

# Package: neopolars (via r-universe)

March 8, 2025

**Title** R Bindings for the 'polars' Rust Library

**Version** 0.0.0.9000

**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

**Depends** R (>= 4.2)

**Imports** rlang (>= 1.1.0)

**Suggests** arrow, bit64, blob, cli, clock, data.table, ggplot2, hms, jsonlite, nanoarrow, patrick (>= 0.3.0), testthat (>= 3.0.0), tibble, vctrs, withr

**Additional\_repositories** <https://community.r-multiverse.org>

**Config/Needs/dev** devtools, lifecycle, readr, glue

**Config/testthat/edition** 3

**Config/testthat/parallel** true

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.3.2

**SystemRequirements** Cargo (Rust's package manager), rustc

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

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

**RemoteRef** next

**RemoteSha** d595af72a6d6ed13a6fe647ae38da685945ec1cd

## Contents

|   |    |
|---|----|
| as.data.frame.polars_data_frame . . . . . | 14 |
| as.list.polars_data_frame . . . . .       | 16 |
| as_polars_df . . . . .                    | 19 |
| as_polars_expr . . . . .                  | 22 |
| as_polars_lf . . . . .                    | 25 |
| as_polars_series . . . . .                | 26 |
| as_tibble.polars_data_frame . . . . .     | 31 |
| check_polars . . . . .                    | 34 |
| cs . . . . .                              | 37 |
| cs__all . . . . .                         | 38 |
| cs__alpha . . . . .                       | 38 |
| cs__alphanumeric . . . . .                | 39 |
| cs__binary . . . . .                      | 40 |
| cs__boolean . . . . .                     | 41 |
| cs__by_dtype . . . . .                    | 42 |
| cs__by_index . . . . .                    | 43 |
| cs__by_name . . . . .                     | 44 |
| cs__categorical . . . . .                 | 45 |
| cs__contains . . . . .                    | 45 |
| cs__date . . . . .                        | 46 |
| cs__datetime . . . . .                    | 47 |
| cs__decimal . . . . .                     | 48 |
| cs__digit . . . . .                       | 49 |
| cs__duration . . . . .                    | 50 |
| cs__ends_with . . . . .                   | 51 |
| cs__exclude . . . . .                     | 52 |
| cs__first . . . . .                       | 53 |
| cs__float . . . . .                       | 53 |
| cs__integer . . . . .                     | 54 |
| cs__last . . . . .                        | 55 |
| cs__matches . . . . .                     | 55 |
| cs__numeric . . . . .                     | 56 |
| cs__signed_integer . . . . .              | 57 |
| cs__starts_with . . . . .                 | 58 |
| cs__string . . . . .                      | 59 |
| cs__temporal . . . . .                    | 60 |
| cs__time . . . . .                        | 60 |
| cs__unsigned_integer . . . . .            | 61 |
| dataframe__bottom_k . . . . .             | 62 |
| dataframe__cast . . . . .                 | 63 |
| dataframe__clear . . . . .                | 64 |
| dataframe__clone . . . . .                | 64 |
| dataframe__drop . . . . .                 | 65 |
| dataframe__drop_nans . . . . .            | 66 |
| dataframe__drop_nulls . . . . .           | 67 |
| dataframe__equals . . . . .               | 68 |
| dataframe__explode . . . . .              | 69 |

|                                       |     |
|---------------------------------------|-----|
| dataframe__fill_nan . . . . .         | 69  |
| dataframe__fill_null . . . . .        | 70  |
| dataframe__filter . . . . .           | 71  |
| dataframe__gather_every . . . . .     | 72  |
| dataframe__get_column . . . . .       | 72  |
| dataframe__get_columns . . . . .      | 73  |
| dataframe__get_column_index . . . . . | 74  |
| dataframe__group_by . . . . .         | 74  |
| dataframe__group_by_dynamic . . . . . | 75  |
| dataframe__head . . . . .             | 79  |
| dataframe__is_duplicated . . . . .    | 79  |
| dataframe__is_empty . . . . .         | 80  |
| dataframe__is_unique . . . . .        | 81  |
| dataframe__join . . . . .             | 81  |
| dataframe__join_asof . . . . .        | 84  |
| dataframe__join_where . . . . .       | 87  |
| dataframe__lazy . . . . .             | 88  |
| dataframe__max . . . . .              | 89  |
| dataframe__max_horizontal . . . . .   | 89  |
| dataframe__mean . . . . .             | 90  |
| dataframe__mean_horizontal . . . . .  | 90  |
| dataframe__median . . . . .           | 91  |
| dataframe__merge_sorted . . . . .     | 91  |
| dataframe__min . . . . .              | 92  |
| dataframe__min_horizontal . . . . .   | 92  |
| dataframe__n_chunks . . . . .         | 93  |
| dataframe__partition_by . . . . .     | 94  |
| dataframe__pivot . . . . .            | 94  |
| dataframe__quantile . . . . .         | 97  |
| dataframe__rechunk . . . . .          | 97  |
| dataframe__rename . . . . .           | 98  |
| dataframe__rolling . . . . .          | 99  |
| dataframe__sample . . . . .           | 100 |
| dataframe__select . . . . .           | 101 |
| dataframe__set_sorted . . . . .       | 102 |
| dataframe__shift . . . . .            | 103 |
| dataframe__slice . . . . .            | 103 |
| dataframe__sort . . . . .             | 104 |
| dataframe__std . . . . .              | 105 |
| dataframe__sum . . . . .              | 106 |
| dataframe__sum_horizontal . . . . .   | 106 |
| dataframe__tail . . . . .             | 107 |
| dataframe__top_k . . . . .            | 107 |
| dataframe__to_dummies . . . . .       | 108 |
| dataframe__to_series . . . . .        | 109 |
| dataframe__to_struct . . . . .        | 110 |
| dataframe__transpose . . . . .        | 110 |
| dataframe__unique . . . . .           | 111 |

|                                       |     |
|---------------------------------------|-----|
| dataframe__unnest . . . . .           | 113 |
| dataframe__unpivot . . . . .          | 113 |
| dataframe__var . . . . .              | 114 |
| dataframe__with_columns . . . . .     | 115 |
| dataframe__with_columns_seq . . . . . | 116 |
| dataframe__with_row_index . . . . .   | 117 |
| dataframe__write_csv . . . . .        | 118 |
| dataframe__write_json . . . . .       | 120 |
| dataframe__write_ndjson . . . . .     | 121 |
| dataframe__write_parquet . . . . .    | 121 |
| expr_arr_all . . . . .                | 124 |
| expr_arr_any . . . . .                | 124 |
| expr_arr_arg_max . . . . .            | 125 |
| expr_arr_arg_min . . . . .            | 125 |
| expr_arr_contains . . . . .           | 126 |
| expr_arr_count_matches . . . . .      | 126 |
| expr_arr_explode . . . . .            | 127 |
| expr_arr_first . . . . .              | 127 |
| expr_arr_get . . . . .                | 128 |
| expr_arr_join . . . . .               | 129 |
| expr_arr_last . . . . .               | 129 |
| expr_arr_max . . . . .                | 130 |
| expr_arr_median . . . . .             | 130 |
| expr_arr_min . . . . .                | 131 |
| expr_arr_n_unique . . . . .           | 131 |
| expr_arr_reverse . . . . .            | 132 |
| expr_arr_shift . . . . .              | 132 |
| expr_arr_sort . . . . .               | 133 |
| expr_arr_std . . . . .                | 133 |
| expr_arr_sum . . . . .                | 134 |
| expr_arr_to_list . . . . .            | 135 |
| expr_arr_unique . . . . .             | 135 |
| expr_arr_var . . . . .                | 136 |
| expr_bin_contains . . . . .           | 136 |
| expr_bin_decode . . . . .             | 137 |
| expr_bin_encode . . . . .             | 138 |
| expr_bin_ends_with . . . . .          | 139 |
| expr_bin_size . . . . .               | 139 |
| expr_bin_starts_with . . . . .        | 140 |
| expr_cat_get_categories . . . . .     | 141 |
| expr_cat_set_ordering . . . . .       | 141 |
| expr_dt_add_business_days . . . . .   | 142 |
| expr_dt_base_utc_offset . . . . .     | 143 |
| expr_dt_cast_time_unit . . . . .      | 144 |
| expr_dt_century . . . . .             | 145 |
| expr_dt_combine . . . . .             | 145 |
| expr_dt_convert_time_zone . . . . .   | 146 |
| expr_dt_date . . . . .                | 147 |

