 > -\text{Inf} > \text{Null}$  `Null == Null` yields often times false, sometimes true, sometimes Null. The documentation or examples do not reveal this variations. The best to do, when in doubt, is to do test sort on a small Series/Column of all values.

```
#' R NaN ~ polars NaN ~ python [float("NaN")] #only floats have NaNs
```

```
R Inf ~ polars inf ~ python [float("inf")] #only floats have Inf
```

**NULL IS NOT Null is not NULL:**

The R NULL does not exist inside polars frames and series and so on. It resembles the `Option::None` in the hidden rust code. It resembles the python `None`. In all three languages the NULL/None/None are used in this context as function argument to signal default behavior or perhaps a deactivated feature. R NULL does NOT translate into the polars bitmask Null, that is NA. R NULL ~ rust-polars `Option::None` ~ pypolars `None` #typically used for function arguments

**LISTS, FRAMES AND DICTS:**

The following translations are relevant when loading data into polars. The R list appears similar to python dictionary (hashmap), but is implemented more similar to the python list (array of pointers). R list do support string naming elements via a string vector. In polars both lists (of vectors or series) and data.frames can be used to construct a polars DataFrame, just a as dictionaries would be used in python. In terms of loading in/out data the follow translation holds: R `data.frame/list` ~ polars `DataFrame` ~ python dictionary

**Series and Vectors:**

The R vector (Integer, Double, Character, ...) resembles the Series as both are external from any frame and can be of any length. The implementation is quite different. E.g. for-loop appending to an R vector is considered quite bad for performance. The vector will be fully rewritten in memory for every append. The polars Series has chunked memory allocation, which allows any append data to be written only. However fragmented memory is not great for fast computations and polars objects have a `rechunk()`-method, to reallocate chunks into one. `Rechunk` might be called implicitly by polars. In the context of constructing Series and extracting data, the following translation holds: R vector ~ polars Series/column ~ python list

**Expressions:**

The polars Expr do not have any base R counterpart. Expr are analogous to how ggplot split plotting instructions from the rendering. Base R plot immediately pushes any instruction by adding e.g. pixels to a .png canvas. ggplot collects instructions and in the end when executed the rendering can be performed with optimization across all instructions. Btw ggplot command-syntax is a monoid meaning the order does not matter, that is not the case for polars Expr. Polars Expr's can be understood as a DSL (domain specific language) that expresses syntax trees of instructions. R expressions evaluate to syntax trees also, but it difficult to optimize the execution order automatically, without rewriting the code. A great selling point of Polars is that any query will be optimized. Expr are very light-weight symbols chained together.

---

DynamicGroupBy\_agg      *Aggregate over a DynamicGroupBy*

---

**Description**

Aggregate a DataFrame over a time or integer window created with `$group_by_dynamic()`.

**Usage**

```
DynamicGroupBy_agg(...)
```

**Arguments**

...                      Exprs to aggregate over. Those can also be passed wrapped in a list, e.g `$agg(list(e1, e2, e3))`.

**Value**

An aggregated [DataFrame](#)

**Examples**

```
df = pl.DataFrame(
  time = pl.datetime_range(
    start = strptime("2021-12-16 00:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC"),
    end = strptime("2021-12-16 03:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC"),
    interval = "30m"
  ),
)
```

```

    n = 0:6
  )

  # get the sum in the following hour relative to the "time" column
  df$group_by_dynamic("time", every = "1h")$agg(
    vals = pl$col("n"),
    sum = pl$col("n")$sum()
  )

  # using "include_boundaries = TRUE" is helpful to see the period considered
  df$group_by_dynamic("time", every = "1h", include_boundaries = TRUE)$agg(
    vals = pl$col("n")
  )

  # in the example above, the values didn't include the one *exactly* 1h after
  # the start because "closed = 'left'" by default.
  # Changing it to "right" includes values that are exactly 1h after. Note that
  # the value at 00:00:00 now becomes included in the interval [23:00:00 - 00:00:00],
  # even if this interval wasn't there originally
  df$group_by_dynamic("time", every = "1h", closed = "right")$agg(
    vals = pl$col("n")
  )
  # To keep both boundaries, we use "closed = 'both'". Some values now belong to
  # several groups:
  df$group_by_dynamic("time", every = "1h", closed = "both")$agg(
    vals = pl$col("n")
  )

  # Dynamic group bys can also be combined with grouping on normal keys
  df = df$with_columns(
    groups = as_polars_series(c("a", "a", "a", "b", "b", "a", "a"))
  )
  df

  df$group_by_dynamic(
    "time",
    every = "1h",
    closed = "both",
    group_by = "groups",
    include_boundaries = TRUE
  )$agg(pl$col("n"))

  # We can also create a dynamic group by based on an index column
  df = pl$LazyFrame(
    idx = 0:5,
    A = c("A", "A", "B", "B", "B", "C")
  )$with_columns(pl$col("idx")$set_sorted())
  df

  df$group_by_dynamic(
    "idx",
    every = "2i",
    period = "3i",

```

```

include_boundaries = TRUE,
closed = "right"
)$agg(A_agg_list = pl$col("A"))

```

---

DynamicGroupBy\_class    *Operations on Polars DataFrame grouped on time or integer values*

---

### Description

This class comes from `<DataFrame>$group_by_dynamic()`.

### Examples

```

df = pl$DataFrame(
  time = pl$datetime_range(
    start = strptime("2021-12-16 00:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC"),
    end = strptime("2021-12-16 03:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC"),
    interval = "30m"
  ),
  n = 0:6
)

# get the sum in the following hour relative to the "time" column
df$group_by_dynamic("time", every = "1h")

```

---

DynamicGroupBy\_ungroup    *Ungroup a DynamicGroupBy object*

---

### Description

Revert the `$group_by_dynamic()` operation. Doing `<DataFrame>$group_by_dynamic(...)$ungroup()` returns the original DataFrame.

### Usage

```
DynamicGroupBy_ungroup()
```

### Value

[DataFrame](#)



**Examples**

```
df = pl$DataFrame(
  time = pl$datetime_range(
    start = strptime("2021-12-16 00:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC"),
    end = strptime("2021-12-16 03:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC"),
    interval = "30m"
  ),
  n = 0:6
)
df

df$group_by_dynamic("time", every = "1h")$ungroup()
```

ExprArr\_all

*Evaluate whether all boolean values in an array are true***Description**

Evaluate whether all boolean values in an array are true

**Usage**

```
ExprArr_all()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(c(TRUE, TRUE), c(FALSE, TRUE), c(FALSE, FALSE), c(NA, NA)),
  schema = list(values = pl$Array(pl$Boolean, 2))
)
df$with_columns(all = pl$col("values")$arr$all())
```

ExprArr\_any

*Evaluate whether any boolean values in an array are true***Description**

Evaluate whether any boolean values in an array are true

**Usage**

```
ExprArr_any()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(c(TRUE, TRUE), c(FALSE, TRUE), c(FALSE, FALSE), c(NA, NA)),
  schema = list(values = pl$Array(pl$Boolean, 2))
)
df$with_columns(any = pl$col("values")$arr$any())
```

---

ExprArr_arg_max	<i>Get the index of the maximal value in an array</i>
-----------------	---

---

**Description**

Get the index of the maximal value in an array

**Usage**

ExprArr\_arg\_max()

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(1:2, 2:1),
  schema = list(values = pl$Array(pl$Int32, 2))
)
df$with_columns(
  arg_max = pl$col("values")$arr$arg_max()
)
```

---

ExprArr_arg_min	<i>Get the index of the minimal value in an array</i>
-----------------	---

---

**Description**

Get the index of the minimal value in an array

**Usage**

ExprArr\_arg\_min()

**Value**

Expr

**Examples**

```
df = pl$DataFrame(  
  values = list(1:2, 2:1),  
  schema = list(values = pl$Array(pl$Int32, 2))  
)  
df$with_columns(  
  arg_min = pl$col("values")$arr$arg_min()  
)
```

---

ExprArr_contains	<i>Check if array contains a given value</i>
------------------	--

---

**Description**

Check if array contains a given value

**Usage**

ExprArr\_contains(item)

**Arguments**item            Expr or something coercible to an Expr. Strings are *not* parsed as columns.**Value**

Expr

**Examples**

```
df = pl$DataFrame(  
  values = list(0:2, 4:6, c(NA_integer_, NA_integer_, NA_integer_)),  
  item = c(0L, 4L, 2L),  
  schema = list(values = pl$Array(pl$Float64, 3))  
)  
df$with_columns(  
  with_expr = pl$col("values")$arr$contains(pl$col("item")),  
  with_lit = pl$col("values")$arr$contains(1)  
)
```

---

ExprArr\_get                      *Get the value by index in an array*

---

### Description

This allows to extract one value per array only.

### Usage

```
ExprArr_get(index, ..., null_on_oob = TRUE)
```

### Arguments

index	An Expr or something coercible to an Expr, that must return a single index. Values are 0-indexed (so index 0 would return the first item of every sub-array) and negative values start from the end (index -1 returns the last item).
...	Ignored.
null_on_oob	If TRUE, return null if an index is out of bounds. Otherwise, raise an error.

### Value

Expr

### Examples

```
df = pl$DataFrame(
  values = list(c(1, 2), c(3, 4), c(NA_real_, 6)),
  idx = c(1, NA, 3),
  schema = list(values = pl$Array(pl$Float64, 2))
)
df$with_columns(
  using_expr = pl$col("values")$arr$get("idx"),
  val_0 = pl$col("values")$arr$get(0),
  val_minus_1 = pl$col("values")$arr$get(-1),
  val_oob = pl$col("values")$arr$get(10)
)
```

---

ExprArr\_join                      *Join elements of an array*

---

### Description

Join all string items in a sub-array and place a separator between them. This only works on columns of type `list[str]`.

**Usage**

```
ExprArr_join(separator, ignore_nulls = FALSE)
```

**Arguments**

**separator** String to separate the items with. Can be an Expr. Strings are *not* parsed as columns.

**ignore\_nulls** If FALSE (default), null values are propagated: if the row contains any null values, the output is null.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(c("a", "b", "c"), c("x", "y", "z"), c("e", NA, NA)),
  separator = c("-", "+", "/"),
  schema = list(values = pl$Array(pl$String, 3))
)
df$with_columns(
  join_with_expr = pl$col("values")$arr$join(pl$col("separator")),
  join_with_lit = pl$col("values")$arr$join(" "),
  join_ignore_null = pl$col("values")$arr$join(" ", ignore_nulls = TRUE)
)
```

---

ExprArr\_max

*Find the maximum value in an array*

---

**Description**

Find the maximum value in an array

**Usage**

```
ExprArr_max()
```

**Details**

This method is only available with the "nightly" feature. See [polars\\_info\(\)](#) for more details.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(c(1, 2), c(3, 4), c(NA_real_, NA_real_)),
  schema = list(values = pl$Array(pl$Float64, 2))
)
df$with_columns(max = pl$col("values")$arr$max())
```

---

ExprArr_median	<i>Find the median in an array</i>
----------------	------------------------------------

---

**Description**

Find the median in an array

**Usage**

```
ExprArr_median()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(c(2, 1, 4), c(8.4, 3.2, 1)),
  schema = list(values = pl$Array(pl$Float64, 3))
)
df$with_columns(median = pl$col("values")$arr$median())
```

---

ExprArr_min	<i>Find the minimum value in an array</i>
-------------	---

---

**Description**

Find the minimum value in an array

**Usage**

```
ExprArr_min()
```

**Details**

This method is only available with the "nightly" feature. See [polars\\_info\(\)](#) for more details.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(  
  values = list(c(1, 2), c(3, 4), c(NA_real_, NA_real_)),  
  schema = list(values = pl$Array(pl$Float64, 2))  
)  
df$with_columns(min = pl$col("values")$arr$min())
```

---

ExprArr_reverse	<i>Reverse values in an array</i>
-----------------	-----------------------------------

---

**Description**

Reverse values in an array

**Usage**

ExprArr\_reverse()

**Value**

Expr

**Examples**

```
df = pl$DataFrame(  
  values = list(c(1, 2), c(3, 4), c(NA_real_, 6)),  
  schema = list(values = pl$Array(pl$Float64, 2))  
)  
df$with_columns(reverse = pl$col("values")$arr$reverse())
```

---

ExprArr_shift	<i>Shift array values by n indices</i>
---------------	--

---

**Description**

Shift array values by n indices

**Usage**

ExprArr\_shift(n = 1)

**Arguments**

`n` Number of indices to shift forward. If a negative value is passed, values are shifted in the opposite direction instead.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(1:3, c(2L, NA_integer_, 5L)),
  idx = 1:2,
  schema = list(values = pl$Array(pl$Int32, 3))
)
df$with_columns(
  shift_by_expr = pl$col("values")$arr$shift(pl$col("idx")),
  shift_by_lit = pl$col("values")$arr$shift(2)
)
```

---

ExprArr\_sort

*Sort values in an array*

---

**Description**

Sort values in an array

**Usage**

```
ExprArr_sort(descending = FALSE, nulls_last = FALSE)
```

**Arguments**

`descending` A logical. If TRUE, sort in descending order.  
`nulls_last` A logical. If TRUE, place null values last instead of first.

**Examples**

```
df = pl$DataFrame(
  values = list(c(2, 1), c(3, 4), c(NA_real_, 6)),
  schema = list(values = pl$Array(pl$Float64, 2))
)
df$with_columns(sort = pl$col("values")$arr$sort(nulls_last = TRUE))
```



---

ExprArr_std	<i>Find the standard deviation in an array</i>
-------------	--

---

**Description**

Find the standard deviation in an array

**Usage**

```
ExprArr_std(ddof = 1)
```

**Arguments**

ddof                      Delta Degrees of Freedom: the divisor used in the calculation is  $N - \text{ddof}$ , where  $N$  represents the number of elements. By default ddof is 1.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(  
  values = list(c(2, 1, 4), c(8.4, 3.2, 1)),  
  schema = list(values = pl$Array(pl$Float64, 3))  
)  
df$with_columns(std = pl$col("values")$arr$std())
```

---

ExprArr_sum	<i>Sum all elements in an array</i>
-------------	-------------------------------------

---

**Description**

Sum all elements in an array

**Usage**

```
ExprArr_sum()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(c(1, 2), c(3, 4), c(NA_real_, 6)),
  schema = list(values = pl$Array(pl$Float64, 2))
)
df$with_columns(sum = pl$col("values")$arr$sum())
```

---

ExprArr_to_list	<i>Convert an Array column into a List column with the same inner data type</i>
-----------------	---

---

**Description**

Convert an Array column into a List column with the same inner data type

**Usage**

```
ExprArr_to_list()
```

**Value**

[Expr](#) of data type [List](#)

**Examples**

```
df = pl$DataFrame(
  a = list(c(1, 2), c(3, 4)),
  schema = list(a = pl$Array(pl$Int8, 2))
)

df$with_columns(
  list = pl$col("a")$arr$to_list()
)
```

---

ExprArr_to_struct	<i>Convert array to struct</i>
-------------------	--------------------------------

---

**Description**

Convert array to struct

**Usage**

```
ExprArr_to_struct(fields = NULL)
```

**Arguments**

`fields` If the name and number of the desired fields is known in advance, a list of field names can be given, which will be assigned by index. Otherwise, to dynamically assign field names, a custom R function that takes an R double and outputs a string value can be used. If NULL (default), fields will be `field_0`, `field_1` ... `field_n`.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(1:3, c(2L, NA_integer_, 5L)),
  schema = list(values = pl$Array(pl$Int32, 3))
)
df$with_columns(
  struct = pl$col("values")$arr$to_struct()
)

# pass a custom function that will name all fields by adding a prefix
df2 = df$with_columns(
  pl$col("values")$arr$to_struct(
    fields = \(idx) paste0("col_", idx)
  )
)
df2

df2$unnest()
```

---

ExprArr\_unique      *Get unique values in an array*

---

**Description**

Get unique values in an array

**Usage**

```
ExprArr_unique(maintain_order = FALSE)
```

**Arguments**

`maintain_order` If TRUE, the unique values are returned in order of appearance.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(c(1, 1, 2), c(4, 4, 4), c(NA_real_, 6, 7)),
  schema = list(values = pl$Array(pl$Float64, 3))
)
df$with_columns(unique = pl$col("values")$arr$unique())
```

ExprArr\_var

*Find the variance in an array***Description**

Find the variance in an array

**Usage**

```
ExprArr_var(ddof = 1)
```

**Arguments**

ddof                   Delta Degrees of Freedom: the divisor used in the calculation is  $N - \text{ddof}$ , where  $N$  represents the number of elements. By default ddof is 1.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(c(2, 1, 4), c(8.4, 3.2, 1)),
  schema = list(values = pl$Array(pl$Float64, 3))
)
df$with_columns(var = pl$col("values")$arr$var())
```

ExprBin\_contains

*Check if binaries contain a binary substring***Description**

Check if binaries contain a binary substring

**Usage**

```
ExprBin_contains(literal)
```

**Arguments**

literal            The binary substring to look for.

**Value**

Expr returning a Boolean

**Examples**

```
colors = pl$DataFrame(
  name = c("black", "yellow", "blue"),
  code = as_polars_series(c("x00x00x00", "xffxffx00", "x00x00xff"))$cast(pl$Binary),
  lit = as_polars_series(c("x00", "xffx00", "xffxff"))$cast(pl$Binary)
)

colors$select(
  "name",
  contains_with_lit = pl$col("code")$bin$contains("xff"),
  contains_with_expr = pl$col("code")$bin$contains(pl$col("lit"))
)
```

---

ExprBin_decode	<i>Decode values using the provided encoding</i>
----------------	--

---

**Description**

Decode values using the provided encoding

**Usage**

```
ExprBin_decode(encoding, ..., strict = TRUE)
```

**Arguments**

encoding            A character, "hex" or "base64". The encoding to use.

...                 Ignored.

strict              Raise an error if the underlying value cannot be decoded, otherwise mask out with a null value.

**Value**

Expr of data type String.

**Examples**

```
df = pl$DataFrame(
  name = c("black", "yellow", "blue"),
  code_hex = as_polars_series(c("000000", "ffff00", "0000ff"))$cast(pl$Binary),
  code_base64 = as_polars_series(c("AAAA", "//8A", "AAD/"))$cast(pl$Binary)
)

df$with_columns(
  decoded_hex = pl$col("code_hex")$bin$decode("hex"),
  decoded_base64 = pl$col("code_base64")$bin$decode("base64")
)

# Set `strict = FALSE` to set invalid values to `null` instead of raising an error.
df = pl$DataFrame(
  colors = as_polars_series(c("000000", "ffff00", "invalid_value"))$cast(pl$Binary)
)
df$select(pl$col("colors")$bin$decode("hex", strict = FALSE))
```

---

**ExprBin\_encode***Encode a value using the provided encoding*

---

**Description**

Encode a value using the provided encoding

**Usage**

```
ExprBin_encode(encoding)
```

**Arguments**

encoding            A character, "hex" or "base64". The encoding to use.

**Value**

[Expr](#) of data type String.

**Examples**

```
df = pl$DataFrame(
  name = c("black", "yellow", "blue"),
  code = as_polars_series(
    c("000000", "ffff00", "0000ff")
  )$cast(pl$Binary)$bin$decode("hex")
)

df$with_columns(encoded = pl$col("code")$bin$encode("hex"))
```

---

ExprBin\_ends\_with      *Check if string values end with a binary substring*

---

**Description**

Check if string values end with a binary substring

**Usage**

```
ExprBin_ends_with(suffix)
```

**Arguments**

suffix                  Suffix substring.

**Value**

Expr returning a Boolean

**Examples**

```
colors = pl$DataFrame(
  name = c("black", "yellow", "blue"),
  code = as_polars_series(c("x00x00x00", "xffxffx00", "x00x00xff"))$cast(pl$Binary),
  suffix = as_polars_series(c("x00", "xffx00", "xffxff"))$cast(pl$Binary)
)

colors$select(
  "name",
  ends_with_lit = pl$col("code")$bin$ends_with("xff"),
  ends_with_expr = pl$col("code")$bin$ends_with(pl$col("suffix"))
)
```

---

ExprBin\_size                  *Get the size of binary values in the given unit*

---

**Description**

Get the size of binary values in the given unit

**Usage**

```
ExprBin_size(unit = "b")
```

**Arguments**

unit                      Scale the returned size to the given unit. Can be "b", "kb", "mb", "gb", "tb", or their full names ("kilobytes", etc.).

**Value**

Expr of data type UInt or Float.

**Examples**

```
df = pl$DataFrame(
  name = c("black", "yellow", "blue"),
  code_hex = as_polars_series(c("000000", "ffff00", "0000ff"))$cast(pl$Binary)
)

df$with_columns(
  n_bytes = pl$col("code_hex")$bin$size(),
  n_kilobytes = pl$col("code_hex")$bin$size("kb")
)
```

---

ExprBin\_starts\_with    *Check if values start with a binary substring*

---

**Description**

Check if values start with a binary substring

**Usage**

```
ExprBin_starts_with(sub)
```

**Arguments**

sub                    Prefix substring.

**Value**

Expr returning a Boolean

**Examples**

```
colors = pl$DataFrame(
  name = c("black", "yellow", "blue"),
  code = as_polars_series(c("x00x00x00", "xffxffx00", "x00x00xff"))$cast(pl$Binary),
  prefix = as_polars_series(c("x00", "xffx00", "xffxff"))$cast(pl$Binary)
)

colors$select(
  "name",
  starts_with_lit = pl$col("code")$bin$starts_with("xff"),
  starts_with_expr = pl$col("code")$bin$starts_with(pl$col("prefix"))
)
```



---

`ExprCat_get_categories`*Get the categories stored in this data type*

---

**Description**

Get the categories stored in this data type

**Usage**

```
ExprCat_get_categories()
```

**Value**

A polars DataFrame with the categories for each categorical Series.

**Examples**

```
df = pl.DataFrame(  
    cats = factor(c("z", "z", "k", "a", "b")),  
    vals = factor(c(3, 1, 2, 2, 3))  
)  
df  
  
df$select(  
    pl$col("cats")$cat$get_categories()  
)  
df$select(  
    pl$col("vals")$cat$get_categories()  
)
```

---

`ExprCat_set_ordering` *Set Ordering*

---

**Description**

Determine how this categorical series should be sorted.

**Usage**

```
ExprCat_set_ordering(ordering)
```

**Arguments**

<code>ordering</code>	string either 'physical' or 'lexical' <ul style="list-style-type: none"><li>• "physical": use the physical representation of the categories to determine the order (default).</li><li>• "lexical": use the string values to determine the order.</li></ul>
-----------------------	--

**Value**

An Expr of datatype Categorical

**Examples**

```
df = pl$DataFrame(
  cats = factor(c("z", "z", "k", "a", "b")),
  vals = c(3, 1, 2, 2, 3)
)

# sort by the string value of categories
df$with_columns(
  pl$col("cats")$cat$set_ordering("lexical")
)$sort("cats", "vals")

# sort by the underlying value of categories
df$with_columns(
  pl$col("cats")$cat$set_ordering("physical")
)$sort("cats", "vals")
```

---

ExprDT\_cast\_time\_unit *cast\_time\_unit*

---

**Description**

Cast the underlying data to another time unit. This may lose precision. The corresponding global timepoint will stay unchanged +/- precision.

**Usage**

```
ExprDT_cast_time_unit(tu = c("ns", "us", "ms"))
```

**Arguments**

tu                    string option either 'ns', 'us', or 'ms'

**Value**

Expr of i64

**Examples**

```
df = pl$DataFrame(
  date = pl$datetime_range(
    start = as.Date("2001-1-1"),
    end = as.Date("2001-1-3"),
    interval = "1d1s"
  )
)
```

```
df$select(
  pl$col("date"),
  pl$col("date")$dt$cast_time_unit()$alias("cast_time_unit_ns"),
  pl$col("date")$dt$cast_time_unit(tu = "ms")$alias("cast_time_unit_ms")
)
```

---

ExprDT_combine	<i>Combine Date and Time</i>
----------------	------------------------------

---

### Description

If the underlying expression is a Datetime then its time component is replaced, and if it is a Date then a new Datetime is created by combining the two values.

### Usage

```
ExprDT_combine(time, time_unit = "us")
```

### Arguments

time	The number of epoch since or before (if negative) the Date. Can be an Expr or a PTime.
time_unit	Unit of time. One of "ms", "us" (default) or "ns".

### Value

Date/Datetime expr

### Examples

```
df = pl$DataFrame(
  dtm = c(
    ISOdatetime(2022, 12, 31, 10, 30, 45),
    ISOdatetime(2023, 7, 5, 23, 59, 59)
  ),
  dt = c(ISOdate(2022, 10, 10), ISOdate(2022, 7, 5)),
  tm = c(pl$time(1, 2, 3, 456000), pl$time(7, 8, 9, 101000))
)$explode("tm")

df

df$select(
  d1 = pl$col("dtm")$dt$combine(pl$col("tm")),
  s2 = pl$col("dt")$dt$combine(pl$col("tm")),
  d3 = pl$col("dt")$dt$combine(pl$time(4, 5, 6))
)
```

---

 ExprDT\_convert\_time\_zone

*Convert to given time zone for an expression of type Datetime.*


---

### Description

If converting from a time-zone-naive datetime, then conversion will happen as if converting from UTC, regardless of your system's time zone.

### Usage

```
ExprDT_convert_time_zone(time_zone)
```

### Arguments

time\_zone      String time zone from `base::OlsonNames()`

### Value

Expr of i64

### Examples

```
df = pl$DataFrame(
  date = pl$datetime_range(
    as.POSIXct("2020-03-01", tz = "UTC"),
    as.POSIXct("2020-05-01", tz = "UTC"),
    "1mo1s"
  )
)

df$select(
  "date",
  London = pl$col("date")$dt$convert_time_zone("Europe/London")
)
```

---

 ExprDT\_day

*Day*


---

### Description

Extract day from underlying Date representation. Applies to Date and Datetime columns. Returns the day of month starting from 1. The return value ranges from 1 to 31. (The last day of month differs by months.)

**Usage**

```
ExprDT_day()
```

**Value**

Expr of day as UInt32

**Examples**

```
df = pl$DataFrame(
  date = pl$date_range(
    as.Date("2020-12-25"),
    as.Date("2021-1-05"),
    interval = "1d",
    time_zone = "GMT"
  )
)
df$with_columns(
  pl$col("date")$dt$day()$alias("day")
)
```

---

ExprDT\_epoch

*Epoch*

---

**Description**

Get the time passed since the Unix EPOCH in the give time unit.

**Usage**

```
ExprDT_epoch(time_unit = "us")
```

**Arguments**

`time_unit` Time unit, one of "ns", "us", "ms", "s" or "d".

**Value**

Expr with datatype Int64

**Examples**

```
df = pl$DataFrame(date = pl$date_range(as.Date("2001-1-1"), as.Date("2001-1-3")))
df$with_columns(
  epoch_ns = pl$col("date")$dt$epoch(),
  epoch_s = pl$col("date")$dt$epoch(time_unit = "s")
)
```

---

ExprDT_hour	<i>Hour</i>
-------------	-------------

---

**Description**

Extract hour from underlying Datetime representation. Applies to Datetime columns. Returns the hour number from 0 to 23.

**Usage**

```
ExprDT_hour()
```

**Value**

Expr of hour as UInt32

**Examples**

```
df = pl$DataFrame(
  date = pl$datetime_range(
    as.Date("2020-12-25"),
    as.Date("2021-1-05"),
    interval = "1d2h",
    time_zone = "GMT"
  )
)
df$with_columns(
  pl$col("date")$dt$hour()$alias("hour")
)
```

---

ExprDT_iso_year	<i>Iso-Year</i>
-----------------	-----------------

---

**Description**

Extract ISO year from underlying Date representation. Applies to Date and Datetime columns. Returns the year number in the ISO standard. This may not correspond with the calendar year.

**Usage**

```
ExprDT_iso_year()
```

**Value**

Expr of iso\_year as Int32

**Examples**

```
df = pl$DataFrame(  
  date = pl$date_range(  
    as.Date("2020-12-25"),  
    as.Date("2021-1-05"),  
    interval = "1d",  
    time_zone = "GMT"  
  )  
)  
df$with_columns(  
  pl$col("date")$dt$year()$alias("year"),  
  pl$col("date")$dt$iso_year()$alias("iso_year")  
)
```

---

ExprDT_is_leap_year	<i>Determine whether the year of the underlying date is a leap year</i>
---------------------	---

---

**Description**

Determine whether the year of the underlying date is a leap year

**Usage**

```
ExprDT_is_leap_year()
```

**Value**

An Expr of datatype Bool

**Examples**

```
df = pl$DataFrame(date = as.Date(c("2000-01-01", "2001-01-01", "2002-01-01")))  
df$with_columns(  
  leap_year = pl$col("date")$dt$is_leap_year()  
)
```

---

ExprDT_microsecond	<i>Extract microseconds from underlying Datetime representation.</i>
--------------------	--

---

**Description**

Applies to Datetime columns.

**Usage**

```
ExprDT_microsecond()
```

**Value**

Expr of data type Int32

**Examples**

```
df = pl$DataFrame(  
  datetime = as.POSIXct(  
    c(  
      "1978-01-01 01:01:01",  
      "2024-10-13 05:30:14.500",  
      "2065-01-01 10:20:30.06"  
    ),  
    "UTC"  
  )  
)  
  
df$with_columns(  
  microsecond = pl$col("datetime")$dt$microsecond()  
)
```

---

ExprDT\_millisecond      *Extract milliseconds from underlying Datetime representation*

---

**Description**

Applies to Datetime columns.

**Usage**

```
ExprDT_millisecond()
```

**Value**

Expr of data type Int32

**Examples**

```
df = pl$DataFrame(  
  datetime = as.POSIXct(  
    c(  
      "1978-01-01 01:01:01",  
      "2024-10-13 05:30:14.500",  
      "2065-01-01 10:20:30.06"  
    ),  
    "UTC"  
  )  
)  
  
df$with_columns(  
  microsecond = pl$col("datetime")$dt$microsecond()  
)
```



```

    millisecond = pl$col("datetime")$dt$millisecond()
  )

```

---

ExprDT_minute	<i>Minute</i>
---------------	---------------

---

### Description

Extract minutes from underlying Datetime representation. Applies to Datetime columns. Returns the minute number from 0 to 59.

### Usage

```
ExprDT_minute()
```

### Value

Expr of minute as UInt32

### Examples

```

df = pl$DataFrame(
  date = pl$datetime_range(
    as.Date("2020-12-25"),
    as.Date("2021-1-05"),
    interval = "1d5s",
    time_zone = "GMT"
  )
)
df$with_columns(
  pl$col("date")$dt$minute()$alias("minute")
)

```

---

ExprDT_month	<i>Month</i>
--------------	--------------

---

### Description

Extract month from underlying Date representation. Applies to Date and Datetime columns. Returns the month number starting from 1. The return value ranges from 1 to 12.

### Usage

```
ExprDT_month()
```

### Value

Expr of month as UInt32

**Examples**

```
df = pl$DataFrame(
  date = pl$date_range(
    as.Date("2020-12-25"),
    as.Date("2021-1-05"),
    interval = "1d",
    time_zone = "GMT"
  )
)
df$with_columns(
  pl$col("date")$dt$month()$alias("month")
)
```

---

ExprDT_nanosecond	<i>Extract nanoseconds from underlying Datetime representation</i>
-------------------	--

---

**Description**

Applies to Datetime columns.

**Usage**

```
ExprDT_nanosecond()
```

**Value**

[Expr](#) of data type Int32

**Examples**

```
df = pl$DataFrame(
  datetime = as.POSIXct(
    c(
      "1978-01-01 01:01:01",
      "2024-10-13 05:30:14.500",
      "2065-01-01 10:20:30.06"
    ),
    "UTC"
  )
)
df$with_columns(
  nanosecond = pl$col("datetime")$dt$nanosecond()
)
```

---

ExprDT_offset_by	<i>Offset By</i>
------------------	------------------

---

**Description**

Offset this date by a relative time offset. This differs from `pl$col("foo_datetime_tu") + value_tu` in that it can take months and leap years into account. Note that only a single minus sign is allowed in the `by` string, as the first character.

**Usage**

```
ExprDT_offset_by(by)
```

**Arguments**

`by` optional string encoding duration see details.

**Details**

The `by` are created with the the following string language:

- `1ns # 1 nanosecond`
- `1us # 1 microsecond`
- `1ms # 1 millisecond`
- `1s # 1 second`
- `1m # 1 minute`
- `1h # 1 hour`
- `1d # 1 day`
- `1w # 1 calendar week`
- `1mo # 1 calendar month`
- `1y # 1 calendar year`
- `1i # 1 index count`

These strings can be combined:

- `3d12h4m25s # 3 days, 12 hours, 4 minutes, and 25 seconds`

**Value**

Date/Datetime expr

**Examples**

```

df = pl$DataFrame(
  dates = pl$date_range(
    as.Date("2000-1-1"),
    as.Date("2005-1-1"),
    "1y"
  )
)
df$select(
  pl$col("dates")$dt$offset_by("1y")$alias("date_plus_1y"),
  pl$col("dates")$dt$offset_by("-1y2mo")$alias("date_min")
)

# the "by" argument also accepts expressions
df = pl$DataFrame(
  dates = pl$datetime_range(
    as.POSIXct("2022-01-01", tz = "GMT"),
    as.POSIXct("2022-01-02", tz = "GMT"),
    interval = "6h", time_unit = "ms", time_zone = "GMT"
  )$to_r(),
  offset = c("1d", "-2d", "1mo", NA, "1y")
)

df

df$with_columns(new_dates = pl$col("dates")$dt$offset_by(pl$col("offset")))

```

---

ExprDT\_ordinal\_day      *Ordinal Day*

---

**Description**

Extract ordinal day from underlying Date representation. Applies to Date and Datetime columns. Returns the day of year starting from 1. The return value ranges from 1 to 366. (The last day of year differs by years.)

**Usage**

```
ExprDT_ordinal_day()
```

**Value**

Expr of ordinal\_day as UInt32

**Examples**

```

df = pl$DataFrame(
  date = pl$date_range(
    as.Date("2020-12-25"),

```

```
    as.Date("2021-1-05"),
    interval = "1d",
    time_zone = "GMT"
  )
)
df$with_columns(
  pl$col("date")$dt$ordinal_day()$alias("ordinal_day")
)
```

---

ExprDT_quarter	<i>Quarter</i>
----------------	----------------

---

## Description

Extract quarter from underlying Date representation. Applies to Date and Datetime columns. Returns the quarter ranging from 1 to 4.

## Usage

```
ExprDT_quarter()
```

## Value

Expr of quarter as UInt32

## Examples

```
df = pl$DataFrame(
  date = pl$date_range(
    as.Date("2020-12-25"),
    as.Date("2021-1-05"),
    interval = "1d",
    time_zone = "GMT"
  )
)
df$with_columns(
  pl$col("date")$dt$quarter()$alias("quarter")
)
```

---

ExprDT\_replace\_time\_zone  
*Replace time zone*

---

### Description

Cast time zone for a Series of type Datetime. This is different from `$convert_time_zone()` as it will also modify the underlying timestamp. Use to correct a wrong time zone annotation. This will change the corresponding global timepoint.

### Usage

```
ExprDT_replace_time_zone(
  time_zone,
  ...,
  ambiguous = "raise",
  non_existent = "raise"
)
```

### Arguments

time_zone	NULL or string time zone from <code>base::OlsonNames()</code>
...	Ignored.
ambiguous	Determine how to deal with ambiguous datetimes: <ul style="list-style-type: none"> <li>• "raise" (default): throw an error</li> <li>• "earliest": use the earliest datetime</li> <li>• "latest": use the latest datetime</li> <li>• "null": return a null value</li> </ul>
non_existent	Determine how to deal with non-existent datetimes: <ul style="list-style-type: none"> <li>• "raise" (default): throw an error</li> <li>• "null": return a null value</li> </ul>

### Value

Expr of i64

### Examples

```
df1 = pl$DataFrame(
  london_timezone = pl$datetime_range(
    as.POSIXct("2020-03-01", tz = "UTC"),
    as.POSIXct("2020-07-01", tz = "UTC"),
    "1mo1s"
  )$dt$convert_time_zone("Europe/London")
)
```

```

df1$select(
  "london_timezone",
  London_to_Amsterdam = pl$col("london_timezone")$dt$replace_time_zone("Europe/Amsterdam")
)

# You can use `ambiguous` to deal with ambiguous datetimes:
dates = c(
  "2018-10-28 01:30",
  "2018-10-28 02:00",
  "2018-10-28 02:30",
  "2018-10-28 02:00"
)
df2 = pl$DataFrame(
  ts = as_polars_series(dates)$str$strptime(pl$Datetime("us")),
  ambiguous = c("earliest", "earliest", "latest", "latest")
)

df2$with_columns(
  ts_localized = pl$col("ts")$dt$replace_time_zone(
    "Europe/Brussels",
    ambiguous = pl$col("ambiguous")
  )
)

```

ExprDT\_round

*Round datetime***Description**

Divide the date/datetime range into buckets. Each date/datetime in the first half of the interval is mapped to the start of its bucket. Each date/datetime in the second half of the interval is mapped to the end of its bucket.

**Usage**

```
ExprDT_round(every)
```

**Arguments**

`every` Either an Expr or a string indicating a column name or a duration (see Details).

**Details**

The `every` and `offset` argument are created with the the following string language:

- `1ns` # 1 nanosecond
- `1us` # 1 microsecond
- `1ms` # 1 millisecond
- `1s` # 1 second

- 1m # 1 minute
- 1h # 1 hour
- 1d # 1 day
- 1w # 1 calendar week
- 1mo # 1 calendar month
- 1y # 1 calendar year These strings can be combined:
  - 3d12h4m25s # 3 days, 12 hours, 4 minutes, and 25 seconds

**Value**

Date/Datetime expr

**Examples**

```
t1 = as.POSIXct("3040-01-01", tz = "GMT")
t2 = t1 + as.difftime(25, units = "secs")
s = pl$datetime_range(t1, t2, interval = "2s", time_unit = "ms")

df = pl$DataFrame(datetime = s)$with_columns(
  pl$col("datetime")$dt$round("4s")$alias("rounded_4s")
)
df
```

ExprDT\_second

*Extract seconds from underlying Datetime representation***Description**

Applies to Datetime columns. Returns the integer second number from 0 to 59, or a floating point number from 0 < 60 if fractional=TRUE that includes any milli/micro/nanosecond component.

**Usage**

```
ExprDT_second(fractional = FALSE)
```

**Arguments**

`fractional` A logical. Whether to include the fractional component of the second.

**Value**

`Expr` of data type Int8 or Float64



**Examples**

```
df = pl$DataFrame(
  datetime = as.POSIXct(
    c(
      "1978-01-01 01:01:01",
      "2024-10-13 05:30:14.500",
      "2065-01-01 10:20:30.06"
    ),
    "UTC"
  )
)

df$with_columns(
  second = pl$col("datetime")$dt$second(),
  second_fractional = pl$col("datetime")$dt$second(fractional = TRUE)
)
```

---

ExprDT_strftime	<i>strftime</i>
-----------------	-----------------

---

**Description**

Format Date/Datetime with a formatting rule. See chrono `strftime/strptime` <<https://docs.rs/chrono/latest/chrono>>

**Usage**

```
ExprDT_strftime(format)
```

**Arguments**

format	string format very much like in R passed to chrono
--------	--

**Value**

Date/Datetime expr

**Examples**

```
pl$lit(as.POSIXct("2021-01-02 12:13:14", tz = "GMT"))$dt$strftime("this is the year: %Y")$to_r()
```

---

 ExprDT\_time

*Extract time from a Datetime Series*


---

**Description**

This only works on Datetime Series, it will error on Date Series.

**Usage**

```
ExprDT_time()
```

**Value**

A Time Expr

**Examples**

```
df = pl$DataFrame(dates = pl$datetime_range(
  as.Date("2000-1-1"),
  as.Date("2000-1-2"),
  "1h"
))

df$with_columns(times = pl$col("dates")$dt$time())
```

---

 ExprDT\_timestamp

*timestamp*


---

**Description**

Return a timestamp in the given time unit.

**Usage**

```
ExprDT_timestamp(tu = c("ns", "us", "ms"))
```

**Arguments**

tu string option either 'ns', 'us', or 'ms'

**Value**

Expr of i64

**Examples**

```
df = pl$DataFrame(  
  date = pl$datetime_range(  
    start = as.Date("2001-1-1"),  
    end = as.Date("2001-1-3"),  
    interval = "1d1s"  
  )  
)  
df$select(  
  pl$col("date"),  
  pl$col("date")$dt$timestamp()$alias("timestamp_ns"),  
  pl$col("date")$dt$timestamp(tu = "ms")$alias("timestamp_ms")  
)
```

---

ExprDT_total_days	<i>Days</i>
-------------------	-------------

---

**Description**

Extract the days from a Duration type.

**Usage**

```
ExprDT_total_days()
```

**Value**

Expr of i64

**Examples**

```
df = pl$DataFrame(  
  date = pl$datetime_range(  
    start = as.Date("2020-3-1"),  
    end = as.Date("2020-5-1"),  
    interval = "1mo1s"  
  )  
)  
df$select(  
  pl$col("date"),  
  diff_days = pl$col("date")$diff()$dt$total_days()  
)
```

---

ExprDT\_total\_hours     *Hours*

---

**Description**

Extract the hours from a Duration type.

**Usage**

```
ExprDT_total_hours()
```

**Value**

Expr of i64

**Examples**

```
df = pl$DataFrame(  
  date = pl$date_range(  
    start = as.Date("2020-1-1"),  
    end = as.Date("2020-1-4"),  
    interval = "1d"  
  )  
)  
df$select(  
  pl$col("date"),  
  diff_hours = pl$col("date")$diff()$dt$total_hours()  
)
```

---

ExprDT\_total\_microseconds  
                          *microseconds*

---

**Description**

Extract the microseconds from a Duration type.

**Usage**

```
ExprDT_total_microseconds()
```

**Value**

Expr of i64

**Examples**

```
df = pl$DataFrame(date = pl$datetime_range(  
  start = as.POSIXct("2020-1-1", tz = "GMT"),  
  end = as.POSIXct("2020-1-1 00:00:01", tz = "GMT"),  
  interval = "1ms"  
))  
df$select(  
  pl$col("date"),  
  diff_microsec = pl$col("date")$diff()$dt$total_microseconds()  
)
```

---

ExprDT\_total\_milliseconds  
*milliseconds*

---

**Description**

Extract the milliseconds from a Duration type.

**Usage**

```
ExprDT_total_milliseconds()
```

**Value**

Expr of i64

**Examples**

```
df = pl$DataFrame(date = pl$datetime_range(  
  start = as.POSIXct("2020-1-1", tz = "GMT"),  
  end = as.POSIXct("2020-1-1 00:00:01", tz = "GMT"),  
  interval = "1ms"  
))  
df$select(  
  pl$col("date"),  
  diff_millisec = pl$col("date")$diff()$dt$total_milliseconds()  
)
```

ExprDT\_total\_minutes *Minutes*

---

**Description**

Extract the minutes from a Duration type.

**Usage**

```
ExprDT_total_minutes()
```

**Value**

Expr of i64

**Examples**

```
df = pl$DataFrame(  
  date = pl$date_range(  
    start = as.Date("2020-1-1"),  
    end = as.Date("2020-1-4"),  
    interval = "1d"  
  )  
)  
df$select(  
  pl$col("date"),  
  diff_minutes = pl$col("date")$diff()$dt$total_minutes()  
)
```

---

ExprDT\_total\_nanoseconds  
*nanoseconds*

---

**Description**

Extract the nanoseconds from a Duration type.

**Usage**

```
ExprDT_total_nanoseconds()
```

**Value**

Expr of i64

**Examples**

```
df = pl$DataFrame(date = pl$datetime_range(  
  start = as.POSIXct("2020-1-1", tz = "GMT"),  
  end = as.POSIXct("2020-1-1 00:00:01", tz = "GMT"),  
  interval = "1ms"  
))  
df$select(  
  pl$col("date"),  
  diff_nanosec = pl$col("date")$diff()$dt$total_nanoseconds()  
)
```

---

ExprDT_total_seconds	<i>Seconds</i>
----------------------	----------------

---

**Description**

Extract the seconds from a Duration type.

**Usage**

```
ExprDT_total_seconds()
```

**Value**

Expr of i64

**Examples**

```
df = pl$DataFrame(date = pl$datetime_range(  
  start = as.POSIXct("2020-1-1", tz = "GMT"),  
  end = as.POSIXct("2020-1-1 00:04:00", tz = "GMT"),  
  interval = "1m"  
))  
df$select(  
  pl$col("date"),  
  diff_sec = pl$col("date")$diff()$dt$total_seconds()  
)
```

---

ExprDT_truncate	<i>Truncate datetime</i>
-----------------	--------------------------

---

**Description**

Divide the date/datetime range into buckets. Each date/datetime is mapped to the start of its bucket.

**Usage**

```
ExprDT_truncate(every)
```

**Arguments**

`every` Either an Expr or a string indicating a column name or a duration (see Details).

**Details**

The `every` and `offset` argument are created with the the following string language:

- 1ns # 1 nanosecond
  - 1us # 1 microsecond
  - 1ms # 1 millisecond
  - 1s # 1 second
  - 1m # 1 minute
  - 1h # 1 hour
  - 1d # 1 day
  - 1w # 1 calendar week
  - 1mo # 1 calendar month
  - 1y # 1 calendar year
- These strings can be combined:
- 3d12h4m25s # 3 days, 12 hours, 4 minutes, and 25 seconds

**Value**

Date/Datetime expr

**Examples**

```
t1 = as.POSIXct("3040-01-01", tz = "GMT")
t2 = t1 + as.difftime(25, units = "secs")
s = pl$datetime_range(t1, t2, interval = "2s", time_unit = "ms")

df = pl$DataFrame(datetime = s)$with_columns(
  pl$col("datetime")$dt$truncate("4s")$alias("truncated_4s")
)
df
```



---

ExprDT_week	<i>Week</i>
-------------	-------------

---

**Description**

Extract the week from the underlying Date representation. Applies to Date and Datetime columns. Returns the ISO week number starting from 1. The return value ranges from 1 to 53. (The last week of year differs by years.)

**Usage**

```
ExprDT_week()
```

**Value**

Expr of week as UInt32

**Examples**

```
df = pl$DataFrame(  
  date = pl$date_range(  
    as.Date("2020-12-25"),  
    as.Date("2021-1-05"),  
    interval = "1d",  
    time_zone = "GMT"  
  )  
)  
df$with_columns(  
  pl$col("date")$dt$week()$alias("week")  
)
```

---

ExprDT_weekday	<i>Weekday</i>
----------------	----------------

---

**Description**

Extract the week day from the underlying Date representation. Applies to Date and Datetime columns. Returns the ISO weekday number where monday = 1 and sunday = 7

**Usage**

```
ExprDT_weekday()
```

**Value**

Expr of weekday as UInt32

**Examples**

```
df = pl$DataFrame(
  date = pl$date_range(
    as.Date("2020-12-25"),
    as.Date("2021-1-05"),
    interval = "1d",
    time_zone = "GMT"
  )
)
df$with_columns(
  pl$col("date")$dt$weekday()$alias("weekday")
)
```

---

ExprDT\_with\_time\_unit *with\_time\_unit*

---

**Description**

Set time unit of a Series of dtype Datetime or Duration. This does not modify underlying data, and should be used to fix an incorrect time unit. The corresponding global timepoint will change.

**Usage**

```
ExprDT_with_time_unit(tu = c("ns", "us", "ms"))
```

**Arguments**

tu string option either 'ns', 'us', or 'ms'

**Value**

Expr of i64

**Examples**

```
df = pl$DataFrame(
  date = pl$datetime_range(
    start = as.Date("2001-1-1"),
    end = as.Date("2001-1-3"),
    interval = "1d1s"
  )
)
df$select(
  pl$col("date"),
  pl$col("date")$dt$with_time_unit()$alias("with_time_unit_ns"),
  pl$col("date")$dt$with_time_unit(tu = "ms")$alias("with_time_unit_ms")
)
```

---

ExprDT_year	<i>Year</i>
-------------	-------------

---

**Description**

Extract year from underlying Date representation. Applies to Date and Datetime columns. Returns the year number in the calendar date.

**Usage**

```
ExprDT_year()
```

**Value**

Expr of Year as Int32

**Examples**

```
df = pl$DataFrame(  
  date = pl$date_range(  
    as.Date("2020-12-25"),  
    as.Date("2021-1-05"),  
    interval = "1d",  
    time_zone = "GMT"  
  )  
)  
df$with_columns(  
  pl$col("date")$dt$year()$alias("year"),  
  pl$col("date")$dt$iso_year()$alias("iso_year")  
)
```

---

ExprList_all	<i>Evaluate whether all boolean values in a list are true</i>
--------------	---

---

**Description**

Evaluate whether all boolean values in a list are true

**Usage**

```
ExprList_all()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  list(a = list(c(TRUE, TRUE), c(FALSE, TRUE), c(FALSE, FALSE), NA, c()))
)
df$with_columns(all = pl$col("a")$list$all())
```

ExprList\_any

*Evaluate whether any boolean values in a list are true***Description**

Evaluate whether any boolean values in a list are true

**Usage**

```
ExprList_any()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  list(a = list(c(TRUE, TRUE), c(FALSE, TRUE), c(FALSE, FALSE), NA, c()))
)
df$with_columns(any = pl$col("a")$list$any())
```

ExprList\_arg\_max

*Get the index of the maximal value in list***Description**

Get the index of the maximal value in list

**Usage**

```
ExprList_arg_max()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(list(s = list(1:2, 2:1)))
df$with_columns(
  arg_max = pl$col("s")$list$arg_max()
)
```

---

ExprList_arg_min	<i>Get the index of the minimal value in list</i>
------------------	---

---

**Description**

Get the index of the minimal value in list

**Usage**

```
ExprList_arg_min()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(list(s = list(1:2, 2:1)))
df$with_columns(
  arg_min = pl$col("s")$list$arg_min()
)
```

---

ExprList_concat	<i>Concat two list variables</i>
-----------------	----------------------------------

---

**Description**

Concat two list variables

**Usage**

```
ExprList_concat(other)
```

**Arguments**

other            Values to concat with. Can be an Expr or something coercible to an Expr.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = list("a", "x"),
  b = list(c("b", "c"), c("y", "z"))
)
df$with_columns(
  conc_to_b = pl$col("a")$list$concat(pl$col("b")),
  conc_to_lit_str = pl$col("a")$list$concat(pl$lit("some string")),
  conc_to_lit_list = pl$col("a")$list$concat(pl$lit(list("hello", c("hello", "world"))))
)
```

---

ExprList_contains	<i>Check if list contains a given value</i>
-------------------	---

---

**Description**

Check if list contains a given value

**Usage**

```
ExprList_contains(item)
```

**Arguments**

`item` Expr or something coercible to an Expr. Strings are *not* parsed as columns.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = list(3:1, NULL, 1:2),
  item = 0:2
)
df$with_columns(
  with_expr = pl$col("a")$list$contains(pl$col("item")),
  with_lit = pl$col("a")$list$contains(1)
)
```

---

ExprList_diff	<i>Compute difference between list values</i>
---------------	---

---

**Description**

This computes the first discrete difference between shifted items of every list. The parameter `n` gives the interval between items to subtract, e.g `n = 2` the output will be the difference between the 1st and the 3rd value, the 2nd and 4th value, etc.

**Usage**

```
ExprList_diff(n = 1, null_behavior = c("ignore", "drop"))
```

**Arguments**

`n` Number of slots to shift. If negative, then it starts from the end.  
`null_behavior` How to handle null values. Either "ignore" (default) or "drop".

**Value**

Expr

**Examples**

```
df = pl$DataFrame(list(s = list(1:4, c(10L, 2L, 1L))))
df$with_columns(diff = pl$col("s")$list$diff(2))

# negative value starts shifting from the end
df$with_columns(diff = pl$col("s")$list$diff(-2))
```

---

ExprList_eval	<i>Run any polars expression on the list values</i>
---------------	---

---

**Description**

Run any polars expression on the list values

**Usage**

```
ExprList_eval(expr, parallel = FALSE)
```

**Arguments**

`expr` Expression to run. Note that you can select an element with `pl$element()`, `pl$first()`, and more. See Examples.  
`parallel` Run all expression parallel. Don't activate this blindly. Parallelism is worth it if there is enough work to do per thread. This likely should not be used in the `$group_by()` context, because we already do parallel execution per group.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = list(c(1, 8, 3), c(3, 2), c(NA, NA, 1)),
  b = list(c("R", "is", "amazing"), c("foo", "bar"), "text")
)

df

# standardize each value inside a list, using only the values in this list
df$select(
  a_stand = pl$col("a")$list$eval(
    (pl$element() - pl$element()$mean()) / pl$element()$std()
  )
)

# count characters for each element in list. Since column "b" is list[str],
# we can apply all `str` functions on elements in the list:
df$select(
  b_len_chars = pl$col("b")$list$eval(
    pl$element()$str$len_chars()
  )
)

# concat strings in each list
df$select(
  pl$col("b")$list$eval(pl$element()$str$join(" "))*list$first()
)
```

ExprList\_explode

*Returns a column with a separate row for every list element***Description**

Returns a column with a separate row for every list element

**Usage**

ExprList\_explode()

**Value**

Expr



**Examples**

```
df = pl$DataFrame(a = list(c(1, 2, 3), c(4, 5, 6)))
df$select(pl$col("a")$list$explode())
```

---

ExprList_first	<i>Get the first value in a list</i>
----------------	--------------------------------------

---

**Description**

Get the first value in a list

**Usage**

```
ExprList_first()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(list(a = list(3:1, NULL, 1:2)))
df$with_columns(
  first = pl$col("a")$list$first()
)
```

---

ExprList_gather	<i>Get several values by index in a list</i>
-----------------	--

---

**Description**

This allows to extract several values per list. To extract a single value by index, use [\\$list\\$get\(\)](#).

**Usage**

```
ExprList_gather(index, null_on_oob = FALSE)
```

**Arguments**

index	An Expr or something coercible to an Expr, that can return several single indices. Values are 0-indexed (so index 0 would return the first item of every sublist) and negative values start from the end (index -1 returns the last item). If the index is out of bounds, it will return a null. Strings are parsed as column names.
null_on_oob	If TRUE, return null if an index is out of bounds. Otherwise, raise an error.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = list(c(3, 2, 1), 1, c(1, 2)),
  idx = list(0:1, integer(), c(1L, 999L))
)
df$with_columns(
  gathered = pl$col("a")$list$gather("idx", null_on_oob = TRUE)
)

df$with_columns(
  gathered = pl$col("a")$list$gather(2, null_on_oob = TRUE)
)

# by some column name, must cast to an Int/UInt type to work
df$with_columns(
  gathered = pl$col("a")$list$gather(pl$col("a")$cast(pl$List(pl$UInt64)), null_on_oob = TRUE)
)
```

---

ExprList\_gather\_every *Gather every nth element in a list*

---

**Description**

Gather every nth element in a list

**Usage**

ExprList\_gather\_every(n, offset = 0)

**Arguments**

n	Positive integer.
offset	Starting index.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = list(1:5, 6:8, 9:12),
  n = c(2, 1, 3),
  offset = c(0, 1, 0)
)

df$with_columns(
  gather_every = pl$col("a")$list$gather_every(pl$col("n"), offset = pl$col("offset"))
)
```

---

ExprList_get	<i>Get the value by index in a list</i>
--------------	---

---

**Description**

This allows to extract one value per list only. To extract several values by index, use [\\$list\\$gather\(\)](#).

**Usage**

```
ExprList_get(index, ..., null_on_oob = TRUE)
```

**Arguments**

index	An Expr or something coercible to an Expr, that must return a single index. Values are 0-indexed (so index 0 would return the first item of every sublist) and negative values start from the end (index -1 returns the last item).
...	Ignored.
null_on_oob	If TRUE, return null if an index is out of bounds. Otherwise, raise an error.

**Value**

[Expr](#)

**Examples**

```
df = pl$DataFrame(
  values = list(c(2, 2, NA), c(1, 2, 3), NA_real_, NULL),
  idx = c(1, 2, NA, 3)
)

df$with_columns(
  using_expr = pl$col("values")$list$get("idx"),
  val_0 = pl$col("values")$list$get(0),
  val_minus_1 = pl$col("values")$list$get(-1),
  val_oob = pl$col("values")$list$get(10)
)
```

---

ExprList\_head                    *Get the first n values of a list*

---

### Description

Get the first n values of a list

### Usage

```
ExprList_head(n = 5L)
```

### Arguments

n                    Number of values to return for each sublist. Can be an Expr. Strings are parsed as column names.

### Value

Expr

### Examples

```
df = pl$DataFrame(
  s = list(1:4, c(10L, 2L, 1L)),
  n = 1:2
)
df$with_columns(
  head_by_expr = pl$col("s")$list$head("n"),
  head_by_lit = pl$col("s")$list$head(2)
)
```

---

ExprList\_join                    *Join elements of a list*

---

### Description

Join all string items in a sublist and place a separator between them. This only works on columns of type `list[str]`.

### Usage

```
ExprList_join(separator, ignore_nulls = FALSE)
```

**Arguments**

separator	String to separate the items with. Can be an Expr. Strings are <i>not</i> parsed as columns.
ignore_nulls	If FALSE (default), null values are propagated: if the row contains any null values, the output is null.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  s = list(c("a", "b", "c"), c("x", "y"), c("e", NA)),
  separator = c("-", "+", "/" )
)
df$with_columns(
  join_with_expr = pl$col("s")$list$join(pl$col("separator")),
  join_with_lit = pl$col("s")$list$join(" "),
  join_ignore_null = pl$col("s")$list$join(" ", ignore_nulls = TRUE)
)
```

ExprList\_last

*Get the last value in a list***Description**

Get the last value in a list

**Usage**

ExprList\_last()

**Value**

Expr

**Examples**

```
df = pl$DataFrame(list(a = list(3:1, NULL, 1:2)))
df$with_columns(
  last = pl$col("a")$list$last()
)
```

---

ExprList\_len            *Get the length of each list*

---

**Description**

Return the number of elements in each list. Null values are counted in the total.

**Usage**

```
ExprList_len()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(list(list_of_strs = list(c("a", "b", NA), "c")))
df$with_columns(len_list = pl$col("list_of_strs")$list$len())
```

---

ExprList\_max            *Find the maximum value in a list*

---

**Description**

Find the maximum value in a list

**Usage**

```
ExprList_max()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA_real_))
df$with_columns(max = pl$col("values")$list$max())
```

---

ExprList_mean	<i>Compute the mean value of a list</i>
---------------	---

---

**Description**

Compute the mean value of a list

**Usage**

```
ExprList_mean()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA_real_))
df$with_columns(mean = pl$col("values")$list$mean())
```

---

ExprList_min	<i>Find the minimum value in a list</i>
--------------	---

---

**Description**

Find the minimum value in a list

**Usage**

```
ExprList_min()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA_real_))
df$with_columns(min = pl$col("values")$list$min())
```

---

ExprList\_n\_unique      *Get the number of unique values in a list*

---

**Description**

Get the number of unique values in a list

**Usage**

```
ExprList_n_unique()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(values = list(c(2, 2, NA), c(1, 2, 3), NA_real_))
df$with_columns(unique = pl$col("values")$list$n_unique())
```

---

ExprList\_reverse      *Reverse values in a list*

---

**Description**

Reverse values in a list

**Usage**

```
ExprList_reverse()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA_real_))
df$with_columns(reverse = pl$col("values")$list$reverse())
```



---

ExprList_sample	<i>Sample from this list</i>
-----------------	------------------------------

---

**Description**

Sample from this list

**Usage**

```
ExprList_sample(
  n = NULL,
  ...,
  fraction = NULL,
  with_replacement = FALSE,
  shuffle = FALSE,
  seed = NULL
)
```

**Arguments**

n	Number of items to return. Cannot be used with fraction.
...	Ignored.
fraction	Fraction of items to return. Cannot be used with n. Can be larger than 1 if with_replacement is TRUE.
with_replacement	If TRUE (default), allow values to be sampled more than once.
shuffle	Shuffle the order of sampled data points (implicitly TRUE if with_replacement = TRUE).
seed	numeric value of 0 to 2 <sup>52</sup> Seed for the random number generator. If NULL (default), a random seed value between 0 and 10000 is picked.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  values = list(1:3, NA_integer_, c(NA_integer_, 3L), 5:7),
  n = c(1, 1, 1, 2)
)

df$with_columns(
  sample = pl$col("values")$list$sample(n = pl$col("n"), seed = 1)
)
```

---

 ExprList\_set\_difference

*Get the difference of two list variables*


---

### Description

This returns the "asymmetric difference", meaning only the elements of the first list that are not in the second list. To get all elements that are in only one of the two lists, use `$set_symmetric_difference()`.

### Usage

```
ExprList_set_difference(other)
```

### Arguments

`other` Other list variable. Can be an Expr or something coercible to an Expr.

### Details

Note that the datatypes inside the list must have a common supertype. For example, the first column can be `list[i32]` and the second one can be `list[i8]` because it can be cast to `list[i32]`. However, the second column cannot be e.g `list[f32]`.

### Value

Expr

### Examples

```
df = pl$DataFrame(
  a = list(1:3, NA_integer_, c(NA_integer_, 3L), 5:7),
  b = list(2:4, 3L, c(3L, 4L, NA_integer_), c(6L, 8L))
)

df$with_columns(difference = pl$col("a")$list$set_difference("b"))
```

---

 ExprList\_set\_intersection

*Get the intersection of two list variables*


---

### Description

Get the intersection of two list variables

### Usage

```
ExprList_set_intersection(other)
```

**Arguments**

other                    Other list variable. Can be an Expr or something coercible to an Expr.

**Details**

Note that the datatypes inside the list must have a common supertype. For example, the first column can be `list[i32]` and the second one can be `list[i8]` because it can be cast to `list[i32]`. However, the second column cannot be e.g `list[f32]`.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = list(1:3, NA_integer_, c(NA_integer_, 3L), 5:7),
  b = list(2:4, 3L, c(3L, 4L, NA_integer_), c(6L, 8L))
)

df$with_columns(intersection = pl$col("a")$list$set_intersection("b"))
```

---

ExprList\_set\_symmetric\_difference

*Get the symmetric difference of two list variables*

---

**Description**

This returns all elements that are in only one of the two lists. To get only elements that are in the first list but not in the second one, use `$set_difference()`.

**Usage**

```
ExprList_set_symmetric_difference(other)
```

**Arguments**

other                    Other list variable. Can be an Expr or something coercible to an Expr.

**Details**

Note that the datatypes inside the list must have a common supertype. For example, the first column can be `list[i32]` and the second one can be `list[i8]` because it can be cast to `list[i32]`. However, the second column cannot be e.g `list[f32]`.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(  
  a = list(1:3, NA_integer_, c(NA_integer_, 3L), 5:7),  
  b = list(2:4, 3L, c(3L, 4L, NA_integer_), c(6L, 8L))  
)  
  
df$with_columns(  
  symmetric_difference = pl$col("a")$list$set_symmetric_difference("b")  
)
```

---

ExprList\_set\_union      *Get the union of two list variables*

---

**Description**

Get the union of two list variables

**Usage**

```
ExprList_set_union(other)
```

**Arguments**

other                      Other list variable. Can be an Expr or something coercible to an Expr.

**Details**

Note that the datatypes inside the list must have a common supertype. For example, the first column can be `list[i32]` and the second one can be `list[i8]` because it can be cast to `list[i32]`. However, the second column cannot be e.g `list[f32]`.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(  
  a = list(1:3, NA_integer_, c(NA_integer_, 3L), 5:7),  
  b = list(2:4, 3L, c(3L, 4L, NA_integer_), c(6L, 8L))  
)  
  
df$with_columns(union = pl$col("a")$list$set_union("b"))
```

---

ExprList_shift	<i>Shift list values by n indices</i>
----------------	---------------------------------------

---

**Description**

Shift list values by n indices

**Usage**

```
ExprList_shift(n = 1)
```

**Arguments**

n                      Number of indices to shift forward. If a negative value is passed, values are shifted in the opposite direction instead.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(  
  s = list(1:4, c(10L, 2L, 1L)),  
  idx = 1:2  
)  
df$with_columns(  
  shift_by_expr = pl$col("s")$list$shift(pl$col("idx")),  
  shift_by_lit = pl$col("s")$list$shift(2)  
)
```

---

ExprList_slice	<i>Slice list</i>
----------------	-------------------

---

**Description**

This extracts length values at most, starting at index offset. This can return less than length values if length is larger than the number of values.

**Usage**

```
ExprList_slice(offset, length = NULL)
```

**Arguments**

offset	Start index. Negative indexing is supported. Can be an Expr. Strings are parsed as column names.
length	Length of the slice. If NULL (default), the slice is taken to the end of the list. Can be an Expr. Strings are parsed as column names.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  s = list(1:4, c(10L, 2L, 1L)),
  idx_off = 1:2,
  len = c(4, 1)
)
df$with_columns(
  slice_by_expr = pl$col("s")$list$slice("idx_off", "len"),
  slice_by_lit = pl$col("s")$list$slice(2, 3)
)
```

ExprList\_sort

*Sort values in a list***Description**

Sort values in a list

**Usage**

ExprList\_sort(descending = FALSE)

**Arguments**

descending	Sort values in descending order
------------	---------------------------------

**Value**

Expr

**Examples**

```
df = pl$DataFrame(values = list(c(NA, 2, 1, 3), c(Inf, 2, 3, NaN), NA_real_))
df$with_columns(sort = pl$col("values")$list$sort())
```

---

ExprList_sum	<i>Sum all elements in a list</i>
--------------	-----------------------------------

---

**Description**

Sum all elements in a list

**Usage**

```
ExprList_sum()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA_real_))
df$with_columns(sum = pl$col("values")$list$sum())
```

---

ExprList_tail	<i>Get the last n values of a list</i>
---------------	--

---

**Description**

Get the last n values of a list

**Usage**

```
ExprList_tail(n = 5L)
```

**Arguments**

n                      Number of values to return for each sublist. Can be an Expr. Strings are parsed as column names.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  s = list(1:4, c(10L, 2L, 1L)),
  n = 1:2
)
df$with_columns(
  tail_by_expr = pl$col("s")$list$tail("n"),
  tail_by_lit = pl$col("s")$list$tail(2)
)
```

---

ExprList\_to\_struct      *Convert a Series of type List to Struct*

---

**Description**

Convert a Series of type List to Struct

**Usage**

```
ExprList_to_struct(
  n_field_strategy = c("first_non_null", "max_width"),
  fields = NULL,
  upper_bound = 0
)
```

**Arguments**

n_field_strategy	Strategy to determine the number of fields of the struct. If "first_non_null" (default), set number of fields equal to the length of the first non zero-length list. If "max_width", the number of fields is the maximum length of a list.
fields	If the name and number of the desired fields is known in advance, a list of field names can be given, which will be assigned by index. Otherwise, to dynamically assign field names, a custom R function that takes an R double and outputs a string value can be used. If NULL (default), fields will be field_0, field_1 ... field_n.
upper_bound	A LazyFrame needs to know the schema at all time. The caller therefore must provide an upper_bound of struct fields that will be set. If set incorrectly, downstream operation may fail. For instance an all()\$sum() expression will look in the current schema to determine which columns to select. When operating on a DataFrame, the schema does not need to be tracked or pre-determined, as the result will be eagerly evaluated, so you can leave this parameter unset.