|   |     |
|---|-----|
| <code>expr_dt_day</code>                | 147 |
| <code>expr_dt_dst_offset</code>         | 148 |
| <code>expr_dt_epoch</code>              | 148 |
| <code>expr_dt_hour</code>               | 149 |
| <code>expr_dt_iso_year</code>           | 150 |
| <code>expr_dt_is_leap_year</code>       | 150 |
| <code>expr_dt_microsecond</code>        | 151 |
| <code>expr_dt_millisecond</code>        | 151 |
| <code>expr_dt_minute</code>             | 152 |
| <code>expr_dt_month</code>              | 153 |
| <code>expr_dt_month_end</code>          | 153 |
| <code>expr_dt_month_start</code>        | 154 |
| <code>expr_dt_nanosecond</code>         | 154 |
| <code>expr_dt_offset_by</code>          | 155 |
| <code>expr_dt_ordinal_day</code>        | 156 |
| <code>expr_dt_quarter</code>            | 157 |
| <code>expr_dt_replace_time_zone</code>  | 158 |
| <code>expr_dt_round</code>              | 159 |
| <code>expr_dt_second</code>             | 161 |
| <code>expr_dt_strftime</code>           | 161 |
| <code>expr_dt_time</code>               | 162 |
| <code>expr_dt_timestamp</code>          | 163 |
| <code>expr_dt_total_days</code>         | 164 |
| <code>expr_dt_total_hours</code>        | 164 |
| <code>expr_dt_total_microseconds</code> | 165 |
| <code>expr_dt_total_milliseconds</code> | 166 |
| <code>expr_dt_total_minutes</code>      | 166 |
| <code>expr_dt_total_nanoseconds</code>  | 167 |
| <code>expr_dt_total_seconds</code>      | 168 |
| <code>expr_dt_to_string</code>          | 168 |
| <code>expr_dt_truncate</code>           | 170 |
| <code>expr_dt_week</code>               | 171 |
| <code>expr_dt_weekday</code>            | 172 |
| <code>expr_dt_with_time_unit</code>     | 172 |
| <code>expr_dt_year</code>               | 173 |
| <code>expr_list_all</code>              | 173 |
| <code>expr_list_any</code>              | 174 |
| <code>expr_list_arg_max</code>          | 174 |
| <code>expr_list_arg_min</code>          | 175 |
| <code>expr_list_concat</code>           | 175 |
| <code>expr_list_contains</code>         | 176 |
| <code>expr_list_count_matches</code>    | 177 |
| <code>expr_list_diff</code>             | 177 |
| <code>expr_list_drop_nulls</code>       | 178 |
| <code>expr_list_eval</code>             | 178 |
| <code>expr_list_explode</code>          | 179 |
| <code>expr_list_first</code>            | 180 |
| <code>expr_list_gather</code>           | 180 |

|   |     |
|---|-----|
| <code>expr_list_gather_every</code> . . . . .             | 181 |
| <code>expr_list_get</code> . . . . .                      | 182 |
| <code>expr_list_head</code> . . . . .                     | 183 |
| <code>expr_list_join</code> . . . . .                     | 183 |
| <code>expr_list_last</code> . . . . .                     | 184 |
| <code>expr_list_len</code> . . . . .                      | 185 |
| <code>expr_list_max</code> . . . . .                      | 185 |
| <code>expr_list_mean</code> . . . . .                     | 186 |
| <code>expr_list_median</code> . . . . .                   | 186 |
| <code>expr_list_min</code> . . . . .                      | 187 |
| <code>expr_list_n_unique</code> . . . . .                 | 187 |
| <code>expr_list_reverse</code> . . . . .                  | 188 |
| <code>expr_list_sample</code> . . . . .                   | 188 |
| <code>expr_list_set_difference</code> . . . . .           | 189 |
| <code>expr_list_set_intersection</code> . . . . .         | 190 |
| <code>expr_list_set_symmetric_difference</code> . . . . . | 191 |
| <code>expr_list_set_union</code> . . . . .                | 192 |
| <code>expr_list_shift</code> . . . . .                    | 192 |
| <code>expr_list_slice</code> . . . . .                    | 193 |
| <code>expr_list_sort</code> . . . . .                     | 194 |
| <code>expr_list_std</code> . . . . .                      | 195 |
| <code>expr_list_sum</code> . . . . .                      | 195 |
| <code>expr_list_tail</code> . . . . .                     | 196 |
| <code>expr_list_to_array</code> . . . . .                 | 196 |
| <code>expr_list_unique</code> . . . . .                   | 197 |
| <code>expr_list_var</code> . . . . .                      | 198 |
| <code>expr_meta_eq</code> . . . . .                       | 198 |
| <code>expr_meta_has_multiple_outputs</code> . . . . .     | 199 |
| <code>expr_meta_is_column</code> . . . . .                | 199 |
| <code>expr_meta_is_column_selection</code> . . . . .      | 200 |
| <code>expr_meta_is_regex_projection</code> . . . . .      | 201 |
| <code>expr_meta_ne</code> . . . . .                       | 201 |
| <code>expr_meta_output_name</code> . . . . .              | 202 |
| <code>expr_meta_pop</code> . . . . .                      | 203 |
| <code>expr_meta_root_names</code> . . . . .               | 203 |
| <code>expr_meta_serialize</code> . . . . .                | 204 |
| <code>expr_meta_tree_format</code> . . . . .              | 205 |
| <code>expr_meta_undo_aliases</code> . . . . .             | 205 |
| <code>expr_struct_field</code> . . . . .                  | 206 |
| <code>expr_struct_json_encode</code> . . . . .            | 207 |
| <code>expr_struct_rename_fields</code> . . . . .          | 207 |
| <code>expr_struct_unnest</code> . . . . .                 | 208 |
| <code>expr_struct_with_fields</code> . . . . .            | 209 |
| <code>expr_str_contains</code> . . . . .                  | 210 |
| <code>expr_str_contains_any</code> . . . . .              | 211 |
| <code>expr_str_count_matches</code> . . . . .             | 212 |
| <code>expr_str_decode</code> . . . . .                    | 212 |
| <code>expr_str_encode</code> . . . . .                    | 213 |

|   |     |
|---|-----|
| <code>expr_str_ends_with</code> . . . . .         | 214 |
| <code>expr_str_extract</code> . . . . .           | 215 |
| <code>expr_str_extract_all</code> . . . . .       | 215 |
| <code>expr_str_extract_groups</code> . . . . .    | 216 |
| <code>expr_str_extract_many</code> . . . . .      | 217 |
| <code>expr_str_find</code> . . . . .              | 218 |
| <code>expr_str_head</code> . . . . .              | 219 |
| <code>expr_str_join</code> . . . . .              | 220 |
| <code>expr_str_json_decode</code> . . . . .       | 221 |
| <code>expr_str_json_path_match</code> . . . . .   | 221 |
| <code>expr_str_len_bytes</code> . . . . .         | 222 |
| <code>expr_str_len_chars</code> . . . . .         | 223 |
| <code>expr_str_pad_end</code> . . . . .           | 224 |
| <code>expr_str_pad_start</code> . . . . .         | 224 |
| <code>expr_str_replace</code> . . . . .           | 225 |
| <code>expr_str_replace_all</code> . . . . .       | 226 |
| <code>expr_str_replace_many</code> . . . . .      | 228 |
| <code>expr_str_reverse</code> . . . . .           | 229 |
| <code>expr_str_slice</code> . . . . .             | 229 |
| <code>expr_str_split</code> . . . . .             | 230 |
| <code>expr_str_splitn</code> . . . . .            | 230 |
| <code>expr_str_split_exact</code> . . . . .       | 231 |
| <code>expr_str_starts_with</code> . . . . .       | 232 |
| <code>expr_str_strip_chars</code> . . . . .       | 232 |
| <code>expr_str_strip_chars_end</code> . . . . .   | 233 |
| <code>expr_str_strip_chars_start</code> . . . . . | 234 |
| <code>expr_str_strip_prefix</code> . . . . .      | 234 |
| <code>expr_str_strip_suffix</code> . . . . .      | 235 |
| <code>expr_str_strptime</code> . . . . .          | 236 |
| <code>expr_str_tail</code> . . . . .              | 238 |
| <code>expr_str_to_date</code> . . . . .           | 239 |
| <code>expr_str_to_datetime</code> . . . . .       | 240 |
| <code>expr_str_to_decimal</code> . . . . .        | 241 |
| <code>expr_str_to_integer</code> . . . . .        | 242 |
| <code>expr_str_to_lowercase</code> . . . . .      | 243 |
| <code>expr_str_to_time</code> . . . . .           | 243 |
| <code>expr_str_to_uppercase</code> . . . . .      | 244 |
| <code>expr_str_zfill</code> . . . . .             | 244 |
| <code>expr__abs</code> . . . . .                  | 245 |
| <code>expr__add</code> . . . . .                  | 246 |
| <code>expr__agg_groups</code> . . . . .           | 247 |
| <code>expr__alias</code> . . . . .                | 247 |
| <code>expr__all</code> . . . . .                  | 248 |
| <code>expr__and</code> . . . . .                  | 249 |
| <code>expr__any</code> . . . . .                  | 249 |
| <code>expr__append</code> . . . . .               | 250 |
| <code>expr__approx_n_unique</code> . . . . .      | 251 |
| <code>expr__arccos</code> . . . . .               | 251 |