**Value**

Expr



**Examples**

```
df = pl$DataFrame(list(a = list(1:2, 1:3)))

# this discards the third value of the second list as the struct length is
# determined based on the length of the first non-empty list
df$with_columns(
  struct = pl$col("a")$list$to_struct()
)

# we can use "max_width" to keep all values
df$with_columns(
  struct = pl$col("a")$list$to_struct(n_field_strategy = "max_width")
)

# pass a custom function that will name all fields by adding a prefix
df2 = df$with_columns(
  pl$col("a")$list$to_struct(
    fields = \(idx) paste0("col_", idx)
  )
)
df2

df2$unnest()

df2$to_list()
```

---

ExprList_unique	<i>Get unique values in a list</i>
-----------------	------------------------------------

---

**Description**

Get unique values in a list

**Usage**

```
ExprList_unique(maintain_order = FALSE)
```

**Arguments**

`maintain_order` Maintain order of data. This requires more work.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(values = list(c(2, 2, NA), c(1, 2, 3), NA_real_))
df$with_columns(unique = pl$col("values")$list$unique())
```

---

ExprMeta_eq	<i>Check if two expressions are equivalent</i>
-------------	--

---

**Description**

Indicate if this expression is the same as another expression. See also the counterpart `$meta$neq()`.

**Usage**

```
ExprMeta_eq(other)
```

**Arguments**

other	Expr to compare with
-------	----------------------

**Value**

A logical value

**Examples**

```
# three naive expression literals
e1 = pl$lit(40) + 2
e2 = pl$lit(42)
e3 = pl$lit(40) + 2

# e1 and e3 are identical expressions
e1$meta$eq(e3)

# when evaluated, e1 and e2 are equal
e1$eq(e2)$to_r()

# however, on the meta-level, e1 and e2 are NOT identical expressions
e1$meta$eq(e2)
```

---

ExprMeta_has_multiple_outputs	<i>Indicate if an expression has multiple outputs</i>
-------------------------------	---

---

**Description**

Indicate if an expression has multiple outputs

**Usage**

```
ExprMeta_has_multiple_outputs()
```

**Value**

Boolean

**Examples**

```
e = (pl$col("alice") + pl$col("eve"))$alias("bob")
e$meta$has_multiple_outputs()
```

```
# pl$all() select multiple cols to modify them, so it has multiple outputs
pl$all()$meta$has_multiple_outputs()
```

---

ExprMeta\_is\_regex\_projection

*Indicate if an expression uses a regex projection*

---

**Description**

Indicate if an expression uses a regex projection

**Usage**

```
ExprMeta_is_regex_projection()
```

**Value**

Boolean

**Examples**

```
pl$col("^Sepal.*$")$meta$is_regex_projection()
pl$col("Sepal.Length")$meta$is_regex_projection()
```

---

ExprMeta\_neq

*Check if two expressions are different*

---

**Description**

Indicate if this expression is different from another expression. See also the counterpart [\\$meta\\$eq\(\)](#).

**Usage**

```
ExprMeta_neq(other)
```

**Arguments**

other            Expr to compare with

**Value**

A logical value

**Examples**

```
# three naive expression literals
e1 = pl$lit(40) + 2
e2 = pl$lit(42)
e3 = pl$lit(40) + 2

# e1 and e3 are identical expressions
e1$meta$neq(e3)

# when evaluated, e1 and e2 are equal
e1$neq(e2)$to_r()

# however, on the meta-level, e1 and e2 are different
e1$meta$neq(e2)
```

---

ExprMeta\_output\_name *Get the column name that this expression would produce*

---

**Description**

It may not always be possible to determine the output name as that can depend on the schema of the context; in that case this will raise an error if `raise_if_undetermined` is `TRUE` (the default), or return `NA` otherwise.

**Usage**

```
ExprMeta_output_name(..., raise_if_undetermined = TRUE)
```

**Arguments**

... Ignored.

`raise_if_undetermined`  
If `TRUE` (default), raise an error if the output name cannot be determined. Otherwise, return `NA`.

**Value**

A character vector

**Examples**

```

e = pl$col("foo") * pl$col("bar")
e$meta$output_name()

e_filter = pl$col("foo")$filter(pl$col("bar") == 13)
e_filter$meta$output_name()

e_sum_over = pl$sum("foo")$over("groups")
e_sum_over$meta$output_name()

e_sum_slice = pl$sum("foo")$slice(pl$len() - 10, pl$col("bar"))
e_sum_slice$meta$output_name()

pl$len()$meta$output_name()

pl$col("*")$meta$output_name(raise_if_undetermined = FALSE)

```

---

ExprMeta_pop	<i>Pop</i>
--------------	------------

---

**Description**

Pop the latest expression and return the input(s) of the popped expression.

**Usage**

```
ExprMeta_pop()
```

**Value**

A list of expressions which in most cases will have a unit length. This is not the case when an expression has multiple inputs, for instance in a `$fold()` expression.

**Examples**

```

e1 = pl$lit(40) + 2
e2 = pl$lit(42)$sum()

e1
e1$meta$pop()

e2
e2$meta$pop()

```

---

ExprMeta\_root\_names    *Get the root column names*

---

### Description

This returns the names of input columns. Use `$meta$output_name()` to get the name of output column.

### Usage

```
ExprMeta_root_names()
```

### Value

A character vector

### Examples

```
e = (pl$col("alice") + pl$col("eve"))$alias("bob")
e$meta$root_names()
```

---

ExprMeta\_tree\_format    *Format an expression as a tree*

---

### Description

Format an expression as a tree

### Usage

```
ExprMeta_tree_format(return_as_string = FALSE)
```

### Arguments

`return_as_string`

Return the tree as a character vector? If FALSE (default), the tree is printed in the console.

### Value

If `return_as_string` is TRUE, a character vector describing the tree.

If `return_as_string` is FALSE, prints the tree in the console but doesn't return any value.

### Examples

```
my_expr = (pl$col("foo") * pl$col("bar"))$sum()$over(pl$col("ham")) / 2
my_expr$meta$tree_format()
```

---

ExprMeta\_undo\_aliases *Undo any renaming operation*

---

### Description

This removes any renaming operation like `$alias()` or `$name$keep()`. Polars uses the "leftmost rule" to determine naming, meaning that the first element of the expression will be used to name the output.

### Usage

```
ExprMeta_undo_aliases()
```

### Value

Expr with aliases undone

### Examples

```
e = (pl$col("alice") + pl$col("eve"))$alias("bob")
e$meta$output_name()
e$meta$undo_aliases()$meta$output_name()
```

---

ExprName\_keep *Keep the original root name of the expression.*

---

### Description

Keep the original root name of the expression.

### Usage

```
ExprName_keep()
```

### Value

Expr

### Examples

```
pl$DataFrame(list(alice = 1:3))$select(pl$col("alice")$alias("bob")$name$keep())
```

---

ExprName\_prefix      *Add a prefix to a column name*

---

**Description**

Add a prefix to a column name

**Usage**

```
ExprName_prefix(prefix)
```

**Arguments**

prefix      Prefix to be added to column name(s)

**Value**

Expr

**See Also**

[\\$suffix\(\)](#) to add a suffix

**Examples**

```
dat = pl$DataFrame(mtcars)

dat$select(
  pl$col("mpg"),
  pl$col("mpg")$name$prefix("name_"),
  pl$col("cyl", "drat")$name$prefix("bar_")
)
```

---

ExprName\_prefix\_fields      *Add a prefix to all fields name of a struct*

---

**Description**

Add a prefix to all fields name of a struct

**Usage**

```
ExprName_prefix_fields(prefix)
```



**Arguments**

prefix            Prefix to add to the field name.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(a = 1, b = 2)$select(  
  pl$struct(pl$all())$alias("my_struct")  
)  
  
df$with_columns(pl$col("my_struct")$name$prefix_fields("col_"))$unnest()
```

---

ExprName_suffix	<i>Add a suffix to a column name</i>
-----------------	--------------------------------------

---

**Description**

Add a suffix to a column name

**Usage**

```
ExprName_suffix(suffix)
```

**Arguments**

suffix            Suffix to be added to column name(s)

**Value**

Expr

**See Also**

[\\$prefix\(\)](#) to add a prefix

**Examples**

```
dat = pl$DataFrame(mtcars)  
  
dat$select(  
  pl$col("mpg"),  
  pl$col("mpg")$name$suffix("_foo"),  
  pl$col("cyl", "drat")$name$suffix("_bar")  
)
```

---

ExprName\_suffix\_fields

*Add a suffix to all fields name of a struct*

---

**Description**

Add a suffix to all fields name of a struct

**Usage**

```
ExprName_suffix_fields(suffix)
```

**Arguments**

suffix                    Suffix to add to the field name.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(a = 1, b = 2)$select(  
  pl$struct(pl$all())$alias("my_struct")  
)  
  
df$with_columns(pl$col("my_struct")$name$suffix_fields("_post"))$unnest()
```

---

ExprName\_to\_lowercase    *Make the root column name lowercase*

---

**Description**

Due to implementation constraints, this method can only be called as the last expression in a chain.

**Usage**

```
ExprName_to_lowercase()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(Alice = 1:3)$with_columns(pl$col("Alice")$name$to_lowercase())
```

---

ExprName\_to\_uppercase *Make the root column name uppercase*

---

**Description**

Due to implementation constraints, this method can only be called as the last expression in a chain.

**Usage**

```
ExprName_to_uppercase()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(Alice = 1:3)$with_columns(pl$col("Alice")$name$to_uppercase())
```

---

ExprStruct\_field *Retrieve one of the fields of this Struct as a new Series*

---

**Description**

Retrieve one of the fields of this Struct as a new Series

**Usage**

```
ExprStruct_field(name)
```

**Arguments**

name            Name of the field.

**Value**

Expr of datatype Struct

**Examples**

```
df = pl$DataFrame(
  aaa = c(1, 2),
  bbb = c("ab", "cd"),
  ccc = c(TRUE, NA),
  ddd = list(c(1, 2), 3)
)$select(
  pl$struct(pl$all())$alias("struct_col")
)
# struct field into a new Series
df$select(
  pl$col("struct_col")$struct$field("bbb"),
  pl$col("struct_col")$struct$field("ddd")
)
```

---

ExprStruct\_rename\_fields

*Rename the fields of the struct*


---

**Description**

Rename the fields of the struct

**Usage**

```
ExprStruct_rename_fields(names)
```

**Arguments**

names	Vector or list of strings given in the same order as the struct's fields. Providing fewer names will drop the latter fields. If too many names are given, the extra names are ignored.
-------	--

**Value**

Expr of datatype Struct

**Examples**

```
df = pl$DataFrame(
  aaa = 1:2,
  bbb = c("ab", "cd"),
  ccc = c(TRUE, NA),
  ddd = list(1:2, 3L)
)$select(
  pl$struct(pl$all())$alias("struct_col")
)$select(
  pl$col("struct_col")$struct$rename_fields(c("www", "xxx", "yyy", "zzz"))
)
df$unnest()
```

---

`ExprStruct_with_fields`*Add or overwrite fields of this struct*

---

## Description

This is similar to `$with_columns()` on `DataFrame`. Use `pl$field()` to quickly select a field in a `$struct$with_fields()` context.

## Usage

```
ExprStruct_with_fields(...)
```

## Arguments

... Field(s) to add. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals.

## Value

An `Expr` of data type `Struct`.

## Examples

```
df = pl$DataFrame(x = c(1, 4, 9), y = c(4, 9, 16), multiply = c(10, 2, 3))$  
  with_columns(coords = pl$struct(c("x", "y")))$  
  select("coords", "multiply")
```

```
df
```

```
df = df$with_columns(  
  pl$col("coords")$struct$with_fields(  
    pl$field("x")$sqrt(),  
    y_mul = pl$field("y") * pl$col("multiply")  
  )  
)
```

```
df
```

```
df$unnest("coords")
```

---

ExprStr\_contains      *Check if string contains a substring that matches a pattern*

---

### Description

Check if string contains a substring that matches a pattern

### Usage

```
ExprStr_contains(pattern, ..., literal = FALSE, strict = TRUE)
```

### Arguments

pattern	A character or something can be coerced to a string <a href="#">Expr</a> of a valid regex pattern, compatible with the <a href="#">regex crate</a> .
...	Ignored.
literal	Logical. If TRUE (default), treat pattern as a literal string, not as a regular expression.
strict	Logical. If TRUE (default), raise an error if the underlying pattern is not a valid regex, otherwise mask out with a null value.

### Details

To modify regular expression behaviour (such as case-sensitivity) with flags, use the inline `(?iLmsuxU)` syntax. See the [regex crate's](#) section on [grouping and flags](#) for additional information about the use of inline expression modifiers.

### Value

[Expr](#) of Boolean data type

### See Also

- [\\$str\\$start\\_with\(\)](#): Check if string values start with a substring.
- [\\$str\\$ends\\_with\(\)](#): Check if string values end with a substring.
- [\\$str\\$find\(\)](#): Return the index position of the first substring matching a pattern.

### Examples

```
# The inline `(?i)` syntax example
pl$DataFrame(s = c("AAA", "aAa", "aaa"))$with_columns(
  default_match = pl$col("s")$str$contains("AA"),
  insensitive_match = pl$col("s")$str$contains("(?i)AA")
)

df = pl$DataFrame(txt = c("Crab", "cat and dog", "rab$bit", NA))
df$with_columns(
```

```

  regex = pl$col("txt")$str$contains("cat|bit"),
  literal = pl$col("txt")$str$contains("rab$", literal = TRUE)
)

```

---

ExprStr\_contains\_any *Use the aho-corasick algorithm to find matches*

---

## Description

This function determines if any of the patterns find a match.

## Usage

```
ExprStr_contains_any(patterns, ..., ascii_case_insensitive = FALSE)
```

## Arguments

patterns	Character vector or something can be coerced to strings <a href="#">Expr</a> of a valid regex pattern, compatible with the <a href="#">regex crate</a> .
...	Ignored.
ascii_case_insensitive	Enable ASCII-aware case insensitive matching. When this option is enabled, searching will be performed without respect to case for ASCII letters (a-z and A-Z) only.

## Value

[Expr](#) of Boolean data type

## See Also

- [<Expr>\\$str\\$contains\(\)](#)

## Examples

```

df = pl$DataFrame(
  lyrics = c(
    "Everybody wants to rule the world",
    "Tell me what you want, what you really really want",
    "Can you feel the love tonight"
  )
)

df$with_columns(
  contains_any = pl$col("lyrics")$str$contains_any(c("you", "me"))
)

```

---

ExprStr\_count\_matches *Count all successive non-overlapping regex matches*

---

### Description

Count all successive non-overlapping regex matches

### Usage

```
ExprStr_count_matches(pattern, ..., literal = FALSE)
```

### Arguments

pattern	A character or something can be coerced to a string <a href="#">Expr</a> of a valid regex pattern, compatible with the <a href="#">regex crate</a> .
...	Ignored.
literal	Logical. If TRUE (default), treat pattern as a literal string, not as a regular expression.

### Value

[Expr](#) of data type UInt32. Returns null if the original value is null.

### Examples

```
df = pl$DataFrame(foo = c("12 dbc 3xy", "cat\\w", "1zy3\\d\\d", NA))

df$with_columns(
  count_digits = pl$col("foo")$str$count_matches(r"(\d)"),
  count_slash_d = pl$col("foo")$str$count_matches(r"(\d)", literal = TRUE)
)
```

---

ExprStr\_decode *Decode a value using the provided encoding*

---

### Description

Decode a value using the provided encoding

### Usage

```
ExprStr_decode(encoding, ..., strict = TRUE)
```



**Arguments**

encoding	Either 'hex' or 'base64'.
...	Not used currently.
strict	If TRUE (default), raise an error if the underlying value cannot be decoded. Otherwise, replace it with a null value.

**Value**

String array with values decoded using provided encoding

**Examples**

```
df = pl$DataFrame(strings = c("foo", "bar", NA))
df$select(pl$col("strings")$str$encode("hex"))
df$with_columns(
  pl$col("strings")$str$encode("base64")$alias("base64"), # notice DataType is not encoded
  pl$col("strings")$str$encode("hex")$alias("hex") # ... and must restored with cast
)$with_columns(
  pl$col("base64")$str$decode("base64")$alias("base64_decoded")$cast(pl$String),
  pl$col("hex")$str$decode("hex")$alias("hex_decoded")$cast(pl$String)
)
```

---

ExprStr_encode	<i>Encode a value using the provided encoding</i>
----------------	---

---

**Description**

Encode a value using the provided encoding

**Usage**

```
ExprStr_encode(encoding)
```

**Arguments**

encoding	Either 'hex' or 'base64'.
----------	---------------------------

**Value**

String array with values encoded using provided encoding

**Examples**

```
df = pl$DataFrame(strings = c("foo", "bar", NA))
df$select(pl$col("strings")$str$encode("hex"))
df$with_columns(
  pl$col("strings")$str$encode("base64")$alias("base64"), # notice DataType is not encoded
  pl$col("strings")$str$encode("hex")$alias("hex") # ... and must be restored with cast
)$with_columns(
  pl$col("base64")$str$decode("base64")$alias("base64_decoded")$cast(pl$String),
  pl$col("hex")$str$decode("hex")$alias("hex_decoded")$cast(pl$String)
)
```

---

ExprStr_ends_with	<i>Check if string ends with a regex</i>
-------------------	--

---

**Description**

Check if string values end with a substring.

**Usage**

```
ExprStr_ends_with(sub)
```

**Arguments**

sub	Suffix substring or Expr.
-----	---------------------------

**Details**

See also `$str$starts_with()` and `$str$contains()`.

**Value**

[Expr](#) of Boolean data type

**Examples**

```
df = pl$DataFrame(fruits = c("apple", "mango", NA))
df$select(
  pl$col("fruits"),
  pl$col("fruits")$str$ends_with("go")$alias("has_suffix")
)
```

---

ExprStr\_extract      *Extract the target capture group from provided patterns*

---

**Description**

Extract the target capture group from provided patterns

**Usage**

```
ExprStr_extract(pattern, group_index)
```

**Arguments**

pattern	A valid regex pattern. Can be an Expr or something coercible to an Expr. Strings are parsed as column names.
group_index	Index of the targeted capture group. Group 0 means the whole pattern, first group begin at index 1 (default).

**Value**

String array. Contains null if original value is null or regex capture nothing.

**Examples**

```
df = pl$DataFrame(
  a = c(
    "http://vote.com/ballon_dor?candidate=messi&ref=polars",
    "http://vote.com/ballon_dor?candidate=jorginho&ref=polars",
    "http://vote.com/ballon_dor?candidate=ronaldo&ref=polars"
  )
)
df$with_columns(
  extracted = pl$col("a")$str$extract(pl$lit(r"(candidate=(\w+))"), 1)
)
```

---

ExprStr\_extract\_all      *Extract all matches for the given regex pattern*

---

**Description**

Extracts all matches for the given regex pattern. Extracts each successive non-overlapping regex match in an individual string as an array.

**Usage**

```
ExprStr_extract_all(pattern)
```

**Arguments**

pattern            A valid regex pattern

**Value**

List[String] array. Contain null if original value is null or regex capture nothing.

**Examples**

```
df = pl$DataFrame(foo = c("123 bla 45 asd", "xyz 678 910t"))
df$select(
  pl$col("foo")$str$extract_all(r"((\d+))")$alias("extracted_nrs")
)
```

---

ExprStr\_extract\_groups

*Extract all capture groups for the given regex pattern*

---

**Description**

Extract all capture groups for the given regex pattern

**Usage**

```
ExprStr_extract_groups(pattern)
```

**Arguments**

pattern            A character of a valid regular expression pattern containing at least one capture group, compatible with the [regex crate](#).

**Details**

All group names are strings. If your pattern contains unnamed groups, their numerical position is converted to a string. See examples.

**Value**

Expr of data type [Struct](#) with fields of data type String.

**Examples**

```
df = pl$DataFrame(
  url = c(
    "http://vote.com/ballon_dor?candidate=messi&ref=python",
    "http://vote.com/ballon_dor?candidate=weghorst&ref=polars",
    "http://vote.com/ballon_dor?error=404&ref=rust"
  )
)
```

```

pattern = r"(candidate=(?<candidate>\w+)&ref=(?<ref>\w+))"

df$with_columns(
  captures = pl$col("url")$str$extract_groups(pattern)
)$unnest("captures")

# If the groups are unnamed, their numerical position (as a string) is used:

pattern = r"(candidate=(\w+)&ref=(\w+))"

df$with_columns(
  captures = pl$col("url")$str$extract_groups(pattern)
)$unnest("captures")

```

---

ExprStr\_extract\_many *Use the aho-corasick algorithm to extract matches*

---

### Description

Use the aho-corasick algorithm to extract matches

### Usage

```

ExprStr_extract_many(
  patterns,
  ...,
  ascii_case_insensitive = FALSE,
  overlapping = FALSE
)

```

### Arguments

patterns	String patterns to search. This can be an Expr or something coercible to an Expr. Strings are parsed as column names.
...	Ignored.
ascii_case_insensitive	Enable ASCII-aware case insensitive matching. When this option is enabled, searching will be performed without respect to case for ASCII letters (a-z and A-Z) only.
overlapping	Whether matches can overlap.

### Value

Expr: Series of dtype String.

**Examples**

```
df = pl$DataFrame(values = "discontent")
patterns = pl$lit(c("winter", "disco", "onte", "discontent"))

df$with_columns(
  matches = pl$col("values")$str$extract_many(patterns),
  matches_overlap = pl$col("values")$str$extract_many(patterns, overlapping = TRUE)
)

df = pl$DataFrame(
  values = c("discontent", "rhapsody"),
  patterns = list(c("winter", "disco", "onte", "discontent"), c("rhap", "ody", "coalesce"))
)

df$select(pl$col("values")$str$extract_many("patterns"))
```

---

**ExprStr\_find***Return the index position of the first substring matching a pattern*

---

**Description**

Return the index position of the first substring matching a pattern

**Usage**

```
ExprStr_find(pattern, ..., literal = FALSE, strict = TRUE)
```

**Arguments**

pattern	A character or something can be coerced to a string <a href="#">Expr</a> of a valid regex pattern, compatible with the <a href="#">regex crate</a> .
...	Ignored.
literal	Logical. If TRUE (default), treat pattern as a literal string, not as a regular expression.
strict	Logical. If TRUE (default), raise an error if the underlying pattern is not a valid regex, otherwise mask out with a null value.

**Details**

To modify regular expression behaviour (such as case-sensitivity) with flags, use the inline `(?iLmsuxU)` syntax. See the [regex crate's](#) section on [grouping and flags](#) for additional information about the use of inline expression modifiers.

**Value**

An Expr of data type UInt32

**See Also**

- `$str$start_with()`: Check if string values start with a substring.
- `$str$ends_with()`: Check if string values end with a substring.
- `$str$contains()`: Check if string contains a substring that matches a pattern.

**Examples**

```
pl$DataFrame(s = c("AAA", "aAa", "aaa"))$with_columns(
  default_match = pl$col("s")$str$find("Aa"),
  insensitive_match = pl$col("s")$str$find("(?i)Aa")
)
```

ExprStr\_head

*Return the first n characters of each string***Description**

Return the first n characters of each string

**Usage**

```
ExprStr_head(n)
```

**Arguments**

`n` Length of the slice (integer or expression). Strings are parsed as column names. Negative indexing is supported.

**Details**

The `n` input is defined in terms of the number of characters in the (UTF-8) string. A character is defined as a Unicode scalar value. A single character is represented by a single byte when working with ASCII text, and a maximum of 4 bytes otherwise.

When the `n` input is negative, `head()` returns characters up to the `nth` from the end of the string. For example, if `n = -3`, then all characters except the last three are returned.

If the length of the string has fewer than `n` characters, the full string is returned.

**Value**

Expr: Series of dtype String.

**Examples**

```
df = pl$DataFrame(
  s = c("pear", NA, "papaya", "dragonfruit"),
  n = c(3, 4, -2, -5)
)

df$with_columns(
  s_head_5 = pl$col("s")$str$head(5),
  s_head_n = pl$col("s")$str$head("n")
)
```

---

ExprStr_join	<i>Vertically concatenate the string values in the column to a single string value.</i>
--------------	---

---

**Description**

Vertically concatenate the string values in the column to a single string value.

**Usage**

```
ExprStr_join(delimiter = "", ..., ignore_nulls = TRUE)
```

**Arguments**

delimiter	The delimiter to insert between consecutive string values.
...	Ignored.
ignore_nulls	Ignore null values (default). If FALSE, null values will be propagated: if the column contains any null values, the output is null.

**Value**

Expr of String concatenated

**Examples**

```
# concatenate a Series of strings to a single string
df = pl$DataFrame(foo = c(1, NA, 2))

df$select(pl$col("foo")$str$join("-"))

df$select(pl$col("foo")$str$join("-", ignore_nulls = FALSE))
```



---

ExprStr\_json\_decode     *Parse string values as JSON.*

---

### Description

Parse string values as JSON.

### Usage

```
ExprStr_json_decode(dtype, infer_schema_length = 100)
```

### Arguments

`dtype`                     The dtype to cast the extracted value to. If NULL, the dtype will be inferred from the JSON value.

`infer_schema_length`        How many rows to parse to determine the schema. If NULL, all rows are used.

### Details

Throw errors if encounter invalid json strings.

### Value

Expr returning a struct

### Examples

```
df = pl$DataFrame(
  json_val = c('{\"a\":1, \"b\": true}', NA, '{\"a\":2, \"b\": false}')
)
dtype = pl$Struct(pl$Field("a", pl$Int64), pl$Field("b", pl$Boolean))
df$select(pl$col("json_val")$str$json_decode(dtype))
```

---

ExprStr\_json\_path\_match

*Extract the first match of JSON string with the provided JSONPath expression*

---

### Description

Extract the first match of JSON string with the provided JSONPath expression

### Usage

```
ExprStr_json_path_match(json_path)
```

**Arguments**

json\_path      A valid JSON path query string.

**Details**

Throw errors if encounter invalid JSON strings. All return value will be cast to String regardless of the original value.

Documentation on JSONPath standard can be found here: <https://goessner.net/articles/JsonPath/>.

**Value**

String array. Contain null if original value is null or the json\_path return nothing.

**Examples**

```
df = pl$DataFrame(
  json_val = c('{\"a\":1}', NA, '{\"a\":2}', '{\"a\":2.1}', '{\"a\":true}')
)
df$select(pl$col("json_val")$str$json_path_match("$.a"))
```

---

ExprStr\_len\_bytes      *Get the number of bytes in strings*

---

**Description**

Get length of the strings as UInt32 (as number of bytes). Use \$str\$len\_chars() to get the number of characters.

**Usage**

```
ExprStr_len_bytes()
```

**Details**

If you know that you are working with ASCII text, lengths will be equivalent, and faster (returns length in terms of the number of bytes).

**Value**

Expr of u32

**Examples**

```
pl$DataFrame(  
  s = c("Café", NA, "345", "æøå")  
)$select(  
  pl$col("s"),  
  pl$col("s")$str$len_bytes()$alias("lengths"),  
  pl$col("s")$str$len_chars()$alias("n_chars")  
)
```

---

ExprStr_len_chars	<i>Get the number of characters in strings</i>
-------------------	--

---

**Description**

Get length of the strings as UInt32 (as number of characters). Use `$str$len_bytes()` to get the number of bytes.

**Usage**

```
ExprStr_len_chars()
```

**Details**

If you know that you are working with ASCII text, lengths will be equivalent, and faster (returns length in terms of the number of bytes).

**Value**

Expr of u32

**Examples**

```
pl$DataFrame(  
  s = c("Café", NA, "345", "æøå")  
)$select(  
  pl$col("s"),  
  pl$col("s")$str$len_bytes()$alias("lengths"),  
  pl$col("s")$str$len_chars()$alias("n_chars")  
)
```

---

ExprStr\_pad\_end      *Left justify strings*

---

**Description**

Return the string left justified in a string of length width.

**Usage**

```
ExprStr_pad_end(width, fillchar = " ")
```

**Arguments**

width	Justify left to this length.
fillchar	Fill with this ASCII character.

**Details**

Padding is done using the specified fillchar. The original string is returned if width is less than or equal to len(s).

**Value**

Expr of String

**Examples**

```
df = pl$DataFrame(a = c("cow", "monkey", NA, "hippopotamus"))
df$select(pl$col("a")$str$pad_end(8, "*"))
```

---

ExprStr\_pad\_start      *Right justify strings*

---

**Description**

Return the string right justified in a string of length width.

**Usage**

```
ExprStr_pad_start(width, fillchar = " ")
```

**Arguments**

width	Justify right to this length.
fillchar	Fill with this ASCII character.

**Details**

Padding is done using the specified fillchar. The original string is returned if width is less than or equal to len(s).

**Value**

Expr of String

**Examples**

```
df = pl$DataFrame(a = c("cow", "monkey", NA, "hippopotamus"))
df$select(pl$col("a")$str$pad_start(8, "*"))
```

---

ExprStr_replace	<i>Replace first matching regex/literal substring with a new string value</i>
-----------------	---

---

**Description**

Replace first matching regex/literal substring with a new string value

**Usage**

```
ExprStr_replace(pattern, value, ..., literal = FALSE, n = 1L)
```

**Arguments**

pattern	A character or something can be coerced to a string <a href="#">Expr</a> of a valid regex pattern, compatible with the <a href="#">regex crate</a> .
value	A character or an <a href="#">Expr</a> of string that will replace the matched substring.
...	Ignored.
literal	Logical. If TRUE (default), treat pattern as a literal string, not as a regular expression.
n	A number of matches to replace. Note that regex replacement with $n > 1$ not yet supported, so raise an error if $n > 1$ and pattern includes regex pattern and <code>literal = FALSE</code> .

**Details**

To modify regular expression behaviour (such as case-sensitivity) with flags, use the inline (`?iLmsuxU`) syntax. See the [regex crate's](#) section on [grouping and flags](#) for additional information about the use of inline expression modifiers.

**Value**

[Expr](#) of String type

## Capture groups

The dollar sign (\$) is a special character related to capture groups. To refer to a literal dollar sign, use \$\$ instead or set `literal` to TRUE.

## See Also

- `<Expr>$str$replace_all()`

## Examples

```
df = pl$DataFrame(id = 1L:2L, text = c("123abc", "abc456"))
df$with_columns(pl$col("text")$str$replace(r"(abc\b)", "ABC"))

# Capture groups are supported.
# Use `${1}` in the value string to refer to the first capture group in the pattern,
# `${2}` to refer to the second capture group, and so on.
# You can also use named capture groups.
df = pl$DataFrame(word = c("hat", "hut"))
df$with_columns(
  positional = pl$col("word")$str$replace("h(.)t", "b${1}d"),
  named = pl$col("word")$str$replace("h(?<vowel>.)t", "b${vowel}d")
)

# Apply case-insensitive string replacement using the `(?i)` flag.
df = pl$DataFrame(
  city = "Philadelphia",
  season = c("Spring", "Summer", "Autumn", "Winter"),
  weather = c("Rainy", "Sunny", "Cloudy", "Snowy")
)
df$with_columns(
  pl$col("weather")$str$replace("(?i)foggy|rainy|cloudy|snowy", "Sunny")
)
```

---

ExprStr\_replace\_all    *Replace all matching regex/literal substrings with a new string value*

---

## Description

Replace all matching regex/literal substrings with a new string value

## Usage

```
ExprStr_replace_all(pattern, value, ..., literal = FALSE)
```

**Arguments**

pattern	A character or something can be coerced to a string <a href="#">Expr</a> of a valid regex pattern, compatible with the <a href="#">regex crate</a> .
value	A character or an <a href="#">Expr</a> of string that will replace the matched substring.
...	Ignored.
literal	Logical. If TRUE (default), treat pattern as a literal string, not as a regular expression.

**Details**

To modify regular expression behaviour (such as case-sensitivity) with flags, use the inline (`?iLmsuxU`) syntax. See the [regex crate's](#) section on [grouping and flags](#) for additional information about the use of inline expression modifiers.

**Value**

[Expr](#) of String type

**Capture groups**

The dollar sign (\$) is a special character related to capture groups. To refer to a literal dollar sign, use \$\$ instead or set `literal` to TRUE.

**See Also**

- [<Expr>\\$str\\$replace\(\)](#)

**Examples**

```
df = pl$DataFrame(id = 1L:2L, text = c("abcabc", "123a123"))
df$with_columns(pl$col("text")$str$replace_all("a", "-"))

# Capture groups are supported.
# Use `${1}` in the value string to refer to the first capture group in the pattern,
# `${2}` to refer to the second capture group, and so on.
# You can also use named capture groups.
df = pl$DataFrame(word = c("hat", "hut"))
df$with_columns(
  positional = pl$col("word")$str$replace_all("h(.)t", "b${1}d"),
  named = pl$col("word")$str$replace_all("h(?<vowel>.)t", "b${vowel}d")
)

# Apply case-insensitive string replacement using the `(?i)` flag.
df = pl$DataFrame(
  city = "Philadelphia",
  season = c("Spring", "Summer", "Autumn", "Winter"),
  weather = c("Rainy", "Sunny", "Cloudy", "Snowy")
)
df$with_columns(
  pl$col("weather")$str$replace_all(
```

```

    "(?i)foggy|rainy|cloudy|snowy", "Sunny"
  )
)

```

---

ExprStr\_replace\_many *Use the aho-corasick algorithm to replace many matches*

---

## Description

This function replaces several matches at once.

## Usage

```
ExprStr_replace_many(patterns, replace_with, ascii_case_insensitive = FALSE)
```

## Arguments

patterns	String patterns to search. Can be an Expr.
replace_with	A vector of strings used as replacements. If this is of length 1, then it is applied to all matches. Otherwise, it must be of same length as the patterns argument.
ascii_case_insensitive	Enable ASCII-aware case insensitive matching. When this option is enabled, searching will be performed without respect to case for ASCII letters (a-z and A-Z) only.

## Value

Expr

## Examples

```

df = pl$DataFrame(
  lyrics = c(
    "Everybody wants to rule the world",
    "Tell me what you want, what you really really want",
    "Can you feel the love tonight"
  )
)

# a replacement of length 1 is applied to all matches
df$with_columns(
  remove_pronouns = pl$col("lyrics")$str$replace_many(c("you", "me"), "")
)

# if there are more than one replacement, the patterns and replacements are
# matched
df$with_columns(
  fake_pronouns = pl$col("lyrics")$str$replace_many(c("you", "me"), c("foo", "bar"))
)

```



---

ExprStr_reverse	Returns string values in reversed order
-----------------	---

---

**Description**

Returns string values in reversed order

**Usage**

```
ExprStr_reverse()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(text = c("foo", "bar", NA))
df$with_columns(reversed = pl$col("text")$str$reverse())
```

---

ExprStr_slice	Create subslices of the string values of a String Series
---------------	--

---

**Description**

Create subslices of the string values of a String Series

**Usage**

```
ExprStr_slice(offset, length = NULL)
```

**Arguments**

offset	Start index. Negative indexing is supported.
length	Length of the slice. If NULL (default), the slice is taken to the end of the string.

**Value**

Expr: Series of dtype String.

**Examples**

```
df = pl$DataFrame(s = c("pear", NA, "papaya", "dragonfruit"))
df$with_columns(
  pl$col("s")$str$slice(-3)$alias("s_sliced")
)
```

---

ExprStr\_split      *Split the string by a substring*

---

**Description**

Split the string by a substring

**Usage**

```
ExprStr_split(by, inclusive = FALSE)
```

**Arguments**

**by**                      Substring to split by. Can be an Expr.  
**inclusive**            If TRUE, include the split character/string in the results.

**Value**

List of String type

**Examples**

```
df = pl$DataFrame(s = c("foo bar", "foo-bar", "foo bar baz"))
df$select(pl$col("s")$str$split(by = " "))

df = pl$DataFrame(
  s = c("foo^bar", "foo_bar", "foo*bar*baz"),
  by = c("_", "-", "*")
)
df
df$select(pl$col("s")$str$split(by = pl$col("by"))$alias("split"))
```

---

ExprStr\_splitn      *Split the string by a substring, restricted to returning at most n items*

---

**Description**

If the number of possible splits is less than n-1, the remaining field elements will be null. If the number of possible splits is n-1 or greater, the last (nth) substring will contain the remainder of the string.

**Usage**

```
ExprStr_splitn(by, n)
```

**Arguments**

by                    Substring to split by.  
n                     Number of splits to make.

**Value**

Struct where each of n fields is of String type

**Examples**

```
df = pl$DataFrame(s = c("a_1", NA, "c", "d_4_e"))
df$with_columns(
  s1 = pl$col("s")$str$splitn(by = "_", 1),
  s2 = pl$col("s")$str$splitn(by = "_", 2),
  s3 = pl$col("s")$str$splitn(by = "_", 3)
)
```

---

ExprStr\_split\_exact    *Split the string by a substring using n splits*

---

**Description**

This results in a struct of n+1 fields. If it cannot make n splits, the remaining field elements will be null.

**Usage**

```
ExprStr_split_exact(by, n, inclusive = FALSE)
```

**Arguments**

by                    Substring to split by.  
n                     Number of splits to make.  
inclusive            If TRUE, include the split character/string in the results.

**Value**

Struct where each of n+1 fields is of String type

**Examples**

```
df = pl$DataFrame(s = c("a_1", NA, "c", "d_4"))
df$with_columns(
  split = pl$col("s")$str$split_exact(by = "_", 1),
  split_inclusive = pl$col("s")$str$split_exact(by = "_", 1, inclusive = TRUE)
)
```

---

ExprStr\_starts\_with    *Check if string starts with a regex*

---

**Description**

Check if string values starts with a substring.

**Usage**

```
ExprStr_starts_with(sub)
```

**Arguments**

sub                    Prefix substring or Expr.

**Details**

See also `$str$contains()` and `$str$ends_with()`.

**Value**

[Expr](#) of Boolean data type

**Examples**

```
df = pl$DataFrame(fruits = c("apple", "mango", NA))
df$select(
  pl$col("fruits"),
  pl$col("fruits")$str$starts_with("app")$alias("has_suffix")
)
```

---

ExprStr\_strip\_chars    *Strip leading and trailing characters*

---

**Description**

Remove leading and trailing characters.

**Usage**

```
ExprStr_strip_chars(matches = NULL)
```

**Arguments**

matches                The set of characters to be removed. All combinations of this set of characters will be stripped. If NULL (default), all whitespace is removed instead. This can be an Expr.

**Details**

This function will not strip any chars beyond the first char not matched. `strip_chars()` removes characters at the beginning and the end of the string. Use `strip_chars_start()` and `strip_chars_end()` to remove characters only from left and right respectively.

**Value**

Expr of String lowercase chars

**Examples**

```
df = pl$DataFrame(foo = c(" hello", "\tworld"))
df$select(pl$col("foo")$str$strip_chars())
df$select(pl$col("foo")$str$strip_chars(" hel rld"))
```

---

ExprStr\_strip\_chars\_end

*Strip trailing characters*

---

**Description**

Remove trailing characters.

**Usage**

```
ExprStr_strip_chars_end(matches = NULL)
```

**Arguments**

matches	The set of characters to be removed. All combinations of this set of characters will be stripped. If NULL (default), all whitespace is removed instead. This can be an Expr.
---------	--

**Details**

This function will not strip any chars beyond the first char not matched. `strip_chars_end()` removes characters at the end of the string only. Use `strip_chars()` and `strip_chars_start()` to remove characters from the left and right or only from the left respectively.

**Value**

Expr of String lowercase chars

**Examples**

```
df = pl$DataFrame(foo = c(" hello", "\tworld"))
df$select(pl$col("foo")$str$strip_chars_end(" hel\trld"))
df$select(pl$col("foo")$str$strip_chars_end("rldhel\t "))
```

---

 ExprStr\_strip\_chars\_start

*Strip leading characters*


---

### Description

Remove leading characters.

### Usage

```
ExprStr_strip_chars_start(matches = NULL)
```

### Arguments

matches	The set of characters to be removed. All combinations of this set of characters will be stripped. If NULL (default), all whitespace is removed instead. This can be an Expr.
---------	--

### Details

This function will not strip any chars beyond the first char not matched. `strip_chars_start()` removes characters at the beginning of the string only. Use `strip_chars()` and `strip_chars_end()` to remove characters from the left and right or only from the right respectively.

### Value

Expr of String lowercase chars

### Examples

```
df = pl$DataFrame(foo = c(" hello", "\tworld"))
df$select(pl$col("foo")$str$strip_chars_start(" hel rld"))
```

---

 ExprStr\_strptime

*Convert a String column into a Date/Datetime/Time column.*


---

### Description

Similar to the [strptime\(\)](#) function.

**Usage**

```
ExprStr_strptime(
  dtype,
  format = NULL,
  ...,
  strict = TRUE,
  exact = TRUE,
  cache = TRUE,
  ambiguous = "raise"
)
```

**Arguments**

<code>dtype</code>	The data type to convert into. Can be either <code>pl\$Date</code> , <code>pl\$Datetime()</code> , or <code>pl\$Time</code> .
<code>format</code>	Format to use for conversion. Refer to <a href="#">the chrono crate documentation</a> for the full specification. Example: <code>"%Y-%m-%d %H:%M:%S"</code> . If <code>NULL</code> (default), the format is inferred from the data. Notice that time zone <code>%Z</code> is not supported and will just ignore timezones. Numeric time zones like <code>%z</code> or <code>:%:z</code> are supported.
<code>...</code>	Not used.
<code>strict</code>	If <code>TRUE</code> (default), raise an error if a single string cannot be parsed. If <code>FALSE</code> , produce a polars null.
<code>exact</code>	If <code>TRUE</code> (default), require an exact format match. If <code>FALSE</code> , allow the format to match anywhere in the target string. Conversion to the Time type is always exact. Note that using <code>exact = FALSE</code> introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.
<code>cache</code>	Use a cache of unique, converted dates to apply the datetime conversion.
<code>ambiguous</code>	Determine how to deal with ambiguous datetimes: <ul style="list-style-type: none"> <li>• <code>"raise"</code> (default): throw an error</li> <li>• <code>"earliest"</code>: use the earliest datetime</li> <li>• <code>"latest"</code>: use the latest datetime</li> <li>• <code>"null"</code>: return a null value</li> </ul>

**Details**

When parsing a Datetime the column precision will be inferred from the format string, if given, e.g.: `"%F %T%.3f" => pl$Datetime("ms")`. If no fractional second component is found then the default is `"us"` (microsecond).

**Value**

`Expr` of Date, Datetime or Time type

**See Also**

- `<Expr>$str$to_date()`
- `<Expr>$str$to_datetime()`
- `<Expr>$str$to_time()`

**Examples**

```
# Dealing with a consistent format
s = as_polars_series(c("2020-01-01 01:00Z", "2020-01-01 02:00Z"))

s$str$strptime(pl$Datetime(), "%Y-%m-%d %H:%M%#z")

# Auto infer format
s$str$strptime(pl$Datetime())

# Datetime with timezone is interpreted as UTC timezone
as_polars_series("2020-01-01T01:00:00+09:00")$str$strptime(pl$Datetime())

# Dealing with different formats.
s = as_polars_series(
  c(
    "2021-04-22",
    "2022-01-04 00:00:00",
    "01/31/22",
    "Sun Jul 8 00:34:60 2001"
  ),
  "date"
)

s$to_frame()$select(
  pl$coalesce(
    pl$col("date")$str$strptime(pl$Date, "%F", strict = FALSE),
    pl$col("date")$str$strptime(pl$Date, "%F %T", strict = FALSE),
    pl$col("date")$str$strptime(pl$Date, "%D", strict = FALSE),
    pl$col("date")$str$strptime(pl$Date, "%c", strict = FALSE)
  )
)

# Ignore invalid time
s = as_polars_series(
  c(
    "2023-01-01 11:22:33 -0100",
    "2023-01-01 11:22:33 +0300",
    "invalid time"
  )
)

s$str$strptime(
  pl$Datetime("ns"),
  format = "%Y-%m-%d %H:%M:%S %z",
  strict = FALSE
)
```



```
)
```

---

ExprStr_tail	<i>Return the last n characters of each string</i>
--------------	--

---

### Description

Return the last n characters of each string

### Usage

```
ExprStr_tail(n)
```

### Arguments

n                    Length of the slice (integer or expression). Strings are parsed as column names. Negative indexing is supported.

### Details

The n input is defined in terms of the number of characters in the (UTF-8) string. A character is defined as a Unicode scalar value. A single character is represented by a single byte when working with ASCII text, and a maximum of 4 bytes otherwise.

When the n input is negative, tail() returns characters starting from the nth from the beginning of the string. For example, if n = -3, then all characters except the first three are returned.

If the length of the string has fewer than n characters, the full string is returned.

### Value

Expr: Series of dtype String.

### Examples

```
df = pl$DataFrame(  
  s = c("pear", NA, "papaya", "dragonfruit"),  
  n = c(3, 4, -2, -5)  
)  
  
df$with_columns(  
  s_tail_5 = pl$col("s")$str$tail(5),  
  s_tail_n = pl$col("s")$str$tail("n")  
)
```

---

ExprStr\_to\_date      *Convert a String column into a Date column*

---

### Description

Convert a String column into a Date column

### Usage

```
ExprStr_to_date(format = NULL, ..., strict = TRUE, exact = TRUE, cache = TRUE)
```

### Arguments

format	Format to use for conversion. Refer to <a href="#">the chrono crate documentation</a> for the full specification. Example: "%Y-%m-%d %H:%M:%S". If NULL (default), the format is inferred from the data. Notice that time zone %Z is not supported and will just ignore timezones. Numeric time zones like %z or %:z are supported.
...	Not used.
strict	If TRUE (default), raise an error if a single string cannot be parsed. If FALSE, produce a polars null.
exact	If TRUE (default), require an exact format match. If FALSE, allow the format to match anywhere in the target string. Conversion to the Time type is always exact. Note that using exact = FALSE introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.
cache	Use a cache of unique, converted dates to apply the datetime conversion.

### Value

[Expr](#) of Date type

### See Also

- [<Expr>\\$str\\$strptime\(\)](#)

### Examples

```
s = as_polars_series(c("2020/01/01", "2020/02/01", "2020/03/01"))
s$str_to_date()

# by default, this errors if some values cannot be converted
s = as_polars_series(c("2020/01/01", "2020 02 01", "2020-03-01"))
try(s$str_to_date())
s$str_to_date(strict = FALSE)
```

---

ExprStr\_to\_datetime     *Convert a String column into a Datetime column*

---

## Description

Convert a String column into a Datetime column

## Usage

```
ExprStr_to_datetime(
  format = NULL,
  ...,
  time_unit = NULL,
  time_zone = NULL,
  strict = TRUE,
  exact = TRUE,
  cache = TRUE,
  ambiguous = "raise"
)
```

## Arguments

format	Format to use for conversion. Refer to <a href="#">the chrono crate documentation</a> for the full specification. Example: "%Y-%m-%d %H:%M:%S". If NULL (default), the format is inferred from the data. Notice that time zone %Z is not supported and will just ignore timezones. Numeric time zones like %z or %:z are supported.
...	Not used.
time_unit	Unit of time for the resulting Datetime column. If NULL (default), the time unit is inferred from the format string if given, e.g.: "%F %T%.3f" => <code>pl\$Datetime("ms")</code> . If no fractional second component is found, the default is "us" (microsecond).
time_zone	for the resulting <a href="#">Datetime</a> column.
strict	If TRUE (default), raise an error if a single string cannot be parsed. If FALSE, produce a polars null.
exact	If TRUE (default), require an exact format match. If FALSE, allow the format to match anywhere in the target string. Note that using exact = FALSE introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.
cache	Use a cache of unique, converted dates to apply the datetime conversion.
ambiguous	Determine how to deal with ambiguous datetimes: <ul style="list-style-type: none"> <li>• "raise" (default): throw an error</li> <li>• "earliest": use the earliest datetime</li> <li>• "latest": use the latest datetime</li> <li>• "null": return a null value</li> </ul>

**Value**

Expr of Datetime type

**See Also**

- `<Expr>$str$strptime()`

**Examples**

```
s = as_polars_series(c("2020-01-01 01:00Z", "2020-01-01 02:00Z"))

s$str$to_datetime("%Y-%m-%d %H:%M%#Z")
s$str$to_datetime(time_unit = "ms")
```

---

ExprStr\_to\_integer      *Convert a String column into an Int64 column with base radix*

---

**Description**

Convert a String column into an Int64 column with base radix

**Usage**

```
ExprStr_to_integer(..., base = 10L, strict = TRUE)
```

**Arguments**

...	Ignored.
base	A positive integer or expression which is the base of the string we are parsing. Characters are parsed as column names. Default: 10L.
strict	A logical. If TRUE (default), parsing errors or integer overflow will raise an error. If FALSE, silently convert to null.

**Value**

Expression of data type Int64.

**Examples**

```
df = pl$DataFrame(bin = c("110", "101", "010", "invalid"))
df$with_columns(
  parsed = pl$col("bin")$str$to_integer(base = 2, strict = FALSE)
)

df = pl$DataFrame(hex = c("fa1e", "ff00", "cafe", NA))
df$with_columns(
  parsed = pl$col("hex")$str$to_integer(base = 16, strict = TRUE)
)
```

---

ExprStr\_to\_lowercase    *Convert a string to lowercase*

---

**Description**

Transform to lowercase variant.

**Usage**

```
ExprStr_to_lowercase()
```

**Value**

Expr of String lowercase chars

**Examples**

```
pl$lit(c("A", "b", "c", "1", NA))$str$to_lowercase()$to_series()
```

---

ExprStr\_to\_time            *Convert a String column into a Time column*

---

**Description**

Convert a String column into a Time column

**Usage**

```
ExprStr_to_time(format = NULL, ..., strict = TRUE, cache = TRUE)
```

**Arguments**

format	Format to use for conversion. Refer to <a href="#">the chrono crate documentation</a> for the full specification. Example: "%Y-%m-%d %H:%M:%S". If NULL (default), the format is inferred from the data. Notice that time zone %Z is not supported and will just ignore timezones. Numeric time zones like %z or %:z are supported.
...	Not used.
strict	If TRUE (default), raise an error if a single string cannot be parsed. If FALSE, produce a polars null.
cache	Use a cache of unique, converted dates to apply the datetime conversion.

**Value**

[Expr](#) of Time type

**See Also**

- [<Expr>\\$str\\$strptime\(\)](#)

**Examples**

```
s = as_polars_series(c("01:00", "02:00", "03:00"))
s$str$to_time("%H:%M")
```

---

ExprStr\_to\_titlecase    *Convert a string to titlecase*

---

**Description**

Transform to titlecase variant.

**Usage**

```
ExprStr_to_titlecase()
```

**Details**

This method is only available with the "nightly" feature. See [polars\\_info\(\)](#) for more details.

**Value**

Expr of String titlecase chars

**Examples**

```
pl$lit(c("hello there", "HI, THERE", NA))$str$to_titlecase()$to_series()
```

---

ExprStr\_to\_uppercase    *Convert a string to uppercase*

---

**Description**

Transform to uppercase variant.

**Usage**

```
ExprStr_to_uppercase()
```

**Value**

Expr of String uppercase chars

**Examples**

```
pl$lit(c("A", "b", "c", "1", NA))$str$to_uppercase()$to_series()
```

---

ExprStr_zfill	<i>Fills the string with zeroes.</i>
---------------	--------------------------------------

---

**Description**

Add zeroes to a string until it reaches `n` characters. If the number of characters is already greater than `n`, the string is not modified.

**Usage**

```
ExprStr_zfill(alignment)
```

**Arguments**

<code>alignment</code>	Fill the value up to this length. This can be an Expr or something coercible to an Expr. Strings are parsed as column names.
------------------------	--

**Details**

Return a copy of the string left filled with ASCII '0' digits to make a string of length `width`.

A leading sign prefix ('+'/'-') is handled by inserting the padding after the sign character rather than before. The original string is returned if `width` is less than or equal to `len(s)`.

**Value**

Expr

**Examples**

```
some_floats_expr = pl$lit(c(0, 10, -5, 5))

# cast to String and ljust alignment = 5, and view as R char vector
some_floats_expr$cast(pl$String)$str$zfill(5)$to_r()

# cast to int and the to utf8 and then ljust alignment = 5, and view as R
# char vector
some_floats_expr$cast(pl$Int64)$cast(pl$String)$str$zfill(5)$to_r()
```

---

Expr_abs	<i>Compute the absolute values</i>
----------	------------------------------------

---

**Description**

Compute the absolute values

**Usage**

```
Expr_abs()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = -1:1)$  
  with_columns(abs = pl$col("a")$abs())
```

---

Expr_add	<i>Add two expressions</i>
----------	----------------------------

---

**Description**

Method equivalent of addition operator `expr + other`.

**Usage**

```
Expr_add(other)
```

**Arguments**

`other` numeric or string value; accepts expression input.

**Value**

Expr

**See Also**

- [Arithmetic operators](#)



**Examples**

```
df = pl$DataFrame(x = 1:5)

df$with_columns(
  `x+int` = pl$col("x")$add(2L),
  `x+expr` = pl$col("x")$add(pl$col("x")$cum_prod())
)

df = pl$DataFrame(
  x = c("a", "d", "g"),
  y = c("b", "e", "h"),
  z = c("c", "f", "i")
)

df$with_columns(
  pl$col("x")$add(pl$col("y"))$add(pl$col("z"))$alias("xyz")
)
```

Expr\_agg\_groups

*Aggregate groups***Description**

Get the group indexes of the group by operation. Should be used in aggregation context only.

**Usage**

```
Expr_agg_groups()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(list(
  group = c("one", "one", "one", "two", "two"),
  value = c(94, 95, 96, 97, 99)
))
df$group_by("group", maintain_order = TRUE)$agg(pl$col("value")$agg_groups())
```

---

Expr_alias	<i>Rename Expr output</i>
------------	---------------------------

---

**Description**

Rename the output of an expression.

**Usage**

```
Expr_alias(name)
```

**Arguments**

name	New name of output
------	--------------------

**Value**

Expr

**Examples**

```
pl$col("bob")$alias("alice")
```

---

Expr_all	<i>Apply logical AND on a column</i>
----------	--------------------------------------

---

**Description**

Check if all values in a Boolean column are TRUE. This method is an expression - not to be confused with `pl$all()` which is a function to select all columns.

**Usage**

```
Expr_all(..., ignore_nulls = TRUE)
```

**Arguments**

...	Ignored.
ignore_nulls	If TRUE (default), ignore null values. If FALSE, <b>Kleene logic</b> is used to deal with nulls: if the column contains any null values and no TRUE values, the output is null.

**Value**

A logical value

**Examples**

```
df = pl$DataFrame(  
  a = c(TRUE, TRUE),  
  b = c(TRUE, FALSE),  
  c = c(NA, TRUE),  
  d = c(NA, NA)  
)  
  
# By default, ignore null values. If there are only nulls, then all() returns  
# TRUE.  
df$select(pl$col("*")$all())  
  
# If we set ignore_nulls = FALSE, then we don't know if all values in column  
# "c" are TRUE, so it returns null  
df$select(pl$col("*")$all(ignore_nulls = FALSE))
```

---

Expr\_and

*Apply logical AND on two expressions*

---

**Description**

Combine two boolean expressions with AND.

**Usage**

```
Expr_and(other)
```

**Arguments**

other            numeric or string value; accepts expression input.

**Value**

Expr

**Examples**

```
pl$lit(TRUE) & TRUE  
pl$lit(TRUE)$and(pl$lit(TRUE))
```

---

Expr_any	<i>Apply logical OR on a column</i>
----------	-------------------------------------

---

**Description**

Check if any boolean value in a Boolean column is TRUE.

**Usage**

```
Expr_any(..., ignore_nulls = TRUE)
```

**Arguments**

...	Ignored.
ignore_nulls	If TRUE (default), ignore null values. If FALSE, <b>Kleene logic</b> is used to deal with nulls: if the column contains any null values and no TRUE values, the output is null.

**Value**

A logical value

**Examples**

```
df = pl$DataFrame(
  a = c(TRUE, FALSE),
  b = c(FALSE, FALSE),
  c = c(NA, FALSE)
)

df$select(pl$col("*")$any())

# If we set ignore_nulls = FALSE, then we don't know if any values in column
# "c" is TRUE, so it returns null
df$select(pl$col("*")$any(ignore_nulls = FALSE))
```

---

Expr_append	<i>Append expressions</i>
-------------	---------------------------

---

**Description**

This is done by adding the chunks of other to this output.

**Usage**

```
Expr_append(other, upcast = TRUE)
```

**Arguments**

other	Expr or something coercible to an Expr.
upcast	Cast both Expr to a common supertype if they have one.

**Value**

Expr

**Examples**

```
# append bottom to to row
df = pl$DataFrame(list(a = 1:3, b = c(NA_real_, 4, 5)))
df$select(pl$all()$head(1)$append(pl$all()$tail(1)))

# implicit upcast, when default = TRUE
pl$DataFrame(list())$select(pl$lit(42)$append(42L))
pl$DataFrame(list())$select(pl$lit(42)$append(FALSE))
pl$DataFrame(list())$select(pl$lit("Bob")$append(FALSE))
```

---

Expr\_approx\_n\_unique    *Approx count unique values*

---

**Description**

This is done using the HyperLogLog++ algorithm for cardinality estimation.

**Usage**

```
Expr_approx_n_unique()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(iris[, 4:5])$
  with_columns(count = pl$col("Species")$approx_n_unique())
```

---

Expr\_arccos

*Compute inverse cosine*

---

**Description**

Compute inverse cosine

**Usage**

Expr\_arccos()

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(-1, cos(0.5), 0, 1, NA_real_))$  
  with_columns(arccos = pl$col("a")$arccos())
```

---

Expr\_arccosh

*Compute inverse hyperbolic cosine*

---

**Description**

Compute inverse hyperbolic cosine

**Usage**

Expr\_arccosh()

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(-1, cosh(0.5), 0, 1, NA_real_))$  
  with_columns(arccosh = pl$col("a")$arccosh())
```

---

Expr_arcsin	<i>Compute inverse sine</i>
-------------	-----------------------------

---

**Description**

Compute inverse sine

**Usage**

```
Expr_arcsin()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(-1, sin(0.5), 0, 1, NA_real_))$  
  with_columns(arcsin = pl$col("a")$arcsin())
```

---

Expr_arcsinh	<i>Compute inverse hyperbolic sine</i>
--------------	--

---

**Description**

Compute inverse hyperbolic sine

**Usage**

```
Expr_arcsinh()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(-1, sinh(0.5), 0, 1, NA_real_))$  
  with_columns(arcsinh = pl$col("a")$arcsinh())
```

---

Expr_arctan	<i>Compute inverse tangent</i>
-------------	--------------------------------

---

**Description**

Compute inverse tangent

**Usage**

```
Expr_arctan()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(-1, tan(0.5), 0, 1, NA_real_))$  
  with_columns(arctan = pl$col("a")$arctan())
```

---

Expr_arctanh	<i>Compute inverse hyperbolic tangent</i>
--------------	---

---

**Description**

Compute inverse hyperbolic tangent

**Usage**

```
Expr_arctanh()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(-1, tanh(0.5), 0, 1, NA_real_))$  
  with_columns(arctanh = pl$col("a")$arctanh())
```



---

Expr_arg_max	<i>Index of max value</i>
--------------	---------------------------

---

**Description**

Get the index of the maximal value.

**Usage**

```
Expr_arg_max()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(  
  a = c(6, 1, 0, NA, Inf, NaN)  
)$with_columns(arg_max = pl$col("a")$arg_max())
```

---

Expr_arg_min	<i>Index of min value</i>
--------------	---------------------------

---

**Description**

Get the index of the minimal value.

**Usage**

```
Expr_arg_min()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(  
  a = c(6, 1, 0, NA, Inf, NaN)  
)$with_columns(arg_min = pl$col("a")$arg_min())
```

---

Expr_arg_sort	<i>Index of a sort</i>
---------------	------------------------

---

**Description**

Get the index values that would sort this column.

**Usage**

```
Expr_arg_sort(descending = FALSE, nulls_last = FALSE)
```

**Arguments**

descending	A logical. If TRUE, sort in descending order.
nulls_last	A logical. If TRUE, place null values last instead of first.

**Value**

Expr

**See Also**

[pl\\$arg\\_sort\\_by\(\)](#) to find the row indices that would sort multiple columns.

**Examples**

```
pl$DataFrame(
  a = c(6, 1, 0, NA, Inf, NaN)
)$with_columns(arg_sorted = pl$col("a")$arg_sort())
```

---

Expr_arg_unique	<i>Index of first unique values</i>
-----------------	-------------------------------------

---

**Description**

This finds the position of first occurrence of each unique value.

**Usage**

```
Expr_arg_unique()
```

**Value**

Expr

**Examples**

```
pl$select(pl$lit(c(1:2, 1:3))$arg_unique())
```

---

Expr\_backward\_fill      *Fill null values backward*

---

**Description**

Fill missing values with the next to be seen values. Syntactic sugar for `$fill_null(strategy = "backward")`.

**Usage**

```
Expr_backward_fill(limit = NULL)
```

**Arguments**

`limit`                  Number of consecutive null values to fill when using the "forward" or "backward" strategy.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(NA, 1, NA, 2, NA))$  
  with_columns(  
    backward = pl$col("a")$backward_fill()  
  )
```

---

Expr\_bottom\_k                  *Bottom k values*

---

**Description**

Return the k smallest elements. This has time complexity:  $O(n + k \log n - \frac{k}{2})$

**Usage**

```
Expr_bottom_k(k)
```

**Arguments**

`k`                          Number of top values to get.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(6, 1, 0, NA, Inf, NaN))$select(pl$col("a")$bottom_k(5))
```

---

Expr_cast	<i>Cast between DataType</i>
-----------	------------------------------

---

**Description**

Cast between DataType

**Usage**

```
Expr_cast(dtype, strict = TRUE)
```

**Arguments**

dtype	DataType to cast to.
strict	If TRUE (default), an error will be thrown if cast failed at resolve time.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(a = 1:3, b = c(1, 2, 3))
df$with_columns(
  pl$col("a")$cast(pl$dtypes$Float64),
  pl$col("b")$cast(pl$dtypes$Int32)
)

# strict FALSE, inserts null for any cast failure
pl$lit(c(100, 200, 300))$cast(pl$dtypes$UInt8, strict = FALSE)$to_series()

# strict TRUE, raise any failure as an error when query is executed.
tryCatch(
  {
    pl$lit("a")$cast(pl$dtypes$Float64, strict = TRUE)$to_series()
  },
  error = function(e) e
)
```

---

Expr\_ceil

*Ceiling*

---

### Description

Rounds up to the nearest integer value. Only works on floating point Series.

### Usage

```
Expr_ceil()
```

### Value

Expr

### Examples

```
pl$DataFrame(a = c(0.33, 0.5, 1.02, 1.5, NaN, NA, Inf, -Inf))$with_columns(  
  ceiling = pl$col("a")$ceil()  
)
```

---

Expr\_class

*Polars Expressions*

---

### Description

Expressions are all the functions and methods that are applicable to a Polars [DataFrame](#) or [LazyFrame](#) object. Some methods are under the sub-namespaces.

### Sub-namespaces

**arr:**

\$arr stores all array related methods.

**bin:**

\$bin stores all binary related methods.

**cat:**

\$cat stores all categorical related methods.

**dt:**

\$dt stores all temporal related methods.

**list:**

\$list stores all list related methods.

**meta:**

\$meta stores all methods for working with the meta data.

**name:**

\$name stores all name related methods.

**str:**

\$str stores all string related methods.

**struct:**

\$struct stores all struct related methods.

**Examples**

```
df = pl$DataFrame(
  a = 1:2,
  b = list(1:2, 3:4),
  schema = list(a = pl$Int64, b = pl$Array(pl$Int64, 2))
)
```

```
df$select(pl$col("a")$first())
```

```
df$select(pl$col("b")$arr$sum())
```

---

 Expr\_clip

*Clip elements*


---

**Description**

Set values outside the given boundaries to the boundary value. This only works for numeric and temporal values.

**Usage**

```
Expr_clip(lower_bound = NULL, upper_bound = NULL)
```

**Arguments**

lower_bound	Lower bound. Accepts expression input. Strings are parsed as column names and other non-expression inputs are parsed as literals.
upper_bound	Upper bound. Accepts expression input. Strings are parsed as column names and other non-expression inputs are parsed as literals.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(foo = c(-50L, 5L, NA_integer_, 50L), bound = c(1, 10, 1, 1))

# With the two bounds
df$with_columns(clipped = pl$col("foo")$clip(1, 10))

# Without lower bound
df$with_columns(clipped = pl$col("foo")$clip(upper_bound = 10))

# Using another column as lower bound
df$with_columns(clipped = pl$col("foo")$clip(lower_bound = "bound"))
```

---

Expr_cos	<i>Compute cosine</i>
----------	-----------------------

---

**Description**

Compute cosine

**Usage**

```
Expr_cos()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(0, pi / 2, pi, NA_real_))$
  with_columns(cosine = pl$col("a")$cos())
```

---

Expr_cosh	<i>Compute hyperbolic cosine</i>
-----------	----------------------------------

---

**Description**

Compute hyperbolic cosine

**Usage**

```
Expr_cosh()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(-1, acosh(2), 0, 1, NA_real_))$
  with_columns(cosh = pl$col("a")$cosh())
```

---

Expr_count	<i>Count elements</i>
------------	-----------------------

---

**Description**

Count the number of elements in this expression. Note that NULL values are also counted. `$len()` is an alias.

**Usage**

```
Expr_count()
```

```
Expr_len()
```

**Value**

```
Expr
```

**Examples**

```
pl$DataFrame(
  all = c(TRUE, TRUE),
  any = c(TRUE, FALSE),
  none = c(FALSE, FALSE)
)$select(
  pl$all()$count()
)
```

---

Expr_cumulative_eval	<i>Cumulative evaluation of expressions</i>
----------------------	---

---

**Description**

Run an expression over a sliding window that increases by 1 slot every iteration.

**Usage**

```
Expr_cumulative_eval(expr, min_periods = 1L, parallel = FALSE)
```



**Arguments**

expr	Expression to evaluate.
min_periods	Number of valid (non-null) values there should be in the window before the expression is evaluated.
parallel	Run in parallel. Don't do this in a groupby or another operation that already has much parallelization.

**Details**

This can be really slow as it can have  $O(n^2)$  complexity. Don't use this for operations that visit all elements.

**Value**

Expr

**Examples**

```
pl$lit(1:5)$cumulative_eval(
  pl$element()$first() - pl$element()$last()^2
)$to_r()
```

---

Expr_cum_count	<i>Cumulative count</i>
----------------	-------------------------

---

**Description**

Get an array with the cumulative count (zero-indexed) computed at every element.

**Usage**

```
Expr_cum_count(reverse = FALSE)
```

**Arguments**

reverse	If TRUE, reverse the count.
---------	-----------------------------

**Details**

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

`$cum_count()` does not seem to count within lists.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = 1:4)$with_columns(
  pl$col("a")$cum_count()$alias("cum_count"),
  pl$col("a")$cum_count(reverse = TRUE)$alias("cum_count_reversed")
)
```

---

Expr_cum_max	<i>Cumulative maximum</i>
--------------	---------------------------

---

**Description**

Get an array with the cumulative max computed at every element.

**Usage**

```
Expr_cum_max(reverse = FALSE)
```

**Arguments**

reverse            If TRUE, start from the last value.

**Details**

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(1:4, 2L))$with_columns(
  pl$col("a")$cum_max()$alias("cummax"),
  pl$col("a")$cum_max(reverse = TRUE)$alias("cum_max_reversed")
)
```

---

Expr_cum_min	<i>Cumulative minimum</i>
--------------	---------------------------

---

**Description**

Get an array with the cumulative min computed at every element.

**Usage**

```
Expr_cum_min(reverse = FALSE)
```

**Arguments**

reverse            If TRUE, start from the last value.

**Details**

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(1:4, 2L))$with_columns(  
  pl$col("a")$cum_min()$alias("cum_min"),  
  pl$col("a")$cum_min(reverse = TRUE)$alias("cum_min_reversed")  
)
```

---

Expr_cum_prod	<i>Cumulative product</i>
---------------	---------------------------

---

**Description**

Get an array with the cumulative product computed at every element.

**Usage**

```
Expr_cum_prod(reverse = FALSE)
```

**Arguments**

reverse            If TRUE, start with the total product of elements and divide each row one by one.

**Details**

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = 1:4)$with_columns(
  pl$col("a")$cum_prod()$alias("cum_prod"),
  pl$col("a")$cum_prod(reverse = TRUE)$alias("cum_prod_reversed")
)
```

---

Expr\_cum\_sum

*Cumulative sum*

---

**Description**

Get an array with the cumulative sum computed at every element.

**Usage**

```
Expr_cum_sum(reverse = FALSE)
```

**Arguments**

reverse            If TRUE, start with the total sum of elements and subtract each row one by one.

**Details**

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = 1:4)$with_columns(
  pl$col("a")$cum_sum()$alias("cum_sum"),
  pl$col("a")$cum_sum(reverse = TRUE)$alias("cum_sum_reversed")
)
```

---

Expr_cut	<i>Bin continuous values into discrete categories</i>
----------	---

---

**Description**

Bin continuous values into discrete categories

**Usage**

```
Expr_cut(  
  breaks,  
  ...,  
  labels = NULL,  
  left_closed = FALSE,  
  include_breaks = FALSE  
)
```

**Arguments**

breaks	Unique cut points.
...	Ignored.
labels	Names of the categories. The number of labels must be equal to the number of cut points plus one.
left_closed	Set the intervals to be left-closed instead of right-closed.
include_breaks	Include a column with the right endpoint of the bin each observation falls in. This will change the data type of the output from a <a href="#">Categorical</a> to a <a href="#">Struct</a> .

**Value**

Expr of data type [Categorical](#) if `include_breaks` is `FALSE` and of data type [Struct](#) if `include_breaks` is `TRUE`.

**See Also**

[\\$qcut\(\)](#)

**Examples**

```
df = pl$DataFrame(foo = c(-2, -1, 0, 1, 2))  
  
df$with_columns(  
  cut = pl$col("foo")$cut(c(-1, 1), labels = c("a", "b", "c"))  
)  
  
# Add both the category and the breakpoint  
df$with_columns(  
  cut = pl$col("foo")$cut(c(-1, 1), include_breaks = TRUE)  
)$unnest("cut")
```

---

Expr_diff	<i>Difference</i>
-----------	-------------------

---

**Description**

Calculate the n-th discrete difference.

**Usage**

```
Expr_diff(n = 1, null_behavior = c("ignore", "drop"))
```

**Arguments**

n	Number of slots to shift.
null_behavior	String, either "ignore" (default), else "drop".

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(20L, 10L, 30L, 40L))$with_columns(
  diff_default = pl$col("a")$diff(),
  diff_2_ignore = pl$col("a")$diff(2, "ignore")
)
```

---

Expr_div	<i>Divide two expressions</i>
----------	-------------------------------

---

**Description**

Method equivalent of float division operator `expr / other`.

**Usage**

```
Expr_div(other)
```

**Arguments**

other	Numeric literal or expression value.
-------	--------------------------------------

**Details**

Zero-division behaviour follows IEEE-754:

- $0/0$ : Invalid operation - mathematically undefined, returns NaN.
- $n/0$ : On finite operands gives an exact infinite result, e.g.:  $\pm\infty$ .

**Value**[Expr](#)**See Also**

- [Arithmetic operators](#)
- `<Expr>$floor_div()`

**Examples**

```
df = pl$DataFrame(  
  x = -2:2,  
  y = c(0.5, 0, 0, -4, -0.5)  
)  
  
df$with_columns(  
  `x/2` = pl$col("x")$div(2),  
  `x/y` = pl$col("x")$div(pl$col("y"))  
)
```

---

`Expr_dot`*Dot product*

---

**Description**

Compute the dot/inner product between two Expressions.

**Usage**

```
Expr_dot(other)
```

**Arguments**

`other` numeric or string value; accepts expression input.

**Value**[Expr](#)**Examples**

```
pl$DataFrame(  
  a = 1:4, b = c(1, 2, 3, 4)  
)$with_columns(  
  pl$col("a")$dot(pl$col("b"))$alias("a dot b"),  
  pl$col("a")$dot(pl$col("a"))$alias("a dot a")  
)
```

---

Expr_drop_nans	<i>Drop NaN</i>
----------------	-----------------

---

**Description**

Drop NaN

**Usage**

```
Expr_drop_nans()
```

**Details**

Note that NaN values are not null values. Null values correspond to NA in R.

**Value**

Expr

**See Also**

drop\_nulls()

**Examples**

```
pl$DataFrame(list(x = c(1, 2, NaN, NA)))$select(pl$col("x")$drop_nans())
```

---

Expr_drop_nulls	<i>Drop missing values</i>
-----------------	----------------------------

---

**Description**

Drop missing values

**Usage**

```
Expr_drop_nulls()
```

**Value**

Expr

**See Also**

drop\_nans()

**Examples**

```
pl$DataFrame(list(x = c(1, 2, NaN, NA)))$select(pl$col("x")$drop_nulls())
```



---

Expr_entropy	<i>Entropy</i>
--------------	----------------

---

**Description**

The entropy is measured with the formula  $-\sum(pk * \log(pk))$  where pk are discrete probabilities.

**Usage**

```
Expr_entropy(base = base::exp(1), normalize = TRUE)
```

**Arguments**

base	Given exponential base, defaults to exp(1).
normalize	Normalize pk if it doesn't sum to 1.

**Value**

Expr

**Examples**

```
pl$DataFrame(x = c(1, 2, 3, 2))$
  with_columns(entropy = pl$col("x")$entropy(base = 2))
```

---

Expr_eq	<i>Check equality</i>
---------	-----------------------

---

**Description**

Method equivalent of addition operator `expr + other`.

**Usage**

```
Expr_eq(other)
```

**Arguments**

other	numeric or string value; accepts expression input.
-------	--

**Value**

Expr

**See Also**

[Expr\\_eq\\_missing](#)

**Examples**

```
pl$lit(2) == 2
pl$lit(2) == pl$lit(2)
pl$lit(2)$eq(pl$lit(2))
```

---

Expr\_eq\_missing

*Check equality without null propagation*

---

**Description**

Method equivalent of addition operator `expr + other`.

**Usage**

```
Expr_eq_missing(other)
```

**Arguments**

`other` numeric or string value; accepts expression input.

**Value**

[Expr](#)

**See Also**

[Expr\\_eq](#)

**Examples**

```
df = pl$DataFrame(x = c(NA, FALSE, TRUE), y = c(TRUE, TRUE, TRUE))
df$with_columns(
  eq = pl$col("x")$eq("y"),
  eq_missing = pl$col("x")$eq_missing("y")
)
```

---

Expr_ewm_mean	<i>Exponentially-weighted moving average</i>
---------------	--

---

**Description**

Exponentially-weighted moving average

**Usage**

```
Expr_ewm_mean(
  com = NULL,
  span = NULL,
  half_life = NULL,
  alpha = NULL,
  adjust = TRUE,
  min_periods = 1L,
  ignore_nulls = TRUE
)
```

**Arguments**

com	Specify decay in terms of center of mass, $\gamma$ , with $\alpha = \frac{1}{1+\gamma} \forall \gamma \geq 0$
span	Specify decay in terms of span, $\theta$ , with $\alpha = \frac{2}{\theta+1} \forall \theta \geq 1$
half_life	Specify decay in terms of half-life, $\lambda$ , with $\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \forall \lambda > 0$
alpha	Specify smoothing factor alpha directly, $0 < \alpha \leq 1$ .
adjust	Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings: <ul style="list-style-type: none"> <li>• When <code>adjust=TRUE</code> the EW function is calculated using weights <math>w_i = (1 - \alpha)^i</math></li> <li>• When <code>adjust=FALSE</code> the EW function is calculated recursively by <math>y_0 = x_0</math> <math>y_t = (1 - \alpha)y_{t-1} + \alpha x_t</math></li> </ul>
min_periods	Minimum number of observations in window required to have a value (otherwise result is null).
ignore_nulls	Ignore missing values when calculating weights: <ul style="list-style-type: none"> <li>• When <code>TRUE</code> (default), weights are based on relative positions. For example, the weights of <math>x_0</math> and <math>x_2</math> used in calculating the final weighted average of <math>[x_0, \text{None}, x_2]</math> are <math>1 - \alpha</math> and <math>\alpha</math> if <code>adjust=TRUE</code>, and <math>1 - \alpha</math> and <math>\alpha</math> if <code>adjust=FALSE</code>.</li> <li>• When <code>FALSE</code>, weights are based on absolute positions. For example, the weights of <math>x_0</math> and <math>x_2</math> used in calculating the final weighted average of <math>[x_0, \text{None}, x_2]</math> are <math>(1 - \alpha)^2</math> and <math>\alpha</math> if <code>adjust=TRUE</code>, and <math>(1 - \alpha)^2</math> and <math>\alpha</math> if <code>adjust=FALSE</code>.</li> </ul>

**Value**

Expr

**Examples**

```
pl$DataFrame(a = 1:3)$
  with_columns(ewm_mean = pl$col("a")$ewm_mean(com = 1))
```

Expr\_ewm\_std

*Exponentially-weighted moving standard deviation***Description**

Exponentially-weighted moving standard deviation

**Usage**

```
Expr_ewm_std(
  com = NULL,
  span = NULL,
  half_life = NULL,
  alpha = NULL,
  adjust = TRUE,
  bias = FALSE,
  min_periods = 1L,
  ignore_nulls = TRUE
)
```

**Arguments**

com	Specify decay in terms of center of mass, $\gamma$ , with $\alpha = \frac{1}{1+\gamma} \forall \gamma \geq 0$
span	Specify decay in terms of span, $\theta$ , with $\alpha = \frac{2}{\theta+1} \forall \theta \geq 1$
half_life	Specify decay in terms of half-life, $\lambda$ , with $\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \forall \lambda > 0$
alpha	Specify smoothing factor alpha directly, $0 < \alpha \leq 1$ .
adjust	Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings: <ul style="list-style-type: none"> <li>• When adjust=TRUE the EW function is calculated using weights <math>w_i = (1 - \alpha)^i</math></li> <li>• When adjust=FALSE the EW function is calculated recursively by <math>y_0 = x_0</math> <math>y_t = (1 - \alpha)y_{t-1} + \alpha x_t</math></li> </ul>
bias	If FALSE, the calculations are corrected for statistical bias.
min_periods	Minimum number of observations in window required to have a value (otherwise result is null).

- ignore\_nulls Ignore missing values when calculating weights:
- When TRUE (default), weights are based on relative positions. For example, the weights of  $x_0$  and  $x_2$  used in calculating the final weighted average of  $[x_0, \text{None}, x_2]$  are  $1 - \alpha$  and 1 if `adjust=TRUE`, and  $1 - \alpha$  and  $\alpha$  if `adjust=FALSE`.
  - When FALSE, weights are based on absolute positions. For example, the weights of `:math:x_0` and `:math:x_2` used in calculating the final weighted average of  $[x_0, \text{None}, x_2]$  are  $(1 - \alpha)^2$  and 1 if `adjust=TRUE`, and  $(1 - \alpha)^2$  and  $\alpha$  if `adjust=FALSE`.

### Value

Expr

### Examples

```
pl$DataFrame(a = 1:3)$
  with_columns(ewm_std = pl$col("a")$ewm_std(com = 1))
```

---

Expr_ewm_var	<i>Exponentially-weighted moving variance</i>
--------------	---

---

### Description

Exponentially-weighted moving variance

### Usage

```
Expr_ewm_var(
  com = NULL,
  span = NULL,
  half_life = NULL,
  alpha = NULL,
  adjust = TRUE,
  bias = FALSE,
  min_periods = 1L,
  ignore_nulls = TRUE
)
```

### Arguments

- |           |  |
|-----------|--|
| com       | Specify decay in terms of center of mass, $\gamma$ , with $\alpha = \frac{1}{1+\gamma} \forall \gamma \geq 0$  |
| span      | Specify decay in terms of span, $\theta$ , with $\alpha = \frac{2}{\theta+1} \forall \theta \geq 1$  |
| half_life | Specify decay in terms of half-life, <code>:math:\lambda</code> , with $\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \forall \lambda > 0$ |
| alpha     | Specify smoothing factor alpha directly, $0 < \alpha \leq 1$ .   |

adjust	<p>Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings:</p> <ul style="list-style-type: none"> <li>• When adjust=TRUE the EW function is calculated using weights <math>w_i = (1 - \alpha)^i</math></li> <li>• When adjust=FALSE the EW function is calculated recursively by <math>y_0 = x_0</math> <math>y_t = (1 - \alpha)y_{t-1} + \alpha x_t</math></li> </ul>
bias	If FALSE, the calculations are corrected for statistical bias.
min_periods	Minimum number of observations in window required to have a value (otherwise result is null).
ignore_nulls	<p>Ignore missing values when calculating weights:</p> <ul style="list-style-type: none"> <li>• When TRUE (default), weights are based on relative positions. For example, the weights of <math>x_0</math> and <math>x_2</math> used in calculating the final weighted average of <math>[x_0, \text{None}, x_2]</math> are <math>1 - \alpha</math> and 1 if adjust=TRUE, and <math>1 - \alpha</math> and <math>\alpha</math> if adjust=FALSE.</li> <li>• When FALSE, weights are based on absolute positions. For example, the weights of <math>x_0</math> and <math>x_2</math> used in calculating the final weighted average of <math>[x_0, \text{None}, x_2]</math> are <math>(1 - \alpha)^2</math> and 1 if adjust=TRUE, and <math>(1 - \alpha)^2</math> and <math>\alpha</math> if adjust=FALSE.</li> </ul>

**Value**

Expr

**Examples**

```
pl$DataFrame(a = 1:3)$
  with_columns(ewm_var = pl$col("a")$ewm_var(com = 1))
```

---

Expr_exclude	<i>Exclude certain columns from selection</i>
--------------	---

---

**Description**

Exclude certain columns from selection

**Usage**

Expr\_exclude(columns)

**Arguments**

columns	<p>Given param type:</p> <ul style="list-style-type: none"> <li>• string: single column name or regex starting with ^ and ending with \$</li> <li>• character vector: exclude all these column names, no regex allowed</li> <li>• DataType: Exclude any of this DataType</li> <li>• List(DataType): Exclude any of these DataType(s)</li> </ul>
---------	---

**Value**

Expr

**Examples**

```
# make DataFrame
df = pl$DataFrame(iris)

# by name(s)
df$select(pl$all()$exclude("Species"))

# by type
df$select(pl$all()$exclude(pl$Categorical()))
df$select(pl$all()$exclude(list(pl$Categorical(), pl$Float64)))

# by regex
df$select(pl$all()$exclude("^Sepal.*$"))
```

---

Expr\_exp

*Compute the exponential of the elements*

---

**Description**

Compute the exponential of the elements

**Usage**

```
Expr_exp()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = -1:3)$with_columns(a_exp = pl$col("a")$exp())
```

---

Expr_explode	<i>Explode a list or String Series</i>
--------------	--

---

**Description**

This means that every item is expanded to a new row.

**Usage**

```
Expr_explode()
```

**Details**

Categorical values are not supported.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(x = c("abc", "ab"), y = c(list(1:3), list(3:5)))
df

df$select(pl$col("y").explode())
```

---

Expr_extend_constant	<i>Extend Series with a constant</i>
----------------------	--------------------------------------

---

**Description**

Extend the Series with given number of values.

**Usage**

```
Expr_extend_constant(value, n)
```

**Arguments**

value	The value to extend the Series with. This value may be NULL to fill with nulls.
n	The number of values to extend.

**Value**

Expr



**Examples**

```
pl$select(pl$lit(1:4)$extend_constant(10.1, 2))
pl$select(pl$lit(1:4)$extend_constant(NULL, 2))
```

---

Expr_fill_nan	<i>Fill floating point NaN value with a fill value</i>
---------------	--

---

**Description**

Fill floating point NaN value with a fill value

**Usage**

```
Expr_fill_nan(value = NULL)
```

**Arguments**

value	Value used to fill NaN values.
-------	--------------------------------

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(NaN, 1, NaN, 2, NA))$
  with_columns(
    literal = pl$col("a")$fill_nan(999),
    # implicit coercion to string
    string = pl$col("a")$fill_nan("invalid")
  )
```

---

Expr_fill_null	<i>Fill null values with a value or strategy</i>
----------------	--

---

**Description**

Fill null values with a value or strategy

**Usage**

```
Expr_fill_null(value = NULL, strategy = NULL, limit = NULL)
```

**Arguments**

value	Expr or something coercible in an Expr
strategy	Possible choice are NULL (default, requires a non-null value), "forward", "backward", "min", "max", "mean", "zero", "one".
limit	Number of consecutive null values to fill when using the "forward" or "backward" strategy.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(NA, 1, NA, 2, NA))$
  with_columns(
    value = pl$col("a")$fill_null(999),
    backward = pl$col("a")$fill_null(strategy = "backward"),
    mean = pl$col("a")$fill_null(strategy = "mean")
  )
```

---

`Expr_filter`*Filter a single column.*

---

**Description**

Mostly useful in an aggregation context. If you want to filter on a DataFrame level, use `DataFrame$filter()` (or `LazyFrame$filter()`).

**Usage**

```
Expr_filter(predicate)
```

**Arguments**

predicate	An Expr or something coercible to an Expr. Must return a boolean.
-----------	---

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  group_col = c("g1", "g1", "g2"),
  b = c(1, 2, 3)
)
df

df$group_by("group_col")$agg(
  lt = pl$col("b")$filter(pl$col("b") < 2),
  gte = pl$col("b")$filter(pl$col("b") >= 2)
)
```

---

Expr_first	<i>Get the first value.</i>
------------	-----------------------------

---

**Description**

Get the first value.

**Usage**

```
Expr_first()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(x = 3:1)$with_columns(first = pl$col("x")$first())
```

---

Expr_flatten	<i>Explode a list or String Series</i>
--------------	--

---

**Description**

This is an alias for `<Expr>$explode()`.

**Usage**

```
Expr_flatten()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(x = c("abc", "ab"), y = c(list(1:3), list(3:5)))
df

df$select(pl$col("y")$flatten())
```

---

Expr_floor	<i>Floor</i>
------------	--------------

---

**Description**

Rounds down to the nearest integer value. Only works on floating point Series.

**Usage**

```
Expr_floor()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(0.33, 0.5, 1.02, 1.5, NaN, NA, Inf, -Inf))$with_columns(
  floor = pl$col("a")$floor()
)
```

---

Expr_floor_div	<i>Floor divide two expressions</i>
----------------	-------------------------------------

---

**Description**

Method equivalent of floor division operator `expr %/% other`.

**Usage**

```
Expr_floor_div(other)
```

**Arguments**

`other` Numeric literal or expression value.

**Value**

Expr

**See Also**

- [Arithmetic operators](#)
- [<Expr>\\$div\(\)](#)
- [<Expr>\\$mod\(\)](#)

**Examples**

```
df = pl$DataFrame(x = 1:5)

df$with_columns(
  `x/2` = pl$col("x")$div(2),
  `x%/%2` = pl$col("x")$floor_div(2)
)
```

---

Expr_forward_fill	<i>Fill null values forward</i>
-------------------	---------------------------------

---

**Description**

Fill missing values with the last seen values. Syntactic sugar for `$fill_null(strategy = "forward")`.

**Usage**

```
Expr_forward_fill(limit = NULL)
```

**Arguments**

limit	Number of consecutive null values to fill when using the "forward" or "backward" strategy.
-------	--

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(NA, 1, NA, 2, NA))$
  with_columns(
    backward = pl$col("a")$forward_fill()
  )
```

---

Expr_gather	<i>Gather values by index</i>
-------------	-------------------------------

---

**Description**

Gather values by index

**Usage**

```
Expr_gather(indices)
```

**Arguments**

indices            R vector or Series, or Expr that leads to a Series of dtype Int64. (0-indexed)

**Value**

Expr

**Examples**

```
df = pl$DataFrame(a = 1:10)
df$select(pl$col("a")$gather(c(0, 2, 4, -1)))
```

---

Expr_gather_every	<i>Gather every nth element</i>
-------------------	---------------------------------

---

**Description**

Gather every nth value in the Series and return as a new Series.

**Usage**

```
Expr_gather_every(n, offset = 0)
```

**Arguments**

n                    Positive integer.  
offset                Starting index.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = 0:24)$select(pl$col("a")$gather_every(6))
```

---

Expr_gt	<i>Check strictly greater inequality</i>
---------	--

---

**Description**

Method equivalent of addition operator `expr + other`.

**Usage**

```
Expr_gt(other)
```

**Arguments**

other                  numeric or string value; accepts expression input.

**Value**

[Expr](#)

**Examples**

```
pl$lit(2) > 1
pl$lit(2) > pl$lit(1)
pl$lit(2)$gt(pl$lit(1))
```

---

Expr_gt_eq	<i>Check greater or equal inequality</i>
------------	--

---

**Description**

Method equivalent of addition operator `expr + other`.

**Usage**

```
Expr_gt_eq(other)
```

**Arguments**

other                  numeric or string value; accepts expression input.

**Value**

[Expr](#)

**Examples**

```
pl$lit(2) >= 2
pl$lit(2) >= pl$lit(2)
pl$lit(2)$gt_eq(pl$lit(2))
```

---

Expr_hash	<i>Hash elements</i>
-----------	----------------------

---

**Description**

The hash value is of type UInt64.

**Usage**

```
Expr_hash(seed = 0, seed_1 = NULL, seed_2 = NULL, seed_3 = NULL)
```

**Arguments**

seed	Random seed parameter. Defaults to 0. Doesn't have any effect for now.
seed_1, seed_2, seed_3	Random seed parameter. Defaults to arg seed. The column will be coerced to UInt32.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(iris[1:3, c(1, 2)])
df$with_columns(pl$all().$hash(1234)$name$suffix("_hash"))
```

---

Expr_has_nulls	<i>Check whether the expression contains one or more null values</i>
----------------	--

---

**Description**

Check whether the expression contains one or more null values

**Usage**

```
Expr_has_nulls()
```

**Value**

Expr



**Examples**

```
df = pl$DataFrame(  
  a = c(NA, 1, NA),  
  b = c(1, NA, 2),  
  c = c(1, 2, 3)  
)  
  
df$select(pl$all())$has_nulls()
```

---

Expr_head	<i>Get the first n elements</i>
-----------	---------------------------------

---

**Description**

Get the first n elements

**Usage**

```
Expr_head(n = 10)
```

**Arguments**

n                    Number of elements to take.

**Value**

Expr

**Examples**

```
pl$DataFrame(x = 1:11)$select(pl$col("x"))$head(3)
```

---

Expr_implode	<i>Wrap column in list</i>
--------------	----------------------------

---

**Description**

Aggregate values into a list.

**Usage**

```
Expr_implode()
```

**Details**

Use `$to_struct()` to wrap a DataFrame.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = 1:3,
  b = 4:6
)
df$select(pl$all())$implode()
```

Expr\_inspect

*Inspect evaluated Series***Description**

Print the value that this expression evaluates to and pass on the value. The printing will happen when the expression evaluates, not when it is formed.

**Usage**

```
Expr_inspect(fmt = "{}")
```

**Arguments**

fmt                    format string, should contain one set of {} where object will be printed. This formatting mimics python "string".format() use in py-polars.

**Value**

Expr

**Examples**

```
pl$select(pl$lit(1:5))$inspect(
  "Here's what the Series looked like before keeping the first two values: {}"
)$head(2)
```

---

Expr_interpolate	<i>Interpolate null values</i>
------------------	--------------------------------

---

### Description

Fill nulls with linear interpolation using non-missing values. Can also be used to regrid data to a new grid - see examples below.

### Usage

```
Expr_interpolate(method = "linear")
```

### Arguments

method           String, either "linear" (default) or "nearest".

### Value

Expr

### Examples

```
pl$DataFrame(x = c(1, NA, 4, NA, 100, NaN, 150))$
  with_columns(
    interp_lin = pl$col("x")$interpolate(),
    interp_near = pl$col("x")$interpolate("nearest")
  )

# x, y interpolation over a grid
df_original_grid = pl$DataFrame(
  grid_points = c(1, 3, 10),
  values = c(2.0, 6.0, 20.0)
)
df_original_grid
df_new_grid = pl$DataFrame(grid_points = (1:10) * 1.0)
df_new_grid

# Interpolate from this to the new grid
df_new_grid$join(
  df_original_grid,
  on = "grid_points", how = "left"
)$with_columns(pl$col("values")$interpolate())
```

---

Expr_is_between	<i>Check if an expression is between the given lower and upper bounds</i>
-----------------	---

---

**Description**

Check if an expression is between the given lower and upper bounds

**Usage**

```
Expr_is_between(lower_bound, upper_bound, closed = "both")
```

**Arguments**

lower_bound	Lower bound, can be an Expr. Strings are parsed as column names.
upper_bound	Upper bound, can be an Expr. Strings are parsed as column names.
closed	Define which sides of the interval are closed (inclusive). This can be either "left", "right", "both" or "none".

**Details**

Note that in polars, NaN are equal to other NaNs, and greater than any non-NaN value.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(num = 1:5)
df$with_columns(
  is_between = pl$col("num")$is_between(2, 4),
  is_between_excl_upper = pl$col("num")$is_between(2, 4, closed = "left"),
  is_between_excl_both = pl$col("num")$is_between(2, 4, closed = "none")
)

# lower and upper bounds can also be column names or expr
df = pl$DataFrame(
  num = 1:5,
  lower = c(0, 2, 3, 3, 3),
  upper = c(6, 4, 4, 8, 3.5)
)
df$with_columns(
  is_between_cols = pl$col("num")$is_between("lower", "upper"),
  is_between_expr = pl$col("num")$is_between(pl$col("lower") / 2, "upper")
)
```

---

Expr_is_duplicated	<i>Check whether each value is duplicated</i>
--------------------	---

---

**Description**

This is syntactic sugar for `$is_unique()$not()`.

**Usage**

```
Expr_is_duplicated()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(head(mtcars[, 1:2]))$  
  with_columns(is_duplicated = pl$col("mpg")$is_duplicated())
```

---

Expr_is_finite	<i>Check if elements are finite</i>
----------------	-------------------------------------

---

**Description**

Returns a boolean Series indicating which values are finite.

**Usage**

```
Expr_is_finite()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(list(alice = c(0, NaN, NA, Inf, -Inf)))$  
  with_columns(finite = pl$col("alice")$is_finite())
```

---

`Expr_is_first_distinct`*Check whether each value is the first occurrence*

---

**Description**

Check whether each value is the first occurrence

**Usage**

```
Expr_is_first_distinct()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(head(mtcars[, 1:2]))$  
  with_columns(is_ufirst = pl$col("mpg")$is_first_distinct())
```

---

`Expr_is_in`*Check whether a value is in a vector*

---

**Description**

Notice that to check whether a factor value is in a vector of strings, you need to use the string cache, either with `pl$enable_string_cache()` or with `pl$with_string_cache()`. See examples.

**Usage**

```
Expr_is_in(other)
```

**Arguments**

`other` numeric or string value; accepts expression input.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(1:4, NA_integer_))$with_columns(  
  in_1_3 = pl$col("a")$is_in(c(1, 3)),  
  in_NA = pl$col("a")$is_in(pl$lit(NA_real_))  
)  
  
# this fails because we can't compare factors to strings  
# pl$DataFrame(a = factor(letters[1:5]))$with_columns(  
#   in_abc = pl$col("a")$is_in(c("a", "b", "c"))  
# )  
  
# need to use the string cache for this  
pl$with_string_cache({  
  pl$DataFrame(a = factor(letters[1:5]))$with_columns(  
    in_abc = pl$col("a")$is_in(c("a", "b", "c"))  
  )  
})
```

---

Expr_is_infinite	<i>Check if elements are infinite</i>
------------------	---------------------------------------

---

**Description**

Returns a boolean Series indicating which values are infinite.

**Usage**

```
Expr_is_infinite()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(list(alice = c(0, NaN, NA, Inf, -Inf)))$  
  with_columns(infinite = pl$col("alice")$is_infinite())
```

---

Expr\_is\_last\_distinct *Check whether each value is the last occurrence*

---

**Description**

Check whether each value is the last occurrence

**Usage**

```
Expr_is_last_distinct()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(head(mtcars[, 1:2]))$  
  with_columns(is_ulast = pl$col("mpg")$is_last_distinct())
```

---

Expr\_is\_nan *Check if elements are NaN*

---

**Description**

Returns a boolean Series indicating which values are NaN.

**Usage**

```
Expr_is_nan()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(list(alice = c(0, NaN, NA, Inf, -Inf)))$  
  with_columns(nan = pl$col("alice")$is_nan())
```



---

Expr_is_not_nan	<i>Check if elements are not NaN</i>
-----------------	--------------------------------------

---

**Description**

Returns a boolean Series indicating which values are not NaN. Syntactic sugar for `$is_nan()$not()`.

**Usage**

```
Expr_is_not_nan()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(list(alice = c(0, NaN, NA, Inf, -Inf)))$  
  with_columns(not_nan = pl$col("alice")$is_not_nan())
```

---

Expr_is_not_null	<i>Check if elements are not NULL</i>
------------------	---------------------------------------

---

**Description**

Returns a boolean Series indicating which values are not null. Syntactic sugar for `$is_null()$not()`.

**Usage**

```
Expr_is_not_null()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(list(x = c(1, NA, 3)))$select(pl$col("x")$is_not_null())
```

---

Expr_is_null	<i>Check if elements are NULL</i>
--------------	-----------------------------------

---

**Description**

Returns a boolean Series indicating which values are null.

**Usage**

```
Expr_is_null()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(list(x = c(1, NA, 3)))$select(pl$col("x")$is_null())
```

---

Expr_is_unique	<i>Check whether each value is unique</i>
----------------	---

---

**Description**

Check whether each value is unique

**Usage**

```
Expr_is_unique()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(head(mtcars[, 1:2]))$  
  with_columns(is_unique = pl$col("mpg")$is_unique())
```

---

Expr_kurtosis	<i>Kurtosis</i>
---------------	-----------------

---

**Description**

Compute the kurtosis (Fisher or Pearson) of a dataset.

**Usage**

```
Expr_kurtosis(fisher = TRUE, bias = TRUE)
```

**Arguments**

fisher	If TRUE (default), Fisher's definition is used (normal, centered at 0). Otherwise, Pearson's definition is used (normal, centered at 3).
bias	If FALSE, the calculations are corrected for statistical bias.

**Details**

Kurtosis is the fourth central moment divided by the square of the variance. If Fisher's definition is used, then 3 is subtracted from the result to give 0 for a normal distribution.

If bias is FALSE, then the kurtosis is calculated using k statistics to eliminate bias coming from biased moment estimators.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(1:3, 2:1))$
  with_columns(kurt = pl$col("a")$kurtosis())
```

---

Expr_last	<i>Get the last value</i>
-----------	---------------------------

---

**Description**

Get the last value

**Usage**

```
Expr_last()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(x = 3:1)$with_columns(last = pl$col("x")$last())
```

Expr\_limit

*Get the first n elements***Description**

This is an alias for `<Expr>$head()`.

**Usage**

```
Expr_limit(n = 10)
```

**Arguments**

`n`                      Number of elements to take.

**Value**

Expr

**Examples**

```
pl$DataFrame(x = 1:11)$select(pl$col("x")$limit(3))
```

Expr\_log

*Compute the logarithm of elements***Description**

Compute the logarithm of elements

**Usage**

```
Expr_log(base = base::exp(1))
```

**Arguments**

`base`                      Numeric base value for logarithm, default is `exp(1)`.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(1, 2, 3, exp(1)))$  
  with_columns(log = pl$col("a")$log())
```

---

`Expr_log10`*Compute the base-10 logarithm of elements*

---

**Description**

Compute the base-10 logarithm of elements

**Usage**`Expr_log10()`**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(1, 2, 3, exp(1)))$  
  with_columns(log10 = pl$col("a")$log10())
```

---

`Expr_lower_bound`*Find the lower bound of a DataType*

---

**Description**

Find the lower bound of a DataType

**Usage**`Expr_lower_bound()`**Value**

Expr

**Examples**

```
pl$DataFrame(
  x = 1:3, y = 1:3,
  schema = list(x = pl$UInt32, y = pl$Int32)
)$
select(pl$all()$lower_bound())
```

---

Expr_lt	<i>Check strictly lower inequality</i>
---------	--

---

**Description**

Method equivalent of addition operator `expr + other`.

**Usage**

```
Expr_lt(other)
```

**Arguments**

`other` numeric or string value; accepts expression input.

**Value**

Expr

**Examples**

```
pl$lit(5) < 10
pl$lit(5) < pl$lit(10)
pl$lit(5)$lt(pl$lit(10))
```

---

Expr_lt_eq	<i>Check lower or equal inequality</i>
------------	--

---

**Description**

Method equivalent of addition operator `expr + other`.

**Usage**

```
Expr_lt_eq(other)
```

**Arguments**

`other` numeric or string value; accepts expression input.

**Value**

Expr

**Examples**

```
pl$lit(2) <= 2
pl$lit(2) <= pl$lit(2)
pl$lit(2)$lt_eq(pl$lit(2))
```

Expr\_map\_batches

*Map an expression with an R function***Description**

Map an expression with an R function

**Usage**

```
Expr_map_batches(
  f,
  output_type = NULL,
  agg_list = FALSE,
  in_background = FALSE
)
```

**Arguments**

f	a function to map with
output_type	NULL or a type available in <code>names(pl\$dtypes)</code> . If NULL (default), the output datatype will match the input datatype. This is used to inform schema of the actual return type of the R function. Setting this wrong could theoretically have some downstream implications to the query.
agg_list	Aggregate list. Map from vector to group in <code>group_by</code> context.
in_background	Logical. Whether to execute the map in a background R process. Combined with setting e.g. <code>options(polars.rpool_cap = 4)</code> it can speed up some slow R functions as they can run in parallel R sessions. The communication speed between processes is quite slower than between threads. This will likely only give a speed-up in a "low IO - high CPU" use case. If there are multiple <code>\$map_batches(in_background = TRUE)</code> calls in the query, they will be run in parallel.

## Details

It is sometimes necessary to apply a specific R function on one or several columns. However, note that using R code in `$map_batches()` is slower than native polars. The user function must take one polars Series as input and the return should be a Series or any Robj convertible into a Series (e.g. vectors). Map fully supports browser().

If `in_background = FALSE` the function can access any global variable of the R session. However, note that several calls to `$map_batches()` will sequentially share the same main R session, so the global environment might change between the start of the query and the moment a `$map_batches()` call is evaluated. Any native polars computations can still be executed meanwhile. If `in_background = TRUE`, the map will run in one or more other R sessions and will not have access to global variables. Use `options(polars.rpool_cap = 4)` and `polars_options()$rpool_cap` to set and view number of parallel R sessions.

## Value

Expr

## Examples

```
pl$DataFrame(iris)$
  select(
    pl$col("Sepal.Length")$map_batches(\(x) {
      paste("cheese", as.character(x$to_vector()))
    }, pl$dtypes$string)
  )

# R parallel process example, use Sys.sleep() to imitate some CPU expensive
# computation.

# map a,b,c,d sequentially
pl$LazyFrame(a = 1, b = 2, c = 3, d = 4)$select(
  pl$all()$map_batches(\(s) {
    Sys.sleep(.1)
    s * 2
  })
)$collect() |> system.time()

# map in parallel 1: Overhead to start up extra R processes / sessions
options(polars.rpool_cap = 0) # drop any previous processes, just to show start-up overhead
options(polars.rpool_cap = 4) # set back to 4, the default
polars_options()$rpool_cap
pl$LazyFrame(a = 1, b = 2, c = 3, d = 4)$select(
  pl$all()$map_batches(\(s) {
    Sys.sleep(.1)
    s * 2
  }, in_background = TRUE)
)$collect() |> system.time()

# map in parallel 2: Reuse R processes in "polars global_rpool".
polars_options()$rpool_cap
pl$LazyFrame(a = 1, b = 2, c = 3, d = 4)$select(
```



```
pl$all()$map_batches(\(s) {
  Sys.sleep(.1)
  s * 2
}, in_background = TRUE)
)$collect() |> system.time()
```

---

Expr_map_elements	<i>Map a custom/user-defined function (UDF) to each element of a column</i>
-------------------	---

---

### Description

The UDF is applied to each element of a column. See Details for more information on specificities related to the context.

### Usage

```
Expr_map_elements(
  f,
  return_type = NULL,
  strict_return_type = TRUE,
  allow_fail_eval = FALSE,
  in_background = FALSE
)
```

### Arguments

f	Function to map
return_type	DataType of the output Series. If NULL, the dtype will be pl\$Unknown.
strict_return_type	If TRUE (default), error if not correct datatype returned from R. If FALSE, the output will be converted to a polars null value.
allow_fail_eval	If FALSE (default), raise an error if the function fails. If TRUE, the result will be converted to a polars null value.
in_background	Whether to run the function in a background R process, default is FALSE. Combined with setting e.g. <code>options(polars.rpool_cap = 4)</code> , this can speed up some slow R functions as they can run in parallel R sessions. The communication speed between processes is quite slower than between threads. This will likely only give a speed-up in a "low IO - high CPU" usecase. A single map will not be paralleled, only in case of multiple <code>\$map_elements()</code> in the query can these run in parallel.

## Details

Note that, in a GroupBy context, the column will have been pre-aggregated and so each element will itself be a Series. Therefore, depending on the context, requirements for function differ:

- in `$select()` or `$with_columns()` (selection context), the function must operate on R values of length 1. Polars will convert each element into an R value and pass it to the function. The output of the user function will be converted back into a polars type (the return type must match, see argument `return_type`). Using `$map_elements()` in this context should be avoided as a `lapply()` has half the overhead.
- in `$agg()` (GroupBy context), the function must take a Series and return a Series or an R object convertible to Series, e.g. a vector. In this context, it is much faster if there are the number of groups is much lower than the number of rows, as the iteration is only across the groups. The R user function could e.g. convert the Series to a vector with `$to_r()` and perform some vectorized operations.

Note that it is preferred to express your function in polars syntax, which will almost always be *significantly* faster and more memory efficient because:

- the native expression engine runs in Rust; functions run in R.
- use of R functions forces the DataFrame to be materialized in memory.
- Polars-native expressions can be parallelized (R functions cannot).
- Polars-native expressions can be logically optimized (R functions cannot).

Wherever possible you should strongly prefer the native expression API to achieve the best performance and avoid using `$map_elements()`.

## Value

Expr

## Examples

```
# apply over groups: here, the input must be a Series
# prepare two expressions, one to compute the sum of each variable, one to
# get the first two values of each variable and store them in a list
e_sum = pl$all()$map_elements(\(s) sum(s$to_r()))$name$suffix("_sum")
e_head = pl$all()$map_elements(\(s) head(s$to_r(), 2))$name$suffix("_head")
pl$DataFrame(iris)$group_by("Species")$agg(e_sum, e_head)

# apply a function on each value (should be avoided): here the input is an R
# value of length 1
# select only Float64 columns
my_selection = pl$col(pl$dtypes$Float64)

# prepare two expressions, the first one only adds 10 to each element, the
# second returns the letter whose index matches the element
e_add10 = my_selection$map_elements(\(x) {
  x + 10
})$name$suffix("_sum")
```

```

e_letter = my_selection$map_elements(\(x) {
  letters[ceiling(x)]
}, return_type = pl$dtypes$string$name$suffix("_letter")
pl$DataFrame(iris)$select(e_add10, e_letter)

# Small benchmark -----

# Using `map_elements()` is much slower than a more polars-native approach.
# First we multiply each element of a Series of 1M elements by 2.
n = 1000000L
set.seed(1)
df = pl$DataFrame(list(
  a = 1:n,
  b = sample(letters, n, replace = TRUE)
))

system.time({
  df$with_columns(
    bob = pl$col("a")$map_elements(\(x) {
      x * 2L
    })
  )
})

# Comparing this to the standard polars syntax:
system.time({
  df$with_columns(
    bob = pl$col("a") * 2L
  )
})

# Running in parallel -----

# here, we use Sys.sleep() to imitate some CPU expensive computation.

# use apply over each Species-group in each column equal to 12 sequential
# runs ~1.2 sec.
system.time({
  pl$LazyFrame(iris)$group_by("Species")$agg(
    pl$all()$map_elements(\(s) {
      Sys.sleep(.1)
      s$sum()
    })
  )$collect()
})

# first run in parallel: there is some overhead to start up extra R processes
# drop any previous processes, just to show start-up overhead here
options(polars.rpool_cap = 0)
# set back to 4, the default
options(polars.rpool_cap = 4)

```

```

polars_options()$rpool_cap

system.time({
  pl$LazyFrame(iris)$group_by("Species")$agg(
    pl$all()$map_elements(\(s) {
      Sys.sleep(.1)
      s$sum()
    }, in_background = TRUE)
  )$collect()
})

# second run in parallel: this reuses R processes in "polars global_rpool".
polars_options()$rpool_cap
system.time({
  pl$LazyFrame(iris)$group_by("Species")$agg(
    pl$all()$map_elements(\(s) {
      Sys.sleep(.1)
      s$sum()
    }, in_background = TRUE)
  )$collect()
})

```

---

Expr\_max

*Get maximum value*

---

### Description

Get maximum value

### Usage

```
Expr_max()
```

### Value

Expr

### Examples

```

pl$DataFrame(x = c(1, NA, 3))$
  with_columns(max = pl$col("x")$max())

```

---

Expr_mean	<i>Get mean value</i>
-----------	-----------------------

---

**Description**

Get mean value

**Usage**

```
Expr_mean()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(x = c(1L, NA, 2L))$  
  with_columns(mean = pl$col("x")$mean())
```

---

Expr_median	<i>Get median value</i>
-------------	-------------------------

---

**Description**

Get median value

**Usage**

```
Expr_median()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(x = c(1L, NA, 2L))$  
  with_columns(median = pl$col("x")$median())
```

---

Expr_min	<i>Get minimum value</i>
----------	--------------------------

---

**Description**

Get minimum value

**Usage**

```
Expr_min()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(x = c(1, NA, 3))$  
  with_columns(min = pl$col("x")$min())
```

---

Expr_mod	<i>Modulo two expressions</i>
----------	-------------------------------

---

**Description**

Method equivalent of modulus operator `expr %% other`.

**Usage**

```
Expr_mod(other)
```

**Arguments**

`other` Numeric literal or expression value.

**Value**

Expr

**See Also**

- [Arithmetic operators](#)
- `<Expr>$floor_div()`

**Examples**

```
df = pl$DataFrame(x = -5L:5L)

df$with_columns(
  `x%%2` = pl$col("x")$mod(2)
)
```

Expr\_mode

*Mode***Description**

Compute the most occurring value(s). Can return multiple values if there are ties.

**Usage**

```
Expr_mode()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(a = 1:6, b = c(1L, 1L, 3L, 3L, 5L, 6L), c = c(1L, 1L, 2L, 2L, 3L, 3L))
df$select(pl$col("a")$mode())
df$select(pl$col("b")$mode())
df$select(pl$col("c")$mode())
```

Expr\_mul

*Multiply two expressions***Description**

Method equivalent of multiplication operator `expr * other`.

**Usage**

```
Expr_mul(other)
```

**Arguments**

`other` Numeric literal or expression value.

**Value**

Expr

**See Also**

- [Arithmetic operators](#)

**Examples**

```
df = pl$DataFrame(x = c(1, 2, 4, 8, 16))

df$with_columns(
  `x*2` = pl$col("x")$mul(2),
  `x * xlog2` = pl$col("x")$mul(pl$col("x")$log(2))
)
```

---

Expr_nan_max	<i>Get maximum value with NaN</i>
--------------	-----------------------------------

---

**Description**

Get maximum value, but returns NaN if there are any.

**Usage**

```
Expr_nan_max()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(x = c(1, NA, 3, NaN, Inf))$
  with_columns(nan_max = pl$col("x")$nan_max())
```

---

Expr_nan_min	<i>Get minimum value with NaN</i>
--------------	-----------------------------------

---

**Description**

Get minimum value, but returns NaN if there are any.

**Usage**

```
Expr_nan_min()
```

**Value**

Expr



**Examples**

```
pl$DataFrame(x = c(1, NA, 3, NaN, Inf))$
  with_columns(nan_min = pl$col("x")$nan_min())
```

---

Expr_neq	<i>Check inequality</i>
----------	-------------------------

---

**Description**

Method equivalent of addition operator `expr + other`.

**Usage**

```
Expr_neq(other)
```

**Arguments**

`other` numeric or string value; accepts expression input.

**Value**

[Expr](#)

**See Also**

[Expr\\_neq\\_missing](#)

**Examples**

```
pl$lit(1) != 2
pl$lit(1) != pl$lit(2)
pl$lit(1)$neq(pl$lit(2))
```

---

Expr_neq_missing	<i>Check inequality without null propagation</i>
------------------	--

---

**Description**

Method equivalent of addition operator `expr + other`.

**Usage**

```
Expr_neq_missing(other)
```

**Arguments**

`other` numeric or string value; accepts expression input.

**Value**

[Expr](#)

**See Also**

[Expr\\_neq](#)

**Examples**

```
df = pl$DataFrame(x = c(NA, FALSE, TRUE), y = c(TRUE, TRUE, TRUE))
df$with_columns(
  neq = pl$col("x")$neq("y"),
  neq_missing = pl$col("x")$neq_missing("y")
)
```

---

Expr\_not

*Negate a boolean expression*

---

**Description**

Method equivalent of negation operator !expr.

**Usage**

```
Expr_not()
```

**Value**

[Expr](#)

**Examples**

```
# two syntaxes same result
pl$lit(TRUE)$not()
!pl$lit(TRUE)
```

---

Expr_null_count	<i>Count missing values</i>
-----------------	-----------------------------

---

**Description**

Count missing values

**Usage**

```
Expr_null_count()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(x = c(NA, "a", NA, "b"))$  
  with_columns(n_missing = pl$col("x")$null_count())
```

---

Expr_n_unique	<i>Count number of unique values</i>
---------------	--------------------------------------

---

**Description**

Count number of unique values

**Usage**

```
Expr_n_unique()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(iris[, 4:5])$with_columns(count = pl$col("Species")$n_unique())
```

---

Expr_or	<i>Apply logical OR on two expressions</i>
---------	--

---

**Description**

Combine two boolean expressions with OR.

**Usage**

```
Expr_or(other)
```

**Arguments**

other            numeric or string value; accepts expression input.

**Value**

[Expr](#)

**Examples**

```
p1$lit(TRUE) | FALSE
p1$lit(TRUE)$or(p1$lit(TRUE))
```

---

Expr_over	<i>Compute expressions over the given groups</i>
-----------	--

---

**Description**

This expression is similar to performing a group by aggregation and joining the result back into the original [DataFrame](#). The outcome is similar to how window functions work in [PostgreSQL](#).

**Usage**

```
Expr_over(..., order_by = NULL, mapping_strategy = "group_to_rows")
```

**Arguments**

...            Column(s) to group by. Accepts expression input. Characters are parsed as column names.

order\_by      Order the window functions/aggregations with the partitioned groups by the result of the expression passed to order\_by. Can be an Expr. Strings are parsed as column names.

mapping\_strategy    One of the following:

- "group\_to\_rows" (default): if the aggregation results in multiple values, assign them back to their position in the DataFrame. This can only be done if the group yields the same elements before aggregation as after.
- "join": join the groups as List<group\_dtype> to the row positions. Note that this can be memory intensive.
- "explode": don't do any mapping, but simply flatten the group. This only makes sense if the input data is sorted.

## Value

Expr

## Examples

```
# Pass the name of a column to compute the expression over that column.
df = pl$DataFrame(
  a = c("a", "a", "b", "b", "b"),
  b = c(1, 2, 3, 5, 3),
  c = c(5, 4, 2, 1, 3)
)

df$with_columns(
  pl$col("c")$max()$over("a")$name$suffix("_max")
)

# Expression input is supported.
df$with_columns(
  pl$col("c")$max()$over(pl$col("b") %% 2)$name$suffix("_max")
)

# Group by multiple columns by passing a character vector of column names
# or list of expressions.
df$with_columns(
  pl$col("c")$min()$over(c("a", "b"))$name$suffix("_min")
)

df$with_columns(
  pl$col("c")$min()$over(list(pl$col("a"), pl$col("b")))$name$suffix("_min")
)

# Or use positional arguments to group by multiple columns in the same way.
df$with_columns(
  pl$col("c")$min()$over("a", pl$col("b") %% 2)$name$suffix("_min")
)

# Alternative mapping strategy: join values in a list output
df$with_columns(
  top_2 = pl$col("c")$top_k(2)$over("a", mapping_strategy = "join")
)

# order_by specifies how values are sorted within a group, which is
# essential when the operation depends on the order of values
```

```
df = pl$DataFrame(
  g = c(1, 1, 1, 1, 2, 2, 2, 2),
  t = c(1, 2, 3, 4, 4, 1, 2, 3),
  x = c(10, 20, 30, 40, 10, 20, 30, 40)
)

# without order_by, the first and second values in the second group would
# be inverted, which would be wrong
df$with_columns(
  x_lag = pl$col("x")$shift(1)$over("g", order_by = "t")
)
```

---

Expr_pct_change	<i>Percentage change</i>
-----------------	--------------------------

---

### Description

Computes percentage change (as fraction) between current element and most- recent non-null element at least *n* period(s) before the current element. Computes the change from the previous row by default.

### Usage

```
Expr_pct_change(n = 1)
```

### Arguments

*n*                      Periods to shift for computing percent change.

### Value

Expr

### Examples

```
pl$DataFrame(a = c(10L, 11L, 12L, NA_integer_, 12L))$
  with_columns(pct_change = pl$col("a")$pct_change())
```

---

Expr_peak_max	<i>Find local maxima</i>
---------------	--------------------------

---

**Description**

A local maximum is the point that marks the transition between an increase and a decrease in a Series. The first and last values of the Series can never be a peak.

**Usage**

```
Expr_peak_max()
```

**Value**

Expr

**See Also**

`$peak_min()`

**Examples**

```
df = pl$DataFrame(x = c(1, 2, 3, 2, 3, 4, 5, 2))
df

df$with_columns(peak_max = pl$col("x")$peak_max())
```

---

Expr_peak_min	<i>Find local minima</i>
---------------	--------------------------

---

**Description**

A local minimum is the point that marks the transition between a decrease and an increase in a Series. The first and last values of the Series can never be a peak.

**Usage**

```
Expr_peak_min()
```

**Value**

Expr

**See Also**

`$peak_max()`

**Examples**

```
df = pl$DataFrame(x = c(1, 2, 3, 2, 3, 4, 5, 2))
df

df$with_columns(peak_min = pl$col("x")$peak_min())
```

---

Expr\_pow

*Exponentiation two expressions*

---

**Description**

Method equivalent of exponentiation operator `expr ^ exponent`.

**Usage**

```
Expr_pow(exponent)
```

**Arguments**

exponent      Numeric literal or expression value.

**Value**

Expr

**See Also**

- [Arithmetic operators](#)

**Examples**

```
df = pl$DataFrame(x = c(1, 2, 4, 8))

df$with_columns(
  cube = pl$col("x")$pow(3),
  `x^xlog2` = pl$col("x")$pow(pl$col("x")$log(2))
)
```



---

Expr_product	<i>Product</i>
--------------	----------------

---

**Description**

Compute the product of an expression.

**Usage**

```
Expr_product()
```

**Value**

Expr

**Examples**

```
p1$DataFrame(x = c(2L, NA, 2L))$
  with_columns(product = p1$col("x")$product())
```

---

Expr_qcut	<i>Bin continuous values into discrete categories based on their quantiles</i>
-----------	--

---

**Description**

Bin continuous values into discrete categories based on their quantiles

**Usage**

```
Expr_qcut(
  quantiles,
  ...,
  labels = NULL,
  left_closed = FALSE,
  allow_duplicates = FALSE,
  include_breaks = FALSE
)
```

**Arguments**

quantiles	Either a vector of quantile probabilities between 0 and 1 or a positive integer determining the number of bins with uniform probability.
...	Ignored.
labels	Names of the categories. The number of labels must be equal to the number of cut points plus one.

- `left_closed` Set the intervals to be left-closed instead of right-closed.
- `allow_duplicates` If set to TRUE, duplicates in the resulting quantiles are dropped, rather than raising an error. This can happen even with unique probabilities, depending on the data.
- `include_breaks` Include a column with the right endpoint of the bin each observation falls in. This will change the data type of the output from a [Categorical](#) to a [Struct](#).

**Value**

Expr of data type [Categorical](#) if `include_breaks` is FALSE and of data type [Struct](#) if `include_breaks` is TRUE.

**See Also**

[\\$cut\(\)](#)

**Examples**

```
df = pl$DataFrame(foo = c(-2, -1, 0, 1, 2))

# Divide a column into three categories according to pre-defined quantile
# probabilities
df$with_columns(
  qcut = pl$col("foo")$qcut(c(0.25, 0.75), labels = c("a", "b", "c"))
)

# Divide a column into two categories using uniform quantile probabilities.
df$with_columns(
  qcut = pl$col("foo")$qcut(2, labels = c("low", "high"), left_closed = TRUE)
)

# Add both the category and the breakpoint
df$with_columns(
  qcut = pl$col("foo")$qcut(c(0.25, 0.75), include_breaks = TRUE)
)$unnest("qcut")
```

---

Expr\_quantile

*Get quantile value.*

---

**Description**

Get quantile value.

**Usage**

```
Expr_quantile(quantile, interpolation = "nearest")
```

**Arguments**

- quantile        Either a numeric value or an Expr whose value must be between 0 and 1.
- interpolation   One of "nearest", "higher", "lower", "midpoint", or "linear".

**Details**

Null values are ignored and NaNs are ranked as the largest value. For linear interpolation NaN poisons Inf, that poisons any other value.

**Value**

Expr

**Examples**

```
pl$DataFrame(x = -5:5)$
  select(pl$col("x")$quantile(0.5))
```

---

Expr_rank	Rank elements
-----------	---------------

---

**Description**

Assign ranks to data, dealing with ties appropriately.

**Usage**

```
Expr_rank(
  method = c("average", "min", "max", "dense", "ordinal", "random"),
  descending = FALSE,
  seed = NULL
)
```

**Arguments**

- method        String, one of "average" (default), "min", "max", "dense", "ordinal", "random".  
The method used to assign ranks to tied elements:
- "average": The average of the ranks that would have been assigned to all the tied values is assigned to each value.
  - "min": The minimum of the ranks that would have been assigned to all the tied values is assigned to each value. (This is also referred to as "competition" ranking.)
  - "max" : The maximum of the ranks that would have been assigned to all the tied values is assigned to each value.
  - "dense": Like 'min', but the rank of the next highest element is assigned the rank immediately after those assigned to the tied elements.

	<ul style="list-style-type: none"> <li>• "ordinal" : All values are given a distinct rank, corresponding to the order that the values occur in the Series.</li> <li>• "random" : Like 'ordinal', but the rank for ties is not dependent on the order that the values occur in the Series.</li> </ul>
descending	Rank in descending order.
seed	string parsed or number converted into uint64. Used if method="random".

**Value**

Expr

**Examples**

```
# The 'average' method:
pl$DataFrame(a = c(3, 6, 1, 1, 6))$
  with_columns(rank = pl$col("a")$rank())

# The 'ordinal' method:
pl$DataFrame(a = c(3, 6, 1, 1, 6))$
  with_columns(rank = pl$col("a")$rank("ordinal"))
```

Expr\_rechunk

*Rechunk memory layout***Description**

Create a single chunk of memory for this Series.

**Usage**

Expr\_rechunk()

**Details**

See rechunk() explained here [docs\\_translations](#).

**Value**

Expr

**Examples**

```
# get chunked lengths with/without rechunk
series_list = pl$DataFrame(list(a = 1:3, b = 4:6))$select(
  pl$col("a")$append(pl$col("b"))$alias("a_chunked"),
  pl$col("a")$append(pl$col("b"))$rechunk()$alias("a_rechunked")
)$get_columns()
lapply(series_list, \(x) x$chunk_lengths())
```

---

Expr_reinterpret	<i>Reinterpret bits</i>
------------------	-------------------------

---

**Description**

Reinterpret the underlying bits as a signed/unsigned integer. This operation is only allowed for Int64. For lower bits integers, you can safely use the cast operation.

**Usage**

```
Expr_reinterpret(signed = TRUE)
```

**Arguments**

signed	If TRUE (default), reinterpret into Int64. Otherwise, it will be reinterpreted in UInt64.
--------	---

**Value**

Expr

**Examples**

```
df = pl$DataFrame(x = 1:5, schema = list(x = pl$Int64))
df$select(pl$all()$reinterpret())
```

---

Expr_rep	<i>Repeat a Series</i>
----------	------------------------

---

**Description**

This expression takes input and repeats it n times and append chunk.

**Usage**

```
Expr_rep(n, rechunk = TRUE)
```

**Arguments**

n	The number of times to repeat, must be non-negative and finite.
rechunk	If TRUE (default), memory layout will be rewritten.

**Details**

If the input has length 1, this uses a special faster implementation that doesn't require rechunking (so rechunk = TRUE has no effect).

**Value**

Expr

**Examples**

```
pl$select(pl$lit("alice")$rep(n = 3))
pl$select(pl$lit(1:3)$rep(n = 2))
```

---

Expr_repeat_by	<i>Repeat values</i>
----------------	----------------------

---

**Description**

Repeat the elements in this Series as specified in the given expression. The repeated elements are expanded into a List.

**Usage**

Expr\_repeat\_by(by)

**Arguments**

by Expr that determines how often the values will be repeated. The column will be coerced to UInt32.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(a = c("w", "x", "y", "z"), n = c(-1, 0, 1, 2))
df$with_columns(repeated = pl$col("a")$repeat_by("n"))
```

---

Expr_replace	<i>Replace the given values by different values of the same data type.</i>
--------------	--

---

**Description**

This allows one to recode values in a column, leaving all other values unchanged. See [\\$replace\\_strict\(\)](#) to give a default value to all other values and to specify the output datatype.

**Usage**

Expr\_replace(old, new)

**Arguments**

old	Can be several things: <ul style="list-style-type: none"> <li>• a vector indicating the values to recode;</li> <li>• if new is missing, this can be a named list e.g <code>list(old = "new")</code> where the names are the old values and the values are the replacements. Note that if old values are numeric, the names must be wrapped in backticks;</li> <li>• an Expr</li> </ul>
new	Either a vector of length 1, a vector of same length as old or an Expr. If missing, old must be a named list.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(a = c(1, 2, 2, 3))

# "old" and "new" can take vectors of length 1 or of same length
df$with_columns(replaced = pl$col("a")$replace(2, 100))
df$with_columns(replaced = pl$col("a")$replace(c(2, 3), c(100, 200)))

# "old" can be a named list where names are values to replace, and values are
# the replacements
mapping = list(`2` = 100, `3` = 200)
df$with_columns(replaced = pl$col("a")$replace(mapping))

df = pl$DataFrame(a = c("x", "y", "z"))
mapping = list(x = 1, y = 2, z = 3)
df$with_columns(replaced = pl$col("a")$replace(mapping))

# "old" and "new" can take Expr
df = pl$DataFrame(a = c(1, 2, 2, 3), b = c(1.5, 2.5, 5, 1))
df$with_columns(
  replaced = pl$col("a")$replace(
    old = pl$col("a")$max(),
    new = pl$col("b")$sum()
  )
)
```

---

Expr\_replace\_strict    *Replace all values by different values.*

---

**Description**

This changes all the values in a column, either using a specific replacement or a default one. See [\\$replace\(\)](#) to replace only a subset of values.