|  |     |
|--|-----|
| expr__arccosh . . . . .                | 252 |
| expr__arcsin . . . . .                 | 252 |
| expr__arcsinh . . . . .                | 253 |
| expr__arctan . . . . .                 | 253 |
| expr__arctanh . . . . .                | 254 |
| expr__arg_max . . . . .                | 254 |
| expr__arg_min . . . . .                | 255 |
| expr__arg_sort . . . . .               | 255 |
| expr__arg_true . . . . .               | 256 |
| expr__arg_unique . . . . .             | 256 |
| expr__backward_fill . . . . .          | 257 |
| expr__bitwise_and . . . . .            | 257 |
| expr__bitwise_count_ones . . . . .     | 258 |
| expr__bitwise_count_zeros . . . . .    | 258 |
| expr__bitwise_leading_ones . . . . .   | 259 |
| expr__bitwise_leading_zeros . . . . .  | 259 |
| expr__bitwise_or . . . . .             | 260 |
| expr__bitwise_trailing_ones . . . . .  | 260 |
| expr__bitwise_trailing_zeros . . . . . | 261 |
| expr__bitwise_xor . . . . .            | 261 |
| expr__bottom_k . . . . .               | 262 |
| expr__bottom_k_by . . . . .            | 262 |
| expr__cast . . . . .                   | 263 |
| expr__cbrt . . . . .                   | 264 |
| expr__ceil . . . . .                   | 265 |
| expr__clip . . . . .                   | 265 |
| expr__cos . . . . .                    | 266 |
| expr__cosh . . . . .                   | 267 |
| expr__cot . . . . .                    | 267 |
| expr__count . . . . .                  | 268 |
| expr__cumulative_eval . . . . .        | 268 |
| expr__cum_count . . . . .              | 269 |
| expr__cum_max . . . . .                | 270 |
| expr__cum_min . . . . .                | 270 |
| expr__cum_prod . . . . .               | 271 |
| expr__cum_sum . . . . .                | 272 |
| expr__cut . . . . .                    | 272 |
| expr__degrees . . . . .                | 273 |
| expr__diff . . . . .                   | 274 |
| expr__dot . . . . .                    | 274 |
| expr__drop_nans . . . . .              | 275 |
| expr__drop_nulls . . . . .             | 275 |
| expr__entropy . . . . .                | 276 |
| expr__eq . . . . .                     | 276 |
| expr__eq_missing . . . . .             | 277 |
| expr__ewm_mean . . . . .               | 278 |
| expr__ewm_mean_by . . . . .            | 279 |
| expr__ewm_std . . . . .                | 280 |



|                                   |     |
|-----------------------------------|-----|
| expr__ewm_var . . . . .           | 282 |
| expr__exclude . . . . .           | 283 |
| expr__exp . . . . .               | 284 |
| expr__explode . . . . .           | 285 |
| expr__extend_constant . . . . .   | 285 |
| expr__fill_nan . . . . .          | 286 |
| expr__fill_null . . . . .         | 286 |
| expr__filter . . . . .            | 287 |
| expr__first . . . . .             | 288 |
| expr__flatten . . . . .           | 288 |
| expr__floor . . . . .             | 289 |
| expr__floor_div . . . . .         | 289 |
| expr__forward_fill . . . . .      | 290 |
| expr__gather . . . . .            | 291 |
| expr__gather_every . . . . .      | 291 |
| expr__ge . . . . .                | 292 |
| expr__get . . . . .               | 292 |
| expr__gt . . . . .                | 293 |
| expr__hash . . . . .              | 294 |
| expr__has_nulls . . . . .         | 294 |
| expr__head . . . . .              | 295 |
| expr__hist . . . . .              | 295 |
| expr__implode . . . . .           | 296 |
| expr__interpolate . . . . .       | 297 |
| expr__interpolate_by . . . . .    | 297 |
| expr__is_between . . . . .        | 298 |
| expr__is_duplicated . . . . .     | 299 |
| expr__is_finite . . . . .         | 299 |
| expr__is_first_distinct . . . . . | 300 |
| expr__is_in . . . . .             | 300 |
| expr__is_infinite . . . . .       | 301 |
| expr__is_last_distinct . . . . .  | 302 |
| expr__is_nan . . . . .            | 302 |
| expr__is_not_nan . . . . .        | 303 |
| expr__is_not_null . . . . .       | 303 |
| expr__is_null . . . . .           | 304 |
| expr__is_unique . . . . .         | 305 |
| expr__kurtosis . . . . .          | 305 |
| expr__last . . . . .              | 306 |
| expr__le . . . . .                | 306 |
| expr__len . . . . .               | 307 |
| expr__limit . . . . .             | 307 |
| expr__log . . . . .               | 308 |
| expr__log10 . . . . .             | 308 |
| expr__log1p . . . . .             | 309 |
| expr__lower_bound . . . . .       | 309 |
| expr__lt . . . . .                | 310 |
| expr__map_batches . . . . .       | 310 |

|                                     |     |
|-------------------------------------|-----|
| expr__max . . . . .                 | 312 |
| expr__mean . . . . .                | 312 |
| expr__median . . . . .              | 313 |
| expr__min . . . . .                 | 313 |
| expr__mod . . . . .                 | 314 |
| expr__mode . . . . .                | 314 |
| expr__mul . . . . .                 | 315 |
| expr__nan_max . . . . .             | 316 |
| expr__nan_min . . . . .             | 316 |
| expr__ne . . . . .                  | 317 |
| expr__ne_missing . . . . .          | 317 |
| expr__not . . . . .                 | 318 |
| expr__null_count . . . . .          | 319 |
| expr__n_unique . . . . .            | 319 |
| expr__or . . . . .                  | 320 |
| expr__over . . . . .                | 320 |
| expr__pct_change . . . . .          | 322 |
| expr__peak_max . . . . .            | 323 |
| expr__peak_min . . . . .            | 323 |
| expr__pow . . . . .                 | 324 |
| expr__product . . . . .             | 324 |
| expr__qcut . . . . .                | 325 |
| expr__quantile . . . . .            | 326 |
| expr__radians . . . . .             | 327 |
| expr__rank . . . . .                | 327 |
| expr__rechunk . . . . .             | 328 |
| expr__reinterpret . . . . .         | 329 |
| expr__repeat_by . . . . .           | 329 |
| expr__replace . . . . .             | 330 |
| expr__replace_strict . . . . .      | 331 |
| expr__reshape . . . . .             | 333 |
| expr__reverse . . . . .             | 334 |
| expr__rle . . . . .                 | 334 |
| expr__rle_id . . . . .              | 335 |
| expr__rolling . . . . .             | 335 |
| expr__rolling_max . . . . .         | 337 |
| expr__rolling_max_by . . . . .      | 338 |
| expr__rolling_mean . . . . .        | 340 |
| expr__rolling_mean_by . . . . .     | 341 |
| expr__rolling_median . . . . .      | 343 |
| expr__rolling_median_by . . . . .   | 344 |
| expr__rolling_min . . . . .         | 346 |
| expr__rolling_min_by . . . . .      | 348 |
| expr__rolling_quantile . . . . .    | 350 |
| expr__rolling_quantile_by . . . . . | 351 |
| expr__rolling_skew . . . . .        | 353 |
| expr__rolling_std . . . . .         | 354 |
| expr__rolling_std_by . . . . .      | 355 |

|                                     |     |
|-------------------------------------|-----|
| expr__rolling_sum . . . . .         | 358 |
| expr__rolling_sum_by . . . . .      | 359 |
| expr__rolling_var . . . . .         | 361 |
| expr__rolling_var_by . . . . .      | 362 |
| expr__round . . . . .               | 364 |
| expr__round_sig_figs . . . . .      | 365 |
| expr__sample . . . . .              | 365 |
| expr__search_sorted . . . . .       | 366 |
| expr__set_sorted . . . . .          | 367 |
| expr__shift . . . . .               | 368 |
| expr__shrink_dtype . . . . .        | 368 |
| expr__shuffle . . . . .             | 369 |
| expr__sign . . . . .                | 370 |
| expr__sin . . . . .                 | 370 |
| expr__sinh . . . . .                | 371 |
| expr__skew . . . . .                | 371 |
| expr__slice . . . . .               | 372 |
| expr__sort . . . . .                | 373 |
| expr__sort_by . . . . .             | 374 |
| expr__sqrt . . . . .                | 375 |
| expr__std . . . . .                 | 376 |
| expr__sub . . . . .                 | 376 |
| expr__sum . . . . .                 | 377 |
| expr__tail . . . . .                | 377 |
| expr__tan . . . . .                 | 378 |
| expr__tanh . . . . .                | 378 |
| expr__top_k . . . . .               | 379 |
| expr__top_k_by . . . . .            | 379 |
| expr__to_physical . . . . .         | 380 |
| expr__true_div . . . . .            | 381 |
| expr__unique . . . . .              | 382 |
| expr__unique_counts . . . . .       | 383 |
| expr__upper_bound . . . . .         | 383 |
| expr__value_counts . . . . .        | 384 |
| expr__var . . . . .                 | 385 |
| expr__xor . . . . .                 | 385 |
| infer__polars_dtype . . . . .       | 386 |
| lazyframe__bottom_k . . . . .       | 388 |
| lazyframe__cast . . . . .           | 389 |
| lazyframe__clear . . . . .          | 390 |
| lazyframe__clone . . . . .          | 391 |
| lazyframe__collect . . . . .        | 392 |
| lazyframe__collect_schema . . . . . | 394 |
| lazyframe__count . . . . .          | 394 |
| lazyframe__drop . . . . .           | 395 |
| lazyframe__drop_nans . . . . .      | 396 |
| lazyframe__drop_nulls . . . . .     | 397 |
| lazyframe__explain . . . . .        | 397 |