**Usage**

```
Expr_replace_strict(old, new, default = NULL, return_dtype = NULL)
```

**Arguments**

old	Can be several things: <ul style="list-style-type: none"> <li>• a vector indicating the values to recode;</li> <li>• if new is missing, this can be a named list e.g <code>list(old = "new")</code> where the names are the old values and the values are the replacements. Note that if old values are numeric, the names must be wrapped in backticks;</li> <li>• an Expr</li> </ul>
new	Either a vector of length 1, a vector of same length as old or an Expr. If missing, old must be a named list.
default	The default replacement if the value is not in old. Can be an Expr. If NULL (default), then the value doesn't change.
return_dtype	The data type of the resulting expression. If set to NULL (default), the data type is determined automatically based on the other inputs.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(a = c(1, 2, 2, 3))

# "old" and "new" can take vectors of length 1 or of same length
df$with_columns(replaced = pl$col("a")$replace_strict(2, 100, default = 1))
df$with_columns(
  replaced = pl$col("a")$replace_strict(c(2, 3), c(100, 200), default = 1)
)

# "old" can be a named list where names are values to replace, and values are
# the replacements
mapping = list(`2` = 100, `3` = 200)
df$with_columns(replaced = pl$col("a")$replace_strict(mapping, default = -1))

# one can specify the data type to return instead of automatically
# inferring it
df$with_columns(
  replaced = pl$col("a")$replace_strict(mapping, default = 1, return_dtype = pl$Int32)
)

# "old", "new", and "default" can take Expr
df = pl$DataFrame(a = c(1, 2, 2, 3), b = c(1.5, 2.5, 5, 1))
df$with_columns(
  replaced = pl$col("a")$replace_strict(
    old = pl$col("a")$max(),
    new = pl$col("b")$sum(),

```



```

    default = pl$col("b"),
  )
)
```

Expr\_reshape

*Reshape this Expr to a flat Series or a Series of Lists***Description**

Reshape this Expr to a flat Series or a Series of Lists

**Usage**

```
Expr_reshape(dimensions, nested_type = pl$List())
```

**Arguments**

dimensions	A integer vector of length of the dimension size. If -1 is used in any of the dimensions, that dimension is inferred. Currently, more than two dimensions not supported.
nested_type	The nested data type to create. <a href="#">List</a> only supports 2 dimensions, whereas <a href="#">Array</a> supports an arbitrary number of dimensions.

**Value**

[Expr](#). If a single dimension is given, results in an expression of the original data type. If a multiple dimensions are given, results in an expression of data type List with shape equal to the dimensions.

**Examples**

```

df = pl$DataFrame(foo = 1:9)

df$select(pl$col("foo")$reshape(9))
df$select(pl$col("foo")$reshape(c(3, 3)))

# Use `-1` to infer the other dimension
df$select(pl$col("foo")$reshape(c(-1, 3)))
df$select(pl$col("foo")$reshape(c(3, -1)))

# One can specify more than 2 dimensions by using the Array type
df = pl$DataFrame(foo = 1:12)
df$select(
  pl$col("foo")$reshape(c(3, 2, 2), nested_type = pl$Array(pl$Float32, 2))
)
```

---

Expr_reverse	<i>Reverse a variable</i>
--------------	---------------------------

---

**Description**

Reverse a variable

**Usage**

```
Expr_reverse()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(list(a = 1:5))$select(pl$col("a")$reverse())
```

---

Expr_rle	<i>Get the lengths of runs of identical values</i>
----------	--

---

**Description**

Get the lengths of runs of identical values

**Usage**

```
Expr_rle()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(s = c(1, 1, 2, 1, NA, 1, 3, 3))
df$select(pl$col("s")$rle())$unnest("s")
```

---

Expr\_rle\_id                      *Map values to run IDs*

---

### Description

Similar to `$rle()`, but it maps each value to an ID corresponding to the run into which it falls. This is especially useful when you want to define groups by runs of identical values rather than the values themselves. Note that the ID is 0-indexed.

### Usage

```
Expr_rle_id()
```

### Value

Expr

### Examples

```
df = pl$DataFrame(a = c(1, 2, 1, 1, 1, 4))
df$with_columns(a_r = pl$col("a")$rle_id())
```

---

Expr\_rolling                      *Create rolling groups based on a time or numeric column*

---

### Description

If you have a time series  $\langle t_0, t_1, \dots, t_n \rangle$ , then by default the windows created will be:

- $(t_0 - \text{period}, t_0]$
- $(t_1 - \text{period}, t_1]$
- ...
- $(t_n - \text{period}, t_n]$

whereas if you pass a non-default offset, then the windows will be:

- $(t_0 + \text{offset}, t_0 + \text{offset} + \text{period}]$
- $(t_1 + \text{offset}, t_1 + \text{offset} + \text{period}]$
- ...
- $(t_n + \text{offset}, t_n + \text{offset} + \text{period}]$

### Usage

```
Expr_rolling(index_column, ..., period, offset = NULL, closed = "right")
```

**Arguments**

index_column	Column used to group based on the time window. Often of type Date/Datetime. This column must be sorted in ascending order. If this column represents an index, it has to be either Int32 or Int64. Note that Int32 gets temporarily cast to Int64, so if performance matters use an Int64 column.
...	Ignored.
period	A character representing the length of the window, must be non-negative. See the Polars duration string language section for details.
offset	A character representing the offset of the window, or NULL (default). If NULL, -period is used. See the Polars duration string language section for details.
closed	Define which sides of the temporal interval are closed (inclusive). This can be either "left", "right", "both" or "none".

**Details**

In case of a rolling operation on an integer column, the windows are defined by:

- "1i" # length 1
- "10i" # length 10

**Value**

Expr

**Polars duration string language**

Polars duration string language is a simple representation of durations. It is used in many Polars functions that accept durations.

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)
- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds

By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".

**Examples**

```
# create a DataFrame with a Datetime column and an f64 column
dates = c(
  "2020-01-01 13:45:48", "2020-01-01 16:42:13", "2020-01-01 16:45:09",
  "2020-01-02 18:12:48", "2020-01-03 19:45:32", "2020-01-08 23:16:43"
)

df = pl$DataFrame(dt = dates, a = c(3, 7, 5, 9, 2, 1))$
  with_columns(
    pl$col("dt")$str$strptime(pl$Datetime("us"), format = "%Y-%m-%d %H:%M:%S")$set_sorted()
  )

df$with_columns(
  sum_a = pl$sum("a")$rolling(index_column = "dt", period = "2d"),
  min_a = pl$min("a")$rolling(index_column = "dt", period = "2d"),
  max_a = pl$max("a")$rolling(index_column = "dt", period = "2d")
)

# we can use "offset" to change the start of the window period. Here, with
# offset = "1d", we start the window one day after the value in "dt", and
# then we add a 2-day window relative to the window start.
df$with_columns(
  sum_a_offset1 = pl$sum("a")$rolling(index_column = "dt", period = "2d", offset = "1d")
)
```

---

Expr_rolling_max	<i>Rolling maximum</i>
------------------	------------------------

---

**Description**

Compute the rolling (= moving) max over the values in this array. A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the weight vector.

**Usage**

```
Expr_rolling_max(
  window_size,
  weights = NULL,
  min_periods = NULL,
  ...,
  center = FALSE
)
```

**Arguments**

<code>window_size</code>	Integer specifying the length of the window.
<code>weights</code>	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.

min_periods	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
...	Ignored.
center	Set the labels at the center of the window

### Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

### Value

Expr

### Examples

```
pl$DataFrame(a = c(1, 3, 2, 4, 5, 6))$
  with_columns(roll_max = pl$col("a")$rolling_max(window_size = 2))
```

---

Expr\_rolling\_max\_by    *Apply a rolling max based on another column.*

---

### Description

Apply a rolling max based on another column.

### Usage

```
Expr_rolling_max_by(by, window_size, ..., min_periods = 1, closed = "right")
```

### Arguments

by	This column must of dtype <a href="#">Date</a> or <a href="#">Datetime</a> .
window_size	The length of the window. Can be a fixed integer size, or a dynamic temporal size indicated by the following string language: <ul style="list-style-type: none"> <li>• 1ns (1 nanosecond)</li> <li>• 1us (1 microsecond)</li> <li>• 1ms (1 millisecond)</li> <li>• 1s (1 second)</li> <li>• 1m (1 minute)</li> <li>• 1h (1 hour)</li> <li>• 1d (1 day)</li> <li>• 1w (1 week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1y (1 calendar year)</li> </ul>

- `li` (1 index count) If the dynamic string language is used, the `by` and `closed` arguments must also be set.
- ...
- `min_periods` Ignored.
- `min_periods` The number of values in the window that should be non-null before computing a result. If `NULL`, it will be set equal to window size.
- `closed` Define which sides of the temporal interval are closed (inclusive). This can be either "left", "right", "both" or "none".

### Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

### Value

Expr

### Examples

```
df_temporal = pl.DataFrame(
  date = pl.datetime_range(as.Date("2001-1-1"), as.Date("2001-1-2"), "1h")
)$with_row_index("index")

df_temporal

df_temporal$with_columns(
  rolling_row_max = pl$col("index")$rolling_max_by("date", window_size = "3h")
)
```

---

Expr_rolling_mean	<i>Rolling mean</i>
-------------------	---------------------

---

### Description

Compute the rolling (= moving) mean over the values in this array. A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the weight vector.

### Usage

```
Expr_rolling_mean(
  window_size,
  weights = NULL,
  min_periods = NULL,
  ...,
  center = FALSE
)
```

**Arguments**

window_size	Integer specifying the length of the window.
weights	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
min_periods	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
...	Ignored.
center	Set the labels at the center of the window

**Details**

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(1, 3, 2, 4, 5, 6))$
  with_columns(roll_mean = pl$col("a")$rolling_mean(window_size = 2))
```

---

Expr\_rolling\_mean\_by *Apply a rolling mean based on another column.*

---

**Description**

Apply a rolling mean based on another column.

**Usage**

```
Expr_rolling_mean_by(by, window_size, ..., min_periods = 1, closed = "right")
```

**Arguments**

by	This column must of dtype <a href="#">Date</a> or <a href="#">Datetime</a> .
window_size	The length of the window. Can be a fixed integer size, or a dynamic temporal size indicated by the following string language: <ul style="list-style-type: none"> <li>• 1ns (1 nanosecond)</li> <li>• 1us (1 microsecond)</li> <li>• 1ms (1 millisecond)</li> <li>• 1s (1 second)</li> <li>• 1m (1 minute)</li> <li>• 1h (1 hour)</li> </ul>



	<ul style="list-style-type: none"> <li>• 1d (1 day)</li> <li>• 1w (1 week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1y (1 calendar year)</li> <li>• 1i (1 index count) If the dynamic string language is used, the <code>by</code> and <code>closed</code> arguments must also be set.</li> </ul>
...	Ignored.
<code>min_periods</code>	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
<code>closed</code>	Define which sides of the temporal interval are closed (inclusive). This can be either "left", "right", "both" or "none".

### Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

### Value

Expr

### Examples

```
df_temporal = pl.DataFrame(
    date = pl.datetime_range(as.Date("2001-1-1"), as.Date("2001-1-2"), "1h")
)$with_row_index("index")

df_temporal

df_temporal$with_columns(
    rolling_row_mean = pl$col("index")$rolling_mean_by("date", window_size = "3h")
)
```

---

Expr\_rolling\_median     *Rolling median*

---

### Description

Compute the rolling (= moving) median over the values in this array. A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the weight vector.

**Usage**

```
Expr_rolling_median(
  window_size,
  weights = NULL,
  min_periods = NULL,
  center = FALSE
)
```

**Arguments**

<code>window_size</code>	Integer specifying the length of the window.
<code>weights</code>	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
<code>min_periods</code>	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
<code>center</code>	Set the labels at the center of the window

**Details**

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(1, 3, 2, 4, 5, 6))$
  with_columns(roll_median = pl$col("a")$rolling_median(window_size = 2))
```

---

Expr\_rolling\_median\_by

*Apply a rolling median based on another column.*

---

**Description**

Apply a rolling median based on another column.

**Usage**

```
Expr_rolling_median_by(by, window_size, ..., min_periods = 1, closed = "right")
```

**Arguments**

by	This column must of dtype <a href="#">Date</a> or <a href="#">Datetime</a> .
window_size	The length of the window. Can be a fixed integer size, or a dynamic temporal size indicated by the following string language: <ul style="list-style-type: none"> <li>• 1ns (1 nanosecond)</li> <li>• 1us (1 microsecond)</li> <li>• 1ms (1 millisecond)</li> <li>• 1s (1 second)</li> <li>• 1m (1 minute)</li> <li>• 1h (1 hour)</li> <li>• 1d (1 day)</li> <li>• 1w (1 week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1y (1 calendar year)</li> <li>• 1i (1 index count) If the dynamic string language is used, the by and closed arguments must also be set.</li> </ul>
...	Ignored.
min_periods	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
closed	Define which sides of the temporal interval are closed (inclusive). This can be either "left", "right", "both" or "none".

**Details**

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

**Value**

Expr

**Examples**

```
df_temporal = pl.DataFrame(
    date = pl.datetime_range(as.Date("2001-1-1"), as.Date("2001-1-2"), "1h")
)$with_row_index("index")

df_temporal

df_temporal$with_columns(
    rolling_row_median = pl$col("index")$rolling_median_by("date", window_size = "3h")
)
```

---

Expr_rolling_min	<i>Rolling minimum</i>
------------------	------------------------

---

### Description

Compute the rolling (= moving) min over the values in this array. A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the `weight` vector.

### Usage

```
Expr_rolling_min(
  window_size,
  weights = NULL,
  min_periods = NULL,
  ...,
  center = FALSE
)
```

### Arguments

<code>window_size</code>	Integer specifying the length of the window.
<code>weights</code>	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
<code>min_periods</code>	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
<code>...</code>	Ignored.
<code>center</code>	Set the labels at the center of the window

### Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

### Value

Expr

### Examples

```
pl$DataFrame(a = c(1, 3, 2, 4, 5, 6))$
  with_columns(roll_min = pl$col("a")$rolling_min(window_size = 2))
```

---

Expr\_rolling\_min\_by    *Apply a rolling min based on another column.*

---

### Description

Apply a rolling min based on another column.

### Usage

```
Expr_rolling_min_by(by, window_size, ..., min_periods = 1, closed = "right")
```

### Arguments

by	This column must of dtype <a href="#">Date</a> or <a href="#">Datetime</a> .
window_size	The length of the window. Can be a fixed integer size, or a dynamic temporal size indicated by the following string language: <ul style="list-style-type: none"> <li>• 1ns (1 nanosecond)</li> <li>• 1us (1 microsecond)</li> <li>• 1ms (1 millisecond)</li> <li>• 1s (1 second)</li> <li>• 1m (1 minute)</li> <li>• 1h (1 hour)</li> <li>• 1d (1 day)</li> <li>• 1w (1 week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1y (1 calendar year)</li> <li>• 1i (1 index count) If the dynamic string language is used, the by and closed arguments must also be set.</li> </ul>
...	Ignored.
min_periods	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
closed	Define which sides of the temporal interval are closed (inclusive). This can be either "left", "right", "both" or "none".

### Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

### Value

Expr

**Examples**

```
df_temporal = pl$DataFrame(
  date = pl$datetime_range(as.Date("2001-1-1"), as.Date("2001-1-2"), "1h")
)$with_row_index("index")

df_temporal

df_temporal$with_columns(
  rolling_row_min = pl$col("index")$rolling_min_by("date", window_size = "3h")
)
```

---

Expr\_rolling\_quantile *Rolling quantile*

---

**Description**

Compute the rolling (= moving) quantile over the values in this array. A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the weight vector.

**Usage**

```
Expr_rolling_quantile(
  quantile,
  interpolation = "nearest",
  window_size,
  weights = NULL,
  min_periods = NULL,
  ...,
  center = FALSE
)
```

**Arguments**

<code>quantile</code>	Quantile between 0 and 1.
<code>interpolation</code>	String, one of "nearest", "higher", "lower", "midpoint", "linear".
<code>window_size</code>	Integer specifying the length of the window.
<code>weights</code>	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
<code>min_periods</code>	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
<code>...</code>	Ignored.
<code>center</code>	Set the labels at the center of the window

## Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

## Value

Expr

## Examples

```
pl$DataFrame(a = c(1, 3, 2, 4, 5, 6))$  
  with_columns(roll_quant = pl$col("a")$rolling_quantile(0.3, window_size = 2))
```

---

Expr\_rolling\_quantile\_by

*Compute a rolling quantile based on another column*

---

## Description

Compute a rolling quantile based on another column

## Usage

```
Expr_rolling_quantile_by(  
  by,  
  window_size,  
  ...,  
  quantile,  
  interpolation = "nearest",  
  min_periods = 1,  
  closed = "right"  
)
```

## Arguments

by	This column must of dtype <a href="#">Date</a> or <a href="#">Datetime</a> .
window_size	The length of the window. Can be a fixed integer size, or a dynamic temporal size indicated by the following string language: <ul style="list-style-type: none"><li>• 1ns (1 nanosecond)</li><li>• 1us (1 microsecond)</li><li>• 1ms (1 millisecond)</li><li>• 1s (1 second)</li><li>• 1m (1 minute)</li><li>• 1h (1 hour)</li><li>• 1d (1 day)</li></ul>

	<ul style="list-style-type: none"> <li>• 1w (1 week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1y (1 calendar year)</li> <li>• li (1 index count) If the dynamic string language is used, the <code>by</code> and <code>closed</code> arguments must also be set.</li> </ul>
...	Ignored.
quantile	Either a numeric value or an Expr whose value must be between 0 and 1.
interpolation	One of "nearest", "higher", "lower", "midpoint", or "linear".
min_periods	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
closed	Define which sides of the temporal interval are closed (inclusive). This can be either "left", "right", "both" or "none".

### Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

### Value

Expr

### Examples

```
df_temporal = pl$DataFrame(
  date = pl$datetime_range(as.Date("2001-1-1"), as.Date("2001-1-2"), "1h")
)$with_row_index("index")

df_temporal

df_temporal$with_columns(
  rolling_row_quantile = pl$col("index")$rolling_quantile_by(
    "date",
    window_size = "2h", quantile = 0.3
  )
)
```

---

Expr_rolling_skew	<i>Rolling skew</i>
-------------------	---------------------

---

### Description

Compute the rolling (= moving) skewness over the values in this array. A window of length `window_size` will traverse the array.



**Usage**

```
Expr_rolling_skew(window_size, bias = TRUE)
```

**Arguments**

`window_size` Integer specifying the length of the window.  
`bias` If FALSE, the calculations are corrected for statistical bias.

**Details**

For normally distributed data, the skewness should be about zero. For uni-modal continuous distributions, a skewness value greater than zero means that there is more weight in the right tail of the distribution.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(1, 3, 2, 4, 5, 6))$  
  with_columns(roll_skew = pl$col("a")$rolling_skew(window_size = 2))
```

---

Expr_rolling_std	<i>Rolling standard deviation</i>
------------------	-----------------------------------

---

**Description**

Compute the rolling (= moving) standard deviation over the values in this array. A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the weight vector.

**Usage**

```
Expr_rolling_std(  
  window_size,  
  weights = NULL,  
  min_periods = NULL,  
  ...,  
  center = FALSE,  
  ddof = 1  
)
```

**Arguments**

window_size	Integer specifying the length of the window.
weights	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
min_periods	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
...	Ignored.
center	Set the labels at the center of the window
ddof	An integer representing "Delta Degrees of Freedom": the divisor used in the calculation is $N - \text{ddof}$ , where $N$ represents the number of elements. By default <code>ddof</code> is 1.

**Details**

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(1, 3, 2, 4, 5, 6))$
  with_columns(roll_std = pl$col("a")$rolling_std(window_size = 2))
```

---

Expr\_rolling\_std\_by     *Compute a rolling standard deviation based on another column*

---

**Description**

Compute a rolling standard deviation based on another column

**Usage**

```
Expr_rolling_std_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = "right",
  ddof = 1
)
```

**Arguments**

by	This column must of dtype <a href="#">Date</a> or <a href="#">Datetime</a> .
window_size	The length of the window. Can be a fixed integer size, or a dynamic temporal size indicated by the following string language: <ul style="list-style-type: none"> <li>• 1ns (1 nanosecond)</li> <li>• 1us (1 microsecond)</li> <li>• 1ms (1 millisecond)</li> <li>• 1s (1 second)</li> <li>• 1m (1 minute)</li> <li>• 1h (1 hour)</li> <li>• 1d (1 day)</li> <li>• 1w (1 week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1y (1 calendar year)</li> <li>• 1i (1 index count) If the dynamic string language is used, the by and closed arguments must also be set.</li> </ul>
...	Ignored.
min_periods	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
closed	Define which sides of the temporal interval are closed (inclusive). This can be either "left", "right", "both" or "none".
ddof	An integer representing "Delta Degrees of Freedom": the divisor used in the calculation is $N - \text{ddof}$ , where $N$ represents the number of elements. By default ddof is 1.

**Details**

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

**Value**

Expr

**Examples**

```
df_temporal = pl$DataFrame(
  date = pl$datetime_range(as.Date("2001-1-1"), as.Date("2001-1-2"), "1h")
)$with_row_index("index")

df_temporal

# Compute the rolling std with the temporal windows closed on the right (default)
df_temporal$with_columns(
  rolling_row_std = pl$col("index")$rolling_std_by("date", window_size = "2h")
)
```

```
# Compute the rolling std with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_std = pl$col("index")$rolling_std_by("date", window_size = "2h", closed = "both")
)
```

---

Expr_rolling_sum	<i>Rolling sum</i>
------------------	--------------------

---

### Description

Compute the rolling (= moving) sum over the values in this array. A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the weight vector.

### Usage

```
Expr_rolling_sum(
  window_size,
  weights = NULL,
  min_periods = NULL,
  center = FALSE
)
```

### Arguments

<code>window_size</code>	Integer specifying the length of the window.
<code>weights</code>	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
<code>min_periods</code>	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
<code>center</code>	Set the labels at the center of the window

### Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

### Value

Expr

### Examples

```
pl$DataFrame(a = c(1, 3, 2, 4, 5, 6))$
  with_columns(roll_sum = pl$col("a")$rolling_sum(window_size = 2))
```

---

Expr\_rolling\_sum\_by    *Apply a rolling sum based on another column.*

---

### Description

Apply a rolling sum based on another column.

### Usage

```
Expr_rolling_sum_by(by, window_size, ..., min_periods = 1, closed = "right")
```

### Arguments

by	This column must of dtype <a href="#">Date</a> or <a href="#">Datetime</a> .
window_size	The length of the window. Can be a fixed integer size, or a dynamic temporal size indicated by the following string language: <ul style="list-style-type: none"> <li>• 1ns (1 nanosecond)</li> <li>• 1us (1 microsecond)</li> <li>• 1ms (1 millisecond)</li> <li>• 1s (1 second)</li> <li>• 1m (1 minute)</li> <li>• 1h (1 hour)</li> <li>• 1d (1 day)</li> <li>• 1w (1 week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1y (1 calendar year)</li> <li>• 1i (1 index count) If the dynamic string language is used, the by and closed arguments must also be set.</li> </ul>
...	Ignored.
min_periods	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
closed	Define which sides of the temporal interval are closed (inclusive). This can be either "left", "right", "both" or "none".

### Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

### Value

Expr

**Examples**

```
df_temporal = pl$DataFrame(
  date = pl$datetime_range(as.Date("2001-1-1"), as.Date("2001-1-2"), "1h")
)$with_row_index("index")

df_temporal

df_temporal$with_columns(
  rolling_row_sum = pl$col("index")$rolling_sum_by("date", window_size = "3h")
)
```

---

Expr_rolling_var	<i>Rolling variance</i>
------------------	-------------------------

---

**Description**

Compute the rolling (= moving) variance over the values in this array. A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the weight vector.

**Usage**

```
Expr_rolling_var(
  window_size,
  weights = NULL,
  min_periods = NULL,
  ...,
  center = FALSE,
  ddof = 1
)
```

**Arguments**

<code>window_size</code>	Integer specifying the length of the window.
<code>weights</code>	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
<code>min_periods</code>	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
<code>...</code>	Ignored.
<code>center</code>	Set the labels at the center of the window
<code>ddof</code>	An integer representing "Delta Degrees of Freedom": the divisor used in the calculation is $N - \text{ddof}$ , where $N$ represents the number of elements. By default <code>ddof</code> is 1.

## Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

## Value

Expr

## Examples

```
pl$DataFrame(a = c(1, 3, 2, 4, 5, 6))$  
  with_columns(roll_var = pl$col("a")$rolling_var(window_size = 2))
```

---

Expr\_rolling\_var\_by     *Compute a rolling variance based on another column*

---

## Description

Compute a rolling variance based on another column

## Usage

```
Expr_rolling_var_by(  
  by,  
  window_size,  
  ...,  
  min_periods = 1,  
  closed = "right",  
  ddof = 1  
)
```

## Arguments

by	This column must of dtype <a href="#">Date</a> or <a href="#">Datetime</a> .
window_size	The length of the window. Can be a fixed integer size, or a dynamic temporal size indicated by the following string language: <ul style="list-style-type: none"><li>• 1ns (1 nanosecond)</li><li>• 1us (1 microsecond)</li><li>• 1ms (1 millisecond)</li><li>• 1s (1 second)</li><li>• 1m (1 minute)</li><li>• 1h (1 hour)</li><li>• 1d (1 day)</li><li>• 1w (1 week)</li><li>• 1mo (1 calendar month)</li></ul>

	<ul style="list-style-type: none"> <li>• 1y (1 calendar year)</li> <li>• 1i (1 index count) If the dynamic string language is used, the by and closed arguments must also be set.</li> </ul>
...	Ignored.
min_periods	The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
closed	Define which sides of the temporal interval are closed (inclusive). This can be either "left", "right", "both" or "none".
ddof	An integer representing "Delta Degrees of Freedom": the divisor used in the calculation is N - ddof, where N represents the number of elements. By default ddof is 1.

### Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` this method can cache the window size computation.

### Value

Expr

### Examples

```
df_temporal = pl$DataFrame(
  date = pl$datetime_range(as.Date("2001-1-1"), as.Date("2001-1-2"), "1h")
)$with_row_index("index")

df_temporal

# Compute the rolling var with the temporal windows closed on the right (default)
df_temporal$with_columns(
  rolling_row_var = pl$col("index")$rolling_var_by("date", window_size = "2h")
)

# Compute the rolling var with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_var = pl$col("index")$rolling_var_by("date", window_size = "2h", closed = "both")
)
```

---

Expr\_round

*Round*

---

### Description

Round underlying floating point data by decimals digits.



**Usage**

```
Expr_round(decimals)
```

**Arguments**

decimals            Number of decimals to round by.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(0.33, 0.5, 1.02, 1.5, NaN, NA, Inf, -Inf))$with_columns(
  round = pl$col("a")$round(1)
)
```

---

Expr_sample	<i>Take a sample</i>
-------------	----------------------

---

**Description**

Take a sample

**Usage**

```
Expr_sample(
  n = NULL,
  ...,
  fraction = NULL,
  with_replacement = FALSE,
  shuffle = FALSE,
  seed = NULL
)
```

**Arguments**

n                    Number of items to return. Cannot be used with fraction.

...                  Ignored.

fraction            Fraction of items to return. Cannot be used with n. Can be larger than 1 if with\_replacement is TRUE.

with\_replacement   If TRUE (default), allow values to be sampled more than once.

shuffle             Shuffle the order of sampled data points (implicitly TRUE if with\_replacement = TRUE).

seed                numeric value of 0 to  $2^{52}$  Seed for the random number generator. If NULL (default), a random seed value between 0 and 10000 is picked.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(a = 1:4)
df$select(pl$col("a")$sample(fraction = 1, with_replacement = TRUE, seed = 1L))
df$select(pl$col("a")$sample(fraction = 2, with_replacement = TRUE, seed = 1L))
df$select(pl$col("a")$sample(n = 2, with_replacement = FALSE, seed = 1L))
```

---

Expr\_search\_sorted      *Where to inject element(s) to maintain sorting*

---

**Description**

Find indices where elements should be inserted to maintain order.

**Usage**

```
Expr_search_sorted(element)
```

**Arguments**

element                  Element to insert. Can be an Expr or something coercible to an Expr.

**Details**

This function looks up where to insert element to keep self column sorted. It is assumed the column is already sorted in ascending order (otherwise this leads to wrong results).

**Value**

Expr

**Examples**

```
df = pl$DataFrame(a = c(1, 3, 4, 4, 6))
df

# in which row should 5 be inserted in order to not break the sort?
# (value is 0-indexed)
df$select(
  zero = pl$col("a")$search_sorted(0),
  three = pl$col("a")$search_sorted(3),
  five = pl$col("a")$search_sorted(5)
)
```

---

Expr_set_sorted	<i>Flag an Expr as "sorted"</i>
-----------------	---------------------------------

---

**Description**

This enables downstream code to use fast paths for sorted arrays. **WARNING:** this doesn't check whether the data is actually sorted, you have to ensure of that yourself.

**Usage**

```
Expr_set_sorted(..., descending = FALSE)
```

**Arguments**

...	Ignored.
descending	Sort the columns in descending order.

**Value**

Expr

**Examples**

```
# correct use flag something correctly as ascendingly sorted
s = pl$select(pl$lit(1:4)$set_sorted())$alias("a")$get_column("a")
s$flags

# incorrect use, flag something as not sorted ascendingly
s2 = pl$select(pl$lit(c(1, 3, 2, 4))$set_sorted())$alias("a")$get_column("a")
s2$sort()
s2$flags # returns TRUE while it's not actually sorted
```

---

Expr_shift	<i>Shift values by the given number of indices</i>
------------	--

---

**Description**

Shift values by the given number of indices

**Usage**

```
Expr_shift(n = 1, fill_value = NULL)
```

**Arguments**

n	Number of indices to shift forward. If a negative value is passed, values are shifted in the opposite direction instead.
fill_value	Fill the resulting null values with this value. Accepts expression input. Non-expression inputs are parsed as literals.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(1, 2, 4, 5, 8))$
  with_columns(
    pl$col("a")$shift(-2)$alias("shift-2"),
    pl$col("a")$shift(2)$alias("shift+2")
  )
```

---

Expr_shrink_dtype	<i>Shrink numeric columns to the minimal required datatype</i>
-------------------	--

---

**Description**

Shrink to the dtype needed to fit the extrema of this Series. This can be used to reduce memory pressure.

**Usage**

```
Expr_shrink_dtype()
```

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = 1:3,
  b = c(1, 2, 3)
)
df

df$with_columns(pl$all()$shrink_dtype()$name$suffix("_shrunk"))
```

---

Expr_shuffle	<i>Shuffle values</i>
--------------	-----------------------

---

**Description**

Shuffle values

**Usage**

```
Expr_shuffle(seed = NULL)
```

**Arguments**

seed                    numeric value of 0 to  $2^{52}$  Seed for the random number generator. If NULL (default), a random seed value between 0 and 10000 is picked.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = 1:4)$with_columns(shuff = pl$col("a")$shuffle(seed = 1))
```

---

Expr_sign	<i>Get the sign of elements</i>
-----------	---------------------------------

---

**Description**

Get the sign of elements

**Usage**

```
Expr_sign()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(.9, -3, -0, 0, 4, NA_real_))$  
  with_columns(sign = pl$col("a")$sign())
```

---

Expr\_sin

*Compute sine*

---

**Description**

Compute sine

**Usage**

```
Expr_sin()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(0, pi / 2, pi, NA_real_))$  
  with_columns(sine = pl$col("a")$sin())
```

---

Expr\_sinh

*Compute hyperbolic sine*

---

**Description**

Compute hyperbolic sine

**Usage**

```
Expr_sinh()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(-1, asinh(0.5), 0, 1, NA_real_))$  
  with_columns(sinh = pl$col("a")$sinh())
```

---

Expr_skew	<i>Skewness</i>
-----------	-----------------

---

**Description**

Compute the sample skewness of a data set.

**Usage**

```
Expr_skew(bias = TRUE)
```

**Arguments**

`bias`                    If FALSE, then the calculations are corrected for statistical bias.

**Details**

For normally distributed data, the skewness should be about zero. For uni-modal continuous distributions, a skewness value greater than zero means that there is more weight in the right tail of the distribution.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(list(a = c(1:3, 2:1)))
df$select(pl$col("a")$skew())
```

---

Expr_slice	<i>Get a slice of an Expr</i>
------------	-------------------------------

---

**Description**

Performing a slice of length 1 on a subset of columns will recycle this value in those columns but will not change the number of rows in the data. See examples.

**Usage**

```
Expr_slice(offset, length = NULL)
```

**Arguments**

`offset`                    Numeric or expression, zero-indexed. Indicates where to start the slice. A negative value is one-indexed and starts from the end.

`length`                    Maximum number of elements contained in the slice. Default is full data.

**Value**

Expr

**Examples**

```
# as head
pl$DataFrame(list(a = 0:100))$select(
  pl$all()$slice(0, 6)
)

# as tail
pl$DataFrame(list(a = 0:100))$select(
  pl$all()$slice(-6, 6)
)

pl$DataFrame(list(a = 0:100))$select(
  pl$all()$slice(80)
)

# recycling
pl$DataFrame(mtcars)$with_columns(pl$col("mpg")$slice(0, 1)$first())
```

Expr\_sort

*Sort an Expr***Description**

Sort this column. If used in a groupby context, the groups are sorted.

**Usage**

```
Expr_sort(..., descending = FALSE, nulls_last = FALSE)
```

**Arguments**

...	Ignored.
descending	A logical. If TRUE, sort in descending order.
nulls_last	A logical. If TRUE, place null values last instead of first.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(6, 1, 0, NA, Inf, NaN))$
  with_columns(sorted = pl$col("a")$sort())
```



---

Expr_sort_by	<i>Sort Expr by order of others</i>
--------------	-------------------------------------

---

**Description**

Sort this column by the ordering of another column, or multiple other columns. If used in a groupby context, the groups are sorted.

**Usage**

```
Expr_sort_by(
  by,
  ...,
  descending = FALSE,
  nulls_last = FALSE,
  multithreaded = TRUE,
  maintain_order = FALSE
)
```

**Arguments**

by	One expression or a list of expressions and/or strings (interpreted as column names).
...	Ignored.
descending	A logical. If TRUE, sort in descending order.
nulls_last	A logical. If TRUE, place null values last instead of first.
multithreaded	A logical. If TRUE, sort using multiple threads.
maintain_order	A logical to indicate whether the order should be maintained if elements are equal.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  group = c("a", "a", "a", "b", "b", "b"),
  value1 = c(98, 1, 3, 2, 99, 100),
  value2 = c("d", "f", "b", "e", "c", "a")
)

# by one column/expression
df$with_columns(
  sorted = pl$col("group")$sort_by("value1")
)
```

```

# by two columns/expressions
df$with_columns(
  sorted = pl$col("group")$sort_by(
    list("value2", pl$col("value1")),
    descending = c(TRUE, FALSE)
  )
)

# by some expression
df$with_columns(
  sorted = pl$col("group")$sort_by(pl$col("value1")$sort(descending = TRUE))
)

```

---

Expr\_sqrt

*Compute the square root of the elements*


---

### Description

Compute the square root of the elements

### Usage

```
Expr_sqrt()
```

### Value

Expr

### Examples

```
pl$DataFrame(a = -1:3)$with_columns(a_sqrt = pl$col("a")$sqrt())
```

---

Expr\_std

*Get standard deviation*


---

### Description

Get standard deviation

### Usage

```
Expr_std(ddof = 1)
```

### Arguments

**ddof** An integer representing "Delta Degrees of Freedom": the divisor used in the calculation is  $N - \text{ddof}$ , where  $N$  represents the number of elements. By default `ddof` is 1.

**Value**

Expr

**Examples**

```
pl$select(pl$lit(1:5)$std())
```

---

Expr_sub	<i>Subtract two expressions</i>
----------	---------------------------------

---

**Description**

Method equivalent of subtraction operator `expr - other`.

**Usage**

```
Expr_sub(other)
```

**Arguments**

`other` Numeric literal or expression value.

**Value**

Expr

**See Also**

- [Arithmetic operators](#)

**Examples**

```
df = pl$DataFrame(x = 0:4)

df$with_columns(
  `x-2` = pl$col("x")$sub(2),
  `x-expr` = pl$col("x")$sub(pl$col("x")$cum_sum())
)
```

---

Expr_sum	<i>Get sum value</i>
----------	----------------------

---

**Description**

Get sum value

**Usage**

```
Expr_sum()
```

**Details**

The dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

**Value**

Expr

**Examples**

```
pl$DataFrame(x = c(1L, NA, 2L))$  
  with_columns(sum = pl$col("x")$sum())
```

---

Expr_tail	<i>Get the last n elements</i>
-----------	--------------------------------

---

**Description**

Get the last n elements

**Usage**

```
Expr_tail(n = 10)
```

**Arguments**

n                      Number of elements to take.

**Value**

Expr

**Examples**

```
pl$DataFrame(x = 1:11)$select(pl$col("x")$tail(3))
```

---

Expr_tan	<i>Compute tangent</i>
----------	------------------------

---

**Description**

Compute tangent

**Usage**

```
Expr_tan()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(0, pi / 2, pi, NA_real_))$  
  with_columns(tangent = pl$col("a")$tan())
```

---

Expr_tanh	<i>Compute hyperbolic tangent</i>
-----------	-----------------------------------

---

**Description**

Compute hyperbolic tangent

**Usage**

```
Expr_tanh()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(-1, atanh(0.5), 0, 1, NA_real_))$  
  with_columns(tanh = pl$col("a")$tanh())
```

---

Expr\_top\_k                      *Top k values*

---

**Description**

Return the k largest elements. This has time complexity:  $O(n + k \log n - \frac{k}{2})$

**Usage**

Expr\_top\_k(k)

**Arguments**

k                      Number of top values to get.

**Value**

Expr

**Examples**

```
pl$DataFrame(a = c(6, 1, 0, NA, Inf, NaN))$select(pl$col("a")$top_k(5))
```

---

Expr\_to\_physical                      *Cast an Expr to its physical representation*

---

**Description**

The following DataTypes will be converted:

- Date -> Int32
- Datetime -> Int64
- Time -> Int64
- Duration -> Int64
- Categorical -> UInt32
- List(inner) -> List(physical of inner) Other data types will be left unchanged.

**Usage**

Expr\_to\_physical()

**Value**

Expr

**Examples**

```
pl$DataFrame(
  list(vals = c("a", "x", NA, "a", "b"))
)$with_columns(
  pl$col("vals")$cast(pl$Categorical()),
  pl$col("vals")
  $cast(pl$Categorical())
  $to_physical()
  $alias("vals_physical")
)
```

Expr\_to\_r

*Convert an Expr to R output***Description**

This is mostly useful to debug an expression. It evaluates the Expr in an empty DataFrame and return the first Series to R.

**Usage**

```
Expr_to_r(
  df = NULL,
  i = 0,
  ...,
  int64_conversion = polars_options()$int64_conversion
)
```

**Arguments**

df	If NULL (default), it evaluates the Expr in an empty DataFrame. Otherwise, provide a DataFrame that the Expr should be evaluated in.
i	Numeric column to extract. Default is zero (which gives the first column).
...	Any args passed to <code>as.data.frame()</code> .
int64_conversion	How should Int64 values be handled when converting a polars object to R? <ul style="list-style-type: none"> <li>• "double" (default) converts the integer values to double.</li> <li>• "bit64" uses <code>bit64::as.integer64()</code> to do the conversion (requires the package <code>bit64</code> to be attached).</li> <li>• "string" converts Int64 values to character.</li> </ul>

**Value**

R object

**Examples**

```
pl$lit(1:3)$to_r()
```

---

Expr_to_series	<i>Convert Literal to Series</i>
----------------	----------------------------------

---

**Description**

Collect an expression based on literals into a Series.

**Usage**

```
Expr_to_series()
```

**Value**

Series

**Examples**

```
pl$lit(1:5)$to_series()
```

---

Expr_unique	<i>Get unique values</i>
-------------	--------------------------

---

**Description**

Get unique values

**Usage**

```
Expr_unique(maintain_order = FALSE)
```

**Arguments**

`maintain_order` If TRUE, the unique values are returned in order of appearance.

**Value**

Expr

**Examples**

```
pl$DataFrame(iris)$select(pl$col("Species")$unique())
```



---

Expr\_unique\_counts      *Count unique values*

---

**Description**

Return a count of the unique values in the order of appearance. This method differs from `$value_counts()` in that it does not return the values, only the counts and it might be faster.

**Usage**

```
Expr_unique_counts()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(iris)$select(pl$col("Species"))$unique_counts()
```

---

Expr\_upper\_bound      *Find the upper bound of a DataType*

---

**Description**

Find the upper bound of a DataType

**Usage**

```
Expr_upper_bound()
```

**Value**

Expr

**Examples**

```
pl$DataFrame(  
  x = c(1, 2, 3), y = -2:0,  
  schema = list(x = pl$Float64, y = pl$Int32)  
)$  
  select(pl$all())$upper_bound()
```

---

Expr_value_counts	<i>Value counts</i>
-------------------	---------------------

---

**Description**

Count all unique values and create a struct mapping value to count.

**Usage**

```
Expr_value_counts(..., sort = FALSE, parallel = FALSE, name, normalize = FALSE)
```

**Arguments**

...	Ignored.
sort	Ensure the output is sorted from most values to least.
parallel	Better to turn this off in the aggregation context, as it can lead to contention.
name	Give the resulting count column a specific name. The default is "count" if <code>normalize = FALSE</code> and "proportion" if <code>normalize = TRUE</code> .
normalize	If TRUE, it gives relative frequencies of the unique values instead of their count.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(iris)
df$select(pl$col("Species")$value_counts())$unnest()
df$select(pl$col("Species")$value_counts(normalize = TRUE))$unnest()
```

---

Expr_var	<i>Get variance</i>
----------	---------------------

---

**Description**

Get variance

**Usage**

```
Expr_var(ddof = 1)
```

**Arguments**

ddof	An integer representing "Delta Degrees of Freedom": the divisor used in the calculation is $N - \text{ddof}$ , where $N$ represents the number of elements. By default <code>ddof</code> is 1.
------	--

**Value**

Expr

**Examples**

```
pl$select(pl$lit(1:5)$var())
```

---

 Expr\_when\_then\_otherwise

*Make a when-then-otherwise expression*


---

**Description**

when-then-otherwise is similar to R `ifelse()`. Always initiated by a `pl$when(<condition>)$then(<value if condition>)` and optionally followed by chaining one or more `$when(<condition>)$then(<value if condition>)` statements.

**Usage**

```
pl_when(...)
```

```
When_then(statement)
```

```
Then_when(...)
```

```
Then_otherwise(statement)
```

```
ChainedWhen_then(statement)
```

```
ChainedThen_when(...)
```

```
ChainedThen_otherwise(statement)
```

**Arguments**

...	Expr or something coercible to an Expr that returns a boolean each row.
statement	Expr or something coercible to an Expr value to insert in <code>\$then()</code> or <code>\$otherwise()</code> . A character vector is parsed as column names.

**Details**

Chained when-then operations should be read like `if, else if, else if, ...` in R, not as `if, if, if, ...`, i.e. the first condition that evaluates to true will be picked.

If none of the conditions are true, an optional `$otherwise(<value if all statements are false>)` can be appended at the end. If not appended, and none of the conditions are true, null will be returned.

RPolarsThen objects and RPolarsChainedThen objects (returned by `$then()`) stores the same methods as Expr.

**Value**

- `pl$when()` returns a When object
- `<When>$then()` returns a Then object
- `<Then>$when()` returns a ChainedWhen object
- `<ChainedWhen>$then()` returns a ChainedThen object
- `$otherwise()` returns an [Expr](#) object.

**Examples**

```
df = pl$DataFrame(foo = c(1, 3, 4), bar = c(3, 4, 0))

# Add a column with the value 1, where column "foo" > 2 and the value -1
# where it isn't.
df$with_columns(
  val = pl$when(pl$col("foo") > 2)$then(1)$otherwise(-1)
)

# With multiple when-then chained:
df$with_columns(
  val = pl$when(pl$col("foo") > 2)
    $then(1)
    $when(pl$col("bar") > 2)
    $then(4)
    $otherwise(-1)
)

# The `otherwise` at the end is optional.
# If left out, any rows where none of the `when()` expressions are evaluated to `true`,
# are set to `null`
df$with_columns(
  val = pl$when(pl$col("foo") > 2)$then(1)
)

# Pass multiple predicates, each of which must be met:
df$with_columns(
  val = pl$when(
    pl$col("bar") > 0,
    pl$col("foo") %% 2 != 0
  )
  $then(99)
  $otherwise(-1)
)

# In `then()`, a character vector is parsed as column names
df$with_columns(baz = pl$when(pl$col("foo") %% 2 == 1)$then("bar"))

# So use `pl$lit()` to insert a string
df$with_columns(baz = pl$when(pl$col("foo") %% 2 == 1)$then(pl$lit("bar")))
```

---

Expr_xor	<i>Apply logical XOR on two expressions</i>
----------	---

---

**Description**

Combine two boolean expressions with XOR.

**Usage**

```
Expr_xor(other)
```

**Arguments**

other                    numeric or string value; accepts expression input.

**Value**

Expr

**Examples**

```
pl$lit(TRUE)$xor(pl$lit(FALSE))
```

---

global_rpool_cap	<i>Get/set global R session pool capacity (DEPRECATED)</i>
------------------	--

---

**Description**

Deprecated. Use `polars_options()` to get, and `pl$set_options()` to set.

**Usage**

```
pl_get_global_rpool_cap()
```

```
pl_set_global_rpool_cap(n)
```

**Arguments**

n                        Integer, the capacity limit R sessions to process R code.

**Details**

Background R sessions communicate via polars arrow IPC (series/vectors) or R serialize + shared memory buffers via the rust crate `ipc-channel`. Multi-process communication has overhead because all data must be serialized/de-serialized and sent via buffers. Using multiple R sessions will likely only give a speed-up in a low io - high cpu scenario. Native polars query syntax runs in threads and have no overhead. Polars has as default double as many thread workers as cores. If any worker are queuing for or using R sessions, other workers can still continue any native polars parts as much as possible.

**Value**

`polars_options()$rpool_cap` returns the capacity ("limit") of co-running external R sessions / processes. `polars_options()$rpool_active` is the number of R sessions are already spawned in the pool. `rpool_cap` is the limit of new R sessions to spawn. Anytime a polars thread worker needs a background R session specifically to run R code embedded in a query via `$map_batches(..., in_background = TRUE)` or `$map_elements(..., in_background = TRUE)`, it will obtain any R session idling in `rpool`, or spawn a new R session (process) if capacity is not already reached. If capacity is already reached, the thread worker will sleep and in a R job queue until an R session is idle.

**Examples**

```
default = polars_options()$rpool_cap |> print()
options(polars.rpool_cap = 8)
polars_options()$rpool_cap
options(polars.rpool_cap = default)
polars_options()$rpool_cap
```

---

GroupBy\_agg

*Aggregate over a GroupBy*


---

**Description**

Aggregate a DataFrame over a groupby

**Usage**

```
GroupBy_agg(...)
```

**Arguments**

... `exprs` to aggregate over. ... args can also be passed wrapped in a list `$agg(list(e1, e2, e3))`

**Value**

aggregated DataFrame

**Examples**

```
pl$DataFrame(
  foo = c("one", "two", "two", "one", "two"),
  bar = c(5, 3, 2, 4, 1)
)$group_by("foo")$agg(
  pl$col("bar")$sum()$name$suffix("_sum"),
  pl$col("bar")$mean()$alias("bar_tail_sum")
)
```

---

GroupBy\_class

*Operations on Polars grouped DataFrame*


---

**Description**

The GroupBy class in R, is just another interface on top of the [DataFrame](#) in rust polars. Groupby does not use the rust api for `<DataFrame>$group_by() + $agg()` because the groupby-struct is a reference to a DataFrame and that reference will share lifetime with its parent DataFrame.

**Details**

There is no way to expose lifetime limited objects via extendr currently (might be quirky anyhow with R GC). Instead the inputs for the `group_by` are just stored on R side, until also `agg` is called. Which will end up in a self-owned DataFrame object and all is fine. `groupby` aggs are performed via the rust polars `LazyGroupBy` methods, see `DataFrame.groupby_agg` method.

**Active bindings****columns:**

`$columns` returns a character vector with the column names.

**Examples**

```
as_polars_df(mtcars)$group_by("cyl")$agg(
  pl$col("mpg")$sum()
)
```

---

GroupBy\_first

*GroupBy First*


---

**Description**

Reduce the groups to the first value.

**Usage**

```
GroupBy_first()
```

**Value**

aggregated DataFrame

**Examples**

```
df = pl$DataFrame(  
  a = c(1, 2, 2, 3, 4, 5),  
  b = c(0.5, 0.5, 4, 10, 13, 14),  
  c = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE),  
  d = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")  
)  
df$group_by("d", maintain_order = TRUE)$first()
```

---

GroupBy\_last

*GroupBy Last*

---

**Description**

Reduce the groups to the last value.

**Usage**

```
GroupBy_last()
```

**Value**

aggregated DataFrame

**Examples**

```
df = pl$DataFrame(  
  a = c(1, 2, 2, 3, 4, 5),  
  b = c(0.5, 0.5, 4, 10, 13, 14),  
  c = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE),  
  d = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")  
)  
df$group_by("d", maintain_order = TRUE)$last()
```

---

GroupBy\_max

*GroupBy Max*

---

**Description**

Reduce the groups to the maximum value.

**Usage**

```
GroupBy_max()
```



**Value**

aggregated DataFrame

**Examples**

```
df = pl$DataFrame(  
  a = c(1, 2, 2, 3, 4, 5),  
  b = c(0.5, 0.5, 4, 10, 13, 14),  
  c = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE),  
  d = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")  
)  
df$group_by("d", maintain_order = TRUE)$max()
```

---

GroupBy\_mean

*GroupBy Mean*

---

**Description**

Reduce the groups to the mean value.

**Usage**

GroupBy\_mean()

**Value**

aggregated DataFrame

**Examples**

```
df = pl$DataFrame(  
  a = c(1, 2, 2, 3, 4, 5),  
  b = c(0.5, 0.5, 4, 10, 13, 14),  
  c = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE),  
  d = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")  
)  
df$group_by("d", maintain_order = TRUE)$mean()
```

---

GroupBy_median	<i>GroupBy Median</i>
----------------	-----------------------

---

**Description**

Reduce the groups to the median value.

**Usage**

```
GroupBy_median()
```

**Value**

aggregated DataFrame

**Examples**

```
df = pl$DataFrame(  
  a = c(1, 2, 2, 3, 4, 5),  
  b = c(0.5, 0.5, 4, 10, 13, 14),  
  c = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE),  
  d = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")  
)  
df$group_by("d", maintain_order = TRUE)$median()
```

---

GroupBy_min	<i>GroupBy Min</i>
-------------	--------------------

---

**Description**

Reduce the groups to the minimum value.

**Usage**

```
GroupBy_min()
```

**Value**

aggregated DataFrame

**Examples**

```
df = pl$DataFrame(  
  a = c(1, 2, 2, 3, 4, 5),  
  b = c(0.5, 0.5, 4, 10, 13, 14),  
  c = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE),  
  d = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")  
)  
df$group_by("d", maintain_order = TRUE)$min()
```

---

GroupBy\_null\_count      *GroupBy null count*

---

**Description**

Create a new DataFrame that shows the null counts per column.

**Usage**

```
GroupBy_null_count()
```

**Value**

DataFrame

**Examples**

```
x = mtcars
x[1:10, 3:5] = NA
pl$DataFrame(x)$group_by("cyl")$null_count()
```

---

GroupBy\_quantile      *Quantile*

---

**Description**

Aggregate the columns in the DataFrame to their quantile value.

**Usage**

```
GroupBy_quantile(quantile, interpolation = "nearest")
```

**Arguments**

quantile      numeric Quantile between 0.0 and 1.0.  
interpolation      string Interpolation method: "nearest", "higher", "lower", "midpoint", or "linear".

**Value**

GroupBy

**Examples**

```
pl$DataFrame(mtcars)$lazy()$quantile(.4)$collect()
```

---

GroupBy_shift	<i>Shift the values by a given period</i>
---------------	---

---

**Description**

Shift the values by a given period

**Usage**

```
GroupBy_shift(n = 1, fill_value = NULL)
```

**Arguments**

n	Number of indices to shift forward. If a negative value is passed, values are shifted in the opposite direction instead.
fill_value	Fill the resulting null values with this value. Accepts expression input. Non-expression inputs are parsed as literals.

**Value**

GroupBy

**Examples**

```
pl$DataFrame(mtcars)$group_by("cyl")$shift(2)
```

---

GroupBy_std	<i>GroupBy Std</i>
-------------	--------------------

---

**Description**

Reduce the groups to the standard deviation value.

**Usage**

```
GroupBy_std()
```

**Value**

aggregated DataFrame

**Examples**

```
df = pl$DataFrame(  
  a = c(1, 2, 2, 3, 4, 5),  
  b = c(0.5, 0.5, 4, 10, 13, 14),  
  c = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE),  
  d = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")  
)  
df$group_by("d", maintain_order = TRUE)$std()
```

---

GroupBy\_sum

*GroupBy Sum*

---

**Description**

Reduce the groups to the sum value.

**Usage**

```
GroupBy_sum()
```

**Value**

aggregated DataFrame

**Examples**

```
df = pl$DataFrame(  
  a = c(1, 2, 2, 3, 4, 5),  
  b = c(0.5, 0.5, 4, 10, 13, 14),  
  c = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE),  
  d = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")  
)  
df$group_by("d", maintain_order = TRUE)$sum()
```

---

GroupBy\_ungroup

*GroupBy\_ungroup*

---

**Description**

Revert the group by operation.

**Usage**

```
GroupBy_ungroup()
```

**Value**

DataFrame

**Examples**

```
gb = pl$DataFrame(mtcars)$group_by("cyl")
gb

gb$ungroup()
```

---

GroupBy\_var

*GroupBy Var*

---

**Description**

Reduce the groups to the variance value.

**Usage**

GroupBy\_var()

**Value**

aggregated DataFrame

**Examples**

```
df = pl$DataFrame(
  a = c(1, 2, 2, 3, 4, 5),
  b = c(0.5, 0.5, 4, 10, 13, 14),
  c = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE),
  d = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")
)
df$group_by("d", maintain_order = TRUE)$var()
```

---

head.RPolarsDataFrame *Return the first or the last n parts of an object*

---

**Description**

They are equivalent to \$head() and \$tail() methods.

**Usage**

```
## S3 method for class 'RPolarsDataFrame'  
head(x, n = 6L, ...)  
  
## S3 method for class 'RPolarsLazyFrame'  
head(x, n = 6L, ...)  
  
## S3 method for class 'RPolarsDataFrame'  
tail(x, n = 6L, ...)  
  
## S3 method for class 'RPolarsLazyFrame'  
tail(x, n = 6L, ...)
```

**Arguments**

x	A polars object
n	An integer vector of length 1. Note that negative values are not supported for if x is a <a href="#">LazyFrame</a> .
...	Ignored

**Value**

A polars object of the same class as x

**See Also**

- [<DataFrame>\\$head\(\)](#)
- [<LazyFrame>\\$head\(\)](#)
- [<DataFrame>\\$tail\(\)](#)
- [<LazyFrame>\\$tail\(\)](#)
- [<LazyFrame>\\$fetch\(\)](#)

**Examples**

```
df = pl$DataFrame(foo = 1:5, bar = 6:10, ham = letters[1:5])  
lf = df$lazy()  
  
head(df, 2)  
tail(df, 2)  
  
head(lf, 2)  
tail(lf, 2)  
  
head(df, -2)  
tail(df, -2)
```

---

```
infer_nanoarrow_schema.RPolarsDataFrame
```

*Infer nanoarrow schema from a Polars object*

---

## Description

Infer nanoarrow schema from a Polars object

## Usage

```
## S3 method for class 'RPolarsDataFrame'  
infer_nanoarrow_schema(x, ..., compat_level = FALSE)  
  
## S3 method for class 'RPolarsSeries'  
infer_nanoarrow_schema(x, ..., compat_level = FALSE)
```

## Arguments

x	A polars object
...	Ignored
compat_level	Use a specific compatibility level when exporting Polars' internal data structures. This can be: <ul style="list-style-type: none"><li>• an integer indicating the compatibility version (currently only 0 for oldest and 1 for newest);</li><li>• a logical value with TRUE for the newest version and FALSE for the oldest version.</li></ul>

## Examples

```
library(nanoarrow)  
  
pl_df = as_polars_df(mtcars)$select("mpg", "cyl")  
pl_s = as_polars_series(letters)  
  
infer_nanoarrow_schema(pl_df)  
infer_nanoarrow_schema(pl_s)
```



---

is_polars_df	<i>Test if the object is a polars DataFrame</i>
--------------	---

---

**Description**

This function tests if the object is a polars DataFrame.

**Usage**

```
is_polars_df(x)
```

**Arguments**

x                    An object

**Value**

A logical value

**Examples**

```
is_polars_df(mtcars)
```

```
is_polars_df(as_polars_df(mtcars))
```

---

is_polars_dtype	<i>Test if the object a polars DataType</i>
-----------------	---

---

**Description**

Test if the object a polars DataType

**Usage**

```
is_polars_dtype(x, include_unknown = FALSE)
```

**Arguments**

x                    An object

include\_unknown

If FALSE (default), pl\$Unknown is considered as an invalid datatype.

**Value**

A logical value

**Examples**

```
is_polars_dtype(pl$Int64)
is_polars_dtype(mtcars)
is_polars_dtype(pl$Unknown)
is_polars_dtype(pl$Unknown, include_unknown = TRUE)
```

---

is_polars_lf	<i>Test if the object is a polars LazyFrame</i>
--------------	---

---

**Description**

This function tests if the object is a polars LazyFrame.

**Usage**

```
is_polars_lf(x)
```

**Arguments**

x	An object
---	-----------

**Value**

A logical value

**Examples**

```
is_polars_lf(mtcars)
is_polars_lf(as_polars_lf(mtcars))
```

---

is_polars_series	<i>Test if the object is a polars Series</i>
------------------	--

---

**Description**

This function tests if the object is a polars Series.

**Usage**

```
is_polars_series(x)
```

**Arguments**

x	An object
---	-----------

**Value**

A logical value

**Examples**

```
is_polars_series(1:3)
is_polars_series(as_polars_series(1:3))
```

---

```
knit_print.RPolarsDataFrame
      knit print polars DataFrame
```

---

**Description**

Mimics Python Polars' NotebookFormatter for HTML outputs.

**Usage**

```
## S3 method for class 'RPolarsDataFrame'
knit_print(x, ...)
```

**Arguments**

x	a polars DataFrame to knit_print
...	additional arguments, not used

**Details**

Outputs HTML tables if the output format is HTML and the document's df\_print option is not "default" or "tibble".

Or, the output format can be enforced with R's options function as follows:

- options(polars.df\_knitr\_print = "default") for the default print method.
- options(polars.df\_knitr\_print = "html") for the HTML table.

**Value**

invisible x or NULL

---

LazyFrame_cast	<i>Cast LazyFrame column(s) to the specified dtype</i>
----------------	--

---

### Description

This allows to convert all columns to a datatype or to convert only specific columns. Contrarily to the Python implementation, it is not possible to convert all columns of a specific datatype to another datatype.

### Usage

```
LazyFrame_cast(dtypes, ..., strict = TRUE)
```

### Arguments

dtypes	Either a datatype or a list where the names are column names and the values are the datatypes to convert to.
...	Ignored.
strict	If TRUE (default), throw an error if a cast could not be done (for instance, due to an overflow). Otherwise, return null.

### Value

A LazyFrame

### Examples

```
lf = pl$LazyFrame(  
  foo = 1:3,  
  bar = c(6, 7, 8),  
  ham = as.Date(c("2020-01-02", "2020-03-04", "2020-05-06"))  
)  
  
# Cast only some columns  
lf$cast(list(foo = pl$Float32, bar = pl$UInt8))$collect()  
  
# Cast all columns to the same type  
lf$cast(pl$String)$collect()
```

## Description

The LazyFrame-class is simply two environments of respectively the public and private methods/function calls to the polars rust side. The instantiated LazyFrame-object is an externalptr to a lowlevel rust polars LazyFrame object. The pointer address is the only statefulness of the LazyFrame object on the R side. Any other state resides on the rust side. The S3 method `.DollarNames.RPolarsLazyFrame` exposes all public `$foobar()`-methods which are callable onto the object.

Most methods return another LazyFrame-class instance or similar which allows for method chaining. This class system in lack of a better name could be called "environment classes" and is the same class system extendr provides, except here there is both a public and private set of methods. For implementation reasons, the private methods are external and must be called from `.pr$LazyFrame$methodname()`. Also, all private methods must take any self as an argument, thus they are pure functions. Having the private methods as pure functions solved/simplified self-referential complications.

DataFrame and LazyFrame can both be said to be a Frame. To convert use `<DataFrame>$lazy()` and `<LazyFrame>$collect()`. You can also create a LazyFrame directly with `pl$LazyFrame()`. This is quite similar to the lazy-collect syntax of the dplyr package to interact with database connections such as SQL variants. Most SQL databases would be able to perform the same optimizations as polars such predicate pushdown and projection pushdown. However polars can interact and optimize queries with both SQL DBs and other data sources such parquet files simultaneously.

## Active bindings

**columns:**

`$columns` returns a character vector with the column names.

**dtypes:**

`$dtypes` returns a unnamed list with the [data type](#) of each column.

**schema:**

`$schema` returns a named list with the [data type](#) of each column.

**width:**

`$width` returns the number of columns in the LazyFrame.

## Conversion to R data types considerations

When converting Polars objects, such as [DataFrames](#) to R objects, for example via the `as.data.frame()` generic function, each type in the Polars object is converted to an R type. In some cases, an error may occur because the conversion is not appropriate. In particular, there is a high possibility of an error when converting a [Datetime](#) type without a time zone. A [Datetime](#) type without a time zone in Polars is converted to the [POSIXct](#) type in R, which takes into account the time zone in which the R session is running (which can be checked with the `Sys.timezone()` function). In this case, if ambiguous times are included, a conversion error will occur. In such cases, change

the session time zone using `Sys.setenv(TZ = "UTC")` and then perform the conversion, or use the `$dt$replace_time_zone()` method on the Datetime type column to explicitly specify the time zone before conversion.

```
# Due to daylight savings, clocks were turned forward 1 hour on Sunday, March 8, 2020, 2:00:00 am
# so this particular date-time doesn't exist
non_existent_time = as_polars_series("2020-03-08 02:00:00")$str$strptime(pl$Datetime(), "%F %T")
```

```
withr::with_timezone(
  "America/New_York",
  {
    tryCatch(
      # This causes an error due to the time zone (the `TZ` env var is affected).
      as.vector(non_existent_time),
      error = function(e) e
    )
  }
)
```

```
#> <error: in to_r: ComputeError(ErrString("datetime '2020-03-08 02:00:00' is non-existent in time zone
```

```
withr::with_timezone(
  "America/New_York",
  {
    # This is safe.
    as.vector(non_existent_time$dt$replace_time_zone("UTC"))
  }
)
#> [1] "2020-03-08 02:00:00 UTC"
```

## Examples

```
# see all exported methods
ls(.pr$env$RPolarsLazyFrame)

# see all private methods (not intended for regular use)
ls(.pr$LazyFrame)

## Practical example ##
# First writing R iris dataset to disk, to illustrate a difference
temp_filepath = tempfile()
write.csv(iris, temp_filepath, row.names = FALSE)

# Following example illustrates 2 ways to obtain a LazyFrame

# The-Okay-way: convert an in-memory DataFrame to LazyFrame

# eager in-mem R data.frame
Rdf = read.csv(temp_filepath)
```

```

# eager in-mem polars DataFrame
Pdf = as_polars_df(Rdf)

# lazy frame starting from in-mem DataFrame
Ldf_okay = Pdf$lazy()

# The-Best-Way: LazyFrame created directly from a data source is best...
Ldf_best = pl$scan_csv(temp_filepath)

# ... as if to e.g. filter the LazyFrame, that filtering also could predicate will be
# pushed down in the execution stack to the csv_reader, and thereby only bringing into
# memory the rows matching to filter.
# apply filter:
filter_expr = pl$col("Species") == "setosa" # get only rows where Species is setosa
Ldf_okay = Ldf_okay$filter(filter_expr) # overwrite LazyFrame with new
Ldf_best = Ldf_best$filter(filter_expr)

# the non optimized plans are similar, on entire in-mem csv, apply filter
Ldf_okay$explain(optimized = FALSE)
Ldf_best$explain(optimized = FALSE)

# NOTE For Ldf_okay, the full time to load csv already paid when creating Rdf and Pdf

# The optimized plan are quite different, Ldf_best will read csv and perform filter simultaneously
Ldf_okay$explain()
Ldf_best$explain()

# To acquire result in-mem use $collect()
Pdf_okay = Ldf_okay$collect()
Pdf_best = Ldf_best$collect()

# verify tables would be the same
all.equal(
  Pdf_okay$to_data_frame(),
  Pdf_best$to_data_frame()
)

# a user might write it as a one-liner like so:
Pdf_best2 = pl$scan_csv(temp_filepath)$filter(pl$col("Species") == "setosa")

```

---

LazyFrame\_clear

*Create an empty or n-row null-filled copy of the LazyFrame*


---

### Description

Returns a n-row null-filled LazyFrame with an identical schema. n can be greater than the current number of rows in the LazyFrame.

**Usage**

```
LazyFrame_clear(n = 0)
```

**Arguments**

n                      Number of (null-filled) rows to return in the cleared frame.

**Value**

A n-row null-filled LazyFrame with an identical schema

**Examples**

```
df = pl$LazyFrame(  
  a = c(NA, 2, 3, 4),  
  b = c(0.5, NA, 2.5, 13),  
  c = c(TRUE, TRUE, FALSE, NA)  
)  
  
df$clear()  
  
df$clear(n = 5)
```

---

LazyFrame\_clone

*Clone a LazyFrame*

---

**Description**

This makes a very cheap deep copy/clone of an existing [LazyFrame](#). Rarely useful as LazyFrames are nearly 100% immutable. Any modification of a LazyFrame should lead to a clone anyways, but this can be useful when dealing with attributes (see examples).

**Usage**

```
LazyFrame_clone()
```

**Value**

A LazyFrame

**Examples**

```
df1 = pl$LazyFrame(iris)  
  
# Make a function to take a LazyFrame, add an attribute, and return a LazyFrame  
give_attr = function(data) {  
  attr(data, "created_on") = "2024-01-29"  
  data  
}
```



```

df2 = give_attr(df1)

# Problem: the original LazyFrame also gets the attribute while it shouldn't!
attributes(df1)

# Use $clone() inside the function to avoid that
give_attr = function(data) {
  data = data$clone()
  attr(data, "created_on") = "2024-01-29"
  data
}
df1 = pl$LazyFrame(iris)
df2 = give_attr(df1)

# now, the original LazyFrame doesn't get this attribute
attributes(df1)

```

---

LazyFrame\_collect      *Collect a query into a DataFrame*

---

## Description

`$collect()` performs the query on the LazyFrame. It returns a DataFrame

## Usage

```

LazyFrame_collect(
  ...,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  comm_subplan_elim = TRUE,
  comm_subexpr_elim = TRUE,
  cluster_with_columns = TRUE,
  streaming = FALSE,
  no_optimization = FALSE,
  collect_in_background = FALSE
)

```

## Arguments

...	Ignored.
type_coercion	Logical. Coerce types such that operations succeed and run on minimal required memory.
predicate_pushdown	Logical. Applies filters as early as possible at scan level.

<code>projection_pushdown</code>	Logical. Select only the columns that are needed at the scan level.
<code>simplify_expression</code>	Logical. Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.
<code>slice_pushdown</code>	Logical. Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. <code>join\$head(10)</code> ).
<code>comm_subplan_elim</code>	Logical. Will try to cache branching subplans that occur on self-joins or unions.
<code>comm_subexpr_elim</code>	Logical. Common subexpressions will be cached and reused.
<code>cluster_with_columns</code>	Combine sequential independent calls to <code>with_columns()</code> .
<code>streaming</code>	Logical. Run parts of the query in a streaming fashion (this is in an alpha state).
<code>no_optimization</code>	Logical. Sets the following parameters to FALSE: <code>predicate_pushdown</code> , <code>projection_pushdown</code> , <code>slice_pushdown</code> , <code>comm_subplan_elim</code> , <code>comm_subexpr_elim</code> , <code>cluster_with_columns</code> .
<code>collect_in_background</code>	Logical. Detach this query from R session. Computation will start in background. Get a handle which later can be converted into the resulting DataFrame. Useful in interactive mode to not lock R session.

## Details

Note: use `$fetch(n)` if you want to run your query on the first `n` rows only. This can be a huge time saver in debugging queries.

## Value

A DataFrame

## See Also

- `$fetch()` - fast limited query check
- `$profile()` - same as `$collect()` but also returns a table with each operation profiled.
- `$collect_in_background()` - non-blocking collect returns a future handle. Can also just be used via `$collect(collect_in_background = TRUE)`.
- `$sink_parquet()` streams query to a parquet file.
- `$sink_ipc()` streams query to a arrow file.

## Examples

```
pl$LazyFrame(iris)$filter(pl$col("Species") == "setosa")$collect()
```

---

LazyFrame\_collect\_in\_background  
*Collect a query in background*

---

### Description

This doesn't block the R session as it calls `$collect()` in a detached thread. This can also be used via `$collect(collect_in_background = TRUE)`.

### Usage

```
LazyFrame_collect_in_background()
```

### Details

This function immediately returns an `RThreadHandle`. Use `<RPolarsRThreadHandle>$is_finished()` to see if done. Use `<RPolarsRThreadHandle>$join()` to wait and get the final result.

It is useful to not block the R session while query executes. If you use `<Expr>$map_batches()` or `<Expr>$map_elements()` to run R functions in the query, then you must pass `in_background = TRUE` in `$map_batches()` (or `$map_elements()`). Otherwise, `$collect_in_background()` will fail because the main R session is not available for polars execution. See also examples below.

### Value

`RThreadHandle`, a future-like thread handle for the task

### Examples

```
# Some expression which does contain a map
expr = pl$col("mpg")$map_batches(
  \(x) {
    Sys.sleep(.1)
    x * 0.43
  },
  in_background = TRUE # set TRUE if collecting in background queries with $map or $apply
)$alias("kml")

# return is immediately a handle to another thread.
handle = pl$LazyFrame(mtcars)$with_columns(expr)$collect_in_background()

# ask if query is done
if (!handle$is_finished()) print("not done yet")

# get result, blocking until polars query is done
df = handle$join()
df
```

---

LazyFrame\_drop      *Drop columns of a LazyFrame*

---

**Description**

Drop columns of a LazyFrame

**Usage**

```
LazyFrame_drop(..., strict = TRUE)
```

**Arguments**

...                    Characters of column names to drop. Passed to `pl$col()`.

strict                Validate that all column names exist in the schema and throw an exception if a column name does not exist in the schema.

**Value**

LazyFrame

**Examples**

```
pl$LazyFrame(mtcars)$drop(c("mpg", "hp"))$collect()

# equivalent
pl$LazyFrame(mtcars)$drop("mpg", "hp")$collect()
```

---

LazyFrame\_drop\_nulls      *Drop nulls (missing values)*

---

**Description**

Drop all rows that contain nulls (which correspond to NA in R).

**Usage**

```
LazyFrame_drop_nulls(subset = NULL)
```

**Arguments**

subset                A character vector with the names of the column(s) for which nulls are considered. If NULL (default), use all columns.

**Value**

LazyFrame

**Examples**

```

tmp = mtcars
tmp[1:3, "mpg"] = NA
tmp[4, "hp"] = NA
tmp = pl$LazyFrame(tmp)

# number of rows in `tmp` before dropping nulls
tmp$collect()$height

tmp$drop_nulls()$collect()$height
tmp$drop_nulls("mpg")$collect()$height
tmp$drop_nulls(c("mpg", "hp"))$collect()$height

```

---

LazyFrame\_explain      *Create a string representation of the query plan*

---

**Description**

The query plan is read from bottom to top. When `optimized = FALSE`, the query as it was written by the user is shown. This is not what Polars runs. Instead, it applies optimizations that are displayed by default by `$explain()`. One classic example is the predicate pushdown, which applies the filter as early as possible (i.e. at the bottom of the plan).

**Usage**

```

LazyFrame_explain(
  ...,
  format = "plain",
  optimized = TRUE,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  comm_subplan_elim = TRUE,
  comm_subexpr_elim = TRUE,
  cluster_with_columns = TRUE,
  streaming = FALSE
)

```

**Arguments**

<code>...</code>	Ignored.
<code>format</code>	The format to use for displaying the logical plan. Must be either "plain" (default) or "tree".
<code>optimized</code>	Return an optimized query plan. If TRUE (default), the subsequent optimization flags control which optimizations run.

type_coercion	Logical. Coerce types such that operations succeed and run on minimal required memory.
predicate_pushdown	Logical. Applies filters as early as possible at scan level.
projection_pushdown	Logical. Select only the columns that are needed at the scan level.
simplify_expression	Logical. Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.
slice_pushdown	Logical. Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. <code>join\$head(10)</code> ).
comm_subplan_elim	Logical. Will try to cache branching subplans that occur on self-joins or unions.
comm_subexpr_elim	Logical. Common subexpressions will be cached and reused.
cluster_with_columns	Combine sequential independent calls to <code>with_columns()</code> .
streaming	Logical. Run parts of the query in a streaming fashion (this is in an alpha state).

**Value**

A character value containing the query plan.

**Examples**

```

lazy_frame = pl$LazyFrame(iris)

# Prepare your query
lazy_query = lazy_frame$sort("Species")$filter(pl$col("Species") != "setosa")

# This is the query that was written by the user, without any optimizations
# (use cat() for better printing)
lazy_query$explain(optimized = FALSE) |> cat()

# This is the query after `polars` optimizes it: instead of sorting first and
# then filtering, it is faster to filter first and then sort the rest.
lazy_query$explain() |> cat()

# Also possible to see this as tree format
lazy_query$explain(format = "tree") |> cat()

```

---

LazyFrame_explode	<i>Explode columns containing a list of values</i>
-------------------	--

---

**Description**

This will take every element of a list column and add it on an additional row.

**Usage**

```
LazyFrame_explode(...)
```

**Arguments**

... Column(s) to be exploded as individual Into<Expr> or list/vector of Into<Expr>. In a handful of places in rust-polars, only the plain variant Expr::Column is accepted. This is currently one of such places. Therefore pl\$col("name") and pl\$all() is allowed, not pl\$col("name")\$alias("newname"). "name" is implicitly converted to pl\$col("name").

**Details**

Only columns of DataType List or Array can be exploded.

Named expressions like \$explode(a = pl\$col("b")) will not implicitly trigger \$alias("a") here, due to only variant Expr::Column is supported in rust-polars.

**Value**

LazyFrame

**Examples**

```
df = pl$LazyFrame(
  letters = c("aa", "aa", "bb", "cc"),
  numbers = list(1, c(2, 3), c(4, 5), c(6, 7, 8)),
  numbers_2 = list(0, c(1, 2), c(3, 4), c(5, 6, 7)) # same structure as numbers
)
df

# explode a single column, append others
df$explode("numbers")$collect()

# explode two columns of same nesting structure, by names or the common dtype
# "List(Float64)"
df$explode("numbers", "numbers_2")$collect()
df$explode(pl$col(pl$List(pl$Float64)))$collect()
```

---

LazyFrame\_fetch

*Fetch n rows of a LazyFrame*

---

**Description**

This is similar to \$collect() but limit the number of rows to collect. It is mostly useful to check that a query works as expected.

**Usage**

```
LazyFrame_fetch(
  n_rows = 500,
  ...,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  comm_subplan_elim = TRUE,
  comm_subexpr_elim = TRUE,
  cluster_with_columns = TRUE,
  streaming = FALSE,
  no_optimization = FALSE
)
```

**Arguments**

<code>n_rows</code>	Integer. Maximum number of rows to fetch.
<code>...</code>	Ignored.
<code>type_coercion</code>	Logical. Coerce types such that operations succeed and run on minimal required memory.
<code>predicate_pushdown</code>	Logical. Applies filters as early as possible at scan level.
<code>projection_pushdown</code>	Logical. Select only the columns that are needed at the scan level.
<code>simplify_expression</code>	Logical. Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.
<code>slice_pushdown</code>	Logical. Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. <code>join\$head(10)</code> ).
<code>comm_subplan_elim</code>	Logical. Will try to cache branching subplans that occur on self-joins or unions.
<code>comm_subexpr_elim</code>	Logical. Common subexpressions will be cached and reused.
<code>cluster_with_columns</code>	Combine sequential independent calls to <code>with_columns()</code> .
<code>streaming</code>	Logical. Run parts of the query in a streaming fashion (this is in an alpha state).
<code>no_optimization</code>	Logical. Sets the following parameters to FALSE: <code>predicate_pushdown</code> , <code>projection_pushdown</code> , <code>slice_pushdown</code> , <code>comm_subplan_elim</code> , <code>comm_subexpr_elim</code> , <code>cluster_with_columns</code> .

**Details**

`$fetch()` does not guarantee the final number of rows in the DataFrame output. It only guarantees that `n` rows are used at the beginning of the query. Filters, join operations and a lower number of rows available in the scanned file influence the final number of rows.



**Value**

A DataFrame of maximum n\_rows

**See Also**

- `$collect()` - regular collect.
- `$profile()` - same as `$collect()` but also returns a table with each operation profiled.
- `$collect_in_background()` - non-blocking collect returns a future handle. Can also just be used via `$collect(collect_in_background = TRUE)`.
- `$sink_parquet()` streams query to a parquet file.
- `$sink_ipc()` streams query to a arrow file.

**Examples**

```
# fetch 3 rows
pl$LazyFrame(iris)$fetch(3)

# this fetch-query returns 4 rows, because we started with 3 and appended one
# row in the query (see section 'Details')
pl$LazyFrame(iris)$
  select(pl$col("Species")$append("flora gigantea, alien"))$
  fetch(3)
```

---

LazyFrame\_fill\_nan      *Fill floating point NaN value with a fill value*

---

**Description**

Fill floating point NaN value with a fill value

**Usage**

```
LazyFrame_fill_nan(value)
```

**Arguments**

value                      Value used to fill NaN values.

**Value**

LazyFrame

**Examples**

```
df = pl$LazyFrame(
  a = c(1.5, 2, NaN, 4),
  b = c(1.5, NaN, NaN, 4)
)
df$fill_nan(99)$collect()
```

---

LazyFrame\_fill\_null     *Fill nulls*

---

**Description**

Fill null values (which correspond to NA in R) using the specified value or strategy.

**Usage**

```
LazyFrame_fill_null(fill_value)
```

**Arguments**

fill\_value     Value to fill nulls with.

**Value**

LazyFrame

**Examples**

```
df = pl$LazyFrame(  
  a = c(1.5, 2, NA, 4),  
  b = c(1.5, NA, NA, 4)  
)  
df$fill_null(99)$collect()
```

---

LazyFrame\_filter     *Apply filter to LazyFrame*

---

**Description**

Filter rows with an Expression defining a boolean column. Multiple expressions are combined with & (AND). This is equivalent to [dplyr::filter\(\)](#).

**Usage**

```
LazyFrame_filter(...)
```

**Arguments**

...     Polars expressions which will evaluate to a boolean.

**Details**

Rows where the condition returns NA are dropped.

**Value**

A new LazyFrame object with add/modified column.

**Examples**

```
lf = pl$LazyFrame(iris)

lf$filter(pl$col("Species") == "setosa")$collect()

# This is equivalent to
# lf$filter(pl$col("Sepal.Length") > 5 & pl$col("Petal.Width") < 1)
lf$filter(pl$col("Sepal.Length") > 5, pl$col("Petal.Width") < 1)
```

---

LazyFrame_first	<i>Get the first row of a LazyFrame</i>
-----------------	---

---

**Description**

Get the first row of a LazyFrame

**Usage**

```
LazyFrame_first()
```

**Value**

A LazyFrame with one row

**Examples**

```
pl$LazyFrame(mtcars)$first()$collect()
```

---

LazyFrame_gather_every	<i>Take every nth row in the LazyFrame</i>
------------------------	--

---

**Description**

Take every nth row in the LazyFrame

**Usage**

```
LazyFrame_gather_every(n, offset = 0)
```

**Arguments**

n	Gather every n-th row.
offset	Starting index.

**Value**

A LazyFrame

**Examples**

```
lf = pl$LazyFrame(a = 1:4, b = 5:8)
lf$gather_every(2)$collect()

lf$gather_every(2, offset = 1)$collect()
```

---

LazyFrame\_group\_by     *Group a LazyFrame*

---

**Description**

This doesn't modify the data but only stores information about the group structure. This structure can then be used by several functions (`$agg()`, `$filter()`, etc.).

**Usage**

```
LazyFrame_group_by(..., maintain_order = polars_options()$maintain_order)
```

**Arguments**

...	Column(s) to group by. Accepts <a href="#">expression</a> input. Characters are parsed as column names.
maintain_order	Ensure that the order of the groups is consistent with the input data. This is slower than a default group by. Setting this to TRUE blocks the possibility to run on the streaming engine. The default value can be changed with <code>options(polars.maintain_order = TRUE)</code> .

**Value**

[LazyGroupBy](#) (a LazyFrame with special groupby methods like `$agg()`)

**Examples**

```

lf = pl$LazyFrame(
  a = c("a", "b", "a", "b", "c"),
  b = c(1, 2, 1, 3, 3),
  c = c(5, 4, 3, 2, 1)
)

lf$group_by("a")$agg(pl$col("b")$sum())$collect()

# Set `maintain_order = TRUE` to ensure the order of the groups is consistent with the input.
lf$group_by("a", maintain_order = TRUE)$agg(pl$col("c"))$collect()

# Group by multiple columns by passing a list of column names.
lf$group_by(c("a", "b"))$agg(pl$max("c"))$collect()

# Or pass some arguments to group by multiple columns in the same way.
# Expressions are also accepted.
lf$group_by("a", pl$col("b") %% 2)$agg(
  pl$col("c")$mean()
)$collect()

# The columns will be renamed to the argument names.
lf$group_by(d = "a", e = pl$col("b") %% 2)$agg(
  pl$col("c")$mean()
)$collect()

```

---

LazyFrame\_group\_by\_dynamic

*Group based on a date/time or integer column*

---

**Description**

If you have a time series  $\langle t_0, t_1, \dots, t_n \rangle$ , then by default the windows created will be:

- $(t_0 - \text{period}, t_0]$
- $(t_1 - \text{period}, t_1]$
- ...
- $(t_n - \text{period}, t_n]$

whereas if you pass a non-default offset, then the windows will be:

- $(t_0 + \text{offset}, t_0 + \text{offset} + \text{period}]$
- $(t_1 + \text{offset}, t_1 + \text{offset} + \text{period}]$
- ...
- $(t_n + \text{offset}, t_n + \text{offset} + \text{period}]$

**Usage**

```

LazyFrame_group_by_dynamic(
    index_column,
    ...,
    every,
    period = NULL,
    offset = NULL,
    include_boundaries = FALSE,
    closed = "left",
    label = "left",
    group_by = NULL,
    start_by = "window"
)

```

**Arguments**

<code>index_column</code>	Column used to group based on the time window. Often of type Date/Datetime. This column must be sorted in ascending order (or, if <code>by</code> is specified, then it must be sorted in ascending order within each group). In case of a rolling group by on indices, <code>dtype</code> needs to be either <code>Int32</code> or <code>Int64</code> . Note that <code>Int32</code> gets temporarily cast to <code>Int64</code> , so if performance matters use an <code>Int64</code> column.
<code>...</code>	Ignored.
<code>every</code>	Interval of the window.
<code>period</code>	A character representing the length of the window, must be non-negative. See the Polars <code>duration string language</code> section for details.
<code>offset</code>	A character representing the offset of the window, or <code>NULL</code> (default). If <code>NULL</code> , <code>-period</code> is used. See the Polars <code>duration string language</code> section for details.
<code>include_boundaries</code>	Add two columns <code>"_lower_boundary"</code> and <code>"_upper_boundary"</code> columns that show the boundaries of the window. This will impact performance because it's harder to parallelize.
<code>closed</code>	Define which sides of the temporal interval are closed (inclusive). This can be either <code>"left"</code> , <code>"right"</code> , <code>"both"</code> or <code>"none"</code> .
<code>label</code>	Define which label to use for the window: <ul style="list-style-type: none"> <li><code>"left"</code>: lower boundary of the window</li> <li><code>"right"</code>: upper boundary of the window</li> <li><code>"datapoint"</code>: the first value of the index column in the given window. If you don't need the label to be at one of the boundaries, choose this option for maximum performance.</li> </ul>
<code>group_by</code>	Also group by this column/these columns.
<code>start_by</code>	The strategy to determine the start of the first window by: <ul style="list-style-type: none"> <li><code>"window"</code>: start by taking the earliest timestamp, truncating it with <code>every</code>, and then adding <code>offset</code>. Note that weekly windows start on Monday.</li> <li><code>"datapoint"</code>: start from the first encountered data point.</li> </ul>

- a day of the week (only takes effect if every contains "w"): "monday" starts the window on the Monday before the first data point, etc.

## Details

In case of a rolling operation on an integer column, the windows are defined by:

- "1i" # length 1
- "10i" # length 10

## Value

A [LazyGroupBy](#) object

## See Also

- [<LazyFrame>\\$rolling\(\)](#)

## Examples

```
lf = pl$LazyFrame(
  time = pl$datetime_range(
    start = strptime("2021-12-16 00:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC"),
    end = strptime("2021-12-16 03:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC"),
    interval = "30m"
  ),
  n = 0:6
)
lf$collect()

# get the sum in the following hour relative to the "time" column
lf$group_by_dynamic("time", every = "1h")$agg(
  vals = pl$col("n"),
  sum = pl$col("n")$sum()
)$collect()

# using "include_boundaries = TRUE" is helpful to see the period considered
lf$group_by_dynamic("time", every = "1h", include_boundaries = TRUE)$agg(
  vals = pl$col("n")
)$collect()

# in the example above, the values didn't include the one *exactly* 1h after
# the start because "closed = 'left'" by default.
# Changing it to "right" includes values that are exactly 1h after. Note that
# the value at 00:00:00 now becomes included in the interval [23:00:00 - 00:00:00],
# even if this interval wasn't there originally
lf$group_by_dynamic("time", every = "1h", closed = "right")$agg(
  vals = pl$col("n")
)$collect()
# To keep both boundaries, we use "closed = 'both'". Some values now belong to
# several groups:
lf$group_by_dynamic("time", every = "1h", closed = "both")$agg(
```

```

    vals = pl$col("n")
  )$collect()

# Dynamic group bys can also be combined with grouping on normal keys
lf = lf$with_columns(
  groups = as_polars_series(c("a", "a", "a", "b", "b", "a", "a"))
)
lf$collect()

lf$group_by_dynamic(
  "time",
  every = "1h",
  closed = "both",
  group_by = "groups",
  include_boundaries = TRUE
)$agg(pl$col("n"))$collect()

# We can also create a dynamic group by based on an index column
lf = pl$LazyFrame(
  idx = 0:5,
  A = c("A", "A", "B", "B", "B", "C")
)$with_columns(pl$col("idx")$set_sorted())
lf$collect()

lf$group_by_dynamic(
  "idx",
  every = "2i",
  period = "3i",
  include_boundaries = TRUE,
  closed = "right"
)$agg(A_agg_list = pl$col("A"))$collect()

```

---

LazyFrame\_head

*Get the first n rows.*

---

### Description

A shortcut for `$slice(0, n)`. Consider using the `$fetch()` method if you want to test your query. The `$fetch()` operation will load the first n rows at the scan level, whereas `$head()` is applied at the end.

### Usage

```
LazyFrame_head(n = 5L)
```

### Arguments

n                      Number of rows to return.



**Details**

`$limit()` is an alias for `$head()`.

**Value**

A new `LazyFrame` object with applied filter.

**Examples**

```
lf = pl$LazyFrame(a = 1:6, b = 7:12)

lf$head()$collect()

lf$head(2)$collect()
```

---

LazyFrame_join	<i>Join LazyFrames</i>
----------------	------------------------

---

**Description**

This function can do both mutating joins (adding columns based on matching observations, for example with `how = "left"`) and filtering joins (keeping observations based on matching observations, for example with `how = "inner"`).

**Usage**

```
LazyFrame_join(
  other,
  on = NULL,
  how = "inner",
  ...,
  left_on = NULL,
  right_on = NULL,
  suffix = "_right",
  validate = "m:m",
  join_nulls = FALSE,
  allow_parallel = TRUE,
  force_parallel = FALSE,
  coalesce = NULL
)
```

**Arguments**

<code>other</code>	LazyFrame to join with.
<code>on</code>	Either a vector of column names or a list of expressions and/or strings. Use <code>left_on</code> and <code>right_on</code> if the column names to match on are different between the two DataFrames.

how	One of the following methods: "inner", "left", "right", "full", "semi", "anti", "cross".
...	Ignored.
left_on, right_on	Same as on but only for the left or the right DataFrame. They must have the same length.
suffix	Suffix to add to duplicated column names.
validate	Checks if join is of specified type: <ul style="list-style-type: none"> <li>• "m:m" (default): many-to-many, doesn't perform any checks;</li> <li>• "1:1": one-to-one, check if join keys are unique in both left and right datasets;</li> <li>• "1:m": one-to-many, check if join keys are unique in left dataset</li> <li>• "m:1": many-to-one, check if join keys are unique in right dataset</li> </ul> <p>Note that this is currently not supported by the streaming engine, and is only supported when joining by single columns.</p>
join_nulls	Join on null values. By default null values will never produce matches.
allow_parallel	Allow the physical plan to optionally evaluate the computation of both DataFrames up to the join in parallel.
force_parallel	Force the physical plan to evaluate the computation of both DataFrames up to the join in parallel.
coalesce	Coalescing behavior (merging of join columns). <ul style="list-style-type: none"> <li>• NULL: join specific.</li> <li>• TRUE: Always coalesce join columns.</li> <li>• FALSE: Never coalesce join columns.</li> </ul>

## Value

LazyFrame

## Examples

```
# inner join by default
df1 = pl$LazyFrame(list(key = 1:3, payload = c("f", "i", NA)))
df2 = pl$LazyFrame(list(key = c(3L, 4L, 5L, NA_integer_)))
df1$join(other = df2, on = "key")

# cross join
df1 = pl$LazyFrame(x = letters[1:3])
df2 = pl$LazyFrame(y = 1:4)
df1$join(other = df2, how = "cross")

# use "validate" to ensure join keys are not duplicated
df1 = pl$LazyFrame(x = letters[1:5], y = 1:5)
df2 = pl$LazyFrame(x = c("a", letters[1:4]), y2 = 6:10)

# this throws an error because there are two keys in df2 that match the key
```

```
# in df1
tryCatch(
  df1$join(df2, on = "x", validate = "1:1")$collect(),
  error = function(e) print(e)
)
```

---

LazyFrame\_join\_asof     *Perform joins on nearest keys*

---

## Description

This is similar to a left-join except that we match on nearest key rather than equal keys.

## Usage

```
LazyFrame_join_asof(
  other,
  ...,
  left_on = NULL,
  right_on = NULL,
  on = NULL,
  by_left = NULL,
  by_right = NULL,
  by = NULL,
  strategy = c("backward", "forward", "nearest"),
  suffix = "_right",
  tolerance = NULL,
  allow_parallel = TRUE,
  force_parallel = FALSE,
  coalesce = TRUE
)
```

## Arguments

<code>other</code>	LazyFrame
<code>...</code>	Not used, blocks use of further positional arguments
<code>left_on, right_on</code>	Same as <code>on</code> but only for the left or the right DataFrame. They must have the same length.
<code>on</code>	Either a vector of column names or a list of expressions and/or strings. Use <code>left_on</code> and <code>right_on</code> if the column names to match on are different between the two DataFrames.
<code>by_left, by_right</code>	Same as <code>by</code> but only for the left or the right table. They must have the same length.

<code>by</code>	Join on these columns before performing asof join. Either a vector of column names or a list of expressions and/or strings. Use <code>left_by</code> and <code>right_by</code> if the column names to match on are different between the two tables.
<code>strategy</code>	Strategy for where to find match: <ul style="list-style-type: none"> <li>• "backward" (default): search for the last row in the right table whose on key is less than or equal to the left key.</li> <li>• "forward": search for the first row in the right table whose on key is greater than or equal to the left key.</li> <li>• "nearest": search for the last row in the right table whose value is nearest to the left key. String keys are not currently supported for a nearest search.</li> </ul>
<code>suffix</code>	Suffix to add to duplicated column names.
<code>tolerance</code>	Numeric tolerance. By setting this the join will only be done if the near keys are within this distance. If an asof join is done on columns of dtype "Date", "Datetime", "Duration" or "Time", use the Polars duration string language. About the language, see the Polars <code>duration string language</code> section for details. There may be a circumstance where R types are not sufficient to express a numeric tolerance. In that case, you can use the expression syntax like <code>tolerance = pl\$lit(42)\$cast(pl\$UInt64)</code>
<code>allow_parallel</code>	Allow the physical plan to optionally evaluate the computation of both DataFrames up to the join in parallel.
<code>force_parallel</code>	Force the physical plan to evaluate the computation of both DataFrames up to the join in parallel.
<code>coalesce</code>	Coalescing behavior (merging of <code>on</code> / <code>left_on</code> / <code>right_on</code> columns): <ul style="list-style-type: none"> <li>• TRUE: Always coalesce join columns;</li> <li>• FALSE: Never coalesce join columns. Note that joining on any other expressions than <code>col</code> will turn off coalescing.</li> </ul>

### Details

Both tables (DataFrames or LazyFrames) must be sorted by the `asof_join` key.

### Polars duration string language

Polars duration string language is a simple representation of durations. It is used in many Polars functions that accept durations.

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)

- 1w (1 calendar week)
- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds

By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".

### Examples

```
#
# create two LazyFrame to join asof
gdp = pl$LazyFrame(
  date = as.Date(c("2015-1-1", "2016-1-1", "2017-5-1", "2018-1-1", "2019-1-1")),
  gdp = c(4321, 4164, 4411, 4566, 4696),
  group = c("b", "a", "a", "b", "b")
)

pop = pl$LazyFrame(
  date = as.Date(c("2016-5-12", "2017-5-12", "2018-5-12", "2019-5-12")),
  population = c(82.19, 82.66, 83.12, 83.52),
  group = c("b", "b", "a", "a")
)

# optional make sure tables are already sorted with "on" join-key
gdp = gdp$sort("date")
pop = pop$sort("date")

# Left-join_asof LazyFrame pop with gdp on "date"
# Look backward in gdp to find closest matching date
pop$join_asof(gdp, on = "date", strategy = "backward")$collect()

# .... and forward
pop$join_asof(gdp, on = "date", strategy = "forward")$collect()

# join by a group: "only look within groups"
pop$join_asof(gdp, on = "date", by = "group", strategy = "backward")$collect()

# only look 2 weeks and 2 days back
pop$join_asof(gdp, on = "date", strategy = "backward", tolerance = "2w2d")$collect()

# only look 11 days back (numeric tolerance depends on polars type, <date> is in days)
pop$join_asof(gdp, on = "date", strategy = "backward", tolerance = 11)$collect()
```

---

LazyFrame\_join\_where *Perform a join based on one or multiple (in)equality predicates*

---

### Description

This performs an inner join, so only rows where all predicates are true are included in the result, and a row from either LazyFrame may be included multiple times in the result.

Note that the row order of the input LazyFrames is not preserved.

### Usage

```
LazyFrame_join_where(other, ..., suffix = "_right")
```

### Arguments

other	LazyFrame to join with.
...	(In)Equality condition to join the two tables on. When a column name occurs in both tables, the proper suffix must be applied in the predicate. For example, if both tables have a column "x" that you want to use in the conditions, you must refer to the column of the right table as "x<suffix>".
suffix	Suffix to append to columns with a duplicate name.

### Value

A LazyFrame

### Examples

```
east = pl$LazyFrame(
  id = c(100, 101, 102),
  dur = c(120, 140, 160),
  rev = c(12, 14, 16),
  cores = c(2, 8, 4)
)

west = pl$LazyFrame(
  t_id = c(404, 498, 676, 742),
  time = c(90, 130, 150, 170),
  cost = c(9, 13, 15, 16),
  cores = c(4, 2, 1, 4)
)

east$join_where(
  west,
  pl$col("dur") < pl$col("time"),
  pl$col("rev") < pl$col("cost")
)$collect()
```

---

LazyFrame_last	<i>Get the last row of a LazyFrame</i>
----------------	--

---

**Description**

Aggregate the columns in the LazyFrame to their maximum value.

**Usage**

```
LazyFrame_last()
```

**Value**

A LazyFrame with one row

**Examples**

```
p1$LazyFrame(mtcars)$last()$collect()
```

---

LazyFrame_max	<i>Max</i>
---------------	------------

---

**Description**

Aggregate the columns in the LazyFrame to their maximum value.

**Usage**

```
LazyFrame_max()
```

**Value**

A LazyFrame with one row

**Examples**

```
p1$LazyFrame(mtcars)$max()$collect()
```

---

LazyFrame_mean	<i>Mean</i>
----------------	-------------

---

**Description**

Aggregate the columns in the LazyFrame to their mean value.

**Usage**

```
LazyFrame_mean()
```

**Value**

A LazyFrame with one row

**Examples**

```
p1$LazyFrame(mtcars)$mean()$collect()
```

---

LazyFrame_median	<i>Median</i>
------------------	---------------

---

**Description**

Aggregate the columns in the LazyFrame to their median value.

**Usage**

```
LazyFrame_median()
```

**Value**

A LazyFrame with one row

**Examples**

```
p1$LazyFrame(mtcars)$median()$collect()
```



---

LazyFrame_min	<i>Min</i>
---------------	------------

---

**Description**

Aggregate the columns in the LazyFrame to their minimum value.

**Usage**

```
LazyFrame_min()
```

**Value**

A LazyFrame with one row

**Examples**

```
pl$LazyFrame(mtcars)$min().collect()
```

---

LazyFrame_print	<i>print LazyFrame internal method</i>
-----------------	--

---

**Description**

can be used in the middle of a method chain

**Usage**

```
LazyFrame_print(x)
```

**Arguments**

x	LazyFrame
---	-----------

**Value**

self

**Examples**

```
pl$LazyFrame(iris)$print()
```

---

LazyFrame\_profile      *Collect and profile a lazy query.*

---

### Description

This will run the query and return a list containing the materialized DataFrame and a DataFrame that contains profiling information of each node that is executed.

### Usage

```
LazyFrame_profile(
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  comm_subplan_elim = TRUE,
  comm_subexpr_elim = TRUE,
  cluster_with_columns = TRUE,
  streaming = FALSE,
  no_optimization = FALSE,
  collect_in_background = FALSE,
  show_plot = FALSE,
  truncate_nodes = 0
)
```

### Arguments

`type_coercion` Logical. Coerce types such that operations succeed and run on minimal required memory.

`predicate_pushdown` Logical. Applies filters as early as possible at scan level.

`projection_pushdown` Logical. Select only the columns that are needed at the scan level.

`simplify_expression` Logical. Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.

`slice_pushdown` Logical. Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. `join$head(10)`).

`comm_subplan_elim` Logical. Will try to cache branching subplans that occur on self-joins or unions.

`comm_subexpr_elim` Logical. Common subexpressions will be cached and reused.

`cluster_with_columns` Combine sequential independent calls to `with_columns()`.

streaming	Logical. Run parts of the query in a streaming fashion (this is in an alpha state).
no_optimization	Logical. Sets the following parameters to FALSE: predicate_pushdown, projection_pushdown, slice_pushdown, comm_subplan_elim, comm_subexpr_elim, cluster_with_columns.
collect_in_background	Logical. Detach this query from R session. Computation will start in background. Get a handle which later can be converted into the resulting DataFrame. Useful in interactive mode to not lock R session.
show_plot	Show a Gantt chart of the profiling result
truncate_nodes	Truncate the label lengths in the Gantt chart to this number of characters. If 0 (default), do not truncate.

### Details

The units of the timings are microseconds.

### Value

List of two DataFrames: one with the collected result, the other with the timings of each step. If show\_graph = TRUE, then the plot is also stored in the list.

### See Also

- `$collect()` - regular collect.
- `$fetch()` - fast limited query check
- `$collect_in_background()` - non-blocking collect returns a future handle. Can also just be used via `$collect(collect_in_background = TRUE)`.
- `$sink_parquet()` streams query to a parquet file.
- `$sink_ipc()` streams query to a arrow file.

### Examples

```
## Simplest use case
pl$LazyFrame()$select(pl$lit(2) + 2)$profile()

## Use $profile() to compare two queries

# -1- map each Species-group with native polars, takes ~120us only
pl$LazyFrame(iris)$
  sort("Sepal.Length")$
  group_by("Species", maintain_order = TRUE)$
  agg(pl$col(pl$Float64)$first() + 5)$
  profile()

# -2- map each Species-group of each numeric column with an R function, takes ~7000us (slow!)

# some R function, prints `.` for each time called by polars
r_func = \(s) {
  cat(".")
}
```

```

    s$to_r()[1] + 5
  }

pl$LazyFrame(iris)$
  sort("Sepal.Length")$
  group_by("Species", maintain_order = TRUE)$
  agg(pl$col(pl$Float64)$map_elements(r_func))$
  profile()

```

---

LazyFrame\_quantile      *Quantile*

---

### Description

Aggregate the columns in the DataFrame to a unique quantile value. Use `$describe()` to specify several quantiles.

### Usage

```
LazyFrame_quantile(quantile, interpolation = "nearest")
```

### Arguments

`quantile`      Numeric of length 1 between 0 and 1.  
`interpolation`      One of "nearest", "higher", "lower", "midpoint", or "linear".

### Value

LazyFrame

### Examples

```
pl$LazyFrame(mtcars)$quantile(.4)$collect()
```

---

LazyFrame\_rename      *Rename column names of a LazyFrame*

---

### Description

Rename column names of a LazyFrame

### Usage

```
LazyFrame_rename(...)
```

**Arguments**

- ...
- One of the following:
- Key value pairs that map from old name to new name, like `old_name = "new_name"`.
  - As above but with params wrapped in a list
  - An R function that takes the old names character vector as input and returns the new names character vector.

**Details**

If existing names are swapped (e.g. A points to B and B points to A), polars will block projection and predicate pushdowns at this node.

**Value**

[LazyFrame](#)

**Examples**

```
lf = pl$LazyFrame(  
  foo = 1:3,  
  bar = 6:8,  
  ham = letters[1:3]  
)  
  
lf$rename(foo = "apple")$collect()  
  
lf$rename(  
  \(\column_name) paste0("c", substr(column_name, 2, 100))  
)$collect()
```

---

LazyFrame\_reverse      *Reverse*

---

**Description**

Reverse the LazyFrame (the last row becomes the first one, etc.).

**Usage**

```
LazyFrame_reverse()
```

**Value**

LazyFrame

**Examples**

```
pl$LazyFrame(mtcars)$reverse()$collect()
```

---

LazyFrame\_rolling      *Create rolling groups based on a date/time or integer column*

---

### Description

If you have a time series  $\langle t_0, t_1, \dots, t_n \rangle$ , then by default the windows created will be:

- $(t_0 - \text{period}, t_0]$
- $(t_1 - \text{period}, t_1]$
- ...
- $(t_n - \text{period}, t_n]$

whereas if you pass a non-default offset, then the windows will be:

- $(t_0 + \text{offset}, t_0 + \text{offset} + \text{period}]$
- $(t_1 + \text{offset}, t_1 + \text{offset} + \text{period}]$
- ...
- $(t_n + \text{offset}, t_n + \text{offset} + \text{period}]$

### Usage

```
LazyFrame_rolling(
    index_column,
    ...,
    period,
    offset = NULL,
    closed = "right",
    group_by = NULL
)
```

### Arguments

<code>index_column</code>	Column used to group based on the time window. Often of type Date/Datetime. This column must be sorted in ascending order (or, if by is specified, then it must be sorted in ascending order within each group). In case of a rolling group by on indices, dtype needs to be either Int32 or Int64. Note that Int32 gets temporarily cast to Int64, so if performance matters use an Int64 column.
<code>...</code>	Ignored.
<code>period</code>	A character representing the length of the window, must be non-negative. See the Polars duration string language section for details.
<code>offset</code>	A character representing the offset of the window, or NULL (default). If NULL, <code>-period</code> is used. See the Polars duration string language section for details.
<code>closed</code>	Define which sides of the temporal interval are closed (inclusive). This can be either "left", "right", "both" or "none".
<code>group_by</code>	Also group by this column/these columns.

## Details

In case of a rolling operation on an integer column, the windows are defined by:

- "1i" # length 1
- "10i" # length 10

## Value

A [LazyGroupBy](#) object

## Polars duration string language

Polars duration string language is a simple representation of durations. It is used in many Polars functions that accept durations.

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)
- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds

By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".

## See Also

- [<LazyFrame>\\$group\\_by\\_dynamic\(\)](#)

## Examples

```
dates = c(
  "2020-01-01 13:45:48",
  "2020-01-01 16:42:13",
  "2020-01-01 16:45:09",
  "2020-01-02 18:12:48",
  "2020-01-03 19:45:32",
  "2020-01-08 23:16:43"
```

```

)

df = pl$LazyFrame(dt = dates, a = c(3, 7, 5, 9, 2, 1))$with_columns(
  pl$col("dt")$str$strptime(pl$Datetime())$set_sorted()
)

df$rolling(index_column = "dt", period = "2d")$agg(
  sum_a = pl$sum("a"),
  min_a = pl$min("a"),
  max_a = pl$max("a")
)$collect()

```

---

LazyFrame_select	<i>Select and modify columns of a LazyFrame</i>
------------------	---

---

### Description

Similar to `dplyr::mutate()`. However, it discards unmentioned columns (like `.` in `data.table`).

### Usage

```
LazyFrame_select(...)
```

### Arguments

`...` Columns to keep. Those can be expressions (e.g `pl$col("a")`), column names (e.g `"a"`), or list containing expressions or column names (e.g `list(pl$col("a"))`).

### Value

A LazyFrame

### Examples

```

pl$LazyFrame(iris)$select(
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")
)

```



---

LazyFrame\_select\_seq *Select and modify columns of a LazyFrame*

---

### Description

Similar to `dplyr::mutate()`. However, it discards unmentioned columns (like `.` in `data.table`). This will run all expression sequentially instead of in parallel. Use this when the work per expression is cheap. Otherwise, `$select()` should be preferred.

### Usage

```
LazyFrame_select_seq(...)
```

### Arguments

... Columns to keep. Those can be expressions (e.g `pl$col("a")`), column names (e.g `"a"`), or list containing expressions or column names (e.g `list(pl$col("a"))`).

### Value

A LazyFrame

### Examples

```
pl$LazyFrame(iris)$select_seq(
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")
)
```

---

LazyFrame\_serialize *Serialize the logical plan of this LazyFrame to a file or string in JSON format*

---

### Description

Note that not all LazyFrames can be serialized. For example, LazyFrames that contain UDFs such as `$map_elements()` cannot be serialized.

### Usage

```
LazyFrame_serialize()
```

### Value

A character of the JSON representation of the logical plan

**See Also**

- [pl\\$deserialize\\_lf\(\)](#)

**Examples**

```
lf = pl$LazyFrame(a = 1:3)$sum()
json = lf$serialize()
json

# The logical plan can later be deserialized back into a LazyFrame.
pl$deserialize_lf(json)$collect()
```

---

LazyFrame_shift	<i>Shift a LazyFrame</i>
-----------------	--------------------------

---

**Description**

Shift the values by a given period. If the period (n) is positive, then n rows will be inserted at the top of the DataFrame and the last n rows will be discarded. Vice-versa if the period is negative. In the end, the total number of rows of the DataFrame doesn't change.

**Usage**

```
LazyFrame_shift(n = 1, fill_value = NULL)
```

**Arguments**

n	Number of indices to shift forward. If a negative value is passed, values are shifted in the opposite direction instead.
fill_value	Fill the resulting null values with this value. Accepts expression input. Non-expression inputs are parsed as literals.

**Value**

LazyFrame

**Examples**

```
lf = pl$LazyFrame(a = 1:4, b = 5:8)

lf$shift(2)$collect()

lf$shift(-2)$collect()

lf$shift(-2, fill_value = 100)$collect()
```

---

LazyFrame\_sink\_csv      *Stream the output of a query to a CSV file*

---

### Description

This writes the output of a query directly to a CSV file without collecting it in the R session first. This is useful if the output of the query is still larger than RAM as it would crash the R session if it was collected into R.

### Usage

```
LazyFrame_sink_csv(  
  path,  
  ...,  
  include_bom = FALSE,  
  include_header = TRUE,  
  separator = ",",  
  line_terminator = "\n",  
  quote_char = "\"",  
  batch_size = 1024,  
  datetime_format = NULL,  
  date_format = NULL,  
  time_format = NULL,  
  float_precision = NULL,  
  null_values = "",  
  quote_style = "necessary",  
  maintain_order = TRUE,  
  type_coercion = TRUE,  
  predicate_pushdown = TRUE,  
  projection_pushdown = TRUE,  
  simplify_expression = TRUE,  
  slice_pushdown = TRUE,  
  no_optimization = FALSE  
)
```

### Arguments

path	A character. File path to which the file should be written.
...	Ignored.
include_bom	Whether to include UTF-8 BOM (byte order mark) in the CSV output.
include_header	Whether to include header in the CSV output.
separator	Separate CSV fields with this symbol.
line_terminator	String used to end each row.
quote_char	Byte to use as quoting character.

<code>batch_size</code>	Number of rows that will be processed per thread.
<code>datetime_format</code>	A format string, with the specifiers defined by the chrono Rust crate. If no format specified, the default fractional-second precision is inferred from the maximum timeunit found in the frame's Datetime cols (if any).
<code>date_format</code>	A format string, with the specifiers defined by the chrono Rust crate.
<code>time_format</code>	A format string, with the specifiers defined by the chrono Rust crate.
<code>float_precision</code>	Number of decimal places to write, applied to both Float32 and Float64 datatypes.
<code>null_values</code>	A string representing null values (defaulting to the empty string).
<code>quote_style</code>	Determines the quoting strategy used. <ul style="list-style-type: none"> <li>• "necessary" (default): This puts quotes around fields only when necessary. They are necessary when fields contain a quote, delimiter or record terminator. Quotes are also necessary when writing an empty record (which is indistinguishable from a record with one empty field). This is the default.</li> <li>• "always": This puts quotes around every field.</li> <li>• "non_numeric": This puts quotes around all fields that are non-numeric. Namely, when writing a field that does not parse as a valid float or integer, then quotes will be used even if they aren't strictly necessary.</li> <li>• "never": This never puts quotes around fields, even if that results in invalid CSV data (e.g. by not quoting strings containing the separator).</li> </ul>
<code>maintain_order</code>	Maintain the order in which data is processed. Setting this to FALSE will be slightly faster.
<code>type_coercion</code>	Logical. Coerce types such that operations succeed and run on minimal required memory.
<code>predicate_pushdown</code>	Logical. Applies filters as early as possible at scan level.
<code>projection_pushdown</code>	Logical. Select only the columns that are needed at the scan level.
<code>simplify_expression</code>	Logical. Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.
<code>slice_pushdown</code>	Logical. Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. <code>join\$head(10)</code> ).
<code>no_optimization</code>	Logical. Sets the following parameters to FALSE: <code>predicate_pushdown</code> , <code>projection_pushdown</code> , <code>slice_pushdown</code> , <code>comm_subplan_elim</code> , <code>comm_subexpr_elim</code> , <code>cluster_with_columns</code> .

**Value**

Invisibly returns the input LazyFrame

**Examples**

```
# sink table 'mtcars' from mem to CSV
tmpf = tempfile()
pl$LazyFrame(mtcars)$sink_csv(tmpf)

# stream a query end-to-end
tmpf2 = tempfile()
pl$scan_csv(tmpf)$select(pl$col("cyl") * 2)$sink_csv(tmpf2)

# load parquet directly into a DataFrame / memory
pl$scan_csv(tmpf2)$collect()
```

---

LazyFrame\_sink\_ipc      *Stream the output of a query to an Arrow IPC file*

---

**Description**

This writes the output of a query directly to an Arrow IPC file without collecting it in the R session first. This is useful if the output of the query is still larger than RAM as it would crash the R session if it was collected into R.

**Usage**

```
LazyFrame_sink_ipc(
  path,
  ...,
  compression = c("zstd", "lz4", "uncompressed"),
  maintain_order = TRUE,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  no_optimization = FALSE
)
```

**Arguments**

path	A character. File path to which the file should be written.
...	Ignored.
compression	NULL or a character of the compression method, "uncompressed" or "lz4" or "zstd". NULL is equivalent to "uncompressed". Choose "zstd" for good compression performance. Choose "lz4" for fast compression/decompression.
maintain_order	Maintain the order in which data is processed. Setting this to FALSE will be slightly faster.
type_coercion	Logical. Coerce types such that operations succeed and run on minimal required memory.

predicate\_pushdown  
 Logical. Applies filters as early as possible at scan level.

projection\_pushdown  
 Logical. Select only the columns that are needed at the scan level.

simplify\_expression  
 Logical. Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.

slice\_pushdown  
 Logical. Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. `join$head(10)`).

no\_optimization  
 Logical. Sets the following parameters to FALSE: `predicate_pushdown`, `projection_pushdown`, `slice_pushdown`, `comm_subplan_elim`, `comm_subexpr_elim`, `cluster_with_columns`.

**Value**

Invisibly returns the input LazyFrame

**Examples**

```
# sink table 'mtcars' from mem to ipc
tmpf = tempfile()
pl$LazyFrame(mtcars)$sink_ipc(tmpf)

# stream a query end-to-end (not supported yet, https://github.com/pola-rs/polars/issues/1040)
# tmpf2 = tempfile()
# pl$scan_ipc(tmpf)$select(pl$col("cyl") * 2)$sink_ipc(tmpf2)

# load ipc directly into a DataFrame / memory
# pl$scan_ipc(tmpf2)$collect()
```

---

LazyFrame\_sink\_ndjson *Stream the output of a query to a JSON file*

---

**Description**

This writes the output of a query directly to a JSON file without collecting it in the R session first. This is useful if the output of the query is still larger than RAM as it would crash the R session if it was collected into R.

**Usage**

```
LazyFrame_sink_ndjson(
  path,
  ...,
  maintain_order = TRUE,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
```

```

    projection_pushdown = TRUE,
    simplify_expression = TRUE,
    slice_pushdown = TRUE,
    no_optimization = FALSE
  )

```

## Arguments

<code>path</code>	A character. File path to which the file should be written.
<code>...</code>	Ignored.
<code>maintain_order</code>	Maintain the order in which data is processed. Setting this to FALSE will be slightly faster.
<code>type_coercion</code>	Logical. Coerce types such that operations succeed and run on minimal required memory.
<code>predicate_pushdown</code>	Logical. Applies filters as early as possible at scan level.
<code>projection_pushdown</code>	Logical. Select only the columns that are needed at the scan level.
<code>simplify_expression</code>	Logical. Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.
<code>slice_pushdown</code>	Logical. Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. <code>join\$head(10)</code> ).
<code>no_optimization</code>	Logical. Sets the following parameters to FALSE: <code>predicate_pushdown</code> , <code>projection_pushdown</code> , <code>slice_pushdown</code> , <code>comm_subplan_elim</code> , <code>comm_subexpr_elim</code> , <code>cluster_with_columns</code> .

## Value

Invisibly returns the input LazyFrame

## Examples

```

# sink table 'mtcars' from mem to JSON
tmpf = tempfile(fileext = ".json")
pl$LazyFrame(mtcars)$sink_ndjson(tmpf)

# load parquet directly into a DataFrame / memory
pl$scan_ndjson(tmpf)$collect()

```

---

 LazyFrame\_sink\_parquet

*Stream the output of a query to a Parquet file*


---

### Description

This writes the output of a query directly to a Parquet file without collecting it in the R session first. This is useful if the output of the query is still larger than RAM as it would crash the R session if it was collected into R.

### Usage

```
LazyFrame_sink_parquet(
  path,
  ...,
  compression = "zstd",
  compression_level = 3,
  statistics = TRUE,
  row_group_size = NULL,
  data_page_size = NULL,
  maintain_order = TRUE,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  no_optimization = FALSE
)
```

### Arguments

path	A character. File path to which the file should be written.
...	Ignored.
compression	String. The compression method. One of: <ul style="list-style-type: none"> <li>• "lz4": fast compression/decompression.</li> <li>• "uncompressed"</li> <li>• "snappy": this guarantees that the parquet file will be compatible with older parquet readers.</li> <li>• "gzip"</li> <li>• "lzo"</li> <li>• "brotli"</li> <li>• "zstd": good compression performance.</li> </ul>
compression_level	NULL or Integer. The level of compression to use. Only used if method is one of 'gzip', 'brotli', or 'zstd'. Higher compression means smaller files on disk:



	<ul style="list-style-type: none"> <li>• "gzip": min-level: 0, max-level: 10.</li> <li>• "brotli": min-level: 0, max-level: 11.</li> <li>• "zstd": min-level: 1, max-level: 22.</li> </ul>
statistics	<p>Whether statistics should be written to the Parquet headers. Possible values:</p> <ul style="list-style-type: none"> <li>• TRUE: enable default set of statistics (default)</li> <li>• FALSE: disable all statistics</li> <li>• "full": calculate and write all available statistics.</li> <li>• A named list where all values must be TRUE or FALSE, e.g. <code>list(min = TRUE, max = FALSE)</code>. Statistics available are "min", "max", "distinct_count", "null_count".</li> </ul>
row_group_size	NULL or Integer. Size of the row groups in number of rows. If NULL (default), the chunks of the DataFrame are used. Writing in smaller chunks may reduce memory pressure and improve writing speeds.
data_page_size	Size of the data page in bytes. If NULL (default), it is set to $1024^2$ bytes. will be ~1MB.
maintain_order	Maintain the order in which data is processed. Setting this to FALSE will be slightly faster.
type_coercion	Logical. Coerce types such that operations succeed and run on minimal required memory.
predicate_pushdown	Logical. Applies filters as early as possible at scan level.
projection_pushdown	Logical. Select only the columns that are needed at the scan level.
simplify_expression	Logical. Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.
slice_pushdown	Logical. Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. <code>join\$head(10)</code> ).
no_optimization	Logical. Sets the following parameters to FALSE: <code>predicate_pushdown</code> , <code>projection_pushdown</code> , <code>slice_pushdown</code> , <code>comm_subplan_elim</code> , <code>comm_subexpr_elim</code> , <code>cluster_with_columns</code> .

**Value**

Invisibly returns the input LazyFrame

**Examples**

```
# sink table 'mtcars' from mem to parquet
tmpf = tempfile()
pl$LazyFrame(mtcars)$sink_parquet(tmpf)

# stream a query end-to-end
tmpf2 = tempfile()
pl$scan_parquet(tmpf)$select(pl$col("cyl") * 2)$sink_parquet(tmpf2)
```

```
# load parquet directly into a DataFrame / memory
pl$scan_parquet(tmpf2)$collect()
```

---

LazyFrame\_slice      *Slice*

---

### Description

Get a slice of the LazyFrame.

### Usage

```
LazyFrame_slice(offset, length = NULL)
```

### Arguments

offset	Start index, can be a negative value. This is 0-indexed, so offset = 1 doesn't include the first row.
length	Length of the slice. If NULL (default), all rows starting at the offset will be selected.

### Value

A [LazyFrame](#)

### Examples

```
pl$LazyFrame(mtcars)$slice(2, 4)$collect()
pl$LazyFrame(mtcars)$slice(30)$collect()
mtcars[2:6, ]
```

---

LazyFrame\_sort      *Sort the LazyFrame by the given columns*

---

### Description

Sort the LazyFrame by the given columns

### Usage

```
LazyFrame_sort(
  by,
  ...,
  descending = FALSE,
  nulls_last = FALSE,
  maintain_order = FALSE,
  multithreaded = TRUE
)
```

**Arguments**

by	Column(s) to sort by. Can be character vector of column names, a list of Expr(s) or a list with a mix of Expr(s) and column names.
...	More columns to sort by as above but provided one Expr per argument.
descending	Logical. Sort in descending order (default is FALSE). This must be either of length 1 or a logical vector of the same length as the number of Expr(s) specified in by and ...
nulls_last	A logical or logical vector of the same length as the number of columns. If TRUE, place null values last instead of first.
maintain_order	Whether the order should be maintained if elements are equal. If TRUE, streaming is not possible and performance might be worse since this requires a stable search.
multithreaded	A logical. If TRUE, sort using multiple threads.

**Value**

LazyFrame

**Examples**

```
df = mtcars
df$mpg[1] = NA
df = pl$LazyFrame(df)
df$sort("mpg")$collect()
df$sort("mpg", nulls_last = TRUE)$collect()
df$sort("cyl", "mpg")$collect()
df$sort(c("cyl", "mpg"))$collect()
df$sort(c("cyl", "mpg"), descending = TRUE)$collect()
df$sort(c("cyl", "mpg"), descending = c(TRUE, FALSE))$collect()
df$sort(pl$col("cyl"), pl$col("mpg"))$collect()
```

---

LazyFrame\_sql

*Execute a SQL query against the LazyFrame*

---

**Description**

The calling frame is automatically registered as a table in the SQL context under the name "self". All [DataFrames](#) and [LazyFrames](#) found in the `envir` are also registered, using their variable name. More control over registration and execution behaviour is available by the [SQLContext](#) object.

**Usage**

```
LazyFrame_sql(query, ..., table_name = NULL, envir = parent.frame())
```

**Arguments**

query	A character of the SQL query to execute.
...	Ignored.
table_name	NULL (default) or a character of an explicit name for the table that represents the calling frame (the alias "self" will always be registered/available).
envir	The environment to search for polars <a href="#">DataFrames/LazyFrames</a> .

**Details**

This functionality is considered **unstable**, although it is close to being considered stable. It may be changed at any point without it being considered a breaking change.

**Value**

[LazyFrame](#)

**See Also**

- [SQLContext](#)

**Examples**

```
lf1 = pl$LazyFrame(a = 1:3, b = 6:8, c = c("z", "y", "x"))
lf2 = pl$LazyFrame(a = 3:1, d = c(125, -654, 888))

# Query the LazyFrame using SQL:
lf1$sql("SELECT c, b FROM self WHERE a > 1")$collect()

# Join two LazyFrames:
lf1$sql(
  "
  SELECT self.*, d
  FROM self
  INNER JOIN lf2 USING (a)
  WHERE a > 1 AND b < 8
  "
)$collect()

# Apply SQL transforms (aliasing "self" to "frame") and subsequently
# filter natively (you can freely mix SQL and native operations):
lf1$sql(
  query = r"(
  SELECT
  a,
  MOD(a, 2) == 0 AS a_is_even,
  (b::float / 2) AS 'b/2',
  CONCAT_WS(':', c, c, c) AS c_c_c
  FROM frame
  ORDER BY a
  )",

```

```
table_name = "frame"
) $filter(!pl$col("c_c_c")$str$starts_with("x"))$collect()
```

---

LazyFrame_std	<i>Std</i>
---------------	------------

---

**Description**

Aggregate the columns of this LazyFrame to their standard deviation values.

**Usage**

```
LazyFrame_std(ddof = 1)
```

**Arguments**

**ddof**                      Delta Degrees of Freedom: the divisor used in the calculation is  $N - \text{ddof}$ , where  $N$  represents the number of elements. By default `ddof` is 1.

**Value**

A LazyFrame with one row

**Examples**

```
pl$LazyFrame(mtcars)$std()$collect()
```

---

LazyFrame_sum	<i>Sum</i>
---------------	------------

---

**Description**

Aggregate the columns of this LazyFrame to their sum values.

**Usage**

```
LazyFrame_sum()
```

**Value**

A LazyFrame with one row

**Examples**

```
pl$LazyFrame(mtcars)$sum()$collect()
```

---

LazyFrame_tail	<i>Get the last n rows.</i>
----------------	-----------------------------

---

**Description**

Get the last n rows.

**Usage**

```
LazyFrame_tail(n = 5L)
```

**Arguments**

n                      Number of rows to return.

**Value**

A new LazyFrame object with applied filter.

**See Also**

[<LazyFrame>\\$head\(\)](#)

**Examples**

```
lf = pl$LazyFrame(a = 1:6, b = 7:12)
lf$tail()$collect()
lf$tail(2)$collect()
```

---

LazyFrame_to_dot	<i>Plot the query plan</i>
------------------	----------------------------

---

**Description**

This only returns the "dot" output that can be passed to other packages, such as `DiagrammeR::grViz()`.

**Usage**

```

LazyFrame_to_dot(
  ...,
  optimized = TRUE,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  comm_subplan_elim = TRUE,
  comm_subexpr_elim = TRUE,
  cluster_with_columns = TRUE,
  streaming = FALSE
)

```

**Arguments**

...	Not used..
optimized	Optimize the query plan.
type_coercion	Logical. Coerce types such that operations succeed and run on minimal required memory.
predicate_pushdown	Logical. Applies filters as early as possible at scan level.
projection_pushdown	Logical. Select only the columns that are needed at the scan level.
simplify_expression	Logical. Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.
slice_pushdown	Logical. Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. <code>join\$head(10)</code> ).
comm_subplan_elim	Logical. Will try to cache branching subplans that occur on self-joins or unions.
comm_subexpr_elim	Logical. Common subexpressions will be cached and reused.
cluster_with_columns	Combine sequential independent calls to <code>with_columns()</code> .
streaming	Logical. Run parts of the query in a streaming fashion (this is in an alpha state).

**Value**

A character vector

**Examples**

```

lf = pl$LazyFrame(
  a = c("a", "b", "a", "b", "b", "c"),

```

```

    b = 1:6,
    c = 6:1
  )

  query = lf$group_by("a", maintain_order = TRUE)$agg(
    pl$all()$sum()
  )$sort(
    "a"
  )

  query$to_dot() |> cat()

# You could print the graph by using DiagrammeR for example, with
# query$to_dot() |> DiagrammeR::grViz().

```

---

LazyFrame\_unique      *Drop duplicated rows*

---

## Description

Drop duplicated rows

## Usage

```
LazyFrame_unique(subset = NULL, ..., keep = "any", maintain_order = FALSE)
```

## Arguments

subset	A character vector with the names of the column(s) to use to identify duplicates. If NULL (default), use all columns.
...	Not used.
keep	Which of the duplicate rows to keep: <ul style="list-style-type: none"> <li>• "any" (default): Does not give any guarantee of which row is kept. This allows more optimizations.</li> <li>• "first": Keep first unique row.</li> <li>• "last": Keep last unique row.</li> <li>• "none": Don't keep duplicate rows.</li> </ul>
maintain_order	Keep the same order as the original data. Setting this to TRUE makes it more expensive to compute and blocks the possibility to run on the streaming engine.

## Value

LazyFrame



**Examples**

```
df = pl$LazyFrame(
  x = sample(10, 100, rep = TRUE),
  y = sample(10, 100, rep = TRUE)
)
df$collect()$height

df$unique()$collect()$height
df$unique(subset = "x")$collect()$height

df$unique(keep = "last")

# only keep unique rows
df$unique(keep = "none")
```

---

 LazyFrame\_unnest

*Unnest the Struct columns of a LazyFrame*


---

**Description**

Unnest the Struct columns of a LazyFrame

**Usage**

```
LazyFrame_unnest(...)
```

**Arguments**

... Names of the struct columns to unnest. This doesn't accept Expr. If nothing is provided, then all columns of datatype [Struct](#) are unnested.

**Value**

A LazyFrame where some or all columns of datatype Struct are unnested.

**Examples**

```
lf = pl$LazyFrame(
  a = 1:5,
  b = c("one", "two", "three", "four", "five"),
  c = 6:10
)$
select(
  pl$struct("b"),
  pl$struct(c("a", "c"))$alias("a_and_c")
)
lf$collect()

# by default, all struct columns are unnested
```

```
lf$unnest()$collect()

# we can specify specific columns to unnest
lf$unnest("a_and_c")$collect()
```

---

LazyFrame\_unpivot      *Unpivot a Frame from wide to long format*

---

### Description

Unpivot a Frame from wide to long format

### Usage

```
LazyFrame_unpivot(
  on = NULL,
  ...,
  index = NULL,
  variable_name = NULL,
  value_name = NULL
)
```

### Arguments

on	Values to use as identifier variables. If value_vars is empty all columns that are not in id_vars will be used.
...	Not used.
index	Columns to use as identifier variables.
variable_name	Name to give to the new column containing the names of the melted columns. Defaults to "variable".
value_name	Name to give to the new column containing the values of the melted columns. Defaults to "value".

### Details

Optionally leaves identifiers set.

This function is useful to massage a Frame into a format where one or more columns are identifier variables (id\_vars), while all other columns, considered measured variables (value\_vars), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

### Value

A LazyFrame

**Examples**

```
lf = pl$LazyFrame(
  a = c("x", "y", "z"),
  b = c(1, 3, 5),
  c = c(2, 4, 6)
)
lf$unpivot(index = "a", on = c("b", "c"))$collect()
```

---

 LazyFrame\_var

 Var
 

---

**Description**

Aggregate the columns of this LazyFrame to their variance values.

**Usage**

```
LazyFrame_var(ddof = 1)
```

**Arguments**

ddof                   Delta Degrees of Freedom: the divisor used in the calculation is  $N - \text{ddof}$ , where  $N$  represents the number of elements. By default ddof is 1.

**Value**

A LazyFrame with one row

**Examples**

```
pl$LazyFrame(mtcars)$var()$collect()
```

---

 LazyFrame\_with\_columns

*Select and modify columns of a LazyFrame*


---

**Description**

Add columns or modify existing ones with expressions. This is the equivalent of `dplyr::mutate()` as it keeps unmentioned columns (unlike `$select()`).

**Usage**

```
LazyFrame_with_columns(...)
```

**Arguments**

... Any expressions or string column name, or same wrapped in a list. If first and only element is a list, it is unwrapped as a list of args.

**Value**

A LazyFrame

**Examples**

```
pl$LazyFrame(iris)$with_columns(
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")
)

# same query
l_expr = list(
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")
)
pl$LazyFrame(iris)$with_columns(l_expr)

pl$LazyFrame(iris)$with_columns(
  pl$col("Sepal.Length")$abs(), # not named expr will keep name "Sepal.Length"
  SW_add_2 = (pl$col("Sepal.Width") + 2)
)
```

---

LazyFrame\_with\_columns\_seq

*Select and modify columns of a LazyFrame*

---

**Description**

Add columns or modify existing ones with expressions. This is the equivalent of `dplyr::mutate()` as it keeps unmentioned columns (unlike `$select()`).

This will run all expression sequentially instead of in parallel. Use this when the work per expression is cheap. Otherwise, `$with_columns()` should be preferred.

**Usage**

```
LazyFrame_with_columns_seq(...)
```

**Arguments**

... Any expressions or string column name, or same wrapped in a list. If first and only element is a list, it is unwrapped as a list of args.

**Value**

A LazyFrame

**Examples**

```
pl$LazyFrame(iris)$with_columns_seq(  
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),  
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")  
)  
  
# same query  
l_expr = list(  
  pl$col("Sepal.Length")$abs()$alias("abs_SL"),  
  (pl$col("Sepal.Length") + 2)$alias("add_2_SL")  
)  
pl$LazyFrame(iris)$with_columns_seq(l_expr)  
  
pl$LazyFrame(iris)$with_columns_seq(  
  pl$col("Sepal.Length")$abs(), # not named expr will keep name "Sepal.Length"  
  SW_add_2 = (pl$col("Sepal.Width") + 2)  
)
```

---

LazyFrame\_with\_context

*Add an external context to the computation graph*

---

**Description**

This allows expressions to also access columns from DataFrames or LazyFrames that are not part of this one.

**Usage**

```
LazyFrame_with_context(other)
```

**Arguments**

other                    Data/LazyFrame to have access to. This can be a list of DataFrames and LazyFrames.

**Value**

A LazyFrame

**Examples**

```
lf = pl$LazyFrame(a = c(1, 2, 3), b = c("a", "c", NA))
lf_other = pl$LazyFrame(c = c("foo", "ham"))

lf$with_context(lf_other)$select(
  pl$col("b") + pl$col("c")$first()
)$collect()

# Fill nulls with the median from another lazyframe:
train_lf = pl$LazyFrame(
  feature_0 = c(-1.0, 0, 1), feature_1 = c(-1.0, 0, 1)
)
test_lf = pl$LazyFrame(
  feature_0 = c(-1.0, NA, 1), feature_1 = c(-1.0, 0, 1)
)

test_lf$with_context(train_lf$select(pl$all()$name$suffix("_train")))$select(
  pl$col("feature_0")$fill_null(pl$col("feature_0_train")$median())
)$collect()
```

---

LazyFrame\_with\_row\_index

*Add a column for row indices*

---

**Description**

Add a new column at index 0 that counts the rows

**Usage**

```
LazyFrame_with_row_index(name, offset = NULL)
```

**Arguments**

name	string name of the created column
offset	positive integer offset for the start of the counter

**Value**

A new LazyFrame with a counter column in front

**Examples**

```
df = pl$LazyFrame(mtcars)

# by default, the index starts at 0 (to mimic the behavior of Python Polars)
df$with_row_index("idx")

# but in R, we use a 1-index
df$with_row_index("idx", offset = 1)
```

---

LazyGroupBy_agg	<i>LazyGroupBy_agg</i>
-----------------	------------------------

---

**Description**

aggregate a polar\_lazy\_group\_by

**Usage**

```
LazyGroupBy_agg(...)
```

**Arguments**

...                   exprs to aggregate over. ... args can also be passed wrapped in a list `$agg(list(e1, e2, e3))`

**Value**

A new LazyFrame object.

**Examples**

```
lgb = pl$DataFrame(
  foo = c("one", "two", "two", "one", "two"),
  bar = c(5, 3, 2, 4, 1)
)$
  lazy()$
  group_by("foo")

print(lgb)

lgb$
  agg(
    pl$col("bar")$sum()$name$suffix("_sum"),
    pl$col("bar")$mean()$alias("bar_tail_sum")
  )
```

---

LazyGroupBy_class	<i>Operations on Polars grouped LazyFrame</i>
-------------------	---

---

**Description**

This class comes from `<LazyFrame>$group_by()`, etc.

**Active bindings****columns:**

`$columns` returns a character vector with the column names.

**Examples**

```
as_polars_lf(mtcars)$group_by("cyl")$agg(
  pl$col("mpg")$sum()
)
```

---

 LazyGroupBy\_head

*LazyGroupBy\_head*


---

**Description**

get n rows of head of group

**Usage**

```
LazyGroupBy_head(n = 1L)
```

**Arguments**

n integer number of rows to get

**Value**

A new LazyFrame object.