|  |     |
|--|-----|
| lazyframe__explode . . . . .               | 399 |
| lazyframe__fill_nan . . . . .              | 400 |
| lazyframe__fill_null . . . . .             | 400 |
| lazyframe__filter . . . . .                | 401 |
| lazyframe__first . . . . .                 | 402 |
| lazyframe__gather_every . . . . .          | 403 |
| lazyframe__group_by . . . . .              | 403 |
| lazyframe__group_by_dynamic . . . . .      | 404 |
| lazyframe__head . . . . .                  | 408 |
| lazyframe__interpolate . . . . .           | 408 |
| lazyframe__join . . . . .                  | 409 |
| lazyframe__join_asof . . . . .             | 411 |
| lazyframe__join_where . . . . .            | 415 |
| lazyframe__last . . . . .                  | 416 |
| lazyframe__max . . . . .                   | 416 |
| lazyframe__mean . . . . .                  | 417 |
| lazyframe__median . . . . .                | 417 |
| lazyframe__merge_sorted . . . . .          | 418 |
| lazyframe__min . . . . .                   | 418 |
| lazyframe__null_count . . . . .            | 419 |
| lazyframe__profile . . . . .               | 419 |
| lazyframe__quantile . . . . .              | 422 |
| lazyframe__rename . . . . .                | 422 |
| lazyframe__reverse . . . . .               | 423 |
| lazyframe__rolling . . . . .               | 424 |
| lazyframe__select . . . . .                | 425 |
| lazyframe__select_seq . . . . .            | 426 |
| lazyframe__serialize . . . . .             | 427 |
| lazyframe__set_sorted . . . . .            | 428 |
| lazyframe__shift . . . . .                 | 428 |
| lazyframe__sink_csv . . . . .              | 429 |
| lazyframe__slice . . . . .                 | 432 |
| lazyframe__sort . . . . .                  | 432 |
| lazyframe__std . . . . .                   | 433 |
| lazyframe__sum . . . . .                   | 434 |
| lazyframe__tail . . . . .                  | 434 |
| lazyframe__top_k . . . . .                 | 435 |
| lazyframe__to_dot . . . . .                | 436 |
| lazyframe__unique . . . . .                | 438 |
| lazyframe__unnest . . . . .                | 439 |
| lazyframe__unpivot . . . . .               | 439 |
| lazyframe__var . . . . .                   | 440 |
| lazyframe__with_columns . . . . .          | 441 |
| lazyframe__with_columns_seq . . . . .      | 442 |
| lazyframe__with_row_index . . . . .        | 443 |
| parquet_statistics . . . . .               | 444 |
| pl . . . . .                               | 447 |
| pl_api_register_series_namespace . . . . . | 448 |

|  |     |
|--|-----|
| <code>pl__all</code> . . . . .             | 449 |
| <code>pl__all_horizontal</code> . . . . .  | 449 |
| <code>pl__any</code> . . . . .             | 450 |
| <code>pl__any_horizontal</code> . . . . .  | 451 |
| <code>pl__arg_where</code> . . . . .       | 452 |
| <code>pl__coalesce</code> . . . . .        | 452 |
| <code>pl__col</code> . . . . .             | 453 |
| <code>pl__concat</code> . . . . .          | 454 |
| <code>pl__concat_list</code> . . . . .     | 455 |
| <code>pl__concat_str</code> . . . . .      | 456 |
| <code>pl__cum_sum</code> . . . . .         | 457 |
| <code>pl__DataFrame</code> . . . . .       | 458 |
| <code>pl__datetime</code> . . . . .        | 459 |
| <code>pl__datetime_range</code> . . . . .  | 461 |
| <code>pl__datetime_ranges</code> . . . . . | 463 |
| <code>pl__date_range</code> . . . . .      | 465 |
| <code>pl__date_ranges</code> . . . . .     | 467 |
| <code>pl__duration</code> . . . . .        | 469 |
| <code>pl__element</code> . . . . .         | 470 |
| <code>pl__first</code> . . . . .           | 471 |
| <code>pl__int_range</code> . . . . .       | 472 |
| <code>pl__int_ranges</code> . . . . .      | 473 |
| <code>pl__last</code> . . . . .            | 473 |
| <code>pl__LazyFrame</code> . . . . .       | 474 |
| <code>pl__len</code> . . . . .             | 475 |
| <code>pl__lit</code> . . . . .             | 476 |
| <code>pl__max</code> . . . . .             | 477 |
| <code>pl__max_horizontal</code> . . . . .  | 478 |
| <code>pl__mean_horizontal</code> . . . . . | 478 |
| <code>pl__min</code> . . . . .             | 479 |
| <code>pl__min_horizontal</code> . . . . .  | 480 |
| <code>pl__nth</code> . . . . .             | 481 |
| <code>pl__read_csv</code> . . . . .        | 481 |
| <code>pl__read_ipc</code> . . . . .        | 484 |
| <code>pl__read_ipc_stream</code> . . . . . | 486 |
| <code>pl__read_ndjson</code> . . . . .     | 487 |
| <code>pl__read_parquet</code> . . . . .    | 489 |
| <code>pl__repeat</code> . . . . .          | 492 |
| <code>pl__scan_csv</code> . . . . .        | 493 |
| <code>pl__scan_ipc</code> . . . . .        | 496 |
| <code>pl__scan_ndjson</code> . . . . .     | 498 |
| <code>pl__scan_parquet</code> . . . . .    | 500 |
| <code>pl__Series</code> . . . . .          | 502 |
| <code>pl__show_versions</code> . . . . .   | 503 |
| <code>pl__struct</code> . . . . .          | 504 |
| <code>pl__sum</code> . . . . .             | 505 |
| <code>pl__sum_horizontal</code> . . . . .  | 506 |
| <code>pl__time_range</code> . . . . .      | 506 |

|                                  |     |
|----------------------------------|-----|
| pl__time_ranges . . . . .        | 508 |
| polars_dtype . . . . .           | 509 |
| polars_duration_string . . . . . | 511 |
| polars_expr . . . . .            | 512 |
| series_struct_unnest . . . . .   | 513 |
| series__alias . . . . .          | 514 |
| series__n_chunks . . . . .       | 514 |
| series__to_frame . . . . .       | 515 |
| series__to_r_vector . . . . .    | 515 |

## Index 520

---

as.data.frame.polars\_data\_frame

*Export the polars object as an R DataFrame*

---

### Description

This S3 method is a shortcut for `as_polars_df(x, ...) $to_struct() $to_r_vector(ensure_vector = FALSE, struct = "dataframe")`.

### Usage

```
## S3 method for class 'polars_data_frame'
as.data.frame(
  x,
  ...,
  int64 = c("double", "character", "integer", "integer64"),
  date = c("Date", "IDate"),
  time = c("hms", "ITime"),
  decimal = c("double", "character"),
  as_clock_class = FALSE,
  ambiguous = c("raise", "earliest", "latest", "null"),
  non_existent = c("raise", "null")
)
```

```
## S3 method for class 'polars_lazy_frame'
as.data.frame(
  x,
  ...,
  int64 = c("double", "character", "integer", "integer64"),
  date = c("Date", "IDate"),
  time = c("hms", "ITime"),
  decimal = c("double", "character"),
  as_clock_class = FALSE,
  ambiguous = c("raise", "earliest", "latest", "null"),
  non_existent = c("raise", "null")
)
```