---

 LazyGroupBy\_print

*LazyGroupBy\_print*


---

**Description**

prints opaque groupby, not much to show

**Usage**

```
LazyGroupBy_print()
```

**Value**

invisible self



---

LazyGroupBy\_tail      *LazyGroupBy\_tail*

---

**Description**

get n tail rows of group

**Usage**

```
LazyGroupBy_tail(n = 1L)
```

**Arguments**

n                    integer number of rows to get

**Value**

A new LazyFrame object.

---

LazyGroupBy\_ungroup      *LazyGroupBy\_ungroup*

---

**Description**

Revert the group by operation.

**Usage**

```
LazyGroupBy_ungroup()
```

**Value**

A new LazyFrame object.

**Examples**

```
lf = pl$LazyFrame(mtcars)
lf

lgb = lf$group_by("cyl")
lgb

lgb$ungroup()
```

---

length.RPolarsDataFrame  
*Get the length*

---

**Description**

Get the length

**Usage**

```
## S3 method for class 'RPolarsDataFrame'
length(x)

## S3 method for class 'RPolarsLazyFrame'
length(x)

## S3 method for class 'RPolarsSeries'
length(x)
```

**Arguments**

x                    A [DataFrame](#), [LazyFrame](#), or [Series](#)

---

max.RPolarsDataFrame    *Compute the maximum value*

---

**Description**

Compute the maximum value

**Usage**

```
## S3 method for class 'RPolarsDataFrame'
max(x, ...)

## S3 method for class 'RPolarsLazyFrame'
max(x, ...)

## S3 method for class 'RPolarsSeries'
max(x, ...)
```

**Arguments**

x                    A [DataFrame](#), [LazyFrame](#), or [Series](#)  
 ...                  Not used.

---

mean.RPolarsDataFrame *Compute the mean*

---

### Description

Compute the mean

### Usage

```
## S3 method for class 'RPolarsDataFrame'  
mean(x, ...)
```

```
## S3 method for class 'RPolarsLazyFrame'  
mean(x, ...)
```

```
## S3 method for class 'RPolarsSeries'  
mean(x, ...)
```

### Arguments

x	A <a href="#">DataFrame</a> , <a href="#">LazyFrame</a> , or <a href="#">Series</a>
...	Not used.

---

median.RPolarsDataFrame  
*Compute the median*

---

### Description

Compute the median

### Usage

```
## S3 method for class 'RPolarsDataFrame'  
median(x, ...)
```

```
## S3 method for class 'RPolarsLazyFrame'  
median(x, ...)
```

```
## S3 method for class 'RPolarsSeries'  
median(x, ...)
```

### Arguments

x	A <a href="#">DataFrame</a> , <a href="#">LazyFrame</a> , or <a href="#">Series</a>
...	Not used.

---

min.RPolarsDataFrame *Compute the minimum value*

---

### Description

Compute the minimum value

### Usage

```
## S3 method for class 'RPolarsDataFrame'
min(x, ...)

## S3 method for class 'RPolarsLazyFrame'
min(x, ...)

## S3 method for class 'RPolarsSeries'
min(x, ...)
```

### Arguments

x	A <a href="#">DataFrame</a> , <a href="#">LazyFrame</a> , or <a href="#">Series</a>
...	Not used.

---

na.omit.RPolarsLazyFrame  
*Drop missing values*

---

### Description

Drop missing values

### Usage

```
## S3 method for class 'RPolarsLazyFrame'
na.omit(object, subset = NULL, ...)

## S3 method for class 'RPolarsDataFrame'
na.omit(object, subset = NULL, ...)
```

### Arguments

object	A <a href="#">DataFrame</a> or <a href="#">LazyFrame</a>
subset	Character vector of column names to drop missing values from.
...	Not used.

**Examples**

```
df = pl$DataFrame(data.frame(a = c(NA, 2:10), b = c(1, NA, 3:10)))$lazy()
na.omit(df)
na.omit(df, subset = "a")
na.omit(df, subset = c("a", "b"))
```

---

names.RPolarsDataFrame

*Get the column names*


---

**Description**

Get the column names

**Usage**

```
## S3 method for class 'RPolarsDataFrame'
names(x)

## S3 method for class 'RPolarsLazyFrame'
names(x)

## S3 method for class 'RPolarsLazyGroupBy'
names(x)

## S3 method for class 'RPolarsGroupBy'
names(x)
```

**Arguments**

x                    A [DataFrame](#) or [LazyFrame](#)

---

pl\_all

*New Expr referring to all columns*


---

**Description**

Not to mix up with `Expr_object$all()` which is a 'reduce Boolean columns by AND' method.

**Usage**

```
pl_all(name = NULL)
```

**Arguments**

name                    Character vector indicating on which columns the AND operation should be applied.

**Value**

Boolean literal

**Examples**

```
test = pl$DataFrame(col_1 = c(TRUE, TRUE), col_2 = c(TRUE, FALSE))
test

# here, the first `all()` selects all columns, and the second `all()` checks
# whether all values are true in each column
test$with_columns(pl$all()$all())
```

---

pl\_all\_horizontal      *Apply the AND logical rowwise*

---

**Description**

Apply the AND logical rowwise

**Usage**

```
pl_all_horizontal(...)
```

**Arguments**

...                      Columns to concatenate into a single string column. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = c(TRUE, FALSE, NA, NA),
  b = c(TRUE, FALSE, NA, NA),
  c = c(TRUE, FALSE, NA, TRUE)
)
df

df$with_columns(
  pl_all_horizontal("a", "b", "c")$alias("all")
)

# drop rows that have at least one missing value
# == keep rows that only have non-missing values
df$filter(
  pl_all_horizontal(pl$all())$is_not_null()
)
```

---

pl\_any\_horizontal      *Apply the OR logical rowwise*

---

**Description**

Apply the OR logical rowwise

**Usage**

```
pl_any_horizontal(...)
```

**Arguments**

...                      Columns to concatenate into a single string column. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = c(FALSE, FALSE, NA, NA),
  b = c(TRUE, FALSE, NA, NA),
  c = c(TRUE, FALSE, NA, TRUE)
)
df

df$with_columns(
  pl$any_horizontal("a", "b", "c")$alias("any")
)

# drop rows that only have missing values == keep rows that have at least one
# non-missing value
df$filter(
  pl$any_horizontal(pl$all())$is_not_null()
)
```

---

pl\_approx\_n\_unique      *Approximate count of unique values*

---

**Description**

This function is syntactic sugar for `pl$col(...)$approx_n_unique()`, and uses the HyperLogLog++ algorithm for cardinality estimation.

**Usage**

```
pl_approx_n_unique(...)
```

**Arguments**

... Characters indicating the column names, passed to `pl$col()`. See `?pl_col` for details.

**Value**

Expr

**See Also**

- `<Expr>$approx_n_unique()`

**Examples**

```
df = pl$DataFrame(
  a = c(1, 8, 1),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

df$select(pl$approx_n_unique("a"))

df$select(pl$approx_n_unique("b", "c"))
```

---

pl\_arg\_sort\_by

*Return the row indices that would sort the columns*

---

**Description**

Return the row indices that would sort the columns

**Usage**

```
pl_arg_sort_by(
  ...,
  descending = FALSE,
  nulls_last = FALSE,
  multithreaded = TRUE,
  maintain_order = FALSE
)
```



**Arguments**

...	Column(s) to arg sort by. Can be Expr(s) or something coercible to Expr(s). Strings are parsed as column names.
descending	Logical. Sort in descending order (default is FALSE). This must be either of length 1 or a logical vector of the same length as the number of Expr(s) specified in by and ...
nulls_last	A logical or logical vector of the same length as the number of columns. If TRUE, place null values last instead of first.
multithreaded	A logical. If TRUE, sort using multiple threads.
maintain_order	Whether the order should be maintained if elements are equal. If TRUE, streaming is not possible and performance might be worse since this requires a stable search.

**Value**

Expr

**See Also**

[\\$arg\\_sort\(\)](#) to find the row indices that would sort an Expr.

**Examples**

```
df = pl$DataFrame(
  a = c(0, 1, 1, 0),
  b = c(3, 2, 3, 2)
)

df$with_columns(
  arg_sort_a = pl$arg_sort_by("a"),
  arg_sort_ab = pl$arg_sort_by(c("a", "b"), descending = TRUE)
)

# we can also pass Expr
df$with_columns(
  arg_sort_a = pl$arg_sort_by(pl$col("a") * -1)
)
```

---

pl\_arg\_where

*Return indices that match a condition*

---

**Description**

Return indices that match a condition

**Usage**

```
pl_arg_where(condition)
```

**Arguments**

condition      An Expr that gives a boolean.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(a = c(1, 2, 3, 4, 5))
df$select(
  pl$arg_where(pl$col("a") %% 2 == 0)
)
```

---

pl_coalesce	<i>Coalesce</i>
-------------	-----------------

---

**Description**

Folds the expressions from left to right, keeping the first non-null value.

**Usage**

```
pl_coalesce(...)
```

**Arguments**

...      is a: If one arg:

- Series or Expr, same as `column$sum()`
- string, same as `pl$col(column)$sum()`
- numeric, same as `pl$lit(column)$sum()`
- list of strings(column names) or expressions to add up as `expr1 + expr2 + expr3 + ...`

If several args, then wrapped in a list and handled as above.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = NA_real_,
  b = c(1L, 4L, NA_real_, NA_real_),
  c = c(2:4, NA_real_)
)

# use coalesce to get first non Null value for each row, otherwise insert 99.9
df$with_columns(
  pl$coalesce("a", "b", "c", 99.9)$alias("d")
)
```

pl\_col

*Create an expression representing column(s) in a dataframe***Description**

Create an expression representing column(s) in a dataframe

**Usage**

```
pl_col(...)
```

**Arguments**

... One of the following:

- character vectors
  - Single wildcard "\*" has a special meaning: check the examples.
- [RPolarsDataTypes](#)
- a list of [RPolarsDataTypes](#)

**Value**

[Expr](#) of a column or columns

**Examples**

```
# a single column by a character
pl$col("foo")

# multiple columns by characters
pl$col("foo", "bar")

# multiple columns by RPolarsDataTypes
pl$col(pl$Float64, pl$String)

# Single "*" is converted to a wildcard expression
pl$col("*")
```

```

# multiple character vectors and a list of RPolarsDataTypes are also allowed
pl$col(c("foo", "bar"), "baz")
pl$col("foo", c("bar", "baz"))
pl$col(list(pl$Float64, pl$String))

# there are some special notations for selecting columns
df = pl$DataFrame(foo = 1:3, bar = 4:6, baz = 7:9)

## select all columns with a wildcard `"*"`
df$select(pl$col("*"))

## select multiple columns by a regular expression
## starts with `^` and ends with `$`
df$select(pl$col(c("^ba.*$")))

```

---

pl\_concat

*Concat polars objects*

---

## Description

Concat polars objects

## Usage

```

pl_concat(
  ...,
  how = c("vertical", "vertical_relaxed", "horizontal", "diagonal", "diagonal_relaxed"),
  rechunk = FALSE,
  parallel = TRUE
)

```

## Arguments

...	Either individual unpacked args or args wrapped in list(). Args can be eager as DataFrame, Series and R vectors, or lazy as LazyFrame and Expr. The first element determines the output of \$concat(): if the first element is lazy, a LazyFrame is returned; otherwise, a DataFrame is returned (note that if the first element is eager, all other elements have to be eager to avoid implicit collect).
how	Bind direction. Can be "vertical" (like rbind()), "horizontal" (like cbind()), or "diagonal". For "vertical" and "diagonal", adding the suffix "_relaxed" will cast columns to their shared supertypes. For example, if we try to vertically concatenate two columns of types i32 and f64, using how = "vertical_relaxed" will cast the column of type i32 to f64 beforehand.
rechunk	Perform a rechunk at last.
parallel	Only used for LazyFrames. If TRUE (default), lazy computations may be executed in parallel.

**Details**

Categorical columns/Series must have been constructed while global string cache enabled. See [pl\\$enable\\_string\\_cache\(\)](#).

**Value**

DataFrame, Series, LazyFrame or Expr

**Examples**

```
# vertical
l_ver = lapply(1:10, function(i) {
  l_internal = list(
    a = 1:5,
    b = letters[1:5]
  )
  pl$DataFrame(l_internal)
})
pl$concat(l_ver, how = "vertical")

# horizontal
l_hor = lapply(1:10, function(i) {
  l_internal = list(
    1:5,
    letters[1:5]
  )
  names(l_internal) = paste0(c("a", "b"), i)
  pl$DataFrame(l_internal)
})
pl$concat(l_hor, how = "horizontal")

# diagonal
pl$concat(l_hor, how = "diagonal")

# if two columns don't share the same type, concat() will error unless we use
# `how = "vertical_relaxed"`:
test = pl$DataFrame(x = 1L) # i32
test2 = pl$DataFrame(x = 1.0) # f64

pl$concat(test, test2, how = "vertical_relaxed")
```

---

pl\_concat\_list

*Concat the arrays in a Series dtype List in linear time.*

---

**Description**

Folds the expressions from left to right, keeping the first non-null value.

**Usage**

```
pl_concat_list(exprs)
```

**Arguments**

exprs                    list of Into, strings interpreted as column names

**Value**

Expr

**Examples**

```
# Create lagged columns and collect them into a list. This mimics a rolling window.
df = pl$DataFrame(A = c(1, 2, 9, 2, 13))
df$with_columns(lapply(
  0:2,
  \(i) pl$col("A")$shift(i)$alias(paste0("A_lag_", i))
))$select(
  pl$concat_list(lapply(2:0, \(i) pl$col(paste0("A_lag_", i))))$alias(
    "A_rolling"
  )
)

# concat Expr a Series and an R object
pl$concat_list(list(
  pl$lit(1:5),
  as_polars_series(5:1),
  rep(0L, 5)
))$alias("alice")$to_series()
```

---

pl\_concat\_str

*Horizontally concatenate columns into a single string column*

---

**Description**

Horizontally concatenate columns into a single string column

**Usage**

```
pl_concat_str(..., separator = "", ignore_nulls = FALSE)
```

**Arguments**

...                    Columns to concatenate into a single string column. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals. Non-String columns are cast to String

separator	String that will be used to separate the values of each column.
ignore_nulls	If FALSE (default), null values are propagated: if the row contains any null values, the output is null.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = 1:3,
  b = c("dogs", "cats", NA),
  c = c("play", "swim", "walk")
)

df$with_columns(
  pl$concat_str(
    pl$col("a") * 2L, "b", "c", pl$lit("!"),
    separator = " "
  )$alias("full_sentence")
)

df$with_columns(
  pl$concat_str(
    pl$col("a") * 2L, "b", "c", pl$lit("!"),
    separator = " ",
    ignore_nulls = TRUE
  )$alias("full_sentence")
)
```

---

pl_corr	<i>Correlation</i>
---------	--------------------

---

**Description**

Calculates the correlation between two columns

**Usage**

```
pl_corr(a, b, method = "pearson", ddof = 1, propagate_nans = FALSE)
```

**Arguments**

a	One column name or Expr or anything convertible Into via pl\$col().
b	Another column name or Expr or anything convertible Into via pl\$col().
method	str One of 'pearson' or 'spearman'

ddof	integer Delta Degrees of Freedom: the divisor used in the calculation is $N - \text{ddof}$ , where $N$ represents the number of elements. By default <code>ddof</code> is 1.
propagate_nans	bool Used only when calculating the spearman rank correlation. If <code>True</code> any <code>NaN</code> encountered will lead to <code>NaN</code> in the output. Defaults to <code>False</code> where <code>NaN</code> are regarded as larger than any finite number and thus lead to the highest rank.

**Value**

Expr for the computed correlation

**Examples**

```
lf = pl$LazyFrame(data.frame(a = c(1, 8, 3), b = c(4, 5, 2)))
lf$select(pl$corr("a", "b", method = "spearman"))$collect()
```

---

pl_count	<i>Return the number of non-null values in the column.</i>
----------	--

---

**Description**

This function is syntactic sugar for `pl$col(...)$count()`.

**Usage**

```
pl_count(...)
```

**Arguments**

... Characters indicating the column names, passed to `pl$col()`. See `?pl_col` for details.

**Details**

Calling this function without any arguments returns the number of rows in the context. This way of using the function is deprecated. Please use `pl$len()` instead.

**Value**

Expression of data type `UInt32`

**See Also**

- `pl$len()`
- `<Expr>$count()`



**Examples**

```
df = pl$DataFrame(
  a = c(1, 2, NA),
  b = c(3, NA, NA),
  c = c("foo", "bar", "foo")
)

df$select(pl$count("a"))

df$select(pl$count(c("b", "c")))
```

---

pl_cov	<i>Covariance</i>
--------	-------------------

---

**Description**

Calculates the covariance between two columns / expressions.

**Usage**

```
pl_cov(a, b, ddof = 1)
```

**Arguments**

a	One column name or Expr or anything convertible Into via pl\$col().
b	Another column name or Expr or anything convertible Into via pl\$col().
ddof	integer Delta Degrees of Freedom: the divisor used in the calculation is N - ddof, where N represents the number of elements. By default ddof is 1.

**Value**

Expr for the computed covariance

**Examples**

```
lf = pl$LazyFrame(data.frame(a = c(1, 8, 3), b = c(4, 5, 2)))
lf$select(pl$cov("a", "b"))$collect()
pl$cov(c(1, 8, 3), c(4, 5, 2))$to_r()
```

---

pl\_DataFrame                      *Create a new polars DataFrame*

---

## Description

Create a new polars DataFrame

## Usage

```
pl_DataFrame(..., make_names_unique = TRUE, schema = NULL)
```

## Arguments

...                      One of the following:

- a list of mixed vectors and Series of equal length
- mixed vectors and/or Series of equal length
- a positional argument of a [data.frame](#) or a [DataFrame](#) (not recommended use). In this case, the object will be passed to [as\\_polars\\_df\(\)](#).

Columns will be named as of named arguments or alternatively by names of Series or given a placeholder name.

make\_names\_unique                      If TRUE (default), any duplicated names will be prefixed a running number.

schema                      A named list that will be used to convert a variable to a specific DataType. See [Examples](#).

## Value

[DataFrame](#)

## See Also

- [as\\_polars\\_df\(\)](#)

## Examples

```
pl$DataFrame(
  a = c(1, 2, 3, 4, 5),
  b = 1:5,
  c = letters[1:5],
  d = list(1:1, 1:2, 1:3, 1:4, 1:5)
) # directly from vectors

# from a list of vectors
pl$DataFrame(list(
  a = c(1, 2, 3, 4, 5),
  b = 1:5,
  c = letters[1:5],
```

```
d = list(1L, 1:2, 1:3, 1:4, 1:5)
))

# from a data.frame
pl$DataFrame(mtcars)

# custom schema
pl$DataFrame(iris, schema = list(Sepal.Length = pl$Float32, Species = pl$String))
```

---

pl\_date

*Create a Date expression*

---

## Description

Create a Date expression

## Usage

```
pl_date(year, month, day)
```

## Arguments

year	An Expr or something coercible to an Expr, that must return an integer. Strings are parsed as column names. Floats are cast to integers.
month	An Expr or something coercible to an Expr, that must return an integer between 1 and 12. Strings are parsed as column names. Floats are cast to integers.
day	An Expr or something coercible to an Expr, that must return an integer between 1 and 31. Strings are parsed as column names. Floats are cast to integers.

## Value

An Expr of type Date

## See Also

- [pl\\$datetime\(\)](#)
- [pl\\$time\(\)](#)

## Examples

```
df = pl$DataFrame(year = 2019:2021, month = 9:11, day = 10:12)

df$with_columns(
  date_from_cols = pl$date("year", "month", "day"),
  date_from_lit = pl$date(2020, 3, 5),
  date_from_mix = pl$date("year", 3, 5)
)
```

```

# floats are coerced to integers
df$with_columns(
  date_floats = pl$date(2018.8, 5.3, 1)
)

# if date can't be constructed, it returns null
df$with_columns(
  date_floats = pl$date(pl$lit("abc"), -2, 1)
)

```

---

pl\_datetime                      *Create a Datetime expression*

---

## Description

Create a Datetime expression

## Usage

```

pl_datetime(
  year,
  month,
  day,
  hour = NULL,
  minute = NULL,
  second = NULL,
  microsecond = NULL,
  ...,
  time_unit = "us",
  time_zone = NULL,
  ambiguous = "raise"
)

```

## Arguments

year	An Expr or something coercible to an Expr, that must return an integer. Strings are parsed as column names. Floats are cast to integers.
month	An Expr or something coercible to an Expr, that must return an integer between 1 and 12. Strings are parsed as column names. Floats are cast to integers.
day	An Expr or something coercible to an Expr, that must return an integer between 1 and 31. Strings are parsed as column names. Floats are cast to integers.
hour	An Expr or something coercible to an Expr, that must return an integer between 0 and 23. Strings are parsed as column names. Floats are cast to integers.
minute	An Expr or something coercible to an Expr, that must return an integer between 0 and 59. Strings are parsed as column names. Floats are cast to integers.
second	An Expr or something coercible to an Expr, that must return an integer between 0 and 59. Strings are parsed as column names. Floats are cast to integers.

microsecond	An Expr or something coercible to an Expr, that must return an integer between 0 and 999,999. Strings are parsed as column names. Floats are cast to integers.
...	Not used.
time_unit	Unit of time. One of "ms", "us" (default) or "ns".
time_zone	Time zone string, as defined in <a href="#">OlsonNames()</a> . Setting timezone = "*" will match any timezone, which can be useful to select all Datetime columns containing a timezone.
ambiguous	Determine how to deal with ambiguous datetimes: <ul style="list-style-type: none"> <li>• "raise" (default): throw an error</li> <li>• "earliest": use the earliest datetime</li> <li>• "latest": use the latest datetime</li> <li>• "null": return a null value</li> </ul>

**Value**

An Expr of type Datetime

**See Also**

- [pl\\$date\(\)](#)
- [pl\\$time\(\)](#)

**Examples**

```
df = pl$DataFrame(
  year = 2019:2021,
  month = 9:11,
  day = 10:12,
  min = 55:57
)

df$with_columns(
  dt_from_cols = pl$datetime("year", "month", "day", minute = "min"),
  dt_from_lit = pl$datetime(2020, 3, 5, hour = 20:22),
  dt_from_mix = pl$datetime("year", 3, 5, second = 1)
)

# floats are coerced to integers
df$with_columns(
  dt_floats = pl$datetime(2018.8, 5.3, 1, second = 2.1)
)

# if datetime can't be constructed, it returns null
df$with_columns(
  dt_floats = pl$datetime(pl$lit("abc"), -2, 1)
)

# can control the time_unit
df$with_columns(
```

```
dt_from_cols = pl$datetime("year", "month", "day", minute = "min", time_unit = "ms")
)
```

---

pl\_datetime\_range      *Generate a datetime range*

---

## Description

Generate a datetime range

## Usage

```
pl_datetime_range(
  start,
  end,
  interval = "1d",
  ...,
  closed = "both",
  time_unit = NULL,
  time_zone = NULL
)
```

## Arguments

start	Lower bound of the date range. Something that can be coerced to a <a href="#">Date</a> or a <a href="#">Datetime</a> expression. See examples for details.
end	Upper bound of the date range. Something that can be coerced to a <a href="#">Date</a> or a <a href="#">Datetime</a> expression. See examples for details.
interval	Interval of the range periods, specified as a <a href="#">difftime</a> object or using the Polars duration string language. See the Polars duration string language section for details.
...	Ignored.
closed	Define which sides of the range are closed (inclusive). One of the followings: "both" (default), "left", "right", "none".
time_unit	Time unit of the resulting the <a href="#">Datetime</a> data type. One of "ns", "us", "ms" or NULL
time_zone	Time zone of the resulting <a href="#">Datetime</a> data type.

## Value

An [Expr](#) of data type [Datetime](#)

### Polars duration string language

Polars duration string language is a simple representation of durations. It is used in many Polars functions that accept durations.

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)
- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds

By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".

### See Also

[pl\\_datetime\\_ranges\(\)](#) to create a simple Series of data type list(Datetime) based on column values.

### Examples

```
# Using Polars duration string to specify the interval:
pl_datetime_range(as.Date("2022-01-01"), as.Date("2022-03-01"), "1mo") |>
  as_polars_series("datetime")

# Using `difftime` object to specify the interval:
pl_datetime_range(
  as.Date("1985-01-01"),
  as.Date("1985-01-10"),
  as.difftime(1, units = "days") + as.difftime(12, units = "hours")
) |>
  as_polars_series("datetime")

# Specifying a time zone:
pl_datetime_range(
  as.Date("2022-01-01"),
  as.Date("2022-03-01"),
  "1mo",
  time_zone = "America/New_York"
```

```
) |>
  as_polars_series("datetime")
```

---

pl\_datetime\_ranges      *Generate a list containing a datetime range*

---

## Description

Generate a list containing a datetime range

## Usage

```
pl_datetime_ranges(
  start,
  end,
  interval = "1d",
  ...,
  closed = "both",
  time_unit = NULL,
  time_zone = NULL
)
```

## Arguments

start	Lower bound of the date range. Something that can be coerced to a <a href="#">Date</a> or a <a href="#">Datetime</a> expression. See examples for details.
end	Upper bound of the date range. Something that can be coerced to a <a href="#">Date</a> or a <a href="#">Datetime</a> expression. See examples for details.
interval	Interval of the range periods, specified as a <a href="#">difftime</a> object or using the Polars duration string language. See the Polars <a href="#">duration string language</a> section for details.
...	Ignored.
closed	Define which sides of the range are closed (inclusive). One of the followings: "both" (default), "left", "right", "none".
time_unit	Time unit of the resulting the <a href="#">Datetime</a> data type. One of "ns", "us", "ms" or NULL
time_zone	Time zone of the resulting <a href="#">Datetime</a> data type.

## Value

An [Expr](#) of data type `list(Datetime)`



## Polars duration string language

Polars duration string language is a simple representation of durations. It is used in many Polars functions that accept durations.

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)
- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds

By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".

## See Also

[pl\\$datetime\\_range\(\)](#) to create a simple Series of data type Datetime.

## Examples

```
df = pl$DataFrame(
  start = as.POSIXct(c("2022-01-01 10:00", "2022-01-01 11:00", NA)),
  end = as.POSIXct("2022-01-01 12:00")
)

df$with_columns(
  dt_range = pl$datetime_ranges("start", "end", interval = "1h"),
  dt_range_cr = pl$datetime_ranges("start", "end", closed = "right", interval = "1h")
)

# provide a custom "end" value
df$with_columns(
  dt_range_lit = pl$datetime_ranges(
    "start", pl$lit(as.POSIXct("2022-01-01 11:00")),
    interval = "1h"
  )
)
```

---

pl_date_range	<i>Generate a date range</i>
---------------	------------------------------

---

### Description

If both start and end are passed as the Date types (not Datetime), and the interval granularity is no finer than "1d", the returned range is also of type Date. All other permutations return a Datetime.

### Usage

```
pl_date_range(start, end, interval = "1d", ..., closed = "both")
```

### Arguments

start	Lower bound of the date range. Something that can be coerced to a Date or a <a href="#">Datetime</a> expression. See examples for details.
end	Upper bound of the date range. Something that can be coerced to a Date or a <a href="#">Datetime</a> expression. See examples for details.
interval	Interval of the range periods, specified as a <a href="#">difftime</a> object or using the Polars duration string language. See the <a href="#">Polars duration string language</a> section for details.
...	Ignored.
closed	Define which sides of the range are closed (inclusive). One of the followings: "both" (default), "left", "right", "none".

### Value

An [Expr](#) of data type Date or [Datetime](#)

### Polars duration string language

Polars duration string language is a simple representation of durations. It is used in many Polars functions that accept durations.

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)
- 1mo (1 calendar month)

- 1q (1 calendar quarter)
- 1y (1 calendar year)

Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds

By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".

### See Also

[pl\\$date\\_ranges\(\)](#) to create a simple Series of data type list(Date) based on column values.

### Examples

```
# Using Polars duration string to specify the interval:
pl$date_range(as.Date("2022-01-01"), as.Date("2022-03-01"), "1mo") |>
  as_polars_series("date")

# Using `difftime` object to specify the interval:
pl$date_range(
  as.Date("1985-01-01"),
  as.Date("1985-01-10"),
  as.difftime(2, units = "days")
) |>
  as_polars_series("date")
```

---

<code>pl_date_ranges</code>	<i>Generate a list containing a date range</i>
-----------------------------	--

---

### Description

If both `start` and `end` are passed as the `Date` types (not `Datetime`), and the `interval` granularity is no finer than "1d", the returned range is also of type `Date`. All other permutations return a `Datetime`.

### Usage

```
pl_date_ranges(start, end, interval = "1d", ..., closed = "both")
```

### Arguments

<code>start</code>	Lower bound of the date range. Something that can be coerced to a <code>Date</code> or a <a href="#">Datetime</a> expression. See examples for details.
<code>end</code>	Upper bound of the date range. Something that can be coerced to a <code>Date</code> or a <a href="#">Datetime</a> expression. See examples for details.
<code>interval</code>	Interval of the range periods, specified as a <a href="#">difftime</a> object or using the Polars duration string language. See the <a href="#">Polars duration string language</a> section for details.

... Ignored.

closed Define which sides of the range are closed (inclusive). One of the followings: "both" (default), "left", "right", "none".

### Value

An [Expr](#) of data type List(Date) or List(Datetime)

### Polars duration string language

Polars duration string language is a simple representation of durations. It is used in many Polars functions that accept durations.

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)
- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds

By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".

### See Also

[pl\\$date\\_range\(\)](#) to create a simple Series of data type Date.

### Examples

```
df = pl.DataFrame(
  start = as.Date(c("2022-01-01", "2022-01-02", NA)),
  end = as.Date("2022-01-03")
)

df$with_columns(
  date_range = pl$date_ranges("start", "end"),
  date_range_cr = pl$date_ranges("start", "end", closed = "right")
)
```

```
# provide a custom "end" value
df$with_columns(
  date_range_lit = pl$date_ranges("start", pl$lit(as.Date("2022-01-02")))
)
```

---

pl\_deserialize\_lf      *Read a logical plan from a JSON file to construct a LazyFrame*

---

### Description

Read a logical plan from a JSON file to construct a LazyFrame

### Usage

```
pl_deserialize_lf(json)
```

### Arguments

json                      A character of the JSON representation of the logical plan.

### Value

[LazyFrame](#)

### See Also

- [<LazyFrame>\\$serialize\(\)](#)

### Examples

```
lf = pl$LazyFrame(a = 1:3)$sum()
json = lf$serialize()
pl$deserialize_lf(json)$collect()
```

---

pl\_disable\_string\_cache

*Disable the global string cache*

---

### Description

Some functions (e.g joins) can be applied on Categorical series only allowed if using the global string cache is enabled. This function disables the string\_cache. In general, you should use pl\$with\_string\_cache() instead.

### Usage

```
pl_disable_string_cache()
```

**Value**

This doesn't return any value.

**See Also**

[pl\\$using\\_string\\_cache](#) [pl\\$enable\\_string\\_cache](#) [pl\\$with\\_string\\_cache](#)

**Examples**

```
pl$enable_string_cache()
pl$using_string_cache()
pl$disable_string_cache()
pl$using_string_cache()
```

---

pl\_dtypes

*DataTypes (RPolarsDataType)*

---

**Description**

DataType any polars type (ported so far)

**Value**

not applicable

**Examples**

```
print(ls(pl$dtypes))
pl$dtypes$Float64
pl$dtypes$string

pl$list(pl$list(pl$uint64))

pl$struct(pl$field("CityNames", pl$string))

# The function changes type from Int32 to String
# Specifying the output DataType: String solves the problem
as_polars_series(1:4)$map_elements(\(x) letters[x], datatype = pl$dtypes$string)
```

---

pl_duration	Create polars Duration from distinct time components
-------------	--

---

### Description

Create polars Duration from distinct time components

### Usage

```
pl_duration(
    ...,
    weeks = NULL,
    days = NULL,
    hours = NULL,
    minutes = NULL,
    seconds = NULL,
    milliseconds = NULL,
    microseconds = NULL,
    nanoseconds = NULL,
    time_unit = "us"
)
```

### Arguments

...	Not used.
weeks	Number of weeks to add. Expr or something coercible to an Expr. Strings are parsed as column names. <i>Same thing for argument days to nanoseconds.</i>
days	Number of days to add.
hours	Number of hours to add.
minutes	Number of minutes to add.
seconds	Number of seconds to add.
milliseconds	Number of milliseconds to add.
microseconds	Number of microseconds to add.
nanoseconds	Number of nanoseconds to add.
time_unit	Time unit of the resulting expression.

### Details

A duration represents a fixed amount of time. For example, `pl$duration(days = 1)` means "exactly 24 hours". By contrast, `Expr$dt$offset_by('1d')` means "1 calendar day", which could sometimes be 23 hours or 25 hours depending on Daylight Savings Time. For non-fixed durations such as "calendar month" or "calendar day", please use `Expr$dt$offset_by()` instead.

**Value**

Expr

**Examples**

```

test = pl$DataFrame(
  dt = c(
    "2022-01-01 00:00:00",
    "2022-01-02 00:00:00"
  ),
  add = 1:2
)$with_columns(
  pl$col("dt")$str$strptime(pl$Datetime("us"), format = NULL)
)

test$with_columns(
  (pl$col("dt") + pl$duration(weeks = "add"))$alias("add_weeks"),
  (pl$col("dt") + pl$duration(days = "add"))$alias("add_days"),
  (pl$col("dt") + pl$duration(seconds = "add"))$alias("add_seconds"),
  (pl$col("dt") + pl$duration(milliseconds = "add"))$alias("add_millis"),
  (pl$col("dt") + pl$duration(hours = "add"))$alias("add_hours")
)

# we can also pass an Expr
test$with_columns(
  (pl$col("dt") + pl$duration(weeks = pl$col("add") + 1))$alias("add_weeks"),
  (pl$col("dt") + pl$duration(days = pl$col("add") + 1))$alias("add_days"),
  (pl$col("dt") + pl$duration(seconds = pl$col("add") + 1))$alias("add_seconds"),
  (pl$col("dt") + pl$duration(milliseconds = pl$col("add") + 1))$alias("add_millis"),
  (pl$col("dt") + pl$duration(hours = pl$col("add") + 1))$alias("add_hours")
)

```

---

pl\_element

*an element in 'eval'-expr*


---

**Description**

Alias for an element in evaluated in an eval expression.

**Usage**

```
pl_element()
```

**Value**

Expr

**Examples**

```
pl$lit(1:5)$cumulative_eval(pl$element())$first() - pl$element()$last()**2)$to_r()
```



---

`pl_enable_string_cache`*Enable the global string cache*

---

**Description**

Some functions (e.g joins) can be applied on Categorical series only allowed if using the global string cache is enabled. This function enables the string\_cache. In general, you should use `pl$with_string_cache()` instead.

**Usage**

```
pl_enable_string_cache()
```

**Value**

This doesn't return any value.

**See Also**

[pl\\$using\\_string\\_cache](#) [pl\\$disable\\_string\\_cache](#) [pl\\$with\\_string\\_cache](#)

**Examples**

```
pl$enable_string_cache()
pl$using_string_cache()
pl$disable_string_cache()
pl$using_string_cache()
```

---

`pl_field`*Quickly select a field in a Struct*

---

**Description**

This is syntactic sugar that should mostly be used in `$struct$with_fields()`. `pl$field("x")` is equivalent to `pl$col("my_struct")$struct$field("x")`.

**Usage**

```
pl_field(name)
```

**Arguments**

name                    Name of the field to select.

**Value**

An Expr with the datatype from the selected field.

**Examples**

```
df = pl$DataFrame(x = c(1, 4, 9), y = c(4, 9, 16), multiply = c(10, 2, 3))$
  with_columns(coords = pl$struct(c("x", "y")))$
  select("coords", "multiply")
```

```
df
```

```
df = df$with_columns(
  pl$col("coords")$struct$with_fields(
    pl$field("x")$sqrt(),
    y_mul = pl$field("y") * pl$col("multiply")
  )
)
```

```
df
```

```
df$unnest("coords")
```

---

pl_Field_class	<i>Create Field</i>
----------------	---------------------

---

**Description**

A Field is composed of a name and a data type. Fields are used in Structs-datatypes and Schemas to represent everything of the Series/Column except the raw values.

**Usage**

```
pl_Field(name, datatype)
```

**Arguments**

name	Field name
datatype	<a href="#">DataType</a>

**Value**

An object of class "RPolarsRField" containing its name and its [data type](#).

**Active Bindings****datatype:**

\$datatype returns the [data type](#) of the Field.

\$datatype = <RPolarsDataType> sets the [data type](#) of the Field.

**name:**

\$name returns the name of the Field.

\$name = "new\_name" sets the name of the Field.

**Examples**

```
field = pl$Field("city_names", pl$String)

field
field$datatype
field$name

# Set the new data type
field$datatype = pl$Categorical()
field$datatype

# Set the new name
field$name = "CityPopulations"
field
```

---

pl\_first

*Get the first value.*

---

**Description**

This function has different behavior depending on arguments:

- Missing -> Takes first column of a context.
- Character vectors -> Syntactic sugar for `pl$col(...)$first()`.

**Usage**

```
pl_first(...)
```

**Arguments**

... Characters indicating the column names (passed to `pl$col()`, see `?pl_col` for details), or empty. If empty (default), returns an expression to take the first column of the context instead.

**Value**

[Expr](#)

**See Also**

- [<Expr>\\$first\(\)](#)

**Examples**

```
df = pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

df$select(pl$first())

df$select(pl$first("b"))

df$select(pl$first(c("a", "c")))
```

---

pl\_fold

*Accumulate over multiple columns horizontally with an R function*

---

**Description**

This allows one to do rowwise operations, starting with an initial value (acc). See [pl\\$reduce\(\)](#) to do rowwise operations without this initial value.

**Usage**

```
pl_fold(acc, lambda, exprs)
```

**Arguments**

acc	an Expr or Into of the initial accumulator.
lambda	R function which takes two polars Series as input and return one.
exprs	Expressions to aggregate over. May also be a wildcard expression.

**Value**

An expression that will be applied rowwise

**Examples**

```
df = pl$DataFrame(mtcars)

# Make the row-wise sum of all columns
df$with_columns(
  pl$fold(
    acc = pl$lit(0),
    lambda = \(acc, x) acc + x,
```

```

    exprs = pl$col("*")
  )$alias("mpg_drat_sum_folded")
)

```

---

pl_from_epoch	<i>Convert a Unix timestamp to date(time)</i>
---------------	---

---

## Description

Depending on the `time_unit` provided, this function will return a different dtype:

- `time_unit = "d"` returns `pl$Date`
- `time_unit = "s"` returns `pl$Datetime("us")` (`pl$Datetime`'s default)
- `time_unit = "ms"` returns `pl$Datetime("ms")`
- `time_unit = "us"` returns `pl$Datetime("us")`
- `time_unit = "ns"` returns `pl$Datetime("ns")`

## Usage

```
pl_from_epoch(column, time_unit = "s")
```

## Arguments

<code>column</code>	An Expr from which integers will be parsed. If this is a float column, then the decimal part of the float will be ignored. Character are parsed as column names, but other literal values must be passed to <code>pl\$lit()</code> .
<code>time_unit</code>	One of "ns", "us", "ms", "s", "d"

## Value

Expr as `Date` or `Datetime` depending on the `time_unit`.

## Examples

```

# pass an integer column
df = pl$DataFrame(timestamp = c(1666683077, 1666683099))
df$with_columns(
  timestamp_to_datetime = pl$from_epoch(pl$col("timestamp"), time_unit = "s")
)

# pass a literal
pl$from_epoch(pl$lit(c(1666683077, 1666683099)), time_unit = "s")$to_series()

# use different time_unit
df = pl$DataFrame(timestamp = c(12345, 12346))
df$with_columns(
  timestamp_to_date = pl$from_epoch(pl$col("timestamp"), time_unit = "d")
)

```

---

pl_head	<i>Get the first n rows.</i>
---------	------------------------------

---

**Description**

This function is syntactic sugar for `pl$col(...)$head(n)`.

**Usage**

```
pl_head(..., n = 10)
```

**Arguments**

...	Characters indicating the column names, passed to <code>pl\$col()</code> . See <code>?pl_col</code> for details.
n	Number of rows to return.

**Value**

`Expr`

**See Also**

- `<Expr>$head()`

**Examples**

```
df = pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

df$select(pl$head("a"))

df$select(pl$head("a", "b", n = 2))
```

---

pl_implode	<i>Aggregate all column values into a list.</i>
------------	---

---

**Description**

This function is syntactic sugar for `pl$col(...)$implode()`.

**Usage**

```
pl_implode(...)
```

**Arguments**

... Characters indicating the column names, passed to `pl$col()`. See `?pl_col` for details.

**Value**

Expr

**Examples**

```
pl$DataFrame(iris)$select(pl$implode("Species"))
```

---

pl_int_range	<i>Generate a range of integers</i>
--------------	-------------------------------------

---

**Description**

Generate a range of integers

**Usage**

```
pl_int_range(start = 0, end = NULL, step = 1, ..., dtype = pl$Int64)
```

**Arguments**

start	Start of the range (inclusive). Defaults to 0.
end	End of the range (exclusive). If NULL (default), the value of start is used and start is set to 0.
step	Step size of the range.
...	Not used.
dtype	Data type of the range.

**Value**

An Expr with the data type specified in dtype (default is Int64).

**See Also**

`pl$int_ranges()` to generate a range of integers for each row of the input columns.

**Examples**

```

pl$int_range(0, 3) |>
  as_polars_series()

# "end" can be omitted for shorter syntax
pl$int_range(3) |>
  as_polars_series()

# custom data type
pl$int_range(3, dtype = pl$Int16) |>
  as_polars_series()

# one can use pl$int_range() and pl$len() to create an index column
df = pl$DataFrame(a = c(1, 3, 5), b = c(2, 4, 6))
df$select(
  index = pl$int_range(pl$len(), dtype = pl$UInt32),
  pl$all()
)

```

---

pl\_int\_ranges

*Generate a range of integers for each row of the input columns*


---

**Description**

Generate a range of integers for each row of the input columns

**Usage**

```
pl_int_ranges(start = 0, end = NULL, step = 1, ..., dtype = pl$Int64)
```

**Arguments**

start	Start of the range (inclusive). Defaults to 0.
end	End of the range (exclusive). If NULL (default), the value of start is used and start is set to 0.
step	Step size of the range.
...	Not used.
dtype	Data type of the range.

**Value**

An Expr with the data type List(dtype) (with Int64 as default of dtype).

**See Also**

[pl\\$int\\_range\(\)](#) to generate a single range of integers.



**Examples**

```
df = pl$DataFrame(start = c(1, -1), end = c(3, 2))

df$with_columns(int_range = pl$int_ranges("start", "end"))

df$with_columns(int_range = pl$int_ranges("start", "end", dtype = pl$Int16))
```

---

pl_is_schema	<i>check if schema</i>
--------------	------------------------

---

**Description**

check if schema

**Usage**

```
pl_is_schema(x)
```

**Arguments**

x                    object to test if schema

**Value**

bool

**Examples**

```
pl$is_schema(pl$DataFrame(iris)$schema)
pl$is_schema(list("alice", "bob"))
```

---

pl_last	<i>Get the last value.</i>
---------	----------------------------

---

**Description**

This function has different behavior depending on the input type:

- Missing -> Takes last column of a context.
- Character vectors -> Syntactic sugar for `pl$col(...)$last()`.

**Usage**

```
pl_last(...)
```

**Arguments**

... Characters indicating the column names (passed to `pl$col()`, see `?pl_col` for details), or empty. If empty (default), returns an expression to take the last column of the context instead.

**Value**

`Expr`

**See Also**

- `<Expr>$last()`

**Examples**

```
df = pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "baz")
)

df$select(pl$last())

df$select(pl$last("a"))

df$select(pl$last(c("b", "c")))
```

---

pl\_LazyFrame

*Create new LazyFrame*

---

**Description**

This is simply a convenience function to create LazyFrames in a quick way. It is a wrapper around `pl$DataFrame()$lazy()`. Note that this should only be used for making examples and quick demonstrations.

**Usage**

```
pl_LazyFrame(...)
```

**Arguments**

... Anything that is accepted by `pl$DataFrame()`

**Value**

`LazyFrame`

**Examples**

```
pl$LazyFrame(  
  a = c(1, 2, 3, 4, 5),  
  b = 1:5,  
  c = letters[1:5],  
  d = list(1:1, 1:2, 1:3, 1:4, 1:5)  
) # directly from vectors  
  
# from a list of vectors or data.frame  
pl$LazyFrame(list(  
  a = c(1, 2, 3, 4, 5),  
  b = 1:5,  
  c = letters[1:5],  
  d = list(1L, 1:2, 1:3, 1:4, 1:5)  
)  
  
# custom schema  
pl$LazyFrame(  
  iris,  
  schema = list(Sepal.Length = pl$Float32, Species = pl$String)  
)$collect()
```

---

pl\_len

*Return the number of rows in the context.*

---

**Description**

This is similar to COUNT(\*) in SQL.

**Usage**

```
pl_len()
```

**Value**

[Expression](#) of data type [UInt32](#)

**See Also**

- [<Expr>\\$count\(\)](#)

**Examples**

```
df = pl$DataFrame(  
  a = c(1, 2, NA),  
  b = c(3, NA, NA),  
  c = c("foo", "bar", "foo")  
)  
  
df$select(pl_len())
```

---

pl_lit	<i>Create a literal value</i>
--------	-------------------------------

---

**Description**

Create a literal value

**Usage**

```
pl_lit(x)
```

**Arguments**

x                    A vector of any length

**Details**

pl\_lit(NULL) translates into a polars null.

**Value**

Expr

**Examples**

```
# values to literal, explicit `pl_lit(42)` implicit `+ 2`
pl$col("some_column") / pl_lit(42) + 2

# vector to literal explicitly via Series and back again
# R vector to expression and back again
pl$select(pl_lit(as_polars_series(1:4)))$to_list()[[1L]]

# r vector to literal and back r vector
pl_lit(1:4)$to_r()

# r vector to literal to dataframe
pl$select(pl_lit(1:4))

# r vector to literal to Series
pl_lit(1:4)$to_series()

# vectors to literal implicitly
(pl_lit(2) + 1:4) / 4:1
```

---

pl_max	<i>Get the maximum value.</i>
--------	-------------------------------

---

### Description

Syntactic sugar for `pl$col(...)$max()`.

### Usage

```
pl_max(...)
```

### Arguments

... Characters indicating the column names, passed to `pl$col()`. See `?pl_col` for details.

### Value

`Expr`

### See Also

- `<Expr>$max()`
- `pl$max_horizontal()`

### Examples

```
df = pl$DataFrame(  
  num_1 = c(1, 8, 3),  
  num_2 = c(4, 5, 2),  
  chr_1 = c("foo", "bar", "foo")  
)  
  
df$select(pl$max("num_1"))  
  
# Get the maximum value of multiple columns.  
df$select(pl$max(r"^num_\d+$"))  
  
df$select(pl$max("num_1", "num_2"))
```

---

pl\_max\_horizontal      *Get the maximum value rowwise*

---

### Description

Get the maximum value rowwise

### Usage

```
pl_max_horizontal(...)
```

### Arguments

...      Columns to concatenate into a single string column. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

### Value

Expr

### Examples

```
df = pl$DataFrame(
  a = NA_real_,
  b = c(2:1, NA_real_, NA_real_),
  c = c(1:2, NA_real_, Inf)
)
df$with_columns(
  pl$max_horizontal("a", "b", "c", 99.9)$alias("max")
)
```

---

pl\_mean      *Get the mean value.*

---

### Description

This function is syntactic sugar for `pl$col(...)$mean()`.

### Usage

```
pl_mean(...)
```

### Arguments

...      Characters indicating the column names, passed to `pl$col()`. See `?pl_col` for details.

**Value**

Expr

**See Also**

- `<Expr>$mean()`
- `pl$mean_horizontal()`

**Examples**

```
df = pl$DataFrame(  
  a = c(1, 8, 3),  
  b = c(4, 5, 2),  
  c = c("foo", "bar", "foo")  
)  
  
df$select(pl$mean("a"))  
  
df$select(pl$mean("a", "b"))
```

---

pl_mean_horizontal	<i>Compute the mean rowwise</i>
--------------------	---------------------------------

---

**Description**

Compute the mean rowwise

**Usage**

```
pl_mean_horizontal(...)
```

**Arguments**

... Columns to concatenate into a single string column. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(  
  a = c(1, 8, 3, 6, 7),  
  b = c(4, 5, NA_real_, Inf, NaN)  
)  
  
df$with_columns(  
  a = pl$mean_horizontal(a, b)  
)
```

```
pl$mean_horizontal("a", "b")$alias("mean"),
pl$mean_horizontal("a", "b", 5)$alias("mean_with_lit")
)
```

---

pl\_median

*Get the median value.*

---

### Description

This function is syntactic sugar for `pl$col(...)$median()`.

### Usage

```
pl_median(...)
```

### Arguments

... Characters indicating the column names, passed to `pl$col()`. See `?pl_col` for details.

### Value

Expr

### See Also

- `<Expr>$median()`

### Examples

```
df = pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

df$select(pl$median("a"))

df$select(pl$median("a", "b"))
```



---

pl\_mem\_address      *Get Memory Address*

---

**Description**

Get underlying mem address a rust object (via ExtPtr). Expert use only.

**Usage**

```
pl_mem_address(robj)
```

**Arguments**

robj              an R object

**Details**

Does not give meaningful answers for regular R objects.

**Value**

String of mem address

**Examples**

```
pl$mem_address(pl$Series(values = 1:3))
```

---

pl\_min              *Get the minimum value.*

---

**Description**

Syntactic sugar for `pl$col(...)$min()`.

**Usage**

```
pl_min(...)
```

**Arguments**

...              Characters indicating the column names, passed to `pl$col()`. See [?pl\\_col](#) for details.

**Value**

[Expr](#)

**See Also**

- `<Expr>$min()`
- `pl$min_horizontal()`

**Examples**

```
df = pl$DataFrame(
  num_1 = c(1, 8, 3),
  num_2 = c(4, 5, 2),
  chr_1 = c("foo", "bar", "foo")
)

df$select(pl$min("num_1"))

# Get the minimum value of multiple columns.
df$select(pl$min(r"^num_\d+$"))

df$select(pl$min("num_1", "num_2"))
```

---

pl\_min\_horizontal      *Get the minimum value rowwise*

---

**Description**

Get the minimum value rowwise

**Usage**

```
pl_min_horizontal(...)
```

**Arguments**

...                      Columns to concatenate into a single string column. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = NA_real_,
  b = c(2:1, NA_real_, NA_real_),
  c = c(1:2, NA_real_, -Inf)
)
df$with_columns(
  pl$min_horizontal("a", "b", "c", 99.9)$alias("min")
)
```

---

pl_n_unique	<i>Count unique values.</i>
-------------	-----------------------------

---

### Description

This function is syntactic sugar for `pl$col(...)$n_unique()`.

### Usage

```
pl_n_unique(...)
```

### Arguments

... Characters indicating the column names, passed to `pl$col()`. See `?pl_col` for details.

### Value

`Expr`

### See Also

- `<Expr>$n_unique()`

### Examples

```
df = pl$DataFrame(  
  a = c(1, 8, 1),  
  b = c(4, 5, 2),  
  c = c("foo", "bar", "foo")  
)  
  
df$select(pl$n_unique("a"))  
  
df$select(pl$n_unique("b", "c"))
```

---

pl_pl	<i>The complete polars public API.</i>
-------	--

---

### Description

pl-object is a environment of all public functions and class constructors. Public functions are not exported as a normal package as it would be huge namespace collision with `base::` and other functions. All object-methods are accessed with `object$method()` via the new class functions.

Having all functions in an namespace is similar to the rust- and python- polars api.

**Usage**

```
pl
```

**Format**

An object of class `pl_polars_env` (inherits from `environment`) of length 109.

**Details**

If someone do not particularly like the letter combination `pl`, they are free to bind the environment to another variable name as `simon_says = pl` or even do `attach(pl)`

**Value**

not applicable

**Examples**

```
# how to use polars via `pl`
pl$col("colname")$sum() / pl$lit(42L) # expression ~ chain-method / literal-expression

# show all public functions, RPolarsDataTypes, classes and methods
pl$show_all_public_functions()
pl$show_all_public_methods()
```

---

pl\_PTime

*Store Time in R*

---

**Description**

Store Time in R

**Usage**

```
pl_PTime(x, tu = c("s", "ms", "us", "ns"), format = "%H:%M:%S")
```

**Arguments**

<code>x</code>	an integer or double vector of <code>n</code> epochs since midnight OR a char vector of char times passed to <code>as.POSIXct</code> converted to seconds.
<code>tu</code>	timeunit either "s","ms","us","ns"
<code>format</code>	a format string passed to <code>as.POSIXct</code> format via ...

## Details

PTime should probably be replaced with package nanotime or similar.

base R is missing encoding of Time since midnight "s" "ms", "us" and "ns". The latter "ns" is the standard for the polars Time type.

Use PTime to convert R doubles and integers and use as input to polars functions which needs a time.

Loosely inspired by data.table::ITime which is i32 only. PTime must support polars native timeunit is nanoseconds. The R double(float64) can imitate a i64 ns with full precision within the full range of 24 hours.

PTime does not have a time zone and always prints the time as is no matter local machine time zone.

An essential difference between R and polars is R prints POSIXct/lm without a timezone in local time. Polars prints Datetime without a timezone label as is (GMT). For POSIXct/lm tagged with a timexone(tzone) and Datetime with a timezone(tz) the behavior is the same conversion is intuitive.

It appears behavior of R timezones is subject to change a bit in R 4.3.0, see polars unit test test-expr\_datetime.R/"pl\$date\_range Date lazy/eager".

## Value

a PTime vector either double or integer, with class "PTime" and attribute "tu" being either "s", "ms", "us" or "ns"

## Examples

```
# make PTime in all time units
pl$PTime(runif(5) * 3600 * 24 * 1E0, tu = "s")
pl$PTime(runif(5) * 3600 * 24 * 1E3, tu = "ms")
pl$PTime(runif(5) * 3600 * 24 * 1E6, tu = "us")
pl$PTime(runif(5) * 3600 * 24 * 1E9, tu = "ns")
pl$PTime("23:59:59")

as_polars_series(pl$PTime(runif(5) * 3600 * 24 * 1E0, tu = "s"))
pl$lit(pl$PTime("23:59:59"))$to_series()

pl$lit(pl$PTime("23:59:59"))$to_r()
```

---

pl\_raw\_list

*Polars raw list*

---

## Description

Create an "rpolars\_raw\_list", which is an R list where all elements must be an R row or NULL.

**Usage**

```
pl_raw_list(...)

## S3 method for class 'rpolars_raw_list'
x[index]

## S3 method for class 'rpolars_raw_list'
as.list(x, ...)
```

**Arguments**

...	Elements
x	A rpolars_raw_list object created with pl\$raw_list()
index	Elements to select

**Details**

In R, raw can contain a binary sequence of bytes, and the length is the number of bytes. In polars a Series of DataType [Binary](#) is more like a vector of vectors of bytes where missing values are allowed, similar to how NAs can be present in vectors.

To ensure correct round-trip conversion, r-polars uses an R list where any elements must be raw or NULL (encoded as missing), and the S3 class is c("rpolars\_raw\_list", "list").

**Value**

An R list where any elements must be raw, and the S3 class is c("rpolars\_raw\_list", "list").

**Examples**

```
# create a rpolars_raw_list
raw_list = pl$raw_list(raw(1), raw(3), charToRaw("alice"), NULL)

# pass it to Series or lit
pl$Series(values = raw_list)
pl$lit(raw_list)

# convert polars binary Series to rpolars_raw_list
pl$Series(values = raw_list)$to_r()

# NB: a plain list of raws yield a polars Series of DataType [list[Binary]]
# which is not the same
pl$Series(values = list(raw(1), raw(2)))

# to regular list, use as.list or unclass
as.list(raw_list)
# subsetting preserves class
pl$raw_list(NULL, raw(2), raw(3))[1:2]
# to regular list, use as.list or unclass
pl$raw_list(NULL, raw(2), raw(3)) |> as.list()
```

---

pl\_read\_csv

*New DataFrame from CSV*


---

## Description

New DataFrame from CSV

## Usage

```
pl_read_csv(
    source,
    ...,
    has_header = TRUE,
    separator = ",",
    comment_prefix = NULL,
    quote_char = "\"",
    skip_rows = 0,
    dtypes = NULL,
    null_values = NULL,
    ignore_errors = FALSE,
    cache = FALSE,
    infer_schema_length = 100,
    n_rows = NULL,
    encoding = "utf8",
    low_memory = FALSE,
    rechunk = TRUE,
    skip_rows_after_header = 0,
    row_index_name = NULL,
    row_index_offset = 0,
    try_parse_dates = FALSE,
    eol_char = "\n",
    raise_if_empty = TRUE,
    truncate_ragged_lines = FALSE,
    reuse_downloaded = TRUE,
    include_file_paths = NULL
)
```

## Arguments

source	Path to a file or URL. It is possible to provide multiple paths provided that all CSV files have the same schema. It is not possible to provide several URLs.
...	Ignored.
has_header	Indicate if the first row of dataset is a header or not. If FALSE, column names will be autogenerated in the following format: "column_x" x being an enumeration over every column in the dataset starting at 1.
separator	Single byte character to use as separator in the file.

comment_prefix	A string, which can be up to 5 symbols in length, used to indicate the start of a comment line. For instance, it can be set to # or //.
quote_char	Single byte character used for quoting. Set to NULL to turn off special handling and escaping of quotes.
skip_rows	Start reading after a particular number of rows. The header will be parsed at this offset.
dtypes	Named list of column names - dtypes or dtype - column names. This list is used while reading to overwrite dtypes. Supported types so far are: <ul style="list-style-type: none"> <li>• "Boolean" or "logical" for <code>DataType::Boolean</code>,</li> <li>• "Categorical" or "factor" for <code>DataType::Categorical</code>,</li> <li>• "Float32" or "double" for <code>DataType::Float32</code>,</li> <li>• "Float64" or "float64" for <code>DataType::Float64</code>,</li> <li>• "Int32" or "integer" for <code>DataType::Int32</code>,</li> <li>• "Int64" or "integer64" for <code>DataType::Int64</code>,</li> <li>• "String" or "character" for <code>DataType::String</code>,</li> </ul>
null_values	Values to interpret as NA values. Can be: <ul style="list-style-type: none"> <li>• a character vector: all values that match one of the values in this vector will be NA;</li> <li>• a named list with column names and null values.</li> </ul>
ignore_errors	Keep reading the file even if some lines yield errors. You can also use <code>infer_schema_length = 0</code> to read all columns as UTF8 to check which values might cause an issue.
cache	Cache the result after reading.
infer_schema_length	Maximum number of rows to read to infer the column types. If set to 0, all columns will be read as UTF-8. If NULL, a full table scan will be done (slow).
n_rows	Maximum number of rows to read.
encoding	Either "utf8" or "utf8-lossy". Lossy means that invalid UTF8 values are replaced with "?" characters.
low_memory	Reduce memory usage (will yield a lower performance).
rechunk	Reallocate to contiguous memory when all chunks / files are parsed.
skip_rows_after_header	Parse the first row as headers, and then skip this number of rows.
row_index_name	If not NULL, this will insert a row index column with the given name into the DataFrame.
row_index_offset	Offset to start the row index column (only used if the name is set).
try_parse_dates	Try to automatically parse dates. Most ISO8601-like formats can be inferred, as well as a handful of others. If this does not succeed, the column remains of data type <code>pl\$String</code> .
eol_char	Single byte end of line character (default: <code>\n</code> ). When encountering a file with Windows line endings ( <code>\r\n</code> ), one can go with the default <code>\n</code> . The extra <code>\r</code> will be removed when processed.



**raise\_if\_empty** If FALSE, parsing an empty file returns an empty DataFrame or LazyFrame.  
**truncate\_ragged\_lines** Truncate lines that are longer than the schema.  
**reuse\_downloaded** If TRUE(default) and a URL was provided, cache the downloaded files in session for an easy reuse.  
**include\_file\_paths** Include the path of the source file(s) as a column with this name.

**Value**

[DataFrame](#)

---

pl_read_ipc	<i>Read into a DataFrame from Arrow IPC (Feather v2) file</i>
-------------	---

---

**Description**

Read into a DataFrame from Arrow IPC (Feather v2) file

**Usage**

```
pl_read_ipc(
  source,
  ...,
  n_rows = NULL,
  memory_map = TRUE,
  row_index_name = NULL,
  row_index_offset = 0L,
  rechunk = FALSE,
  cache = TRUE
)
```

**Arguments**

source	A single character or a raw vector of Apache Arrow IPC file. You can use globbing with * to scan/read multiple files in the same directory (see examples).
...	Ignored.
n_rows	Maximum number of rows to read.
memory_map	A logical. If TRUE, try to memory map the file. This can greatly improve performance on repeated queries as the OS may cache pages. Only uncompressed Arrow IPC files can be memory mapped.
row_index_name	If not NULL, this will insert a row index column with the given name into the DataFrame.
row_index_offset	Offset to start the row index column (only used if the name is set).

rechunk	In case of reading multiple files via a glob pattern, rechunk the final DataFrame into contiguous memory chunks.
cache	Cache the result after reading.

**Value**

[DataFrame](#)

**Examples**

```
temp_dir = tempfile()
# Write a hive-style partitioned arrow file dataset
arrow::write_dataset(
  mtcars,
  temp_dir,
  partitioning = c("cyl", "gear"),
  format = "arrow",
  hive_style = TRUE
)
list.files(temp_dir, recursive = TRUE)

# Read the dataset
# Since hive-style partitioning is not supported,
# the `cyl` and `gear` columns are not contained in the result
pl$read_ipc(
  file.path(temp_dir, "**/*.arrow")
)

# Read a raw vector
arrow::arrow_table(
  foo = 1:5,
  bar = 6:10,
  ham = letters[1:5]
) |>
  arrow::write_to_raw(format = "file") |>
  pl$read_ipc()
```

---

pl\_read\_ndjson

*New DataFrame from NDJSON*

---

**Description**

Read a file from path into a polars DataFrame.

**Usage**

```
pl_read_ndjson(
  source,
  ...,
  infer_schema_length = 100,
  batch_size = NULL,
  n_rows = NULL,
  low_memory = FALSE,
  rechunk = FALSE,
  row_index_name = NULL,
  row_index_offset = 0,
  ignore_errors = FALSE
)
```

**Arguments**

source	Path to a file or URL. It is possible to provide multiple paths provided that all NDJSON files have the same schema. It is not possible to provide several URLs.
...	Ignored.
infer_schema_length	Maximum number of rows to read to infer the column types. If set to 0, all columns will be read as UTF-8. If NULL, a full table scan will be done (slow).
batch_size	Number of rows that will be processed per thread.
n_rows	Maximum number of rows to read.
low_memory	Reduce memory usage (will yield a lower performance).
rechunk	Reallocate to contiguous memory when all chunks / files are parsed.
row_index_name	If not NULL, this will insert a row index column with the given name into the DataFrame.
row_index_offset	Offset to start the row index column (only used if the name is set).
ignore_errors	Keep reading the file even if some lines yield errors. You can also use <code>infer_schema_length = 0</code> to read all columns as UTF8 to check which values might cause an issue.

**Value**

A DataFrame

**Examples**

```
if (require("jsonlite", quietly = TRUE)) {
  ndjson_filename = tempfile()
  jsonlite::stream_out(iris, file(ndjson_filename), verbose = FALSE)
  pl$read_ndjson(ndjson_filename)
}
```

---

pl\_read\_parquet      *Read a parquet file*

---

### Description

Read a parquet file

### Usage

```
pl_read_parquet(
  source,
  ...,
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  parallel = c("auto", "columns", "row_groups", "none"),
  hive_partitioning = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  glob = TRUE,
  rechunk = TRUE,
  low_memory = FALSE,
  storage_options = NULL,
  use_statistics = TRUE,
  cache = TRUE,
  include_file_paths = NULL
)
```

### Arguments

source	Path to a file. You can use globbing with * to scan/read multiple files in the same directory (see examples).
...	Ignored.
n_rows	Maximum number of rows to read.
row_index_name	If not NULL, this will insert a row index column with the given name into the DataFrame.
row_index_offset	Offset to start the row index column (only used if the name is set).
parallel	This determines the direction of parallelism. "auto" will try to determine the optimal direction. Can be "auto", "columns", "row_groups", "prefiltered", or "none". See 'Details'.
hive_partitioning	Infer statistics and schema from Hive partitioned URL and use them to prune reads. If NULL (default), it is automatically enabled when a single directory is passed, and otherwise disabled.

hive_schema	A list containing the column names and data types of the columns by which the data is partitioned, e.g. <code>list(a = pl\$String, b = pl\$Float32)</code> . If NULL (default), the schema of the Hive partitions is inferred.
try_parse_hive_dates	Whether to try parsing hive values as date/datetime types.
glob	Expand path given via globbing rules.
rechunk	In case of reading multiple files via a glob pattern, rechunk the final DataFrame into contiguous memory chunks.
low_memory	Reduce memory usage (will yield a lower performance).
storage_options	Experimental. List of options necessary to scan parquet files from different cloud storage providers (GCP, AWS, Azure, HuggingFace). See the 'Details' section.
use_statistics	Use statistics in the parquet file to determine if pages can be skipped from reading.
cache	Cache the result after reading.
include_file_paths	Include the path of the source file(s) as a column with this name.

## Details

### On parallel strategies:

The prefiltered strategy first evaluates the pushed-down predicates in parallel and determines a mask of which rows to read. Then, it parallelizes over both the columns and the row groups while filtering out rows that do not need to be read. This can provide significant speedups for large files (i.e. many row-groups) with a predicate that filters clustered rows or filters heavily. In other cases, prefiltered may slow down the scan compared other strategies.

The prefiltered settings falls back to auto if no predicate is given.

### Connecting to cloud providers:

Polars supports scanning parquet files from different cloud providers. The cloud providers currently supported are AWS, GCP, and Azure. The supported keys to pass to the `storage_options` argument can be found here:

- `aws`
- `gcp`
- `azure`

Currently it is impossible to scan public parquet files from GCP without a valid service account. Be sure to always include a service account in the `storage_options` argument.

### Scanning from HuggingFace:

It is possible to scan data stored on HuggingFace using a path starting with `hf://`. The `hf://` path format is defined as `hf://BUCKET/REPOSITORY@REVISION/PATH`, where:

- BUCKET is one of datasets or spaces
- REPOSITORY is the location of the repository. this is usually in the format of `username/repo_name`. A branch can also be optionally specified by appending `@branch`.

- REVISION is the name of the branch (or commit) to use. This is optional and defaults to main if not given.
- PATH is a file or directory path, or a glob pattern from the repository root.

A Hugging Face API key can be passed to access private locations using either of the following methods:

- Passing a token in storage\_options to the scan function, e.g. `scan_parquet(..., storage_options = list(token = "token"))`.
- Setting the HF\_TOKEN environment variable, e.g. `Sys.setenv(HF_TOKEN = <your HF token>)`.

## Value

[DataFrame](#)

## Examples