**Arguments**

|                             |  |
|-----------------------------|--|
| <code>x</code>              | A polars object  |
| <code>...</code>            | Passed to <code>as_polars_df()</code> .  |
| <code>int64</code>          | Determine how to convert Polars' Int64, UInt32, or UInt64 type values to R type. One of the followings: <ul style="list-style-type: none"> <li>• <code>"double"</code> (default): Convert to the R's <code>double</code> type. Accuracy may be degraded.</li> <li>• <code>"character"</code>: Convert to the R's <code>character</code> type.</li> <li>• <code>"integer"</code>: Convert to the R's <code>integer</code> type. If the value is out of the range of R's integer type, export as <code>NA_integer_</code>.</li> <li>• <code>"integer64"</code>: Convert to the <code>bit64::integer64</code> class. The <code>bit64</code> package must be installed. If the value is out of the range of <code>bit64::integer64</code>, export as <code>bit64::NA_integer64_</code>.</li> </ul> |
| <code>date</code>           | Determine how to convert Polars' Date type values to R class. One of the followings: <ul style="list-style-type: none"> <li>• <code>"Date"</code> (default): Convert to the R's <code>Date</code> class.</li> <li>• <code>"IDate"</code>: Convert to the <code>data.table::IDate</code> class.</li> </ul>  |
| <code>time</code>           | Determine how to convert Polars' Time type values to R class. One of the followings: <ul style="list-style-type: none"> <li>• <code>"hms"</code> (default): Convert to the <code>hms::hms</code> class.</li> <li>• <code>"ITime"</code>: Convert to the <code>data.table::ITime</code> class. The <code>data.table</code> package must be installed.</li> </ul>  |
| <code>decimal</code>        | Determine how to convert Polars' Decimal type values to R type. One of the followings: <ul style="list-style-type: none"> <li>• <code>"double"</code> (default): Convert to the R's <code>double</code> type.</li> <li>• <code>"character"</code>: Convert to the R's <code>character</code> type.</li> </ul>  |
| <code>as_clock_class</code> | A logical value indicating whether to export datetimes and duration as the <code>clock</code> package's classes. <ul style="list-style-type: none"> <li>• <code>FALSE</code> (default): Duration values are exported as <code>difftime</code> and datetime values are exported as <code>POSIXct</code>. Accuracy may be degraded.</li> <li>• <code>TRUE</code>: Duration values are exported as <code>clock_duration</code>, datetime without timezone values are exported as <code>clock_naive_time</code>, and datetime with timezone values are exported as <code>clock_zoned_time</code>. For this case, the <code>clock</code> package must be installed. Accuracy will be maintained.</li> </ul>   |
| <code>ambiguous</code>      | Determine how to deal with ambiguous datetimes. Only applicable when <code>as_clock_class</code> is set to <code>FALSE</code> and datetime without timezone values are exported as <code>POSIXct</code> . Character vector or <code>expression</code> containing the followings: <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> </ul>  |

- "null": Return a NA value
- `non_existent` Determine how to deal with non-existent datetimes. Only applicable when `as_clock_class` is set to `FALSE` and datetime without timezone values are exported as `POSIXct`. One of the followings:
- "raise" (default): Throw an error
  - "null": Return a NA value

## Value

An R data frame

## Examples

```
df <- as_polars_df(list(a = 1:3, b = 4:6))

as.data.frame(df)
as.data.frame(df$lazy())
```

---

```
as.list.polars_data_frame
```

*Export the polars object as an R list*

---

## Description

This S3 method calls `as_polars_df(x, ...)$get_columns()` or `as_polars_df(x, ...)$to_struct()$to_r_vector(TRUE)` depending on the `as_series` argument.

## Usage

```
## S3 method for class 'polars_data_frame'
as.list(
  x,
  ...,
  as_series = FALSE,
  int64 = c("double", "character", "integer", "integer64"),
  date = c("Date", "IDate"),
  time = c("hms", "ITime"),
  struct = c("dataframe", "tibble"),
  decimal = c("double", "character"),
  as_clock_class = FALSE,
  ambiguous = c("raise", "earliest", "latest", "null"),
  non_existent = c("raise", "null")
)

## S3 method for class 'polars_lazy_frame'
as.list(
  x,
```



```

...,
as_series = FALSE,
int64 = c("double", "character", "integer", "integer64"),
date = c("Date", "IDate"),
time = c("hms", "ITime"),
struct = c("dataframe", "tibble"),
decimal = c("double", "character"),
as_clock_class = FALSE,
ambiguous = c("raise", "earliest", "latest", "null"),
non_existent = c("raise", "null")
)

```

## Arguments

|                        |   |
|------------------------|---|
| <code>x</code>         | A polars object   |
| <code>...</code>       | Passed to <code>as_polars_df()</code> .   |
| <code>as_series</code> | Whether to convert each column to an R vector or a Series. If TRUE, return a list of Series, otherwise a list of vectors (default).   |
| <code>int64</code>     | Determine how to convert Polars' Int64, UInt32, or UInt64 type values to R type. One of the followings: <ul style="list-style-type: none"> <li>"double" (default): Convert to the R's double type. Accuracy may be degraded.</li> <li>"character": Convert to the R's character type.</li> <li>"integer": Convert to the R's integer type. If the value is out of the range of R's integer type, export as <code>NA_integer_</code>.</li> <li>"integer64": Convert to the <code>bit64::integer64</code> class. The <code>bit64</code> package must be installed. If the value is out of the range of <code>bit64::integer64</code>, export as <code>bit64::NA_integer64_</code>.</li> </ul> |
| <code>date</code>      | Determine how to convert Polars' Date type values to R class. One of the followings: <ul style="list-style-type: none"> <li>"Date" (default): Convert to the R's Date class.</li> <li>"IDate": Convert to the <code>data.table::IDate</code> class.</li> </ul>  |
| <code>time</code>      | Determine how to convert Polars' Time type values to R class. One of the followings: <ul style="list-style-type: none"> <li>"hms" (default): Convert to the <code>hms::hms</code> class.</li> <li>"ITime": Convert to the <code>data.table::ITime</code> class. The <code>data.table</code> package must be installed.</li> </ul>   |
| <code>struct</code>    | Determine how to convert Polars' Struct type values to R class. One of the followings: <ul style="list-style-type: none"> <li>"dataframe" (default): Convert to the R's <code>data.frame</code> class.</li> <li>"tibble": Convert to the <code>tibble</code> class. If the <code>tibble</code> package is not installed, a warning will be shown.</li> </ul>  |
| <code>decimal</code>   | Determine how to convert Polars' Decimal type values to R type. One of the followings:  |

- "double" (default): Convert to the R's [double](#) type.
  - "character": Convert to the R's [character](#) type.
- as\_clock\_class** A logical value indicating whether to export datetimes and duration as the [clock](#) package's classes.
- FALSE (default): Duration values are exported as [difftime](#) and date-time values are exported as [POSIXct](#). Accuracy may be degraded.
  - TRUE: Duration values are exported as [clock\\_duration](#), datetime without timezone values are exported as [clock\\_naive\\_time](#), and datetime with timezone values are exported as [clock\\_zoned\\_time](#). For this case, the [clock](#) package must be installed. Accuracy will be maintained.
- ambiguous** Determine how to deal with ambiguous datetimes. Only applicable when `as_clock_class` is set to FALSE and datetime without timezone values are exported as [POSIXct](#). Character vector or [expression](#) containing the followings:
- "raise" (default): Throw an error
  - "earliest": Use the earliest datetime
  - "latest": Use the latest datetime
  - "null": Return a NA value
- non\_existent** Determine how to deal with non-existent datetimes. Only applicable when `as_clock_class` is set to FALSE and datetime without timezone values are exported as [POSIXct](#). One of the followings:
- "raise" (default): Throw an error
  - "null": Return a NA value

## Details

Arguments other than `x` and `as_series` are passed to `<Series>$to_r_vector()`, so they are ignored when `as_series=TRUE`.

## Value

A [list](#)

## See Also

- [<DataFrame>\\$get\\_columns\(\)](#)

## Examples

```
df <- as_polars_df(list(a = 1:3, b = 4:6))
```

```
as.list(df, as_series = TRUE)
as.list(df, as_series = FALSE)
```

```
as.list(df$lazy(), as_series = TRUE)
as.list(df$lazy(), as_series = FALSE)
```

---

`as_polars_df`*Create a Polars DataFrame from an R object*

---

## Description

The `as_polars_df()` function creates a [Polars DataFrame](#) from various R objects. Because [Polars DataFrame](#) can be converted to a [struct type Series](#) and vice versa, objects that are converted to a [struct type type Series](#) by `as_polars_series()` are supported by this function.

## Usage

```
as_polars_df(x, ...)  
  
## Default S3 method:  
as_polars_df(x, ...)  
  
## S3 method for class 'polars_series'  
as_polars_df(x, ..., column_name = NULL, from_struct = TRUE)  
  
## S3 method for class 'polars_data_frame'  
as_polars_df(x, ...)  
  
## S3 method for class 'polars_group_by'  
as_polars_df(x, ...)  
  
## S3 method for class 'polars_lazy_frame'  
as_polars_df(  
  x,  
  ...,  
  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,  
  no_optimization = FALSE,  
  streaming = FALSE  
)  
  
## S3 method for class 'list'  
as_polars_df(x, ...)  
  
## S3 method for class 'data.frame'  
as_polars_df(x, ...)
```

```
## S3 method for class '`NULL`'  
as_polars_df(x, ...)
```

### Arguments

|                                   |  |
|-----------------------------------|--|
| <code>x</code>                    | An R object.   |
| <code>...</code>                  | Additional arguments passed to the methods.  |
| <code>column_name</code>          | A character or <code>NULL</code> . If not <code>NULL</code> , name/rename the <a href="#">Series</a> column in the new <a href="#">DataFrame</a> . If <code>NULL</code> , the column name is taken from the <a href="#">Series</a> name.   |
| <code>from_struct</code>          | A logical. If <code>TRUE</code> (default) and the <a href="#">Series</a> data type is a struct, the <code>&lt;Series&gt;\$struct\$unnest()</code> method is used to create a <a href="#">DataFrame</a> from the struct <a href="#">Series</a> . In this case, the <code>column_name</code> argument is ignored.    |
| <code>type_coercion</code>        | A logical, indicates type coercion optimization.   |
| <code>predicate_pushdown</code>   | A logical, indicates predicate pushdown optimization.  |
| <code>projection_pushdown</code>  | A logical, indicates projection pushdown optimization.   |
| <code>simplify_expression</code>  | A logical, indicates simplify expression optimization.   |
| <code>slice_pushdown</code>       | A logical, indicates slice pushdown optimization.  |
| <code>comm_subplan_elim</code>    | A logical, indicates try to cache branching subplans that occur on self-joins or unions.   |
| <code>comm_subexpr_elim</code>    | A logical, indicates try to cache common subexpressions.   |
| <code>cluster_with_columns</code> | A logical, indicates to combine sequential independent calls to <code>with_columns</code> .  |
| <code>no_optimization</code>      | A logical. If <code>TRUE</code> , turn off (certain) optimizations.  |
| <code>streaming</code>            | A logical. If <code>TRUE</code> , process the query in batches to handle larger-than-memory data. If <code>FALSE</code> (default), the entire query is processed in a single batch. Note that streaming mode is considered unstable. It may be changed at any point without it being considered a breaking change. |

### Details

#### Default S3 method:

Basically, this method is a shortcut for `as_polars_series(x, ...)$struct$unnest()`. Before converting the object to a [Series](#), the `infer_polars_dtype()` function is used to check if the object can be converted to a [struct dtype](#).

#### S3 method for [list](#):

- The argument `...` (except `name`) is passed to `as_polars_series()` for each element of the list.

- All elements of the list must be converted to the same length of `Series` by `as_polars_series()`.
- The name of the each element is used as the column name of the `DataFrame`. For unnamed elements, the column name will be an empty string "" or if the element is a `Series`, the column name will be the name of the `Series`.

### S3 method for `data.frame`:

- The argument ... (except `name`) is passed to `as_polars_series()` for each column.
- All columns must be converted to the same length of `Series` by `as_polars_series()`.

### S3 method for `polars_series`:

This is a shortcut for `<Series>$to_frame()` or `<Series>$struct$unnest()`, depending on the `from_struct` argument and the `Series` data type. The `column_name` argument is passed to the `name` argument of the `$to_frame()` method.

### S3 method for `polars_lazy_frame`:

This is a shortcut for `<LazyFrame>$collect()`.

## Value

A polars `DataFrame`

## See Also

- `as.list(<polars_data_frame>)`: Export the `DataFrame` as an R list.
- `as.data.frame(<polars_data_frame>)`: Export the `DataFrame` as an R data frame.

## Examples

```
# list
as_polars_df(list(a = 1:2, b = c("foo", "bar")))

# data.frame
as_polars_df(data.frame(a = 1:2, b = c("foo", "bar")))

# polars_series
s_int <- as_polars_series(1:2, "a")
s_struct <- as_polars_series(
  data.frame(a = 1:2, b = c("foo", "bar")),
  "struct"
)

## Use the Series as a column
as_polars_df(s_int)
as_polars_df(s_struct, column_name = "values", from_struct = FALSE)

## Unnest the struct data
as_polars_df(s_struct)
```

---

|                |  |
|----------------|--|
| as_polars_expr | <i>Create a Polars expression from an R object</i> |
|----------------|--|

---

## Description

The `as_polars_expr()` function creates a polars [expression](#) from various R objects. This function is used internally by various polars functions that accept [expressions](#). In most cases, users should use `pl$lit()` instead of this function, which is a shorthand for `as_polars_expr(x, as_lit = TRUE)`. (In other words, this function can be considered as an internal implementation to realize the `lit` function of the Polars API in other languages.)

## Usage

```
as_polars_expr(x, ...)

## Default S3 method:
as_polars_expr(x, ...)

## S3 method for class 'polars_expr'
as_polars_expr(x, ..., structify = FALSE)

## S3 method for class 'polars_series'
as_polars_expr(x, ...)

## S3 method for class 'character'
as_polars_expr(x, ..., as_lit = FALSE)

## S3 method for class 'logical'
as_polars_expr(x, ...)

## S3 method for class 'integer'
as_polars_expr(x, ...)

## S3 method for class 'double'
as_polars_expr(x, ...)

## S3 method for class 'raw'
as_polars_expr(x, ...)

## S3 method for class '`NULL`'
as_polars_expr(x, ...)
```

## Arguments

|     |   |
|-----|---|
| x   | An R object.                                |
| ... | Additional arguments passed to the methods. |

|                        |  |
|------------------------|--|
| <code>structify</code> | A logical. If <code>TRUE</code> , convert multi-column expressions to a single struct expression by calling <code>pl\$struct()</code> . Otherwise (default), done nothing.   |
| <code>as_lit</code>    | A logical value indicating whether to treat vector as literal values or not. This argument is always set to <code>TRUE</code> when calling this function from <code>pl\$lit()</code> , and expects to return literal values. See examples for details. |

## Details

Because R objects are typically mapped to `Series`, this function often calls `as_polars_series()` internally. However, unlike R, Polars has scalars of length 1, so if an R object is converted to a `Series` of length 1, this function get the first value of the `Series` and convert it to a scalar literal. If you want to implement your own conversion from an R class to a Polars object, define an S3 method for `as_polars_series()` instead of this function.

### Default S3 method:

Create a `Series` by calling `as_polars_series()` and then convert that `Series` to an `Expr`. If the length of the `Series` is 1, it will be converted to a scalar value.

Additional arguments ... are passed to `as_polars_series()`.

### S3 method for character:

If the `as_lit` argument is `FALSE` (default), this function will call `pl$col()` and the character vector is treated as column names.

## Value

A polars [expression](#)

## Literal scalar mapping

Since R has no scalar class, each of the following types of length 1 cases is specially converted to a scalar literal.

- character: String
- logical: Boolean
- integer: Int32
- double: Float64

These types' NA is converted to a `null` literal with casting to the corresponding Polars type. The `raw` type vector is converted to a Binary scalar.

- raw: Binary

`NULL` is converted to a Null type `null` literal.

- `NULL`: Null

For other R class, the default S3 method is called and R object will be converted via `as_polars_series()`. So the type mapping is defined by `as_polars_series()`.

**See Also**

- `as_polars_series()`: R -> Polars type mapping is mostly defined by this function.

**Examples**

```

# character
## as_lit = FALSE (default)
as_polars_expr("a") # Same as `pl$col("a")`
as_polars_expr(c("a", "b")) # Same as `pl$col("a", "b")`

## as_lit = TRUE
as_polars_expr(character(0), as_lit = TRUE)
as_polars_expr("a", as_lit = TRUE)
as_polars_expr(NA_character_, as_lit = TRUE)
as_polars_expr(c("a", "b"), as_lit = TRUE)

# logical
as_polars_expr(logical(0))
as_polars_expr(TRUE)
as_polars_expr(NA)
as_polars_expr(c(TRUE, FALSE))

# integer
as_polars_expr(integer(0))
as_polars_expr(1L)
as_polars_expr(NA_integer_)
as_polars_expr(c(1L, 2L))

# double
as_polars_expr(double(0))
as_polars_expr(1)
as_polars_expr(NA_real_)
as_polars_expr(c(1, 2))

# raw
as_polars_expr(raw(0))
as_polars_expr(charToRaw("foo"))

# NULL
as_polars_expr(NULL)

# default method (for list)
as_polars_expr(list())
as_polars_expr(list(1))
as_polars_expr(list(1, 2))

# default method (for Date)
as_polars_expr(as.Date(integer(0)))
as_polars_expr(as.Date("2021-01-01"))
as_polars_expr(as.Date(c("2021-01-01", "2021-01-02")))

# polars_series

```



```
## Unlike the default method, this method does not extract the first value
as_polars_series(1) |>
  as_polars_expr()

# polars_expr
as_polars_expr(pl$col("a", "b"))
as_polars_expr(pl$col("a", "b"), structify = TRUE)
```

---

as\_polars\_lf

*Create a Polars LazyFrame from an R object*


---

## Description

The `as_polars_lf()` function creates a [LazyFrame](#) from various R objects. 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 'polars_lazy_frame'
as_polars_lf(x, ...)
```

## Arguments

`x` An R object.

`...` Additional arguments passed to the methods.

## Details

### Default S3 method:

Create a [DataFrame](#) by calling `as_polars_df()` and then create a [LazyFrame](#) from the [DataFrame](#). Additional arguments `...` are passed to `as_polars_df()`.

## Value

A polars [LazyFrame](#)

---

as\_polars\_series      *Create a Polars Series from an R object*

---

## Description

The `as_polars_series()` function creates a [polars Series](#) from various R objects. The Data Type of the Series is determined by the class of the input object.

## Usage

```
as_polars_series(x, name = NULL, ...)  
  
## Default S3 method:  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'polars_series'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'polars_data_frame'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'polars_lazy_frame'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'double'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'integer'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'character'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'logical'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'raw'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'factor'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'Date'  
as_polars_series(x, name = NULL, ...)  
  