```
# Write a Parquet file than we can then import as DataFrame
temp_file = withr::local_tempfile(fileext = ".parquet")
pl$DataFrame(mtcars)$write_parquet(temp_file)

pl$read_parquet(temp_file)

# Write a hive-style partitioned parquet dataset
temp_dir = withr::local_tempdir()
pl$DataFrame(mtcars)$write_parquet(temp_dir, partition_by = c("cyl", "gear"))
list.files(temp_dir, recursive = TRUE)

# If the path is a folder, Polars automatically tries to detect partitions
# and includes them in the output
pl$read_parquet(temp_dir)
```

---

pl\_reduce

*Accumulate over multiple columns horizontally with an R function*

---

## Description

This allows one to do rowwise operations. See `pl$fold()` to do rowwise operations with an initial value.

## Usage

```
pl_reduce(lambda, exprs)
```

## Arguments

lambda	R function which takes two polars Series as input and return one.
exprs	Expressions to aggregate over. May also be a wildcard expression.

**Value**

An expression that will be applied rowwise

**Examples**

```
df = pl$DataFrame(mtcars)

# Make the row-wise sum of all columns
df$with_columns(
  pl$reduce(
    lambda = \(acc, x) acc + x,
    exprs = pl$col("*")
  )$alias("mpg_drat_sum_reduced")
)
```

---

pl_rolling_corr	<i>Rolling correlation</i>
-----------------	----------------------------

---

**Description**

Calculates the rolling correlation between two columns

**Usage**

```
pl_rolling_corr(a, b, window_size, min_periods = NULL, ddof = 1)
```

**Arguments**

a	One column name or Expr or anything convertible Into via pl\$col().
b	Another column name or Expr or anything convertible Into via pl\$col().
window_size	int The length of the window
min_periods	NULL or int The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
ddof	integer Delta Degrees of Freedom: the divisor used in the calculation is N - ddof, where N represents the number of elements. By default ddof is 1.

**Value**

Expr for the computed rolling correlation

**Examples**

```
lf = pl$LazyFrame(data.frame(a = c(1, 8, 3), b = c(4, 5, 2)))
lf$select(pl$rolling_corr("a", "b", window_size = 2))$collect()
```

---

pl_rolling_cov	<i>Rolling covariance</i>
----------------	---------------------------

---

**Description**

Calculates the rolling covariance between two columns

**Usage**

```
pl_rolling_cov(a, b, window_size, min_periods = NULL, ddof = 1)
```

**Arguments**

a	One column name or Expr or anything convertible Into via pl\$col().
b	Another column name or Expr or anything convertible Into via pl\$col().
window_size	int The length of the window
min_periods	NULL or int The number of values in the window that should be non-null before computing a result. If NULL, it will be set equal to window size.
ddof	integer Delta Degrees of Freedom: the divisor used in the calculation is N - ddof, where N represents the number of elements. By default ddof is 1.

**Value**

Expr for the computed rolling covariance

**Examples**

```
lf = pl$LazyFrame(data.frame(a = c(1, 8, 3), b = c(4, 5, 2)))
lf$select(pl$rolling_cov("a", "b", window_size = 2))$collect()
```

---

pl_scan_csv	<i>New LazyFrame from CSV</i>
-------------	-------------------------------

---

**Description**

Read a file from path into a polars LazyFrame.



**Usage**

```

pl_scan_csv(
  source,
  ...,
  has_header = TRUE,
  separator = ",",
  comment_prefix = NULL,
  quote_char = "\"",
  skip_rows = 0,
  dtypes = NULL,
  null_values = NULL,
  ignore_errors = FALSE,
  cache = FALSE,
  infer_schema_length = 100,
  n_rows = NULL,
  encoding = "utf8",
  low_memory = FALSE,
  rechunk = TRUE,
  skip_rows_after_header = 0,
  row_index_name = NULL,
  row_index_offset = 0,
  try_parse_dates = FALSE,
  eol_char = "\n",
  raise_if_empty = TRUE,
  truncate_ragged_lines = FALSE,
  reuse_downloaded = TRUE,
  include_file_paths = NULL
)

```

**Arguments**

source	Path to a file or URL. It is possible to provide multiple paths provided that all CSV files have the same schema. It is not possible to provide several URLs.
...	Ignored.
has_header	Indicate if the first row of dataset is a header or not. If FALSE, column names will be autogenerated in the following format: "column_x" x being an enumeration over every column in the dataset starting at 1.
separator	Single byte character to use as separator in the file.
comment_prefix	A string, which can be up to 5 symbols in length, used to indicate the start of a comment line. For instance, it can be set to # or //.
quote_char	Single byte character used for quoting. Set to NULL to turn off special handling and escaping of quotes.
skip_rows	Start reading after a particular number of rows. The header will be parsed at this offset.
dtypes	Named list of column names - dtypes or dtype - column names. This list is used while reading to overwrite dtypes. Supported types so far are:

	<ul style="list-style-type: none"> <li>• "Boolean" or "logical" for <code>DataType::Boolean</code>,</li> <li>• "Categorical" or "factor" for <code>DataType::Categorical</code>,</li> <li>• "Float32" or "double" for <code>DataType::Float32</code>,</li> <li>• "Float64" or "float64" for <code>DataType::Float64</code>,</li> <li>• "Int32" or "integer" for <code>DataType::Int32</code>,</li> <li>• "Int64" or "integer64" for <code>DataType::Int64</code>,</li> <li>• "String" or "character" for <code>DataType::String</code>,</li> </ul>
<code>null_values</code>	<p>Values to interpret as NA values. Can be:</p> <ul style="list-style-type: none"> <li>• a character vector: all values that match one of the values in this vector will be NA;</li> <li>• a named list with column names and null values.</li> </ul>
<code>ignore_errors</code>	Keep reading the file even if some lines yield errors. You can also use <code>infer_schema_length = 0</code> to read all columns as UTF8 to check which values might cause an issue.
<code>cache</code>	Cache the result after reading.
<code>infer_schema_length</code>	Maximum number of rows to read to infer the column types. If set to 0, all columns will be read as UTF-8. If NULL, a full table scan will be done (slow).
<code>n_rows</code>	Maximum number of rows to read.
<code>encoding</code>	Either "utf8" or "utf8-lossy". Lossy means that invalid UTF8 values are replaced with "?" characters.
<code>low_memory</code>	Reduce memory usage (will yield a lower performance).
<code>rechunk</code>	Reallocate to contiguous memory when all chunks / files are parsed.
<code>skip_rows_after_header</code>	Parse the first row as headers, and then skip this number of rows.
<code>row_index_name</code>	If not NULL, this will insert a row index column with the given name into the DataFrame.
<code>row_index_offset</code>	Offset to start the row index column (only used if the name is set).
<code>try_parse_dates</code>	Try to automatically parse dates. Most ISO8601-like formats can be inferred, as well as a handful of others. If this does not succeed, the column remains of data type <code>pl\$String</code> .
<code>eol_char</code>	Single byte end of line character (default: <code>\n</code> ). When encountering a file with Windows line endings ( <code>\r\n</code> ), one can go with the default <code>\n</code> . The extra <code>\r</code> will be removed when processed.
<code>raise_if_empty</code>	If FALSE, parsing an empty file returns an empty DataFrame or LazyFrame.
<code>truncate_ragged_lines</code>	Truncate lines that are longer than the schema.
<code>reuse_downloaded</code>	If TRUE(default) and a URL was provided, cache the downloaded files in session for an easy reuse.
<code>include_file_paths</code>	Include the path of the source file(s) as a column with this name.

**Value**[LazyFrame](#)**Examples**

```
my_file = tempfile()
write.csv(iris, my_file)
lazy_frame = pl$scan_csv(my_file)
lazy_frame$collect()
unlink(my_file)
```

---

pl_scan_ipc	<i>Lazily read from an Arrow IPC (Feather v2) file or multiple files via glob patterns</i>
-------------	--

---

**Description**

This allows the query optimizer to push down predicates and projections to the scan level, thereby potentially reducing memory overhead.

**Usage**

```
pl_scan_ipc(
  source,
  ...,
  n_rows = NULL,
  memory_map = TRUE,
  row_index_name = NULL,
  row_index_offset = 0L,
  rechunk = FALSE,
  cache = TRUE,
  hive_partitioning = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  include_file_paths = NULL
)
```

**Arguments**

source	Path to a file. You can use globbing with * to scan/read multiple files in the same directory (see examples).
...	Ignored.
n_rows	Maximum number of rows to read.
memory_map	A logical. If TRUE, try to memory map the file. This can greatly improve performance on repeated queries as the OS may cache pages. Only uncompressed Arrow IPC files can be memory mapped.

row_index_name	If not NULL, this will insert a row index column with the given name into the DataFrame.
row_index_offset	Offset to start the row index column (only used if the name is set).
rechunk	In case of reading multiple files via a glob pattern, rechunk the final DataFrame into contiguous memory chunks.
cache	Cache the result after reading.
hive_partitioning	Infer statistics and schema from Hive partitioned URL and use them to prune reads. If NULL (default), it is automatically enabled when a single directory is passed, and otherwise disabled.
hive_schema	A list containing the column names and data types of the columns by which the data is partitioned, e.g. <code>list(a = pl\$String, b = pl\$Float32)</code> . If NULL (default), the schema of the Hive partitions is inferred.
try_parse_hive_dates	Whether to try parsing hive values as date/datetime types.
include_file_paths	Character value indicating the column name that will include the path of the source file(s).

### Details

Hive-style partitioning is not supported yet.

### Value

[LazyFrame](#)

### Examples

```
temp_dir = tempfile()
# Write a hive-style partitioned arrow file dataset
arrow::write_dataset(
  mtcars,
  temp_dir,
  partitioning = c("cyl", "gear"),
  format = "arrow",
  hive_style = TRUE
)
list.files(temp_dir, recursive = TRUE)

# If the path is a folder, Polars automatically tries to detect partitions
# and includes them in the output
pl$scan_ipc(temp_dir)$collect()

# We can also impose a schema to the partition
pl$scan_ipc(temp_dir, hive_schema = list(cyl = pl$String, gear = pl$Int32))$collect()
```

---

pl\_scan\_ndjson      *New LazyFrame from NDJSON*

---

## Description

Read a file from path into a polars LazyFrame.

## Usage

```
pl_scan_ndjson(
    source,
    ...,
    infer_schema_length = 100,
    batch_size = NULL,
    n_rows = NULL,
    low_memory = FALSE,
    rechunk = FALSE,
    row_index_name = NULL,
    row_index_offset = 0,
    reuse_downloaded = TRUE,
    ignore_errors = FALSE
)
```

## Arguments

source	Path to a file or URL. It is possible to provide multiple paths provided that all NDJSON files have the same schema. It is not possible to provide several URLs.
...	Ignored.
infer_schema_length	Maximum number of rows to read to infer the column types. If set to 0, all columns will be read as UTF-8. If NULL, a full table scan will be done (slow).
batch_size	Number of rows that will be processed per thread.
n_rows	Maximum number of rows to read.
low_memory	Reduce memory usage (will yield a lower performance).
rechunk	Reallocate to contiguous memory when all chunks / files are parsed.
row_index_name	If not NULL, this will insert a row index column with the given name into the DataFrame.
row_index_offset	Offset to start the row index column (only used if the name is set).
reuse_downloaded	If TRUE(default) and a URL was provided, cache the downloaded files in session for an easy reuse.
ignore_errors	Keep reading the file even if some lines yield errors. You can also use infer_schema_length = 0 to read all columns as UTF8 to check which values might cause an issue.

**Value**

A LazyFrame

**Examples**

```
if (require("jsonlite", quietly = TRUE)) {
  ndjson_filename = tempfile()
  jsonlite::stream_out(iris, file(ndjson_filename), verbose = FALSE)
  pl$scan_ndjson(ndjson_filename)$collect()
}
```

---

pl_scan_parquet	<i>Scan a parquet file</i>
-----------------	----------------------------

---

**Description**

Scan a parquet file

**Usage**

```
pl_scan_parquet(
  source,
  ...,
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  parallel = c("auto", "columns", "row_groups", "none"),
  hive_partitioning = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  glob = TRUE,
  rechunk = FALSE,
  low_memory = FALSE,
  storage_options = NULL,
  use_statistics = TRUE,
  cache = TRUE,
  include_file_paths = NULL
)
```

**Arguments**

source	Path to a file. You can use globbing with * to scan/read multiple files in the same directory (see examples).
...	Ignored.
n_rows	Maximum number of rows to read.
row_index_name	If not NULL, this will insert a row index column with the given name into the DataFrame.

row_index_offset	Offset to start the row index column (only used if the name is set).
parallel	This determines the direction of parallelism. "auto" will try to determine the optimal direction. Can be "auto", "columns", "row_groups", "prefiltered", or "none". See 'Details'.
hive_partitioning	Infer statistics and schema from Hive partitioned URL and use them to prune reads. If NULL (default), it is automatically enabled when a single directory is passed, and otherwise disabled.
hive_schema	A list containing the column names and data types of the columns by which the data is partitioned, e.g. <code>list(a = pl\$String, b = pl\$Float32)</code> . If NULL (default), the schema of the Hive partitions is inferred.
try_parse_hive_dates	Whether to try parsing hive values as date/datetime types.
glob	Expand path given via globbing rules.
rechunk	In case of reading multiple files via a glob pattern, rechunk the final DataFrame into contiguous memory chunks.
low_memory	Reduce memory usage (will yield a lower performance).
storage_options	Experimental. List of options necessary to scan parquet files from different cloud storage providers (GCP, AWS, Azure, HuggingFace). See the 'Details' section.
use_statistics	Use statistics in the parquet file to determine if pages can be skipped from reading.
cache	Cache the result after reading.
include_file_paths	Include the path of the source file(s) as a column with this name.

## Details

### On parallel strategies:

The prefiltered strategy first evaluates the pushed-down predicates in parallel and determines a mask of which rows to read. Then, it parallelizes over both the columns and the row groups while filtering out rows that do not need to be read. This can provide significant speedups for large files (i.e. many row-groups) with a predicate that filters clustered rows or filters heavily. In other cases, prefiltered may slow down the scan compared other strategies.

The prefiltered settings falls back to auto if no predicate is given.

### Connecting to cloud providers:

Polars supports scanning parquet files from different cloud providers. The cloud providers currently supported are AWS, GCP, and Azure. The supported keys to pass to the `storage_options` argument can be found here:

- [aws](#)
- [gcp](#)
- [azure](#)

Currently it is impossible to scan public parquet files from GCP without a valid service account. Be sure to always include a service account in the `storage_options` argument.

### Scanning from HuggingFace:

It is possible to scan data stored on HuggingFace using a path starting with `hf://`. The `hf://` path format is defined as `hf://BUCKET/REPOSITORY@REVISION/PATH`, where:

- `BUCKET` is one of datasets or spaces
- `REPOSITORY` is the location of the repository. this is usually in the format of `username/repo_name`. A branch can also be optionally specified by appending `@branch`.
- `REVISION` is the name of the branch (or commit) to use. This is optional and defaults to `main` if not given.
- `PATH` is a file or directory path, or a glob pattern from the repository root.

A Hugging Face API key can be passed to access private locations using either of the following methods:

- Passing a token in `storage_options` to the scan function, e.g. `scan_parquet(..., storage_options = list(token = ...))`
- Setting the `HF_TOKEN` environment variable, e.g. `Sys.setenv(HF_TOKEN = <your HF token>)`.

## Value

[LazyFrame](#)

## Examples

```
# Write a Parquet file than we can then import as DataFrame
temp_file = withr::local_tempfile(fileext = ".parquet")
pl$DataFrame(mtcars)$write_parquet(temp_file)

pl$scan_parquet(temp_file)$collect()

# Write a hive-style partitioned parquet dataset
temp_dir = withr::local_tempdir()
pl$DataFrame(mtcars)$write_parquet(temp_dir, partition_by = c("cyl", "gear"))
list.files(temp_dir, recursive = TRUE)

# If the path is a folder, Polars automatically tries to detect partitions
# and includes them in the output
pl$scan_parquet(temp_dir)$collect()
```

---

pl\_select

*Select from an empty DataFrame*

---

## Description

`pl$select(...)` is a shorthand for `pl$DataFrame(list())$select(...)`



**Usage**

```
pl_select(...)
```

**Arguments**

```
... Expressions
```

**Value**

a [DataFrame](#)

**Examples**

```
pl$select(
  pl$lit(1:4)$alias("ints"),
  pl$lit(letters[1:4])$alias("letters")
)
```

---

pl\_Series

*Create new Series*

---

**Description**

This function is a simple way to convert R vectors to [the Series class object](#). Internally, this function is a simple wrapper of [as\\_polars\\_series\(\)](#).

**Usage**

```
pl_Series(
  name = NULL,
  values = NULL,
  dtype = NULL,
  ...,
  strict = TRUE,
  nan_to_null = FALSE
)
```

**Arguments**

name	A character to use as the name of the Series, or NULL (default). Passed to the name argument in <a href="#">as_polars_series()</a> .
values	Object to convert into a polars Series. Passed to the x argument in <a href="#">as_polars_series()</a> .
dtype	One of <a href="#">polars data type</a> or NULL. If not NULL, that data type is used to <a href="#">cast</a> the Series created from the vector to a specific data type internally.
...	Ignored.

strict	A logical. If TRUE (default), throw an error if any value does not exactly match the given data type by the dtype argument. If FALSE, values that do not match the data type are cast to that data type or, if casting is not possible, set to null instead. Passed to the strict argument of the <code>\$cast()</code> method internally.
nan_to_null	If TRUE, NaN values contained in the Series are replaced to null. Using the <code>\$fill_nan()</code> method internally.

### Details

Python Polars has a feature that automatically interprets something like `polars.Series([1])` as `polars.Series(values=[1])` if you specify Array like objects as the first argument. This feature is not available in R Polars, so something like `pl$Series(1)` will raise an error. You should use `pl$Series(values = 1)` or `as_polars_series(1)` instead.

### Value

[Series](#)

### See Also

- [as\\_polars\\_series\(\)](#)

### Examples

```
# Constructing a Series by specifying name and values positionally:
s = pl$Series("a", 1:3)
s

# Notice that the dtype is automatically inferred as a polars Int32:
s$dtype

# Constructing a Series with a specific dtype:
s2 = pl$Series(values = 1:3, name = "a", dtype = pl$Float32)
s2
```

---

pl\_SQLContext

*Initialise a new SQLContext*

---

### Description

Create a new SQLContext and register the given LazyFrames.

### Usage

```
pl_SQLContext(...)
```

### Arguments

... Name-value pairs of [LazyFrame](#) like objects to register.

**Value**

An [SQLContext](#)

**Examples**

```
ctx = pl$SQLContext(mtcars = mtcars)
ctx
```

---

pl_std	<i>Get the standard deviation.</i>
--------	------------------------------------

---

**Description**

This function is syntactic sugar for `pl$col(...)$std(ddof)`.

**Usage**

```
pl_std(..., ddof = 1)
```

**Arguments**

...	Characters indicating the column names, passed to <code>pl\$col()</code> . See <a href="#">?pl_col</a> for details.
ddof	An integer representing "Delta Degrees of Freedom": the divisor used in the calculation is $N - \text{ddof}$ , where $N$ represents the number of elements. By default <code>ddof</code> is 1.

**Value**

[Expr](#)

**See Also**

- [<Expr>\\$std\(\)](#)

**Examples**

```
df = pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

df$select(pl$std("a"))

df$select(pl$std(c("a", "b")))
```

---

pl_struct	<i>Collect columns into a struct column</i>
-----------	---

---

### Description

Collect columns into a struct column

### Usage

```
pl_struct(exprs, schema = NULL)
```

### Arguments

exprs	Columns/Expressions to collect into a Struct.
schema	Optional schema named list that explicitly defines the struct field dtypes. Each name must match a column name wrapped in the struct. Can only be used to cast some or all dtypes, not to change the names. If NULL (default), columns datatype are not modified. Columns that do not exist are silently ignored and not included in the final struct.

### Details

pl\$struct() creates an Expr of DataType [Struct\(\)](#).

Compared to the Python implementation, pl\$struct() doesn't have the argument eager and always returns an Expr. Use \$to\_series() to return a Series.

### Value

Expr with dtype Struct

### Examples

```
# isolated expression to wrap all columns in a struct aliased 'my_struct'
pl$struct(pl$all())$alias("my_struct")

# wrap all column into on column/Series
df = pl$DataFrame(
  int = 1:2,
  str = c("a", "b"),
  bool = c(TRUE, NA),
  list = list(1:2, 3L)
)$select(
  pl$struct(pl$all())$alias("my_struct")
)

print(df)
print(df$schema) # returns a schema, a named list containing one element a Struct named my_struct
```

```

# wrap two columns in a struct and provide a schema to set all or some DataTypes by name
e1 = pl$struct(
  pl$col(c("int", "str")),
  schema = list(int = pl$Int64, str = pl$String)
)$alias("my_struct")
# same result as e.g. wrapping the columns in a struct and casting afterwards
e2 = pl$struct(
  list(pl$col("int"), pl$col("str"))
)$cast(
  pl$Struct(int = pl$Int64, str = pl$String)
)$alias("my_struct")

df = pl$DataFrame(
  int = 1:2,
  str = c("a", "b"),
  bool = c(TRUE, NA),
  list = list(1:2, 3L)
)

# verify equality in R
identical(df$select(e1)$to_list(), df$select(e2)$to_list())

df$select(e2)
df$select(e2)$to_data_frame()

```

---

pl\_sum

*Sum all values.*

---

### Description

Syntactic sugar for `pl$col(...)$sum()`.

### Usage

```
pl_sum(...)
```

### Arguments

... Characters indicating the column names, passed to `pl$col()`. See `?pl_col` for details.

### Value

`Expr`

### See Also

- `<Expr>$sum()`
- `pl$sum_horizontal()`

**Examples**

```
df = pl$DataFrame(col_a = 1:2, col_b = 3:4, c = 5:6)

df$select(pl$sum("col_a"))

# Sum multiple columns
df$select(pl$sum("col_a", "col_b"))

df$select(pl$sum("^col_.*$"))
```

---

pl_sum_horizontal	<i>Compute the sum rowwise</i>
-------------------	--------------------------------

---

**Description**

Compute the sum rowwise

**Usage**

```
pl_sum_horizontal(...)
```

**Arguments**

... Columns to concatenate into a single string column. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

**Value**

Expr

**Examples**

```
df = pl$DataFrame(
  a = NA_real_,
  b = c(3:4, NA_real_, NA_real_),
  c = c(1:2, NA_real_, -Inf)
)
df$with_columns(
  pl$sum_horizontal("a", "b", "c", 2)$alias("sum")
)
```

---

pl\_tail                      *Get the last n rows.*

---

### Description

This function is syntactic sugar for `pl$col(...)$tail(n)`.

### Usage

```
pl_tail(..., n = 10)
```

### Arguments

...                      Characters indicating the column names, passed to `pl$col()`. See `?pl_col` for details.

n                        Number of rows to return.

### Value

`Expr`

### See Also

- `<Expr>$tail()`

### Examples

```
df = pl$DataFrame(  
  a = c(1, 8, 3),  
  b = c(4, 5, 2),  
  c = c("foo", "bar", "foo")  
)  
  
df$select(pl$tail("a"))  
  
df$select(pl$tail("a", "b", n = 2))
```

---

pl\_thread\_pool\_size      *Get the number of threads in the Polars thread pool.*

---

### Description

The threadpool size can be overridden by setting the `POLARS_MAX_THREADS` environment variable before process start. It cannot be modified once `polars` is loaded. It is strongly recommended not to override this value as it will be set automatically by the engine.

**Usage**

```
pl_thread_pool_size()
```

**Details**

For compatibility with CRAN, the threadpool size is set to 2 by default. To disable this behavior and let the engine determine the threadpool size, one of the following ways can be used:

- Enable the `disable_limit_max_threads` feature of the library. This can be done by setting the feature flag when installing the package. See the installation vignette (`vignette("install", "polars")`) for details.
- Set the `polars.limit_max_threads` option to `FALSE` with the `options()` function. Same as setting the `POLARS_MAX_THREADS` environment variable, this option must be set before loading the package.

**Value**

The number of threads

**Examples**

```
pl$thread_pool_size()
```

---

pl_time	<i>Create a Time expression</i>
---------	---------------------------------

---

**Description**

Create a Time expression

**Usage**

```
pl_time(hour = NULL, minute = NULL, second = NULL, microsecond = NULL)
```

**Arguments**

hour	An Expr or something coercible to an Expr, that must return an integer between 0 and 23. Strings are parsed as column names. Floats are cast to integers.
minute	An Expr or something coercible to an Expr, that must return an integer between 0 and 59. Strings are parsed as column names. Floats are cast to integers.
second	An Expr or something coercible to an Expr, that must return an integer between 0 and 59. Strings are parsed as column names. Floats are cast to integers.
microsecond	An Expr or something coercible to an Expr, that must return an integer between 0 and 999,999. Strings are parsed as column names. Floats are cast to integers.



**Value**

An Expr of type Time

**See Also**

- [pl\\$datetime\(\)](#)
- [pl\\$date\(\)](#)

**Examples**

```
df = pl$DataFrame(hour = 19:21, min = 9:11, sec = 10:12, micro = 1)

df$with_columns(
  time_from_cols = pl$time("hour", "min", "sec", "micro"),
  time_from_lit = pl$time(12, 3, 5),
  time_from_mix = pl$time("hour", 3, 5)
)

# floats are coerced to integers
df$with_columns(
  time_floats = pl$time(12.5, 5.3, 1)
)

# if time can't be constructed, it returns null
df$with_columns(
  time_floats = pl$time(pl$lit("abc"), -2, 1)
)
```

---

`pl_using_string_cache` *Check if the global string cache is enabled*

---

**Description**

This function simply checks if the global string cache is active.

**Usage**

```
pl_using_string_cache()
```

**Value**

A logical value

**See Also**

[pl\\$with\\_string\\_cache](#) [pl\\$enable\\_enable\\_cache](#)

**Examples**

```
pl$enable_string_cache()
pl$using_string_cache()
pl$disable_string_cache()
pl$using_string_cache()
```

---

pl\_var

*Get the variance.*

---

**Description**

This function is syntactic sugar for `pl$col(...)$var(ddof)`.

**Usage**

```
pl_var(..., ddof = 1)
```

**Arguments**

`...` Characters indicating the column names, passed to `pl$col()`. See `?pl_col` for details.

`ddof` An integer representing "Delta Degrees of Freedom": the divisor used in the calculation is  $N - \text{ddof}$ , where  $N$  represents the number of elements. By default `ddof` is 1.

**Value**

`Expr`

**See Also**

- `<Expr>$var()`

**Examples**

```
df = pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

df$select(pl$var("a"))

df$select(pl$var("a", "b"))
```

---

pl\_with\_string\_cache *Evaluate one or several expressions with global string cache*

---

**Description**

This function only temporarily enables the global string cache.

**Usage**

```
pl_with_string_cache(expr)
```

**Arguments**

expr                    An Expr to evaluate while the string cache is enabled.

**Value**

return value of expression

**See Also**

[pl\\$using\\_string\\_cache](#) [pl\\$enable\\_enable\\_cache](#)

**Examples**

```
# activate string cache temporarily when constructing two DataFrame's
pl$with_string_cache({
  df1 = pl$DataFrame(head(iris, 2))
  df2 = pl$DataFrame(tail(iris, 2))
})
pl$concat(list(df1, df2))
```

---

polars\_class\_object *Any polars class object is made of this*

---

**Description**

One SEXP of Rtype: "externalptr" + a class attribute

**Details**

- object\$method() calls are facilitated by a \$.ClassName-s3method see 'R/after-wrappers.R'
- Code completion is facilitated by .DollarNames.ClassName-s3method see e.g. 'R/dataframe\_\_frame.R'
- Implementation of property-methods as DataFrame\_columns() and syntax checking is an extension to \$.ClassName See function macro\_add\_syntax\_check\_to\_class().

**Value**

not applicable

**Examples**

```
# all a polars object is only made of:
some_polars_object = pl$DataFrame(iris)
str(some_polars_object) # External Pointer tagged with a class attribute.

# All state is stored on rust side.

# The single exception from the rule is class "GroupBy", where objects also have
# two private attributes "groupby_input" and "maintain_order".
str(pl$DataFrame(iris)$group_by("Species"))
```

---

```
polars_code_completion_activate
      Polars code completion
```

---

**Description**

Polars code completion

**Usage**

```
polars_code_completion_activate(
  mode = c("auto", "rstudio", "native"),
  verbose = TRUE
)

polars_code_completion_deactivate()
```

**Arguments**

mode	One of "auto", "rstudio", or "native". Automatic mode picks "rstudio" if <code>.Platform\$GUI</code> is "RStudio". "native" registers a custom line buffer completer with <code>utils::rc.getOption("custom.completer")</code> . "rstudio" modifies RStudio code internal <code>.DollarNames</code> and function args completion, as the IDE does not behave well with <code>utils::rc.getOption("custom.completer")</code> .
verbose	Print message of what mode is started.

**Details**

Polars code completion has one implementation for a native terminal via `utils::rc.getOption("custom.completer")` and one for Rstudio by intercepting Rstudio internal functions `.rs.getCompletionsFunction` & `.rs.getCompletionsDollar` in the loaded session environment `tools:rstudio`. Therefore, any error or slowness in the completion is likely to come from r-polars implementation.

Either completers will evaluate the full line-buffer to decide what methods are available. Pressing tab will literally evaluate left-hand-side with any following side. This works swiftly for the polars lazy API, but it can take some time for the eager API depending on the size of the data and of the query.

### Examples

```
if (interactive()) {
  # activate completion
  polars_code_completion_activate()

  # method / property completion for chained expressions
  # add a $ and press tab to see methods of LazyFrame
  pl$LazyFrame(iris)

  # Arg + column-name completion
  # press tab inside group_by() to see args and/or column names.
  pl$LazyFrame(iris)$group_by()

  # deactivate like this or restart R session
  polars_code_completion_deactivate()
}
```

---

polars\_duration\_string

*The Polars duration string language*

---

### Description

The Polars duration string language

### Polars duration string language

Polars duration string language is a simple representation of durations. It is used in many Polars functions that accept durations.

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)
- 1mo (1 calendar month)

- 1q (1 calendar quarter)
- 1y (1 calendar year)

Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds

By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".

---

polars\_envvars

*Get polars environment variables*

---

## Description

Get polars environment variables

## Usage

```
polars_envvars()
```

## Details

The following envvars are available (in alphabetical order, with the default value in parenthesis):

- POLARS\_FMT\_MAX\_COLS (5): Set the number of columns that are visible when displaying tables. If negative, all columns are displayed.
- POLARS\_FMT\_MAX\_ROWS (8): Set the number of rows that are visible when displaying tables. If negative, all rows are displayed. This applies to both [DataFrame](#) and [Series](#).
- POLARS\_FMT\_STR\_LEN (32): Maximum number of characters to display;
- POLARS\_FMT\_TABLE\_CELL\_ALIGNMENT ("LEFT"): set the table cell alignment. Can be "LEFT", "CENTER", "RIGHT";
- POLARS\_FMT\_TABLE\_CELL\_LIST\_LEN (3): Maximum number of elements of list variables to display;
- POLARS\_FMT\_TABLE\_CELL\_NUMERIC\_ALIGNMENT ("LEFT"): Set the table cell alignment for numeric columns. Can be "LEFT", "CENTER", "RIGHT";
- POLARS\_FMT\_TABLE\_DATAFRAME\_SHAPE\_BELOW ("0"): print the DataFrame shape information below the data when displaying tables. Can be "0" or "1".
- POLARS\_FMT\_TABLE\_FORMATTING ("UTF8\_FULL\_CONDENSED"): Set table formatting style. Possible values:
  - "ASCII\_FULL": ASCII, with all borders and lines, including row dividers.
  - "ASCII\_FULL\_CONDENSED": Same as ASCII\_FULL, but with dense row spacing.
  - "ASCII\_NO\_BORDERS": ASCII, no borders.
  - "ASCII\_BORDERS\_ONLY": ASCII, borders only.
  - "ASCII\_BORDERS\_ONLY\_CONDENSED": ASCII, borders only, dense row spacing.
  - "ASCII\_HORIZONTAL\_ONLY": ASCII, horizontal lines only.

- "ASCII\_MARKDOWN": ASCII, Markdown compatible.
  - "UTF8\_FULL": UTF8, with all borders and lines, including row dividers.
  - "UTF8\_FULL\_CONDENSED": Same as UTF8\_FULL, but with dense row spacing.
  - "UTF8\_NO\_BORDERS": UTF8, no borders.
  - "UTF8\_BORDERS\_ONLY": UTF8, borders only.
  - "UTF8\_HORIZONTAL\_ONLY": UTF8, horizontal lines only.
  - "NOTHING": No borders or other lines.
- POLARS\_FMT\_TABLE\_HIDE\_COLUMN\_DATA\_TYPES ("0"): Hide table column data types (i64, f64, str etc.). Can be "0" or "1".
  - POLARS\_FMT\_TABLE\_HIDE\_COLUMN\_NAMES ("0"): Hide table column names. Can be "0" or "1".
  - POLARS\_FMT\_TABLE\_HIDE\_COLUMN\_SEPARATOR ("0"): Hide the "---" separator between the column names and column types. Can be "0" or "1".
  - POLARS\_FMT\_TABLE\_HIDE\_DATAFRAME\_SHAPE\_INFORMATION ("0"): Hide the DataFrame shape information when displaying tables. Can be "0" or "1".
  - POLARS\_FMT\_TABLE\_INLINE\_COLUMN\_DATA\_TYPE ("0"): Moves the data type inline with the column name (to the right, in parentheses). Can be "0" or "1".
  - POLARS\_FMT\_TABLE\_ROUNDED\_CORNERS ("0"): Apply rounded corners to UTF8-styled tables (only applies to UTF8 formats).
  - POLARS\_MAX\_THREADS (<variable>): Maximum number of threads used to initialize the thread pool. The thread pool is locked once polars is loaded, so this envvar must be set before loading the package.
  - POLARS\_STREAMING\_CHUNK\_SIZE (<variable>): Chunk size used in the streaming engine. Integer larger than 1. By default, the chunk size is determined by the schema and size of the thread pool. For some datasets (esp. when you have large string elements) this can be too optimistic and lead to Out of Memory errors.
  - POLARS\_TABLE\_WIDTH (<variable>): Set the maximum width of a table in characters.
  - POLARS\_VERBOSE ("0"): Enable additional verbose/debug logging.
  - POLARS\_WARN\_UNSTABLE ("0"): Issue a warning when unstable functionality is used. Enabling this setting may help avoid functionality that is still evolving, potentially reducing maintenance burden from API changes and bugs. Can be "0" or "1".

The following configuration options are present in the Python API but currently cannot be changed in R: decimal separator, thousands separator, float precision, float formatting, trimming decimal zeros.

## Value

`polars_envvars()` returns a named list where the names are the names of environment variables and values are their values.

## Examples

```
polars_envvars()

pl$DataFrame(x = "This is a very very very long sentence.")

Sys.setenv(POLARS_FMT_STR_LEN = 50)
pl$DataFrame(x = "This is a very very very long sentence.")

# back to default
Sys.setenv(POLARS_FMT_STR_LEN = 32)
```

---

polars\_info

*Report information of the package*

---

## Description

This function reports the following information:

- Package versions (the Polars R package version and the dependent Rust Polars crate version)
- [Number of threads used by Polars](#)
- Rust feature flags (See `vignette("install", "polars")` for details)
- Code completion mode: either "deactivated", "rstudio", or "native". See [polars\\_code\\_completion\\_activate\(\)](#)

## Usage

```
polars_info()
```

## Value

A list with information of the package

## Examples

```
polars_info()

polars_info()$versions

polars_info()$features$nightly
```



---

polars\_options      *Get and reset polars options*

---

### Description

`polars_options()` returns a list of options for polars. Options can be set with `options()`. Note that **options must be prefixed with "polars."**, e.g to modify the option `strictly_immutable` you need to pass `options(polars.strictly_immutable =)`. See below for a description of all options.

`polars_options_reset()` brings all polars options back to their default value.

### Usage

```
polars_options()
```

```
polars_options_reset()
```

### Details

The following options are available (in alphabetical order, with the default value in parenthesis):

- `debug_polars` (FALSE): Print additional information to debug Polars.
- `do_not_repeat_call` (FALSE): Do not print the call causing the error in error messages. The default is to show them.
- `int64_conversion` ("double"): How should Int64 values be handled when converting a polars object to R?
  - "double" converts the integer values to double.
  - "bit64" uses `bit64::as.integer64()` to do the conversion (requires the package `bit64` to be attached).
  - "string" converts Int64 values to character.
- `limit_max_threads` (`!polars_info()$features$disable_limit_max_threads`): See `?pl_thread_pool_size` for details. This option should be set before the package is loaded.
- `maintain_order` (FALSE): Default for the `maintain_order` argument in `<LazyFrame>$group_by()` and `<DataFrame>$group_by()`.
- `no_messages` (FALSE): Hide messages.
- `rpool_cap`: The maximum number of R sessions that can be used to process R code in the background. See the section "About pool options" below.
- `strictly_immutable` (TRUE): Keep polars strictly immutable. Polars/arrow is in general pro "immutable objects". Immutability is also classic in R. To mimic the Python-polars API, set this to FALSE.

### Value

`polars_options()` returns a named list where the names are option names and values are option values.

`polars_options_reset()` doesn't return anything.

**About pool options**

`polars_options()$rpool_active` indicates the number of R sessions already spawned in pool. `polars_options()$rpool_cap` indicates the maximum number of new R sessions that can be spawned. Anytime a polars thread worker needs a background R session specifically to run R code embedded in a query via `$map_batches(..., in_background = TRUE)` or `$map_elements(..., in_background = TRUE)`, it will obtain any R session idling in rpool, or spawn a new R session (process) and add it to the rpool if `rpool_cap` is not already reached. If `rpool_cap` is already reached, the thread worker will sleep until an R session is idling.

Background R sessions communicate via polars arrow IPC (series/vectors) or R serialize + shared memory buffers via the rust crate `ipc-channel`. Multi-process communication has overhead because all data must be serialized/de-serialized and sent via buffers. Using multiple R sessions will likely only give a speed-up in a low io - high cpu scenario. Native polars query syntax runs in threads and have no overhead.

**Examples**

```
options(polars.maintain_order = TRUE, polars.strictly_immutable = FALSE)
polars_options()

# option checks are run when calling polars_options(), not when setting
# options
options(polars.maintain_order = 42, polars.int64_conversion = "foobar")
tryCatch(
  polars_options(),
  error = function(e) print(e)
)

# reset options to their default value
polars_options_reset()
```

---

print.RPolarsSeries    *Print values*

---

**Description**

Print values

**Usage**

```
## S3 method for class 'RPolarsSeries'
print(x, ...)
```

**Arguments**

<code>x</code>	A Polars Series
<code>...</code>	Not used

---

RollingGroupBy\_agg     *Aggregate over a RollingGroupBy*

---

**Description**

Aggregate a `DataFrame` over a rolling window created with `$rolling()`.

**Usage**

```
RollingGroupBy_agg(...)
```

**Arguments**

...                    Exprs to aggregate over. Those can also be passed wrapped in a list, e.g `$agg(list(e1, e2, e3))`.

**Value**

An aggregated `DataFrame`

**Examples**

```
df = pl$DataFrame(
  dt = c("2020-01-01", "2020-01-01", "2020-01-01", "2020-01-02", "2020-01-03", "2020-01-08"),
  a = c(3, 7, 5, 9, 2, 1)
)$with_columns(
  pl$col("dt")$str$strptime(pl$Date, format = NULL)$set_sorted()
)

df$rolling(index_column = "dt", period = "2d")$agg(
  pl$col("a"),
  pl$sum("a")$alias("sum_a"),
  pl$min("a")$alias("min_a"),
  pl$max("a")$alias("max_a")
)
```

---

RollingGroupBy\_class     *Operations on Polars DataFrame grouped by rolling windows*

---

**Description**

This class comes from `<DataFrame>$rolling()`.

**Examples**

```
df = pl$DataFrame(  
  dt = c("2020-01-01", "2020-01-01", "2020-01-01", "2020-01-02", "2020-01-03", "2020-01-08"),  
  a = c(3, 7, 5, 9, 2, 1)  
)$with_columns(  
  pl$col("dt")$str$strptime(pl$Date, format = NULL)$set_sorted()  
)  
  
df$rolling(index_column = "dt", period = "2d")
```

---

RollingGroupBy\_ungroup

*Ungroup a RollingGroupBy object*

---

**Description**

Revert the `$rolling()` operation. Doing `<DataFrame>$rolling(...)$ungroup()` returns the original `DataFrame`.

**Usage**

```
RollingGroupBy_ungroup()
```

**Value**

[DataFrame](#)

**Examples**

```
df = pl$DataFrame(  
  dt = c("2020-01-01", "2020-01-01", "2020-01-01", "2020-01-02", "2020-01-03", "2020-01-08"),  
  a = c(3, 7, 5, 9, 2, 1)  
)$with_columns(  
  pl$col("dt")$str$strptime(pl$Date, format = NULL)$set_sorted()  
)  
  
df$rolling(index_column = "dt", period = "2d")$ungroup()
```

---

```
row.names.RPolarsDataFrame
      Get the row names
```

---

**Description**

Get the row names

**Usage**

```
## S3 method for class 'RPolarsDataFrame'
row.names(x)
```

**Arguments**

x                    A Polars DataFrame

---

```
RThreadHandle_class    The RPolarsRThreadHandle class
```

---

**Description**

A handle to some polars query running in a background thread.

**Details**

`<LazyFrame>$collect_in_background()` will execute a polars query detached from the R session and return an `RPolarsRThreadHandle` immediately. This `RPolarsRThreadHandle`-class has the methods `is_finished()` and `join()`.

**NOTICE**

The background thread cannot use the main R session, but can access the pool of extra R sessions to process R code embedded in polars query via `$map_batches(..., in_background = TRUE)` or `$map_elements(background=TRUE)`. Use `options(polars.rpool_cap = XX)` to limit number of parallel R sessions. Starting polars `<LazyFrame>$collect_in_background()` with e.g. some `$map_batches(..., in_background = FALSE)` will raise an Error as the main R session is not available to process the R part of the polars query. Native polars query does not need any R session.

**See Also**

- `<LazyFrame>$collect_in_background()`
- `<Expr>$map_batches()`
- `<Expr>$map_elements()`

**Examples**

```
prexpr = pl$col("mpg")$map_batches(\(x) {
  Sys.sleep(.1)
  x * 0.43
}, in_background = TRUE)$alias("kml")
handle = pl$LazyFrame(mtcars)$with_columns(prexpr)$collect_in_background()
if (!handle$is_finished()) print("not done yet")
df = handle$join() # get result
df
```

---

RThreadHandle\_is\_finished

*Ask if RThreadHandle is finished?*

---

**Description**

Ask if RThreadHandle is finished?

**Usage**

```
RThreadHandle_is_finished()
```

**Value**

trinary value: TRUE if finished, FALSE if not, and NULL if the handle was exhausted with `<RThreadHandle>$join()`.

---

RThreadHandle\_join

*Join a RThreadHandle*

---

**Description**

Join a RThreadHandle

**Usage**

```
RThreadHandle_join()
```

**Details**

method `<RThreadHandle>$join()`: will block until job is done and then return some value or raise an error from the thread. Calling `<RThreadHandle>$join()` a second time will raise an error because handle is already exhausted.

**Value**

return value from background thread

**See Also**[RThreadHandle\\_class](#)

---

**S3\_arithmetic***Arithmetic operators for RPolars objects*

---

**Description**

Arithmetic operators for RPolars objects

**Usage**

```
## S3 method for class 'RPolarsExpr'  
x + y  
  
## S3 method for class 'RPolarsExpr'  
x - y  
  
## S3 method for class 'RPolarsExpr'  
x * y  
  
## S3 method for class 'RPolarsExpr'  
x / y  
  
## S3 method for class 'RPolarsExpr'  
x ^ y  
  
## S3 method for class 'RPolarsExpr'  
x %% y  
  
## S3 method for class 'RPolarsExpr'  
x %/% y  
  
## S3 method for class 'RPolarsSeries'  
x + y  
  
## S3 method for class 'RPolarsSeries'  
x - y  
  
## S3 method for class 'RPolarsSeries'  
x * y  
  
## S3 method for class 'RPolarsSeries'  
x / y  
  
## S3 method for class 'RPolarsSeries'
```

```
x ^ y

## S3 method for class 'RPolarsSeries'
x %% y

## S3 method for class 'RPolarsSeries'
x %/% y
```

### Arguments

`x, y` numeric type of RPolars objects or objects that can be coerced such. Only + can take strings.

### Value

A Polars object the same type as the input.

### See Also

- `<Expr>$add()`
- `<Expr>$sub()`
- `<Expr>$mul()`
- `<Expr>$div()`
- `<Expr>$pow()`
- `<Expr>$mod()`
- `<Expr>$floor_div()`
- `<Series>$add()`
- `<Series>$sub()`
- `<Series>$mul()`
- `<Series>$div()`
- `<Series>$pow()`
- `<Series>$mod()`
- `<Series>$floor_div()`

### Examples

```
pl$lit(5) + 10
5 + pl$lit(10)
pl$lit(5) + pl$lit(10)
+pl$lit(1)

# This will not raise an error as it is not actually evaluated.
expr = pl$lit(5) + "10"
expr

# Will raise an error as it is evaluated.
```



```
tryCatch(  
  expr$to_series(),  
  error = function(e) e  
)  
  
as_polars_series(5) + 10  
+as_polars_series(5)  
-as_polars_series(5)
```

---

Series\_add

*Add Series*

---

## Description

Method equivalent of addition operator `series + other`.

## Usage

```
Series_add(other)
```

## Arguments

`other` [Series](#) like object of numeric or string values. Converted to [Series](#) by `as_polars_series()` in this method.

## Value

[Series](#)

## See Also

- [Arithmetic operators](#)

## Examples

```
as_polars_series(1:3)$add(as_polars_series(11:13))  
as_polars_series(1:3)$add(11:13)  
as_polars_series(1:3)$add(1L)  
  
as_polars_series("a")$add("-z")
```

---

Series_alias	<i>Change name of Series</i>
--------------	------------------------------

---

**Description**

Change name of Series

**Usage**

```
Series_alias(name)
```

**Arguments**

name	New name.
------	-----------

**Value**

[Series](#)

**Examples**

```
as_polars_series(1:3, name = "alice")$alias("bob")
```

---

Series_all	<i>Reduce Boolean Series with ALL</i>
------------	---------------------------------------

---

**Description**

Reduce Boolean Series with ALL

**Usage**

```
Series_all()
```

**Value**

A logical value

**Examples**

```
as_polars_series(c(TRUE, TRUE, NA))$all()
```

---

Series_any	<i>Reduce boolean Series with ANY</i>
------------	---------------------------------------

---

**Description**

Reduce boolean Series with ANY

**Usage**

```
Series_any()
```

**Value**

A logical value

**Examples**

```
as_polars_series(c(TRUE, FALSE, NA))$any()
```

---

Series_append	<i>Append two Series</i>
---------------	--------------------------

---

**Description**

Append two Series

**Usage**

```
Series_append(other, immutable = TRUE)
```

**Arguments**

other	Series to append.
immutable	Should the other Series be immutable? Default is TRUE.

**Details**

If `immutable = FALSE`, the Series object will not behave as immutable. This means that appending to this Series will affect any variable pointing to this memory location. This will break normal scoping rules of R. Setting `immutable = FALSE` is discouraged as it can have undesirable side effects and cloning Polars Series is a cheap operation.

**Value**

[Series](#)

**Examples**

```

# default immutable behavior, s_imut and s_imut_copy stay the same
s_imut = as_polars_series(1:3)
s_imut_copy = s_imut
s_new = s_imut$append(as_polars_series(1:3))
s_new

# the original Series didn't change
s_imut
s_imut_copy

# enabling mutable behavior requires setting a global option
withr::with_options(
  list(polars.strictly_immutable = FALSE),
  {
    s_mut = as_polars_series(1:3)
    s_mut_copy = s_mut
    s_new = s_mut$append(as_polars_series(1:3), immutable = FALSE)
    print(s_new)

    # the original Series also changed since it's mutable
    print(s_mut)
    print(s_mut_copy)
  }
)

```

---

Series_arg_max	<i>Index of max value</i>
----------------	---------------------------

---

**Description**

Note that this is 0-indexed.

**Usage**

```
Series_arg_max()
```

**Value**

A numeric value

**Examples**

```
as_polars_series(c(5, 1))$arg_max()
```

---

Series_arg_min	<i>Index of min value</i>
----------------	---------------------------

---

**Description**

Note that this is 0-indexed.

**Usage**

```
Series_arg_min()
```

**Value**

A numeric value

**Examples**

```
as_polars_series(c(5, 1))$arg_min()
```

---

Series_chunk_lengths	<i>Lengths of Series memory chunks</i>
----------------------	--

---

**Description**

Lengths of Series memory chunks

**Usage**

```
Series_chunk_lengths()
```

**Value**

Numeric vector. Output length is the number of chunks, and the sum of the output is equal to the length of the full Series.

**Examples**

```
chunked_series = c(as_polars_series(1:3), as_polars_series(1:10))  
chunked_series$chunk_lengths()
```

**Description**

The Series-class is simply two environments of respectively the public and private methods/function calls to the polars rust side. The instantiated Series-object is an externalptr to a lowlevel rust polars Series object. The pointer address is the only statefullness of the Series object on the R side. Any other state resides on the rust side. The S3 method `.DollarNames.RPolarsSeries` exposes all public `$foobar()`-methods which are callable onto the object. Most methods return another Series-class instance or similar which allows for method chaining. This class system in lack of a better name could be called "environment classes" and is the same class system extendr provides, except here there is both a public and private set of methods. For implementation reasons, the private methods are external and must be called from `.pr$Series$methodname()`, also all private methods must take any self as an argument, thus they are pure functions. Having the private methods as pure functions solved/simplified self-referential complications.

**Details**

Check out the source code in `R/Series_frame.R` how public methods are derived from private methods. Check out `extendr-wrappers.R` to see the extendr-auto-generated methods. These are moved to `.pr` and converted into pure external functions in `after-wrappers.R`. In `zzz.R` (named `zzz` to be last file sourced) the extendr-methods are removed and replaced by any function prefixed `Series_`.

**Active bindings****dtype:**

`$dtype` returns the [data type](#) of the Series.

**flags:**

`$flags` returns a named list with flag names and their values.

Flags are used internally to avoid doing unnecessary computations, such as sorting a variable that we know is already sorted. The number of flags varies depending on the column type: columns of type `array` and `list` have the flags `SORTED_ASC`, `SORTED_DESC`, and `FAST_EXPLODE`, while other column types only have the former two.

- `SORTED_ASC` is set to `TRUE` when we sort a column in increasing order, so that we can use this information later on to avoid re-sorting it.
- `SORTED_DESC` is similar but applies to sort in decreasing order.

**name:**

`$name` returns the name of the Series.

**shape:**

`$shape` returns a numeric vector of length two with the number of length of the Series and width of the Series (always 1).

## Expression methods

Series stores most of all [Expr](#) methods.

Some of these are stored in sub-namespaces.

### **arr:**

\$arr stores all array related methods.

### **bin:**

\$bin stores all binary related methods.

### **cat:**

\$cat stores all categorical related methods.

### **dt:**

\$dt stores all temporal related methods.

### **list:**

\$list stores all list related methods.

### **str:**

\$str stores all string related methods.

### **struct:**

\$struct stores all struct related methods and active bindings.

Active bindings specific to Series:

- \$struct\$fields: Returns a character vector of the fields in the struct.

## Conversion to R data types considerations

When converting Polars objects, such as [DataFrames](#) to R objects, for example via the `as.data.frame()` generic function, each type in the Polars object is converted to an R type. In some cases, an error may occur because the conversion is not appropriate. In particular, there is a high possibility of an error when converting a [Datetime](#) type without a time zone. A [Datetime](#) type without a time zone in Polars is converted to the [POSIXct](#) type in R, which takes into account the time zone in which the R session is running (which can be checked with the `Sys.timezone()` function). In this case, if ambiguous times are included, a conversion error will occur. In such cases, change the session time zone using `Sys.setenv(TZ = "UTC")` and then perform the conversion, or use the `$dt$replace_time_zone()` method on the Datetime type column to explicitly specify the time zone before conversion.

```
# Due to daylight savings, clocks were turned forward 1 hour on Sunday, March 8, 2020, 2:00:00 am
# so this particular date-time doesn't exist
non_existent_time = as_polars_series("2020-03-08 02:00:00")$str$strptime(pl$Datetime(), "%F %T")

withr::with_timezone(
  "America/New_York",
  {
    tryCatch(
      # This causes an error due to the time zone (the `TZ` env var is affected).

```

```

        as.vector(non_existent_time),
        error = function(e) e
    )
}
)
#> <error: in to_r: ComputeError(ErrString("datetime '2020-03-08 02:00:00' is non-existent in time zone

withr::with_timezone(
  "America/New_York",
  {
    # This is safe.
    as.vector(non_existent_time$dt$replace_time_zone("UTC"))
  }
)
#> [1] "2020-03-08 02:00:00 UTC"

```

## Examples

```

# make a Series
s = as_polars_series(c(1:3, 1L))

# call an active binding
s$shape

# show flags
s$sort()$flags

# use Expr method
s$cos()

# use Expr method in subnamespaces
as_polars_series(list(3:1, 1:2, NULL))$list$first()
as_polars_series(c(1, NA, 2))$str$join("-")

s = pl$date_range(
  as.Date("2024-02-18"), as.Date("2024-02-24"),
  interval = "1d"
)$to_series()
s
s$dt$day()

# Other active bindings in subnamespaces
as_polars_series(data.frame(a = 1:2, b = 3:4))$struct$fields

# show all available methods for Series
pl$show_all_public_methods("RPolarsSeries")

```



---

Series_clear	Create an empty or n-row null-filled copy of the Series
--------------	---

---

**Description**

Returns a n-row null-filled Series with an identical schema. n can be greater than the current number of values in the Series.

**Usage**

```
Series_clear(n = 0)
```

**Arguments**

n                      Number of (null-filled) rows to return in the cleared frame.

**Value**

A n-value null-filled Series with an identical schema

**Examples**

```
s = pl$Series(name = "a", values = 1:3)
s$clear()
s$clear(n = 5)
```

---

Series_clone	Clone a Series
--------------	----------------

---

**Description**

This makes a very cheap deep copy/clone of an existing [Series](#). Rarely useful as Series are nearly 100% immutable. Any modification of a Series should lead to a clone anyways, but this can be useful when dealing with attributes (see examples).

**Usage**

```
Series_clone()
```

**Value**

[Series](#)

**Examples**

```

df1 = as_polars_series(1:10)

# Make a function to take a Series, add an attribute, and return a Series
give_attr = function(data) {
  attr(data, "created_on") = "2024-01-29"
  data
}
df2 = give_attr(df1)

# Problem: the original Series also gets the attribute while it shouldn't!
attributes(df1)

# Use $clone() inside the function to avoid that
give_attr = function(data) {
  data = data$clone()
  attr(data, "created_on") = "2024-01-29"
  data
}
df1 = as_polars_series(1:10)
df2 = give_attr(df1)

# now, the original Series doesn't get this attribute
attributes(df1)

```

---

Series\_compare

*Compare Series*


---

**Description**

Check the (in)equality of two Series.

**Usage**

```

Series_compare(other, op)

## S3 method for class 'RPolarsSeries'
s1 == s2

## S3 method for class 'RPolarsSeries'
s1 != s2

## S3 method for class 'RPolarsSeries'
s1 < s2

## S3 method for class 'RPolarsSeries'
s1 > s2

```

```
## S3 method for class 'RPolarsSeries'
s1 <= s2
```

```
## S3 method for class 'RPolarsSeries'
s1 >= s2
```

### Arguments

other	A Series or something a Series can be created from
op	The chosen operator, must be one of "equal", "not_equal", "lt", "gt", "lt_eq" or "gt_eq"
s1	lhs Series
s2	rhs Series or any into Series

### Value

[Series](#)

### Examples

```
# We can either use `compare()`...
as_polars_series(1:5)$compare(as_polars_series(c(1:3, NA_integer_, 10L)), op = "equal")

# ... or the more classic way
as_polars_series(1:5) == as_polars_series(c(1:3, NA_integer_, 10L))
```

---

Series\_div

*Divide Series*

---

### Description

Method equivalent of division operator `series / other`.

### Usage

```
Series_div(other)
```

### Arguments

other	<a href="#">Series</a> like object of numeric. Converted to <a href="#">Series</a> by <code>as_polars_series()</code> in this method.
-------	---

### Value

[Series](#)

**See Also**

- [Arithmetic operators](#)

**Examples**

```
as_polars_series(1:3)$div(11:13)
as_polars_series(1:3)$div(as_polars_series(11:13))
as_polars_series(1:3)$div(1L)
```

---

Series_equals	<i>Are two Series equal?</i>
---------------	------------------------------

---

**Description**

This checks whether two Series are equal in values and in their name.

**Usage**

```
Series_equals(other, null_equal = FALSE, strict = FALSE)
```

**Arguments**

other	Series to compare with.
null_equal	If TRUE, consider that null values are equal. Overridden by strict.
strict	If TRUE, do not allow similar DataType comparison. Overrides null_equal.

**Value**

A logical value

**Examples**

```
as_polars_series(1:4)$equals(as_polars_series(1:4))

# names are different
as_polars_series(1:4, "bob")$equals(as_polars_series(1:4))

# nulls are different by default
as_polars_series(c(1:4, NA))$equals(as_polars_series(c(1:4, NA)))
as_polars_series(c(1:4, NA))$equals(as_polars_series(c(1:4, NA)), null_equal = TRUE)

# datatypes are ignored by default
as_polars_series(1:4)$cast(pl$Int16)$equals(as_polars_series(1:4))
as_polars_series(1:4)$cast(pl$Int16)$equals(as_polars_series(1:4), strict = TRUE)
```

---

Series_floor_div	<i>Floor Divide Series</i>
------------------	----------------------------

---

**Description**

Method equivalent of floor division operator `series %/% other`.

**Usage**

```
Series_floor_div(other)
```

**Arguments**

`other` [Series](#) like object of numeric. Converted to [Series](#) by `as_polars_series()` in this method.

**Value**

[Series](#)

**See Also**

- [Arithmetic operators](#)

**Examples**

```
as_polars_series(1:3)$floor_div(11:13)
as_polars_series(1:3)$floor_div(as_polars_series(11:13))
as_polars_series(1:3)$floor_div(1L)
```

---

Series_is_numeric	<i>Check if the Series is numeric</i>
-------------------	---------------------------------------

---

**Description**

This checks whether the Series DataType is in `pl$numeric_dtypes`.

**Usage**

```
Series_is_numeric()
```

**Value**

A logical value

**Examples**

```
as_polars_series(1:4)$is_numeric()
as_polars_series(c("a", "b", "c"))$is_numeric()
pl$numeric_dtypes
```

---

Series\_is\_sorted      *Check if the Series is sorted*

---

**Description**

Check if the Series is sorted

**Usage**

```
Series_is_sorted(descending = FALSE)
```

**Arguments**

descending      Check if the Series is sorted in descending order.

**Value**

A logical value

**See Also**

Use `$set_sorted()` to add a "sorted" flag to the Series that could be used for faster operations later on.

**Examples**

```
as_polars_series(1:4)$sort()$is_sorted()
```

---

Series\_item      *Return the element at the given index*

---

**Description**

Return the element at the given index

**Usage**

```
Series_item(index = NULL)
```

**Arguments**

index      Index of the item to return.

**Value**

A value of length 1

**Examples**

```
s1 = pl$Series(values = 1)
```

```
s1$item()
```

```
s2 = pl$Series(values = 9:7)
```

```
s2$cum_sum()$item(-1)
```

---

Series_len	<i>Length of a Series</i>
------------	---------------------------

---

**Description**

Length of a Series

**Usage**

```
Series_len()
```

**Value**

A numeric value

**Examples**

```
as_polars_series(1:10)$len()
```

---

Series_map_elements	<i>Apply every value with an R fun</i>
---------------------	--

---

**Description**

About as slow as regular non-vectorized R. Similar to using R sapply on a vector.

**Usage**

```
Series_map_elements(  
  fun,  
  datatype = NULL,  
  strict_return_type = TRUE,  
  allow_fail_eval = FALSE  
)
```

**Arguments**

**fun**                    r function, should take a single value as input and return one.  
**datatype**             DataType of return value. Default NULL means same as input.  
**strict\_return\_type**    bool, default TRUE: fail on wrong return type, FALSE: convert to polars Null  
**allow\_fail\_eval**        bool, default FALSE: raise R fun error, TRUE: convert to polars Null

**Value**

[Series](#)

**Examples**

```

s = as_polars_series(letters[1:5], "ltrs")
f = \(x) paste(x, ":", as.integer(charToRaw(x)))
s$map_elements(f, pl$String)

# same as
as_polars_series(sapply(s$to_r(), f), s$name)
  
```

---

Series\_max

*Find the max of a Series*

---

**Description**

Find the max of a Series

**Usage**

```
Series_max()
```

**Details**

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

**Value**

A numeric value

**Examples**

```

as_polars_series(c(1:2, NA, 3, 5))$max() # a NA is dropped always
as_polars_series(c(1:2, NA, 3, NaN, 4, Inf))$max() # NaN carries / poisons
as_polars_series(c(1:2, 3, Inf, 4, -Inf, 5))$max() # Inf-Inf is NaN
  
```



---

Series_mean	<i>Compute the mean of a Series</i>
-------------	-------------------------------------

---

**Description**

Compute the mean of a Series

**Usage**

```
Series_mean()
```

**Details**

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

**Value**

A numeric value

**Examples**

```
as_polars_series(c(1:2, NA, 3, 5))$mean() # a NA is dropped always  
as_polars_series(c(1:2, NA, 3, NaN, 4, Inf))$mean() # NaN carries / poisons  
as_polars_series(c(1:2, 3, Inf, 4, -Inf, 5))$mean() # Inf-Inf is NaN
```

---

Series_median	<i>Compute the median of a Series</i>
---------------	---------------------------------------

---

**Description**

Compute the median of a Series

**Usage**

```
Series_median()
```

**Details**

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

**Value**

A numeric value

**Examples**

```
as_polars_series(c(1:2, NA, 3, 5))$median() # a NA is dropped always
as_polars_series(c(1:2, NA, 3, NaN, 4, Inf))$median() # NaN carries / poisons
as_polars_series(c(1:2, 3, Inf, 4, -Inf, 5))$median() # Inf-Inf is NaN
```

Series\_min

*Find the min of a Series***Description**

Find the min of a Series

**Usage**

```
Series_min()
```

**Details**

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

**Value**

A numeric value

**Examples**

```
as_polars_series(c(1:2, NA, 3, 5))$min() # a NA is dropped always
as_polars_series(c(1:2, NA, 3, NaN, 4, Inf))$min() # NaN carries / poisons
as_polars_series(c(1:2, 3, Inf, 4, -Inf, 5))$min() # Inf-Inf is NaN
```

Series\_mod

*Modulo Series***Description**

Method equivalent of modulo operator `series %% other`.

**Usage**

```
Series_mod(other)
```

**Arguments**

`other` [Series](#) like object of numeric. Converted to [Series](#) by `as_polars_series()` in this method.

**Value**

[Series](#)

**See Also**

- [Arithmetic operators](#)

**Examples**

```
as_polars_series(1:4)$mod(2L)
as_polars_series(1:3)$mod(as_polars_series(11:13))
as_polars_series(1:3)$mod(1L)
```

---

Series\_mul

*Multiply Series*

---

**Description**

Method equivalent of multiplication operator `series * other`.

**Usage**

```
Series_mul(other)
```

**Arguments**

`other` [Series](#) like object of numeric. Converted to [Series](#) by `as_polars_series()` in this method.

**Value**

[Series](#)

**See Also**

- [Arithmetic operators](#)

**Examples**

```
as_polars_series(1:3)$mul(11:13)
as_polars_series(1:3)$mul(as_polars_series(11:13))
as_polars_series(1:3)$mul(1L)
```

---

Series_n_chunks	<i>Get the number of chunks that this Series contains.</i>
-----------------	--

---

**Description**

Get the number of chunks that this Series contains.

**Usage**

```
Series_n_chunks()
```

**Value**

A numeric value

**Examples**

```
s = as_polars_series(1:3)
s$n_chunks()

# Concatenate Series with rechunk = TRUE
s2 = as_polars_series(4:6)
pl$concat(s, s2, rechunk = TRUE)$n_chunks()

# Concatenate Series with rechunk = FALSE
pl$concat(s, s2, rechunk = FALSE)$n_chunks()
```

---

Series_n_unique	<i>Count unique values in Series</i>
-----------------	--------------------------------------

---

**Description**

Count unique values in Series

**Usage**

```
Series_n_unique()
```

**Value**

A numeric value

**Examples**

```
as_polars_series(c(1, 2, 1, 4, 4, 1, 5))$n_unique()
```

---

Series_pow	<i>Power Series</i>
------------	---------------------

---

**Description**

Method equivalent of power operator `series ^ other`.

**Usage**

```
Series_pow(exponent)
```

**Arguments**

exponent      [Series](#) like object of numeric. Converted to [Series](#) by `as_polars_series()` in this method.

**Value**

[Series](#)

**See Also**

- [Arithmetic operators](#)

**Examples**

```
s = as_polars_series(1:4, name = "foo")
s$pow(3L)
```

---

Series_print	<i>Print Series</i>
--------------	---------------------

---

**Description**

Print Series

**Usage**

```
Series_print()
```

**Value**

self

**Examples**

```
as_polars_series(1:3)
```

---

Series_rename	<i>Rename a series</i>
---------------	------------------------

---

**Description**

Rename a series

**Usage**

```
Series_rename(name, in_place = FALSE)
```

**Arguments**

name	New name.
in_place	Rename in-place, which breaks immutability. If TRUE, you need to run <code>options(polars.strictly_immutable = FALSE)</code> before, otherwise it will throw an error.

**Value**

[Series](#)

**Examples**

```
as_polars_series(1:4, "bob")$rename("alice")
```

---

Series_rep	<i>Duplicate and concatenate a series</i>
------------	---

---

**Description**

Note that this function doesn't exist in Python Polars.

**Usage**

```
Series_rep(n, rechunk = TRUE)
```

**Arguments**

n	Number of times to repeat
rechunk	If TRUE (default), reallocate object in memory which can speed up some calculations. If FALSE, the Series will take less space in memory.

**Value**

[Series](#)

**Examples**

```
as_polars_series(1:2, "bob")$rep(3)
```

---

Series_set_sorted	<i>Set a sorted flag on a Series</i>
-------------------	--------------------------------------

---

**Description**

Set a sorted flag on a Series

**Usage**

```
Series_set_sorted(..., descending = FALSE, in_place = FALSE)
```

**Arguments**

...	Ignored.
descending	Sort the columns in descending order.
in_place	If TRUE, this will set the flag mutably and return NULL. Remember to use <code>options(polars.strictly_immutable = FALSE)</code> before using this parameter, otherwise an error will occur. If FALSE (default), it will return a cloned Series with the flag.

**Details**

Use [\\$flags](#) to see the values of the sorted flags.

**Value**

A [Series](#) with a flag

**Examples**

```
s = as_polars_series(1:4)$set_sorted()
s$flags
```

---

Series_sort	<i>Sort a Series</i>
-------------	----------------------

---

## Description

Sort a Series

## Usage

```
Series_sort(
  ...,
  descending = FALSE,
  nulls_last = FALSE,
  multithreaded = TRUE,
  in_place = FALSE
)
```

## Arguments

...	Ignored.
descending	A logical. If TRUE, sort in descending order.
nulls_last	A logical. If TRUE, place null values last instead of first.
multithreaded	A logical. If TRUE, sort using multiple threads.
in_place	If TRUE, this will set the flag mutably and return NULL. Remember to use <code>options(polars.strictly_immutable = FALSE)</code> before using this parameter, otherwise an error will occur. If FALSE (default), it will return a cloned Series with the flag.

## Value

[Series](#)

## Examples

```
as_polars_series(c(1.5, NA, 1, NaN, Inf, -Inf))$sort()
as_polars_series(c(1.5, NA, 1, NaN, Inf, -Inf))$sort(nulls_last = TRUE)
```



---

Series_std	<i>Compute the standard deviation of a Series</i>
------------	---

---

**Description**

Compute the standard deviation of a Series

**Usage**

```
Series_std(ddof = 1)
```

**Arguments**

ddof	Delta Degrees of Freedom: the divisor used in the calculation is $N - \text{ddof}$ , where $N$ represents the number of elements. By default ddof is 1.
------	---

**Value**

A numeric value

**Examples**

```
as_polars_series(1:10)$std()
```

---

Series_sub	<i>Subtract Series</i>
------------	------------------------

---

**Description**

Method equivalent of subtraction operator `series - other`.

**Usage**

```
Series_sub(other)
```

**Arguments**

other	<a href="#">Series</a> like object of numeric. Converted to <a href="#">Series</a> by <code>as_polars_series()</code> in this method.
-------	---

**Value**

[Series](#)

**See Also**

- [Arithmetic operators](#)

**Examples**

```

as_polars_series(1:3)$sub(11:13)
as_polars_series(1:3)$sub(as_polars_series(11:13))
as_polars_series(1:3)$sub(1L)
1L - as_polars_series(1:3)
as_polars_series(1:3) - 1L

```

---

Series_sum	<i>Compute the sum of a Series</i>
------------	------------------------------------

---

**Description**

Compute the sum of a Series

**Usage**

```
Series_sum()
```

**Details**

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

**Value**

A numeric value

**Examples**

```

as_polars_series(c(1:2, NA, 3, 5))$sum() # a NA is dropped always
as_polars_series(c(1:2, NA, 3, NaN, 4, Inf))$sum() # NaN poisons the result
as_polars_series(c(1:2, 3, Inf, 4, -Inf, 5))$sum() # Inf-Inf is NaN

```

---

Series_to_frame	<i>Convert Series to DataFrame</i>
-----------------	------------------------------------

---

**Description**

Convert Series to DataFrame

**Usage**

```
Series_to_frame()
```

**Value**

DataFrame

**Examples**

```
# default will be a DataFrame with empty name
as_polars_series(1:4)$to_frame()

as_polars_series(1:4, "bob")$to_frame()
```

Series\_to\_lit

*Convert a Series to literal***Description**

Convert a Series to literal

**Usage**

```
Series_to_lit()
```

**Value**

Expr

**Examples**

```
as_polars_series(list(1:1, 1:2, 1:3, 1:4))$
  print()$
  to_lit()$
  list$len()$
  sum()$
  cast(pl$dtypes$Int8)$
  to_series()
```

Series\_to\_r

*Convert Series to R vector or list***Description**

`$to_r()` automatically returns an R vector or list based on the Polars DataType. It is possible to force the output type by using `$to_vector()` or `$to_list()`.

**Usage**

```
Series_to_r(int64_conversion = polars_options()$int64_conversion)

Series_to_vector(int64_conversion = polars_options()$int64_conversion)

Series_to_list(int64_conversion = polars_options()$int64_conversion)
```

**Arguments**

int64\_conversion

How should Int64 values be handled when converting a polars object to R?

- "double" (default) converts the integer values to double.
- "bit64" uses `bit64::as.integer64()` to do the conversion (requires the package `bit64` to be attached).
- "string" converts Int64 values to character.

**Value**

R list or vector

**Conversion to R data types considerations**

When converting Polars objects, such as [DataFrames](#) to R objects, for example via the `as.data.frame()` generic function, each type in the Polars object is converted to an R type. In some cases, an error may occur because the conversion is not appropriate. In particular, there is a high possibility of an error when converting a [Datetime](#) type without a time zone. A [Datetime](#) type without a time zone in Polars is converted to the [POSIXct](#) type in R, which takes into account the time zone in which the R session is running (which can be checked with the `Sys.timezone()` function). In this case, if ambiguous times are included, a conversion error will occur. In such cases, change the session time zone using `Sys.setenv(TZ = "UTC")` and then perform the conversion, or use the `$dt$replace_time_zone()` method on the [Datetime](#) type column to explicitly specify the time zone before conversion.