## S3 method for class 'POSIXct'  
as_polars_series(x, name = NULL, ...)
```

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

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

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

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

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

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

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

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

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

## S3 method for class 'data.frame'
as_polars_series(x, name = NULL, ...)

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

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

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

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

## S3 method for class 'integer64'
as_polars_series(x, name = NULL, ...)

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

```

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

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

## 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 'clock_duration'
as_polars_series(x, name = NULL, ...)

```

## Arguments

|                     |   |
|---------------------|---|
| <code>x</code>      | An R object.  |
| <code>name</code>   | A single string or <code>NULL</code> . Name of the Series. Will be used as a column name when used in a <a href="#">polars DataFrame</a> . When not specified, name is set to an empty string.  |
| <code>...</code>    | Additional arguments passed to the methods.   |
| <code>strict</code> | A logical value to indicate whether throwing an error when the input <a href="#">list</a> 's elements have different data types. If <code>FALSE</code> (default), all elements are automatically cast to the super type, or, casting to the super type is failed, the value will be <code>null</code> . If <code>TRUE</code> , the first non- <code>NULL</code> element's data type is used as the data type of the inner Series. |

## Details

The default method of [as\\_polars\\_series\(\)](#) throws an error, so we need to define S3 methods for the classes we want to support.

### S3 method for [list](#) and [list](#) based classes:

In R, a [list](#) can contain elements of different types, but in Polars (Apache Arrow), all elements must have the same type. So the [as\\_polars\\_series\(\)](#) function automatically casts all elements to the same type or throws an error, depending on the `strict` argument. We can check the [data type](#) of the [Series](#) that will be created from the [list](#) by using the [infer\\_polars\\_dtype\(\)](#) function in advance. If you want to create a list with all elements of the same type in R, consider using the [vctrs::list\\_of\(\)](#) function.

Since a [list](#) can contain another [list](#), the `strict` argument is also used when creating [Series](#) from the inner [list](#) in the case of classes constructed on top of a [list](#), such as [data.frame](#) or [vctrs\\_rcrd](#).

**S3 method for [Date](#):**

Sub-day values will be ignored (floored to the day).

**S3 method for [POSIXct](#):**

Sub-millisecond values will be ignored (floored to the millisecond).

If the `tzzone` attribute is not present or an empty string (`""`), the [Series](#)' `dtype` will be `Datetime without timezone`.

**S3 method for [POSIXlt](#):**

Sub-nanosecond values will be ignored (floored to the nanosecond).

**S3 method for [difftime](#):**

Sub-millisecond values will be rounded to milliseconds.

**S3 method for [hms](#):**

Sub-nanosecond values will be ignored (floored to the nanosecond).

If the `hms` vector contains values greater-equal to 24-o'clock or less than 0-o'clock, an error will be thrown.

**S3 method for [clock\\_duration](#):**

Calendrical durations (years, quarters, months) are treated as chronologically with the internal representation of seconds. Please check the [clock\\_duration](#) documentation for more details.

**S3 methods for [polars\\_data\\_frame](#), [polars\\_lazy\\_frame](#), and `data.frame`:**

These methods are shortcuts for `as_polars_df(x, ...)$to_struct()`. See `as_polars_df()` and `<DataFrame>$to_struct()` for more details.

**Value**

A [polars Series](#)

**See Also**

- `<Series>$to_r_vector()`: Export the Series as an R vector.
- `as_polars_df()`: Create a Polars DataFrame from an R object.
- `infer_polars_dtype()`: Infer the Polars [DataType](#) corresponding to an R object.

**Examples**

```
# double
as_polars_series(c(NA, 1, 2))

# integer
as_polars_series(c(NA, 1:2))

# character
as_polars_series(c(NA, "foo", "bar"))
```

```
# logical
as_polars_series(c(NA, TRUE, FALSE))

# raw
as_polars_series(charToRaw("foo"))

# factor
as_polars_series(factor(c(NA, "a", "b")))

# Date
as_polars_series(as.Date(c(NA, "2021-01-01")))

## Sub-day precision will be ignored
as.Date(c(-0.5, 0, 0.5)) |>
  as_polars_series()

# POSIXct with timezone
as_polars_series(as.POSIXct(c(NA, "2021-01-01 00:00:00.123456789"), "UTC"))

# POSIXct without timezone
as_polars_series(as.POSIXct(c(NA, "2021-01-01 00:00:00.123456789")))

# POSIXlt
as_polars_series(as.POSIXlt(c(NA, "2021-01-01 00:00:00.123456789"), "UTC"))

# difftime
as_polars_series(as.difftime(c(NA, 1), units = "days"))

## Sub-millisecond values will be rounded to milliseconds
as.difftime(c(0.0005, 0.0010, 0.0015, 0.0020), units = "secs") |>
  as_polars_series()

as.difftime(c(0.0005, 0.0010, 0.0015, 0.0020), units = "weeks") |>
  as_polars_series()

# numeric_version
as_polars_series(getRversion())

# NULL
as_polars_series(NULL)

# list
as_polars_series(list(NA, NULL, list(), 1, "foo", TRUE))

## 1st element will be `null` due to the casting failure
as_polars_series(list(list("bar"), "foo"))

# data.frame
as_polars_series(
  data.frame(x = 1:2, y = c("foo", "bar"), z = I(list(1, 2)))
)

# vctrs_unspecified
```

```

if (requireNamespace("vctrs", quietly = TRUE)) {
  as_polars_series(vctrs::unspecified(3L))
}

# hms
if (requireNamespace("hms", quietly = TRUE)) {
  as_polars_series(hms::as_hms(c(NA, "01:00:00")))
}

# blob
if (requireNamespace("blob", quietly = TRUE)) {
  as_polars_series(blob::as_blob(c(NA, "foo", "bar")))
}

# integer64
if (requireNamespace("bit64", quietly = TRUE)) {
  as_polars_series(bit64::as_integer64(c(NA, "9223372036854775807")))
}

# clock_naive_time
if (requireNamespace("clock", quietly = TRUE)) {
  as_polars_series(clock::naive_time_parse(c(
    NA,
    "1900-01-01T12:34:56.123456789",
    "2020-01-01T12:34:56.123456789"
  )), precision = "nanosecond"))
}

# clock_duration
if (requireNamespace("clock", quietly = TRUE)) {
  as_polars_series(clock::duration_nanoseconds(c(NA, 1)))
}

## Calendrical durations are treated as chronologically
if (requireNamespace("clock", quietly = TRUE)) {
  as_polars_series(clock::duration_years(c(NA, 1)))
}

```

---

```
as_tibble.polars_data_frame
```

*Export the polars object as a tibble data frame*

---

## Description

This S3 method is basically a shortcut of `as_polars_df(x, ...)$to_struct()$to_r_vector(ensure_vector = FALSE, struct = "tibble")`. Additionally, you can check or repair the column names by specifying the `.name_repair` argument. Because polars `DataFrame` allows empty column name, which is not generally valid column name in R data frame.

## Usage

```
## S3 method for class 'polars_data_frame'
as_tibble(
  x,
  ...,
  .name_repair = c("check_unique", "unique", "universal", "minimal"),
  int64 = c("double", "character", "integer", "integer64"),
  date = c("Date", "IDate"),
  time = c("hms", "ITime"),
  decimal = c("double", "character"),
  as_clock_class = FALSE,
  ambiguous = c("raise", "earliest", "latest", "null"),
  non_existent = c("raise", "null")
)