```
# Due to daylight savings, clocks were turned forward 1 hour on Sunday, March 8, 2020, 2:00:00 am
# so this particular date-time doesn't exist
non_existent_time = as_polars_series("2020-03-08 02:00:00")$str$strptime(pl$Datetime(), "%F %T")
```

```
withr::with_timezone(
  "America/New_York",
  {
    tryCatch(
      # This causes an error due to the time zone (the `TZ` env var is affected).
      as.vector(non_existent_time),
      error = function(e) e
    )
  }
)
#> <error: in to_r: ComputeError(ErrString("datetime '2020-03-08 02:00:00' is non-existent in time zone
```

```
withr::with_timezone(
  "America/New_York",
  {
    # This is safe.
    as.vector(non_existent_time$dt$replace_time_zone("UTC"))
  }
)
```

```
#> [1] "2020-03-08 02:00:00 UTC"
```

## Examples

```
# Series with non-list type
series_vec = as_polars_series(letters[1:3])

series_vec$to_r() # as vector because Series DataType is not list (is String)
series_vec$to_list() # implicit call as.list(), convert to list
series_vec$to_vector() # implicit call unlist(), same as to_r() as already vector

# make a Series with nested lists
series_list = as_polars_series(
  list(
    list(c(1:5, NA_integer_)),
    list(1:2, NA_integer_)
  )
)
series_list

series_list$to_r() # as list because Series DataType is list
series_list$to_list() # implicit call as.list(), same as to_r() as already list
series_list$to_vector() # implicit call unlist(), append into a vector
```

---

Series\_value\_counts    *Count the occurrences of unique values*

---

## Description

Count the occurrences of unique values

## Usage

```
Series_value_counts(
  ...,
  sort = TRUE,
  parallel = FALSE,
  name = "count",
  normalize = FALSE
)
```

## Arguments

...	Ignored.
sort	Ensure the output is sorted from most values to least.
parallel	Better to turn this off in the aggregation context, as it can lead to contention.
name	Give the resulting count column a specific name. The default is "count" if normalize = FALSE and "proportion" if normalize = TRUE.
normalize	If TRUE, it gives relative frequencies of the unique values instead of their count.

**Value**

DataFrame

**Examples**

```
as_polars_series(iris$Species, name = "flower species")$value_counts()
```

---

Series\_var

*Compute the variance of a Series*

---

**Description**

Compute the variance of a Series

**Usage**

```
Series_var(ddof = 1)
```

**Arguments**

ddof                   Delta Degrees of Freedom: the divisor used in the calculation is  $N - \text{ddof}$ , where  $N$  represents the number of elements. By default ddof is 1.

**Value**

A numeric value

**Examples**

```
as_polars_series(1:10)$var()
```

---

show\_all\_public\_functions

*show all public functions / objects*

---

**Description**

print any object(function, RPolarsDataType) available via pl\$.

**Usage**

```
pl_show_all_public_functions()
```

**Examples**

```
pl$show_all_public_functions()
```

---

```
show_all_public_methods
      show all public methods
```

---

**Description**

methods are listed by their Class

**Usage**

```
pl_show_all_public_methods(class_names = NULL)
```

**Arguments**

class\_names      character vector of polars class names to show, Default NULL is all.

**Examples**

```
pl$show_all_public_methods()
```

---

```
SQLContext_class      Run SQL queries against DataFrame/LazyFrame data.
```

---

**Description**

Run SQL queries against [DataFrame/LazyFrame](#) data.

**Examples**

```
lf = pl$LazyFrame(a = 1:3, b = c("x", NA, "z"))

res = pl$SQLContext(frame = lf)$execute(
  "SELECT b, a*2 AS two_a FROM frame WHERE b IS NOT NULL"
)
res$collect()
```

---

SQLContext_execute	<i>Execute SQL query against the registered data</i>
--------------------	--

---

**Description**

Parse the given SQL query and execute it against the registered frame data.

**Usage**

```
SQLContext_execute(query)
```

**Arguments**

query	A character of the SQL query to execute.
-------	--

**Value**

A [LazyFrame](#)

**Examples**

```
query = "SELECT * FROM mtcars WHERE cyl = 4"
pl$SQLContext(mtcars = mtcars)$execute(query)
```

---

SQLContext_register	<i>Register a single data as a table</i>
---------------------	--

---

**Description**

Register a single frame as a table, using the given name.

**Usage**

```
SQLContext_register(name, frame)
```

**Arguments**

name	A string name to register the frame as.
frame	A <a href="#">LazyFrame</a> like object to register.

**Details**

If a table with the same name is already registered, it will be overwritten.



**Value**

Returns the [SQLContext](#) object invisibly.

**Examples**

```
ctx = pl$SQLContext()
ctx$register("mtcars", mtcars)

ctx$execute("SELECT * FROM mtcars LIMIT 5")$collect()
```

---

SQLContext\_register\_globals

*Register all polars DataFrames/LazyFrames found in the environment*

---

**Description**

Automatically maps variable names to table names.

**Usage**

```
SQLContext_register_globals(..., envir = parent.frame())
```

**Arguments**

...	Ignored.
envir	The environment to search for polars <a href="#">DataFrames/LazyFrames</a> .

**Details**

If a table with the same name is already registered, it will be overwritten.

**Value**

Returns the [SQLContext](#) object invisibly.

**See Also**

- [<SQLContext>\\$register\(\)](#)
- [<SQLContext>\\$register\\_many\(\)](#)
- [<SQLContext>\\$unregister\(\)](#)

**Examples**

```
df1 = pl$DataFrame(a = 1:3, b = c("x", NA, "z"))
df2 = pl$LazyFrame(a = 2:4, c = c("t", "w", "v"))

# Register frames directly from variables found in the current environment.
ctx = pl$SQLContext()$register_globals()
ctx$tables()

ctx$execute(
  "SELECT a, b, c FROM df1 LEFT JOIN df2 USING (a) ORDER BY a DESC"
)$collect()
```

---

SQLContext\_register\_many

*Register multiple data as tables*


---

**Description**

Register multiple frames as tables.

**Usage**

```
SQLContext_register_many(...)
```

**Arguments**

... Name-value pairs of [LazyFrame](#) like objects to register.

**Details**

If a table with the same name is already registered, it will be overwritten.

**Value**

Returns the [SQLContext](#) object invisibly.

**Examples**

```
ctx = pl$SQLContext()
r_df = mtcars
pl_df = pl$DataFrame(mtcars)
pl_lf = pl$LazyFrame(mtcars)

ctx$register_many(r_df = r_df, pl_df = pl_df, pl_lf = pl_lf)

ctx$execute(
  "SELECT * FROM r_df
  UNION ALL
```

```
SELECT * FROM p1_df  
UNION ALL  
SELECT * FROM p1_1f"  
)$collect()
```

---

SQLContext\_tables      *List registered tables*

---

### Description

Return a character vector of the registered table names.

### Usage

```
SQLContext_tables()
```

### Value

A character vector of the registered table names.

### Examples

```
ctx = pl$SQLContext()  
ctx$tables()  
ctx$register("df1", mtcars)  
ctx$tables()  
ctx$register("df2", mtcars)  
ctx$tables()
```

---

SQLContext\_unregister      *Unregister tables by name*

---

### Description

Unregister tables by name.

### Usage

```
SQLContext_unregister(names)
```

### Arguments

names                      A character vector of table names to unregister.

**Value**

Returns the [SQLContext](#) object invisibly.

**Examples**

```
# Initialise a new SQLContext and register the given tables.
ctx = pl$SQLContext(x = mtcars, y = mtcars, z = mtcars)
ctx$tables()

# Unregister some tables.
ctx$unregister(c("x", "y"))
ctx$tables()
```

---

sum.RPolarsDataFrame    *Compute the sum*

---

**Description**

Compute the sum

**Usage**

```
## S3 method for class 'RPolarsDataFrame'
sum(x, ...)

## S3 method for class 'RPolarsLazyFrame'
sum(x, ...)

## S3 method for class 'RPolarsSeries'
sum(x, ...)
```

**Arguments**

x	A <a href="#">DataFrame</a> , <a href="#">LazyFrame</a> , or <a href="#">Series</a>
...	Not used.

---

`unique.RPolarsDataFrame`*Drop duplicated rows*

---

**Description**

Drop duplicated rows

**Usage**

```
## S3 method for class 'RPolarsDataFrame'  
unique(x, incomparables = FALSE, subset = NULL, keep = "first", ...)
```

```
## S3 method for class 'RPolarsLazyFrame'  
unique(x, incomparables = FALSE, subset = NULL, keep = "first", ...)
```

**Arguments**

<code>x</code>	A <a href="#">DataFrame</a> or <a href="#">LazyFrame</a>
<code>incomparables</code>	Not used.
<code>subset</code>	Character vector of column names to drop duplicated values from.
<code>keep</code>	Either "first", "last", or "none".
<code>...</code>	Not used.

**Examples**

```
df = pl$DataFrame(  
  x = as.numeric(c(1, 1:5)),  
  y = as.numeric(c(1, 1:5)),  
  z = as.numeric(c(1, 1, 1:4))  
)  
unique(df)
```

---

`[.RPolarsDataFrame`      *Extract Parts of a Polars Object*

---

**Description**Mimics the behavior of `[x[i, j, drop = TRUE]][Extract]` for [data.frame](#) or R vector.

**Usage**

```
## S3 method for class 'RPolarsDataFrame'
x[i, j, drop = TRUE]

## S3 method for class 'RPolarsLazyFrame'
x[i, j, drop = TRUE]

## S3 method for class 'RPolarsSeries'
x[i]
```

**Arguments**

x	A <a href="#">DataFrame</a> , <a href="#">LazyFrame</a> , or <a href="#">Series</a>
i	Rows to select. Integer vector, logical vector, or an <a href="#">Expression</a> .
j	Columns to select. Integer vector, logical vector, character vector, or an <a href="#">Expression</a> . For <a href="#">LazyFrames</a> , only an <a href="#">Expression</a> can be used.
drop	Convert to a Polars Series if only one column is selected. For <a href="#">LazyFrames</a> , if the result has one column and drop = TRUE, an error will occur.

**Details**

`<Series>[i]` is equivalent to `pl$select(<Series>)[i, , drop = TRUE]`.

**See Also**

[<DataFrame>\\$select\(\)](#), [<LazyFrame>\\$select\(\)](#), [<DataFrame>\\$filter\(\)](#), [<LazyFrame>\\$filter\(\)](#)

**Examples**

```
df = pl$DataFrame(data.frame(a = 1:3, b = letters[1:3]))
lf = df$lazy()

# Select a row
df[1, ]

# If only `i` is specified, it is treated as `j`
# Select a column
df[1]

# Select a column by name (and convert to a Series)
df[, "b"]

# Can use Expression for filtering and column selection
lf[pl$col("a") >= 2, pl$col("b")$alias("new"), drop = FALSE] |>
  as.data.frame()
```

# Index

- !=.RPolarsSeries (Series\_compare), 538
- \* **DataFrame\_new**
  - LazyFrame\_collect, 377
  - LazyFrame\_collect\_in\_background, 379
- \* **DataFrame**
  - DataFrame\_class, 32
  - DataFrame\_describe, 36
  - DataFrame\_drop\_in\_place, 38
  - DataFrame\_drop\_nulls, 38
  - DataFrame\_dtype\_strings, 39
  - DataFrame\_equals, 39
  - DataFrame\_estimated\_size, 40
  - DataFrame\_explode, 41
  - DataFrame\_fill\_null, 42
  - DataFrame\_filter, 43
  - DataFrame\_first, 44
  - DataFrame\_get\_column, 45
  - DataFrame\_join, 52
  - DataFrame\_last, 58
  - DataFrame\_lazy, 58
  - DataFrame\_max, 59
  - DataFrame\_mean, 59
  - DataFrame\_median, 60
  - DataFrame\_min, 60
  - DataFrame\_n\_chunks, 61
  - DataFrame\_null\_count, 61
  - DataFrame\_pivot, 64
  - DataFrame\_quantile, 66
  - DataFrame\_rechunk, 66
  - DataFrame\_sample, 71
  - DataFrame\_select, 72
  - DataFrame\_sort, 75
  - DataFrame\_std, 77
  - DataFrame\_sum, 78
  - DataFrame\_to\_data\_frame, 79
  - DataFrame\_to\_series, 84
  - DataFrame\_to\_struct, 85
  - DataFrame\_transpose, 86
  - DataFrame\_unpivot, 88
  - DataFrame\_var, 90
  - DataFrame\_with\_columns, 90
  - DataFrame\_with\_row\_index, 92
  - GroupBy\_null\_count, 363
  - knit\_print.RPolarsDataFrame, 371
  - LazyFrame\_first, 387
  - pl\_select, 504
- \* **ExprDT**
  - ExprDT\_cast\_time\_unit, 138
  - ExprDT\_day, 140
  - ExprDT\_hour, 142
  - ExprDT\_iso\_year, 142
  - ExprDT\_minute, 145
  - ExprDT\_month, 145
  - ExprDT\_offset\_by, 147
  - ExprDT\_ordinal\_day, 148
  - ExprDT\_quarter, 149
  - ExprDT\_replace\_time\_zone, 150
  - ExprDT\_strftime, 153
  - ExprDT\_timestamp, 154
  - ExprDT\_week, 161
  - ExprDT\_weekday, 161
  - ExprDT\_with\_time\_unit, 162
  - ExprDT\_year, 163
- \* **ExprStr**
  - ExprStr\_decode, 200
  - ExprStr\_encode, 201
  - ExprStr\_ends\_with, 202
  - ExprStr\_extract, 203
  - ExprStr\_extract\_all, 203
  - ExprStr\_json\_decode, 209
  - ExprStr\_json\_path\_match, 209
  - ExprStr\_len\_bytes, 210
  - ExprStr\_len\_chars, 211
  - ExprStr\_pad\_end, 212
  - ExprStr\_pad\_start, 212
  - ExprStr\_slice, 217
  - ExprStr\_split\_exact, 219

- ExprStr\_splitn, 218
- ExprStr\_starts\_with, 220
- ExprStr\_strip\_chars, 220
- ExprStr\_to\_lowercase, 229
- ExprStr\_to\_titlecase, 230
- ExprStr\_to\_uppercase, 230
- ExprStr\_zfill, 231
- \* **Expr\_new**
  - pl\_coalesce, 442
  - pl\_concat\_list, 445
- \* **Expr**
  - pl\_element, 464
- \* **GroupBy**
  - GroupBy\_first, 359
  - GroupBy\_last, 360
  - GroupBy\_max, 360
  - GroupBy\_mean, 361
  - GroupBy\_median, 362
  - GroupBy\_min, 362
  - GroupBy\_quantile, 363
  - GroupBy\_std, 364
  - GroupBy\_sum, 365
  - GroupBy\_var, 366
- \* **LazyFrame\_new**
  - DataFrame\_lazy, 58
- \* **LazyFrame**
  - LazyFrame\_class, 373
  - LazyFrame\_collect, 377
  - LazyFrame\_collect\_in\_background, 379
  - LazyFrame\_explode, 382
  - LazyFrame\_fetch, 383
  - LazyFrame\_fill\_null, 386
  - LazyFrame\_filter, 386
  - LazyFrame\_group\_by, 388
  - LazyFrame\_last, 399
  - LazyFrame\_max, 399
  - LazyFrame\_mean, 400
  - LazyFrame\_median, 400
  - LazyFrame\_min, 401
  - LazyFrame\_print, 401
  - LazyFrame\_profile, 402
  - LazyFrame\_reverse, 405
  - LazyFrame\_sort, 418
  - LazyFrame\_std, 421
  - LazyFrame\_sum, 421
  - LazyFrame\_unpivot, 426
  - LazyFrame\_var, 427
- \* **RThreadHandle**
  - RThreadHandle\_is\_finished, 526
  - RThreadHandle\_join, 526
- \* **Series**
  - Series\_class, 534
  - Series\_map\_elements, 543
- \* **api\_object**
  - polars\_class\_object, 515
- \* **api**
  - pl\_pl, 483
- \* **docs**
  - docs\_translations, 116
- \* **functions**
  - pl\_is\_schema, 473
  - pl\_raw\_list, 485
  - show\_all\_public\_functions, 558
  - show\_all\_public\_methods, 559
- \* **options**
  - global\_rpool\_cap, 357
  - pl\_disable\_string\_cache, 461
  - pl\_enable\_string\_cache, 465
  - pl\_using\_string\_cache, 513
  - pl\_with\_string\_cache, 515
- \*.RPolarsExpr (S3\_arithmetic), 527
- \*.RPolarsSeries (S3\_arithmetic), 527
- + .RPolarsExpr (S3\_arithmetic), 527
- + .RPolarsSeries (S3\_arithmetic), 527
- .RPolarsExpr (S3\_arithmetic), 527
- .RPolarsSeries (S3\_arithmetic), 527
- / .RPolarsExpr (S3\_arithmetic), 527
- / .RPolarsSeries (S3\_arithmetic), 527
- <.RPolarsSeries (Series\_compare), 538
- <=.RPolarsSeries (Series\_compare), 538
- <DataFrame>\$filter(), 566
- <DataFrame>\$get\_columns(), 83
- <DataFrame>\$group\_by(), 63, 359, 521
- <DataFrame>\$group\_by\_dynamic(), 71, 120
- <DataFrame>\$head(), 367
- <DataFrame>\$lazy(), 373
- <DataFrame>\$n\_chunks(), 67
- <DataFrame>\$partition\_by(), 47
- <DataFrame>\$rechunk(), 62
- <DataFrame>\$rolling(), 50, 523
- <DataFrame>\$select(), 566
- <DataFrame>\$tail(), 367
- <DataFrame>\$to\_data\_frame(), 19
- <DataFrame>\$to\_list(), 45
- <DataFrame>\$to\_raw\_ipc(), 95



- <DataFrame>\$write\_ipc(), 84
- <Expr>\$add(), 528
- <Expr>\$approx\_n\_unique(), 440
- <Expr>\$count(), 448, 475
- <Expr>\$div(), 269, 528
- <Expr>\$first(), 468
- <Expr>\$floor\_div(), 255, 294, 528
- <Expr>\$head(), 470
- <Expr>\$last(), 474
- <Expr>\$map\_batches(), 379, 525
- <Expr>\$map\_elements(), 379, 525
- <Expr>\$max(), 477
- <Expr>\$mean(), 479
- <Expr>\$median(), 480
- <Expr>\$min(), 482
- <Expr>\$mod(), 269, 528
- <Expr>\$mul(), 528
- <Expr>\$n\_unique(), 483
- <Expr>\$pow(), 528
- <Expr>\$std(), 507
- <Expr>\$str\$contains(), 199
- <Expr>\$str\$replace(), 215
- <Expr>\$str\$replace\_all(), 214
- <Expr>\$str\$strptime(), 226, 228, 230
- <Expr>\$str\$to\_date(), 224
- <Expr>\$str\$to\_datetime(), 224
- <Expr>\$str\$to\_time(), 224
- <Expr>\$sub(), 528
- <Expr>\$sum(), 509
- <Expr>\$tail(), 511
- <Expr>\$var(), 514
- <LazyFrame>\$collect(), 373
- <LazyFrame>\$collect\_in\_background(), 525
- <LazyFrame>\$fetch(), 367
- <LazyFrame>\$filter(), 566
- <LazyFrame>\$group\_by(), 431, 521
- <LazyFrame>\$group\_by\_dynamic(), 407
- <LazyFrame>\$head(), 367, 422
- <LazyFrame>\$rolling(), 391
- <LazyFrame>\$select(), 566
- <LazyFrame>\$serialize(), 461
- <LazyFrame>\$tail(), 367
- <RPolarsRThreadHandle>\$is\_finished(), 379
- <RPolarsRThreadHandle>\$join(), 379
- <RThreadHandle>\$join(), 526
- <SQLContext>\$register(), 561
- <SQLContext>\$register\_many(), 561
- <SQLContext>\$unregister(), 561
- <Series>\$add(), 528
- <Series>\$div(), 528
- <Series>\$floor\_div(), 528
- <Series>\$mod(), 528
- <Series>\$mul(), 528
- <Series>\$pow(), 528
- <Series>\$sub(), 528
- ==.RPolarsSeries (Series\_compare), 538
- >.RPolarsSeries (Series\_compare), 538
- >=.RPolarsSeries (Series\_compare), 538
- ?pl\_col, 440, 448, 467, 470, 471, 474, 477, 478, 480, 481, 483, 507, 509, 511, 514
- ?pl\_thread\_pool\_size, 521
- [.RPolarsDataFrame, 565
- [.RPolarsLazyFrame ( [.RPolarsDataFrame), 565
- [.RPolarsSeries ( [.RPolarsDataFrame), 565
- [.rpolars\_raw\_list (pl\_raw\_list), 485
- \$agg(), 359
- \$alias(), 191
- \$arg\_sort(), 441
- \$cast(), 506
- \$collect(), 26, 385, 403
- \$collect\_in\_background(), 378, 385, 403
- \$convert\_time\_zone(), 150
- \$cut(), 306
- \$dt\$replace\_time\_zone(), 19, 21, 33, 80, 82, 374, 535, 556
- \$fetch(), 26, 378, 392, 403
- \$fill\_nan(), 506
- \$flags, 551
- \$fold(), 189
- \$group\_by(), 63
- \$lazy(), 27
- \$list\$gather(), 171
- \$list\$get(), 169
- \$map\_batches(), 288, 379
- \$map\_elements(), 379, 409
- \$meta\$eq(), 187
- \$meta\$neq(), 186
- \$meta\$output\_name(), 190
- \$name\$keep(), 191
- \$prefix(), 193
- \$profile(), 378, 385

- \$qcut(), 253
- \$replace(), 311
- \$replace\_strict(), 310
- \$set\_difference(), 179
- \$set\_sorted(), 542
- \$set\_symmetric\_difference(), 178
- \$sink\_ipc(), 378, 385, 403
- \$sink\_parquet(), 378, 385, 403
- \$str\$contains(), 207
- \$str\$ends\_with(), 198, 207
- \$str\$find(), 198
- \$str\$start\_with(), 198, 207
- \$struct\$with\_fields(), 465
- \$suffix(), 192
- \$with\_columns(), 197
- %%.RPolarsExpr (S3\_arithmetic), 527
- %%.RPolarsSeries (S3\_arithmetic), 527
- %.RPolarsExpr (S3\_arithmetic), 527
- %.RPolarsSeries (S3\_arithmetic), 527
- ^.RPolarsExpr (S3\_arithmetic), 527
- ^.RPolarsSeries (S3\_arithmetic), 527
- A Polars DataFrame, 22, 30
- a polars Series, 27, 29
- abs(n), 51, 78
- agg (GroupBy\_agg), 358
- apply (Series\_map\_elements), 543
- arg\_unique (Expr\_arg\_unique), 242
- Arithmetic operators, 232, 255, 269, 294, 296, 304, 347, 529, 540, 541, 547, 549, 553
- Array, 313
- as.character.RPolarsSeries, 17
- as.data.frame(), 19, 21, 33, 80, 82, 373, 535, 556
- as.data.frame.RPolarsDataFrame, 17
- as.data.frame.RPolarsLazyFrame (as.data.frame.RPolarsDataFrame), 17
- as.list.rpolars\_raw\_list (pl\_raw\_list), 485
- as.matrix.RPolarsDataFrame, 20
- as.matrix.RPolarsLazyFrame (as.matrix.RPolarsDataFrame), 20
- as.vector.RPolarsSeries, 20
- as\_arrow\_table.RPolarsDataFrame, 21
- as\_nanoarrow\_array\_stream.RPolarsDataFrame, 22
- as\_nanoarrow\_array\_stream.RPolarsSeries (as\_nanoarrow\_array\_stream.RPolarsDataFrame), 22
- as\_polars\_df, 23
- as\_polars\_df(), 19, 23, 450
- as\_polars\_df(x, ...), 27
- as\_polars\_lf, 27
- as\_polars\_lf(), 27
- as\_polars\_series, 27
- as\_polars\_series(), 27, 505, 506, 529, 539, 541, 546, 547, 549, 553
- as\_polars\_series(1), 506
- as\_record\_batch\_reader.RPolarsDataFrame, 29
- base::OlsonNames(), 140, 150
- Binary, 486
- c.RPolarsSeries, 30
- cast, 505
- Categorical, 103, 253, 306
- ChainedThen (Expr\_when\_then\_otherwise), 355
- ChainedThen\_otherwise (Expr\_when\_then\_otherwise), 355
- ChainedThen\_when (Expr\_when\_then\_otherwise), 355
- ChainedWhen (Expr\_when\_then\_otherwise), 355
- ChainedWhen\_then (Expr\_when\_then\_otherwise), 355
- data type, 32, 33, 373, 466, 467, 534
- data type List, 130
- data.frame, 18, 450, 565
- DataFrame, 26, 35, 51, 52, 68, 76, 79, 116, 118, 120, 197, 245, 300, 359, 366, 434–437, 450, 489, 490, 494, 505, 518, 523, 524, 559, 564–566
- DataFrame\_cast, 31
- DataFrame\_class, 32
- DataFrame\_clear, 34
- DataFrame\_clone, 35
- DataFrame\_describe, 36
- DataFrame\_drop, 37
- DataFrame\_drop\_in\_place, 38
- DataFrame\_drop\_nulls, 38
- DataFrame\_dtype\_strings, 39
- DataFrame\_equals, 39

- DataFrame\_estimated\_size, 40
- DataFrame\_explode, 41
- DataFrame\_fill\_nan, 42
- DataFrame\_fill\_null, 42
- DataFrame\_filter, 43
- DataFrame\_first, 44
- DataFrame\_gather\_every, 44
- DataFrame\_get\_column, 45
- DataFrame\_get\_columns, 45
- DataFrame\_glimpse, 46
- DataFrame\_group\_by, 47
- DataFrame\_group\_by\_dynamic, 48
- DataFrame\_head, 51
- DataFrame\_item, 52
- DataFrame\_join, 52
- DataFrame\_join\_asof, 54
- DataFrame\_join\_where, 57
- DataFrame\_last, 58
- DataFrame\_lazy, 58
- DataFrame\_limit (DataFrame\_head), 51
- DataFrame\_max, 59
- DataFrame\_mean, 59
- DataFrame\_median, 60
- DataFrame\_min, 60
- DataFrame\_n\_chunks, 61
- DataFrame\_null\_count, 61
- DataFrame\_partition\_by, 63
- DataFrame\_pivot, 64
- DataFrame\_quantile, 66
- DataFrame\_rechunk, 66
- DataFrame\_rename, 68
- DataFrame\_reverse, 69
- DataFrame\_rolling, 69
- DataFrame\_sample, 71
- DataFrame\_select, 72
- DataFrame\_select\_seq, 73
- DataFrame\_shift, 74
- DataFrame\_slice, 74
- DataFrame\_sort, 75
- DataFrame\_sql, 76
- DataFrame\_std, 77
- DataFrame\_sum, 78
- DataFrame\_tail, 78
- DataFrame\_to\_data\_frame, 79
- DataFrame\_to\_dummies, 80
- DataFrame\_to\_list, 81
- DataFrame\_to\_raw\_ipc, 83
- DataFrame\_to\_series, 84
- DataFrame\_to\_struct, 85
- DataFrame\_transpose, 86
- DataFrame\_unique, 87
- DataFrame\_unnest, 88
- DataFrame\_unpivot, 88
- DataFrame\_var, 90
- DataFrame\_with\_columns, 90
- DataFrame\_with\_columns\_seq, 91
- DataFrame\_with\_row\_index, 92
- DataFrame\_write\_csv, 93
- DataFrame\_write\_ipc, 94
- DataFrame\_write\_json, 95
- DataFrame\_write\_ndjson, 96
- DataFrame\_write\_parquet, 97
- DataFrames, 19, 21, 33, 63, 76, 80, 82, 373, 419, 420, 535, 556, 561
- DataType, 466
- DataType\_Array, 99
- DataType\_Categorical, 99
- DataType\_contains\_categoricals, 100
- DataType\_contains\_views, 101
- DataType\_Datetime, 101
- DataType\_Duration, 102
- DataType\_Enum, 103
- DataType\_is\_array, 104
- DataType\_is\_binary, 104
- DataType\_is\_bool, 105
- DataType\_is\_categorical, 105
- DataType\_is\_enum, 106
- DataType\_is\_float, 106
- DataType\_is\_integer, 107
- DataType\_is\_known, 107
- DataType\_is\_list, 108
- DataType\_is\_logical, 108
- DataType\_is\_nested, 109
- DataType\_is\_null, 109
- DataType\_is\_numeric, 110
- DataType\_is\_ord, 110
- DataType\_is\_primitive, 111
- DataType\_is\_signed\_integer, 111
- DataType\_is\_string, 112
- DataType\_is\_struct, 112
- DataType\_is\_temporal, 113
- DataType\_is\_unsigned\_integer, 113
- DataType\_List, 114
- DataType\_Struct, 114
- Date, 318, 320, 323, 325, 327, 331, 333, 335
- Datetime, 19, 21, 33, 80, 82, 227, 228, 318,

- 320, 323, 325, 327, 331, 333, 335,  
373, 454, 456, 458–460, 469, 535,  
556
- difftime, 454, 456, 458, 459
- dim.RPolarsDataFrame, 115
- dim.RPolarsLazyFrame  
(dim.RPolarsDataFrame), 115
- dimnames.RPolarsDataFrame, 116
- dimnames.RPolarsLazyFrame  
(dimnames.RPolarsDataFrame),  
116
- docs\_translations, 30, 116, 308
- dplyr::filter(), 43, 386
- DynamicGroupBy\_agg, 118
- DynamicGroupBy\_class, 120
- DynamicGroupBy\_ungroup, 120
- element (pl\_element), 464
- exclude (Expr\_exclude), 262
- Expr, 124, 130, 133, 134, 136, 144, 146, 152,  
171, 197–200, 202, 206, 213, 215,  
220, 223, 226, 228, 229, 232, 235,  
255, 257, 258, 268, 271, 286, 287,  
294, 295, 297, 298, 300, 304, 313,  
347, 355–357, 440, 443, 454, 456,  
458, 460, 467, 470, 471, 474, 477,  
479–481, 483, 507, 509, 511, 514,  
535, 555
- Expr\_abs, 232
- Expr\_add, 232
- Expr\_agg\_groups, 233
- Expr\_alias, 234
- Expr\_all, 234
- Expr\_and, 235
- Expr\_any, 236
- Expr\_append, 236
- Expr\_approx\_n\_unique, 237
- Expr\_arccos, 238
- Expr\_arccosh, 238
- Expr\_arcsin, 239
- Expr\_arcsinh, 239
- Expr\_arctan, 240
- Expr\_arctanh, 240
- Expr\_arg\_max, 241
- Expr\_arg\_min, 241
- Expr\_arg\_sort, 242
- Expr\_arg\_unique, 242
- Expr\_backward\_fill, 243
- Expr\_bottom\_k, 243
- Expr\_cast, 244
- Expr\_ceil, 245
- Expr\_class, 245
- Expr\_clip, 246
- Expr\_cos, 247
- Expr\_cosh, 247
- Expr\_count, 248
- Expr\_cum\_count, 249
- Expr\_cum\_max, 250
- Expr\_cum\_min, 251
- Expr\_cum\_prod, 251
- Expr\_cum\_sum, 252
- Expr\_cumulative\_eval, 248
- Expr\_cut, 253
- Expr\_diff, 254
- Expr\_div, 254
- Expr\_dot, 255
- Expr\_drop\_nans, 256
- Expr\_drop\_nulls, 256
- Expr\_entropy, 257
- Expr\_eq, 257, 258
- Expr\_eq\_missing, 257, 258
- Expr\_ewm\_mean, 259
- Expr\_ewm\_std, 260
- Expr\_ewm\_var, 261
- Expr\_exclude, 262
- Expr\_exp, 263
- Expr\_explode, 264
- Expr\_extend\_constant, 264
- Expr\_fill\_nan, 265
- Expr\_fill\_null, 265
- Expr\_filter, 266
- Expr\_first, 267
- Expr\_flatten, 267
- Expr\_floor, 268
- Expr\_floor\_div, 268
- Expr\_forward\_fill, 269
- Expr\_gather, 270
- Expr\_gather\_every, 270
- Expr\_gt, 271
- Expr\_gt\_eq, 271
- Expr\_has\_nulls, 272
- Expr\_hash, 272
- Expr\_head, 273
- Expr\_implode, 273
- Expr\_inspect, 274
- Expr\_interpolate, 275
- Expr\_is\_between, 276

- Expr\_is\_duplicated, 277
- Expr\_is\_finite, 277
- Expr\_is\_first\_distinct, 278
- Expr\_is\_in, 278
- Expr\_is\_infinite, 279
- Expr\_is\_last\_distinct, 280
- Expr\_is\_nan, 280
- Expr\_is\_not\_nan, 281
- Expr\_is\_not\_null, 281
- Expr\_is\_null, 282
- Expr\_is\_unique, 282
- Expr\_kurtosis, 283
- Expr\_last, 283
- Expr\_len (Expr\_count), 248
- Expr\_limit, 284
- Expr\_log, 284
- Expr\_log10, 285
- Expr\_lower\_bound, 285
- Expr\_lt, 286
- Expr\_lt\_eq, 286
- Expr\_map\_batches, 287
- Expr\_map\_elements, 289
- Expr\_max, 292
- Expr\_mean, 293
- Expr\_median, 293
- Expr\_min, 294
- Expr\_mod, 294
- Expr\_mode, 295
- Expr\_mul, 295
- Expr\_n\_unique, 299
- Expr\_nan\_max, 296
- Expr\_nan\_min, 296
- Expr\_neq, 297, 298
- Expr\_neq\_missing, 297, 297
- Expr\_not, 298
- Expr\_null\_count, 299
- Expr\_or, 300
- Expr\_over, 300
- Expr\_pct\_change, 302
- Expr\_peak\_max, 303
- Expr\_peak\_min, 303
- Expr\_pow, 304
- Expr\_product, 305
- Expr\_qcut, 305
- Expr\_quantile, 306
- Expr\_rank, 307
- Expr\_rechunk, 308
- Expr\_reinterpret, 309
- Expr\_rep, 309
- Expr\_repeat\_by, 310
- Expr\_replace, 310
- Expr\_replace\_strict, 311
- Expr\_reshape, 313
- Expr\_reverse, 314
- Expr\_rle, 314
- Expr\_rle\_id, 315
- Expr\_rolling, 315
- Expr\_rolling\_max, 317
- Expr\_rolling\_max\_by, 318
- Expr\_rolling\_mean, 319
- Expr\_rolling\_mean\_by, 320
- Expr\_rolling\_median, 321
- Expr\_rolling\_median\_by, 322
- Expr\_rolling\_min, 324
- Expr\_rolling\_min\_by, 325
- Expr\_rolling\_quantile, 326
- Expr\_rolling\_quantile\_by, 327
- Expr\_rolling\_skew, 328
- Expr\_rolling\_std, 329
- Expr\_rolling\_std\_by, 330
- Expr\_rolling\_sum, 332
- Expr\_rolling\_sum\_by, 333
- Expr\_rolling\_var, 334
- Expr\_rolling\_var\_by, 335
- Expr\_round, 336
- Expr\_sample, 337
- Expr\_search\_sorted, 338
- Expr\_set\_sorted, 339
- Expr\_shift, 339
- Expr\_shrink\_dtype, 340
- Expr\_shuffle, 341
- Expr\_sign, 341
- Expr\_sin, 342
- Expr\_sinh, 342
- Expr\_skew, 343
- Expr\_slice, 343
- Expr\_sort, 344
- Expr\_sort\_by, 345
- Expr\_sqrt, 346
- Expr\_std, 346
- expr\_str\_strip\_chars
  - (ExprStr\_strip\_chars), 220
- expr\_str\_strip\_chars\_end
  - (ExprStr\_strip\_chars\_end), 221
- expr\_str\_strip\_chars\_start
  - (ExprStr\_strip\_chars\_start),

- 222
- expr\_str\_zfill (ExprStr\_zfill), 231
- Expr\_sub, 347
- Expr\_sum, 348
- Expr\_tail, 348
- Expr\_tan, 349
- Expr\_tanh, 349
- Expr\_to\_physical, 350
- Expr\_to\_r, 351
- Expr\_to\_series, 352
- Expr\_top\_k, 350
- Expr\_unique, 352
- Expr\_unique\_counts, 353
- Expr\_upper\_bound, 353
- Expr\_value\_counts, 354
- Expr\_var, 354
- Expr\_when\_then\_otherwise, 355
- Expr\_xor, 357
- ExprArr\_all, 121
- ExprArr\_any, 121
- ExprArr\_arg\_max, 122
- ExprArr\_arg\_min, 122
- ExprArr\_contains, 123
- ExprArr\_get, 124
- ExprArr\_join, 124
- ExprArr\_max, 125
- ExprArr\_median, 126
- ExprArr\_min, 126
- ExprArr\_reverse, 127
- ExprArr\_shift, 127
- ExprArr\_sort, 128
- ExprArr\_std, 129
- ExprArr\_sum, 129
- ExprArr\_to\_list, 130
- ExprArr\_to\_struct, 130
- ExprArr\_unique, 131
- ExprArr\_var, 132
- ExprBin\_contains, 132
- ExprBin\_decode, 133
- ExprBin\_encode, 134
- ExprBin\_ends\_with, 135
- ExprBin\_size, 135
- ExprBin\_starts\_with, 136
- ExprCat\_get\_categories, 137
- ExprCat\_set\_ordering, 137
- ExprDT\_cast\_time\_unit, 138
- ExprDT\_combine, 139
- ExprDT\_convert\_time\_zone, 140
- ExprDT\_day, 140
- ExprDT\_epoch, 141
- ExprDT\_hour, 142
- ExprDT\_is\_leap\_year, 143
- ExprDT\_iso\_year, 142
- ExprDT\_microsecond, 143
- ExprDT\_millisecond, 144
- ExprDT\_minute, 145
- ExprDT\_month, 145
- ExprDT\_nanosecond, 146
- ExprDT\_offset\_by, 147
- ExprDT\_ordinal\_day, 148
- ExprDT\_quarter, 149
- ExprDT\_replace\_time\_zone, 150
- ExprDT\_round, 151
- ExprDT\_second, 152
- ExprDT\_strftime, 153
- ExprDT\_time, 154
- ExprDT\_timestamp, 154
- ExprDT\_total\_days, 155
- ExprDT\_total\_hours, 156
- ExprDT\_total\_microseconds, 156
- ExprDT\_total\_milliseconds, 157
- ExprDT\_total\_minutes, 158
- ExprDT\_total\_nanoseconds, 158
- ExprDT\_total\_seconds, 159
- ExprDT\_truncate, 160
- ExprDT\_week, 161
- ExprDT\_weekday, 161
- ExprDT\_with\_time\_unit, 162
- ExprDT\_year, 163
- Expression, 228, 448, 475, 566
- expression, 47, 388
- Expressions, 505
- ExprList\_all, 163
- ExprList\_any, 164
- ExprList\_arg\_max, 164
- ExprList\_arg\_min, 165
- ExprList\_concat, 165
- ExprList\_contains, 166
- ExprList\_diff, 167
- ExprList\_eval, 167
- ExprList\_explode, 168
- ExprList\_first, 169
- ExprList\_gather, 169
- ExprList\_gather\_every, 170
- ExprList\_get, 171
- ExprList\_head, 172

- ExprList\_join, 172
- ExprList\_last, 173
- ExprList\_len, 174
- ExprList\_max, 174
- ExprList\_mean, 175
- ExprList\_min, 175
- ExprList\_n\_unique, 176
- ExprList\_reverse, 176
- ExprList\_sample, 177
- ExprList\_set\_difference, 178
- ExprList\_set\_intersection, 178
- ExprList\_set\_symmetric\_difference, 179
- ExprList\_set\_union, 180
- ExprList\_shift, 181
- ExprList\_slice, 181
- ExprList\_sort, 182
- ExprList\_sum, 183
- ExprList\_tail, 183
- ExprList\_to\_struct, 184
- ExprList\_unique, 185
- ExprMeta\_eq, 186
- ExprMeta\_has\_multiple\_outputs, 186
- ExprMeta\_is\_regex\_projection, 187
- ExprMeta\_neq, 187
- ExprMeta\_output\_name, 188
- ExprMeta\_pop, 189
- ExprMeta\_root\_names, 190
- ExprMeta\_tree\_format, 190
- ExprMeta\_undo\_aliases, 191
- ExprName\_keep, 191
- ExprName\_prefix, 192
- ExprName\_prefix\_fields, 192
- ExprName\_suffix, 193
- ExprName\_suffix\_fields, 194
- ExprName\_to\_lowercase, 194
- ExprName\_to\_uppercase, 195
- ExprStr\_contains, 198
- ExprStr\_contains\_any, 199
- ExprStr\_count\_matches, 200
- ExprStr\_decode, 200
- ExprStr\_encode, 201
- ExprStr\_ends\_with, 202
- ExprStr\_extract, 203
- ExprStr\_extract\_all, 203
- ExprStr\_extract\_groups, 204
- ExprStr\_extract\_many, 205
- ExprStr\_find, 206
- ExprStr\_head, 207
- ExprStr\_join, 208
- ExprStr\_json\_decode, 209
- ExprStr\_json\_path\_match, 209
- ExprStr\_len\_bytes, 210
- ExprStr\_len\_chars, 211
- ExprStr\_pad\_end, 212
- ExprStr\_pad\_start, 212
- ExprStr\_replace, 213
- ExprStr\_replace\_all, 214
- ExprStr\_replace\_many, 216
- ExprStr\_reverse, 217
- ExprStr\_slice, 217
- ExprStr\_split, 218
- ExprStr\_split\_exact, 219
- ExprStr\_splitn, 218
- ExprStr\_starts\_with, 220
- ExprStr\_strip\_chars, 220
- ExprStr\_strip\_chars\_end, 221
- ExprStr\_strip\_chars\_start, 222
- ExprStr\_strptime, 222
- ExprStr\_tail, 225
- ExprStr\_to\_date, 226
- ExprStr\_to\_datetime, 227
- ExprStr\_to\_integer, 228
- ExprStr\_to\_lowercase, 229
- ExprStr\_to\_time, 229
- ExprStr\_to\_titlecase, 230
- ExprStr\_to\_uppercase, 230
- ExprStr\_zfill, 231
- ExprStruct\_field, 195
- ExprStruct\_rename\_fields, 196
- ExprStruct\_with\_fields, 197
- global\_rpool\_cap, 357
- GroupBy, 47, 49
- GroupBy\_agg, 358
- GroupBy\_class, 359
- GroupBy\_first, 359
- GroupBy\_last, 360
- GroupBy\_max, 360
- GroupBy\_mean, 361
- GroupBy\_median, 362
- GroupBy\_min, 362
- GroupBy\_null\_count, 363
- GroupBy\_quantile, 363
- GroupBy\_shift, 364
- GroupBy\_std, 364
- GroupBy\_sum, 365
- GroupBy\_ungroup, 365



- GroupBy\_var, 366
- hash (Expr\_hash), 272
- head.RPolarsDataFrame, 366
- head.RPolarsLazyFrame  
(head.RPolarsDataFrame), 366
- ifelse(), 355
- infer\_nanoarrow\_schema.RPolarsDataFrame,  
368
- infer\_nanoarrow\_schema.RPolarsSeries  
(infer\_nanoarrow\_schema.RPolarsDataFrame),  
368
- is\_finished(), 525
- is\_infinite (Expr\_is\_infinite), 279
- is\_nan (Expr\_is\_nan), 280
- is\_not\_nan (Expr\_is\_not\_nan), 281
- is\_polars\_df, 369
- is\_polars\_dtype, 369
- is\_polars\_lf, 370
- is\_polars\_series, 370
- join(), 525
- knit\_print.RPolarsDataFrame, 371
- lapply(), 63
- lazy (DataFrame\_lazy), 58
- LazyFrame, 26, 27, 116, 245, 367, 376, 405,  
418, 420, 434–437, 461, 474, 499,  
500, 504, 506, 559, 560, 562,  
564–566
- LazyFrame\_cast, 372
- LazyFrame\_class, 373
- LazyFrame\_clear, 375
- LazyFrame\_clone, 376
- LazyFrame\_collect, 377
- LazyFrame\_collect\_in\_background, 379
- LazyFrame\_drop, 380
- LazyFrame\_drop\_nulls, 380
- LazyFrame\_explain, 381
- LazyFrame\_explode, 382
- LazyFrame\_fetch, 383
- LazyFrame\_fill\_nan, 385
- LazyFrame\_fill\_null, 386
- LazyFrame\_filter, 386
- LazyFrame\_first, 387
- LazyFrame\_gather\_every, 387
- LazyFrame\_group\_by, 388
- LazyFrame\_group\_by\_dynamic, 389
- LazyFrame\_head, 392
- LazyFrame\_join, 393
- LazyFrame\_join\_asof, 395
- LazyFrame\_join\_where, 398
- LazyFrame\_last, 399
- LazyFrame\_max, 399
- LazyFrame\_mean, 400
- LazyFrame\_median, 400
- LazyFrame\_min, 401
- LazyFrame\_print, 401
- LazyFrame\_profile, 402
- LazyFrame\_quantile, 404
- LazyFrame\_rename, 404
- LazyFrame\_reverse, 405
- LazyFrame\_rolling, 406
- LazyFrame\_select, 408
- LazyFrame\_select\_seq, 409
- LazyFrame\_serialize, 409
- LazyFrame\_shift, 410
- LazyFrame\_sink\_csv, 411
- LazyFrame\_sink\_ipc, 413
- LazyFrame\_sink\_ndjson, 414
- LazyFrame\_sink\_parquet, 416
- LazyFrame\_slice, 418
- LazyFrame\_sort, 418
- LazyFrame\_sql, 419
- LazyFrame\_std, 421
- LazyFrame\_sum, 421
- LazyFrame\_tail, 422
- LazyFrame\_to\_dot, 422
- LazyFrame\_unique, 424
- LazyFrame\_unnest, 425
- LazyFrame\_unpivot, 426
- LazyFrame\_var, 427
- LazyFrame\_with\_columns, 427
- LazyFrame\_with\_columns\_seq, 428
- LazyFrame\_with\_context, 429
- LazyFrame\_with\_row\_index, 430
- LazyFrames, 76, 419, 420, 561
- LazyGroupBy, 388, 391, 407
- LazyGroupBy\_agg, 431
- LazyGroupBy\_class, 431
- LazyGroupBy\_head, 432
- LazyGroupBy\_print, 432
- LazyGroupBy\_tail, 433
- LazyGroupBy\_ungroup, 433
- length.RPolarsDataFrame, 434



- length.RPolarsLazyFrame
  - (length.RPolarsDataFrame), 434
- length.RPolarsSeries
  - (length.RPolarsDataFrame), 434
- List, 313
- list, 63
- matrix, 20
- max.RPolarsDataFrame, 434
- max.RPolarsLazyFrame
  - (max.RPolarsDataFrame), 434
- max.RPolarsSeries
  - (max.RPolarsDataFrame), 434
- mean.RPolarsDataFrame, 435
- mean.RPolarsLazyFrame
  - (mean.RPolarsDataFrame), 435
- mean.RPolarsSeries
  - (mean.RPolarsDataFrame), 435
- median.RPolarsDataFrame, 435
- median.RPolarsLazyFrame
  - (median.RPolarsDataFrame), 435
- median.RPolarsSeries
  - (median.RPolarsDataFrame), 435
- min.RPolarsDataFrame, 436
- min.RPolarsLazyFrame
  - (min.RPolarsDataFrame), 436
- min.RPolarsSeries
  - (min.RPolarsDataFrame), 436
- na.omit.RPolarsDataFrame
  - (na.omit.RPolarsLazyFrame), 436
- na.omit.RPolarsLazyFrame, 436
- names.RPolarsDataFrame, 437
- names.RPolarsGroupBy
  - (names.RPolarsDataFrame), 437
- names.RPolarsLazyFrame
  - (names.RPolarsDataFrame), 437
- names.RPolarsLazyGroupBy
  - (names.RPolarsDataFrame), 437
- Number of threads used by Polars, 520
- OlsonNames(), 101, 453
- options(), 512, 521
- otherwise (Expr\_when\_then\_otherwise), 355
- pl (pl\_pl), 483
- pl\$all(), 234
- pl\$arg\_sort\_by(), 242
- pl\$col(), 37, 63, 380, 440, 448, 467, 470, 471, 474, 477, 478, 480, 481, 483, 507, 509, 511, 514
- pl\$date(), 453, 513
- pl\$date\_range(), 460
- pl\$date\_ranges(), 459
- pl\$Datetime(), 223
- pl\$datetime(), 451, 513
- pl\$Datetime(ms), 223, 227, 469
- pl\$Datetime(ns), 469
- pl\$Datetime(us), 469
- pl\$datetime\_range(), 457
- pl\$datetime\_ranges(), 455
- pl\$deserialize\_lf(), 410
- pl\$disable\_string\_cache, 465
- pl\$enable\_enable\_cache, 513, 515
- pl\$enable\_string\_cache, 462
- pl\$enable\_string\_cache(), 445
- pl\$Field(), 114
- pl\$field(), 197
- pl\$fold(), 494
- pl\$int\_range(), 472
- pl\$int\_ranges(), 471
- pl\$LazyFrame(), 373
- pl\$len(), 448
- pl\$lit(), 469
- pl\$max\_horizontal(), 477
- pl\$mean\_horizontal(), 479
- pl\$min\_horizontal(), 482
- pl\$reduce(), 468
- pl\$sum\_horizontal(), 509
- pl\$time(), 451, 453
- pl\$using\_string\_cache, 462, 465, 515
- pl\$with\_string\_cache, 462, 465, 513
- pl\_all, 437
- pl\_all\_horizontal, 438
- pl\_any\_horizontal, 439
- pl\_approx\_n\_unique, 439
- pl\_arg\_sort\_by, 440
- pl\_arg\_where, 441
- pl\_coalesce, 442
- pl\_col, 443
- pl\_concat, 444
- pl\_concat\_list, 445
- pl\_concat\_str, 446
- pl\_corr, 447
- pl\_count, 448
- pl\_cov, 449

- pl\_DataFrame, 450
- pl\_date, 451
- pl\_date\_range, 458
- pl\_date\_ranges, 459
- pl\_Datetime (DataType\_Datetime), 101
- pl\_datetime, 452
- pl\_datetime\_range, 454
- pl\_datetime\_ranges, 456
- pl\_deserialize\_lf, 461
- pl\_disable\_string\_cache, 461
- pl\_dtypes, 462
- pl\_duration, 463
- pl\_element, 464
- pl\_enable\_string\_cache, 465
- pl\_Field (pl\_Field\_class), 466
- pl\_field, 465
- pl\_Field\_class, 466
- pl\_first, 467
- pl\_fold, 468
- pl\_from\_epoch, 469
- pl\_get\_global\_rpool\_cap  
(global\_rpool\_cap), 357
- pl\_head, 470
- pl\_implode, 470
- pl\_int\_range, 471
- pl\_int\_ranges, 472
- pl\_is\_schema, 473
- pl\_last, 473
- pl\_LazyFrame, 474
- pl\_len, 475
- pl\_lit, 476
- pl\_max, 477
- pl\_max\_horizontal, 478
- pl\_mean, 478
- pl\_mean\_horizontal, 479
- pl\_median, 480
- pl\_mem\_address, 481
- pl\_min, 481
- pl\_min\_horizontal, 482
- pl\_n\_unique, 483
- pl\_pl, 483
- pl\_PTime, 484
- pl\_raw\_list, 485
- pl\_read\_csv, 487
- pl\_read\_ipc, 489
- pl\_read\_ndjson, 490
- pl\_read\_parquet, 492
- pl\_reduce, 494
- pl\_rolling\_corr, 495
- pl\_rolling\_cov, 496
- pl\_scan\_csv, 496
- pl\_scan\_ipc, 499
- pl\_scan\_ndjson, 501
- pl\_scan\_parquet, 502
- pl\_select, 504
- pl\_Series, 505
- pl\_set\_global\_rpool\_cap  
(global\_rpool\_cap), 357
- pl\_show\_all\_public\_functions  
(show\_all\_public\_functions),  
558
- pl\_show\_all\_public\_methods  
(show\_all\_public\_methods), 559
- pl\_SQLContext, 506
- pl\_std, 507
- pl\_struct, 508
- pl\_sum, 509
- pl\_sum\_horizontal, 510
- pl\_tail, 511
- pl\_thread\_pool\_size, 511
- pl\_time, 512
- pl\_using\_string\_cache, 513
- pl\_var, 514
- pl\_when (Expr\_when\_then\_otherwise), 355
- pl\_with\_string\_cache, 515
- polars data type, 505
- polars DataFrame, 23
- polars\_class\_object, 515
- polars\_code\_completion\_activate, 516
- polars\_code\_completion\_activate(), 520
- polars\_code\_completion\_deactivate  
(polars\_code\_completion\_activate),  
516
- polars\_duration\_string, 517
- polars\_envvars, 518
- polars\_info, 520
- polars\_info(), 125, 126, 230
- polars\_options, 521
- polars\_options\_reset (polars\_options),  
521
- POSIXct, 19, 21, 33, 80, 82, 373, 535, 556
- print.RPolarsSeries, 522
- PTIME (pl\_PTime), 484
- RollingGroupBy, 70
- RollingGroupBy\_agg, 523
- RollingGroupBy\_class, 523

- RollingGroupBy\_ungroup, [524](#)
- row.names.RPolarsDataFrame, [525](#)
- RPolarsChainedThen
  - (Expr\_when\_then\_otherwise), [355](#)
- RPolarsChainedWhen
  - (Expr\_when\_then\_otherwise), [355](#)
- RPolarsDataFrame (DataFrame\_class), [32](#)
- RPolarsDataType (pl\_dtypes), [462](#)
- RPolarsDataTypes, [443](#)
- RPolarsDynamicGroupBy
  - (DynamicGroupBy\_class), [120](#)
- RPolarsExpr (Expr\_class), [245](#)
- RPolarsGroupBy (GroupBy\_class), [359](#)
- RPolarsLazyFrame (LazyFrame\_class), [373](#)
- RPolarsLazyGroupBy (LazyGroupBy\_class), [431](#)
- RPolarsRollingGroupBy
  - (RollingGroupBy\_class), [523](#)
- RPolarsRThreadHandle
  - (RThreadHandle\_class), [525](#)
- RPolarsSeries (Series\_class), [534](#)
- RPolarsSQLContext (SQLContext\_class), [559](#)
- RPolarsThen (Expr\_when\_then\_otherwise), [355](#)
- RPolarsWhen (Expr\_when\_then\_otherwise), [355](#)
- RThreadHandle, [379](#)
- RThreadHandle\_class, [525](#), [527](#)
- RThreadHandle\_is\_finished, [526](#)
- RThreadHandle\_join, [526](#)
  
- S3\_arithmetic, [527](#)
- select (DataFrame\_select), [72](#)
- Series, [29](#), [45](#), [83](#), [434–436](#), [506](#), [518](#), [529–531](#), [537](#), [539](#), [541](#), [544](#), [546](#), [547](#), [549–553](#), [564](#), [566](#)
- Series (pl\_Series), [505](#)
- Series\_add, [529](#)
- Series\_alias, [530](#)
- Series\_all, [530](#)
- Series\_any, [531](#)
- Series\_append, [531](#)
- Series\_arg\_max, [532](#)
- Series\_arg\_min, [533](#)
- Series\_chunk\_lengths, [533](#)
- Series\_class, [534](#)
- Series\_clear, [537](#)
- Series\_clone, [537](#)
- Series\_compare, [538](#)
- Series\_div, [539](#)
- Series\_equals, [540](#)
- Series\_floor\_div, [541](#)
- Series\_is\_numeric, [541](#)
- Series\_is\_sorted, [542](#)
- Series\_item, [542](#)
- Series\_len, [543](#)
- Series\_map\_elements, [543](#)
- Series\_max, [544](#)
- Series\_mean, [545](#)
- Series\_median, [545](#)
- Series\_min, [546](#)
- Series\_mod, [546](#)
- Series\_mul, [547](#)
- Series\_n\_chunks, [548](#)
- Series\_n\_unique, [548](#)
- Series\_pow, [549](#)
- Series\_print, [549](#)
- Series\_rename, [550](#)
- Series\_rep, [550](#)
- Series\_set\_sorted, [551](#)
- Series\_sort, [552](#)
- Series\_std, [553](#)
- Series\_sub, [553](#)
- Series\_sum, [554](#)
- Series\_to\_frame, [554](#)
- Series\_to\_list (Series\_to\_r), [555](#)
- Series\_to\_lit, [555](#)
- Series\_to\_r, [555](#)
- Series\_to\_vector (Series\_to\_r), [555](#)
- Series\_value\_counts, [557](#)
- Series\_var, [558](#)
- set\_global\_rpool\_cap
  - (global\_rpool\_cap), [357](#)
- show\_all\_public\_functions, [558](#)
- show\_all\_public\_methods, [559](#)
- SQLContext, [76](#), [77](#), [419](#), [420](#), [507](#), [561](#), [562](#), [564](#)
- SQLContext\_class, [559](#)
- SQLContext\_execute, [560](#)
- SQLContext\_register, [560](#)
- SQLContext\_register\_globals, [561](#)
- SQLContext\_register\_many, [562](#)
- SQLContext\_tables, [563](#)
- SQLContext\_unregister, [563](#)
- strptime(), [222](#)
- Struct, [88](#), [204](#), [253](#), [306](#), [425](#)

`Struct()`, [508](#)  
`sum.RPolarsDataFrame`, [564](#)  
`sum.RPolarsLazyFrame`  
    (`sum.RPolarsDataFrame`), [564](#)  
`sum.RPolarsSeries`  
    (`sum.RPolarsDataFrame`), [564](#)  
`Sys.timezone()`, [19](#), [21](#), [33](#), [80](#), [82](#), [373](#), [535](#),  
    [556](#)  
  
`tail.RPolarsDataFrame`  
    (`head.RPolarsDataFrame`), [366](#)  
`tail.RPolarsLazyFrame`  
    (`head.RPolarsDataFrame`), [366](#)  
the Series class object, [505](#)  
`Then (Expr_when_then_otherwise)`, [355](#)  
`then (Expr_when_then_otherwise)`, [355](#)  
`Then_otherwise`  
    (`Expr_when_then_otherwise`), [355](#)  
`Then_when (Expr_when_then_otherwise)`,  
    [355](#)  
`to_physical (Expr_to_physical)`, [350](#)  
`to_struct (DataFrame_to_struct)`, [85](#)  
  
`UInt32`, [448](#), [475](#)  
`unique.RPolarsDataFrame`, [565](#)  
`unique.RPolarsLazyFrame`  
    (`unique.RPolarsDataFrame`), [565](#)  
  
`When (Expr_when_then_otherwise)`, [355](#)  
`when (Expr_when_then_otherwise)`, [355](#)  
`When_then (Expr_when_then_otherwise)`,  
    [355](#)  
`with_columns (DataFrame_with_columns)`,  
    [90](#)  
`with_columns()`, [18](#), [25](#), [378](#), [382](#), [384](#), [402](#),  
    [423](#)