## S3 method for class 'polars_lazy_frame'
as_tibble(
  x,
  ...,
  .name_repair = c("check_unique", "unique", "universal", "minimal"),
  int64 = c("double", "character", "integer", "integer64"),
  date = c("Date", "IDate"),
  time = c("hms", "ITime"),
  decimal = c("double", "character"),
  as_clock_class = FALSE,
  ambiguous = c("raise", "earliest", "latest", "null"),
  non_existent = c("raise", "null")
)
```

## Arguments

|                           |   |
|---------------------------|---|
| <code>x</code>            | A polars object   |
| <code>...</code>          | Passed to <code>as_polars_df()</code> .   |
| <code>.name_repair</code> | <p>Treatment of problematic column names:</p> <ul style="list-style-type: none"> <li>• "minimal": No name repair or checks, beyond basic existence,</li> <li>• "unique": Make sure names are unique and not empty,</li> <li>• "check_unique": (default value), no name repair, but check they are unique,</li> <li>• "universal": Make the names unique and syntactic</li> <li>• a function: apply custom name repair (e.g., <code>.name_repair = make.names</code> for names in the style of base R).</li> <li>• A purrr-style anonymous function, see <code>rlang::as_function()</code></li> </ul> <p>This argument is passed on as <code>repair</code> to <code>vctrs::vec_as_names()</code>. See there for more details on these terms and the strategies used to enforce them.</p> |
| <code>int64</code>        | Determine how to convert Polars' Int64, UInt32, or UInt64 type values to R type. One of the followings:   |



|                |   |
|----------------|---|
|                | <ul style="list-style-type: none"> <li>• "double" (default): Convert to the R's <a href="#">double</a> type. Accuracy may be degraded.</li> <li>• "character": Convert to the R's <a href="#">character</a> type.</li> <li>• "integer": Convert to the R's <a href="#">integer</a> type. If the value is out of the range of R's integer type, export as <a href="#">NA_integer_</a>.</li> <li>• "integer64": Convert to the <a href="#">bit64::integer64</a> class. The <a href="#">bit64</a> package must be installed. If the value is out of the range of <a href="#">bit64::integer64</a>, export as <a href="#">bit64::NA_integer64_</a>.</li> </ul>  |
| date           | <p>Determine how to convert Polars' Date type values to R class. One of the followings:</p> <ul style="list-style-type: none"> <li>• "Date" (default): Convert to the R's <a href="#">Date</a> class.</li> <li>• "IDate": Convert to the <a href="#">data.table::IDate</a> class.</li> </ul>  |
| time           | <p>Determine how to convert Polars' Time type values to R class. One of the followings:</p> <ul style="list-style-type: none"> <li>• "hms" (default): Convert to the <a href="#">hms::hms</a> class.</li> <li>• "ITime": Convert to the <a href="#">data.table::ITime</a> class. The <a href="#">data.table</a> package must be installed.</li> </ul>   |
| decimal        | <p>Determine how to convert Polars' Decimal type values to R type. One of the followings:</p> <ul style="list-style-type: none"> <li>• "double" (default): Convert to the R's <a href="#">double</a> type.</li> <li>• "character": Convert to the R's <a href="#">character</a> type.</li> </ul>  |
| as_clock_class | <p>A logical value indicating whether to export datetimes and duration as the <a href="#">clock</a> package's classes.</p> <ul style="list-style-type: none"> <li>• FALSE (default): Duration values are exported as <a href="#">difftime</a> and date-time values are exported as <a href="#">POSIXct</a>. Accuracy may be degraded.</li> <li>• TRUE: Duration values are exported as <a href="#">clock_duration</a>, datetime without timezone values are exported as <a href="#">clock_naive_time</a>, and datetime with timezone values are exported as <a href="#">clock_zoned_time</a>. For this case, the <a href="#">clock</a> package must be installed. Accuracy will be maintained.</li> </ul> |
| ambiguous      | <p>Determine how to deal with ambiguous datetimes. Only applicable when <code>as_clock_class</code> is set to FALSE and datetime without timezone values are exported as <a href="#">POSIXct</a>. Character vector or <a href="#">expression</a> containing the followings:</p> <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 NA value</li> </ul>  |
| non_existent   | <p>Determine how to deal with non-existent datetimes. Only applicable when <code>as_clock_class</code> is set to FALSE and datetime without timezone values are exported as <a href="#">POSIXct</a>. One of the followings:</p> <ul style="list-style-type: none"> <li>• "raise" (default): Throw an error</li> <li>• "null": Return a NA value</li> </ul>  |

**Value**

A [tibble](#)

**See Also**

- [as.data.frame\(<polars\\_object>\)](#): Export the polars object as a basic data frame.

**Examples**

```
# Polars DataFrame may have empty column name
df <- pl$DataFrame(x = 1:2, c("a", "b"))
df

# Without checking or repairing the column names
tibble::as_tibble(df, .name_repair = "minimal")
tibble::as_tibble(df$lazy(), .name_repair = "minimal")

# You can make that unique
tibble::as_tibble(df, .name_repair = "unique")
tibble::as_tibble(df$lazy(), .name_repair = "unique")
```

---

check\_polars

*Check if the object is a polars object*

---

**Description**

Functions to check if the object is a polars object. `is_*` functions return TRUE or FALSE depending on the class of the object. `check_*` functions throw an informative error if the object is not the correct class. Suffixes are corresponding to the polars object classes:

- `*_dtype`: For polars data types.
- `*_df`: For [polars data frames](#).
- `*_expr`: For [polars expressions](#).
- `*_lf`: For [polars lazy frames](#).
- `*_selector`: For [polars selectors](#).
- `*_series`: For [polars series](#).

**Usage**

```
is_polars_dtype(x)
```

```
is_polars_df(x)
```

```
is_polars_expr(x, ...)
```

```
is_polars_lf(x)
```

```
is_polars_selector(x, ...)  
  
is_polars_series(x)  
  
is_list_of_polars_dtype(x, n = NULL)  
  
check_polars_dtype(  
  x,  
  ...,  
  allow_null = FALSE,  
  arg = caller_arg(x),  
  call = caller_env()  
)  
  
check_polars_df(  
  x,  
  ...,  
  allow_null = FALSE,  
  arg = caller_arg(x),  
  call = caller_env()  
)  
  
check_polars_expr(  
  x,  
  ...,  
  allow_null = FALSE,  
  arg = caller_arg(x),  
  call = caller_env()  
)  
  
check_polars_lf(  
  x,  
  ...,  
  allow_null = FALSE,  
  arg = caller_arg(x),  
  call = caller_env()  
)  
  
check_polars_selector(  
  x,  
  ...,  
  allow_null = FALSE,  
  arg = caller_arg(x),  
  call = caller_env()  
)  
  
check_polars_series(  
  x,  
  ...,  
  allow_null = FALSE,  
  arg = caller_arg(x),  
  call = caller_env()  
)
```

```

    x,
    ...,
    allow_null = FALSE,
    arg = caller_arg(x),
    call = caller_env()
)

check_list_of_polars_dtype(
  x,
  ...,
  allow_null = FALSE,
  arg = caller_arg(x),
  call = caller_env()
)

```

### Arguments

|                         |   |
|-------------------------|---|
| <code>x</code>          | An object to check.   |
| <code>...</code>        | Arguments passed to <code>rlang::abort()</code> .   |
| <code>n</code>          | Expected length of a vector.  |
| <code>allow_null</code> | If TRUE, NULL is allowed as a valid input.  |
| <code>arg</code>        | An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.  |
| <code>call</code>       | The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of <code>abort()</code> for more information. |

### Details

`check_polars_*` functions are derived from the `standalone-types-check` functions from the `rlang` package (Can be installed with `usethis::use_standalone("r-lib/rlang", file = "types-check")`).

### Value

- `is_polars_*` functions return TRUE or FALSE.
- `check_polars_*` functions return NULL invisibly if the input is valid.

### See Also

- `infer_polars_dtype()`: Check if the object can be converted to a [Series](#).

### Examples

```

is_polars_df(as_polars_df(mtcars))
is_polars_df(mtcars)

# Use `check_polars_*` functions in a function
# to ensure the input is a polars object

```

```

sample_func <- function(x) {
  check_polars_df(x)
  TRUE
}

sample_func(as_polars_df(mtcars))
try(sample_func(mtcars))

```

---

cs

*Polars column selector function namespace*


---

## Description

`cs` is an [environment class](#) object that stores all selector functions of the R Polars API which mimics the Python Polars API. It is intended to work the same way in Python as if you had imported Python Polars Selectors with `import polars.selectors as cs`.

## Usage

```
cs
```

## Format

An object of class `polars_object` of length 29.

## Supported operators

There are 4 supported operators for selectors:

- `&` to combine conditions with AND, e.g. select columns that contain "oo" *and* end with "t" with `cs$contains("oo") & cs$ends_with("t");`
- `|` to combine conditions with OR, e.g. select columns that contain "oo" *or* end with "t" with `cs$contains("oo") | cs$ends_with("t");`
- `-` to subtract conditions, e.g. select all columns that have alphanumeric names except those that contain "a" with `cs$alphanumeric() - cs$contains("a");`
- `!` to invert the selection, e.g. select all columns that *are not* of data type `String` with `!cs$string()`.

Note that Python Polars uses `~` instead of `!` to invert selectors.

## Examples

```

cs

# How many members are in the `cs` environment?
length(cs)

```

---

|         |                           |
|---------|---------------------------|
| cs__all | <i>Select all columns</i> |
|---------|---------------------------|

---

**Description**

Select all columns

**Usage**

```
cs__all()
```

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(dt = as.Date(c("2000-1-1")), value = 10)

# Select all columns, casting them to string:
df$select(cs$all())$cast(pl$String)

# Select all columns except for those matching the given dtypes:
df$select(cs$all() - cs$numeric())
```

---

|           |   |
|-----------|---|
| cs__alpha | <i>Select all columns with alphabetic names (e.g. only letters)</i> |
|-----------|---|

---

**Description**

Select all columns with alphabetic names (e.g. only letters)

**Usage**

```
cs__alpha(ascii_only = FALSE, ..., ignore_spaces = FALSE)
```

**Arguments**

|                            |   |
|----------------------------|---|
| <code>ascii_only</code>    | Indicate whether to consider only ASCII alphabetic characters, or the full Unicode range of valid letters (accented, idiographic, etc). |
| <code>...</code>           | These dots are for future extensions and must be empty.   |
| <code>ignore_spaces</code> | Indicate whether to ignore the presence of spaces in column names; if so, only the other (non-space) characters are considered.         |

**Details**

Matching column names cannot contain any non-alphabetic characters. Note that the definition of “alphabetic” consists of all valid Unicode alphabetic characters (`p{Alphabetic}`) by default; this can be changed by setting `ascii_only = TRUE`.

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(
  no1 = c(100, 200, 300),
  café = c("espresso", "latte", "mocha"),
  `t or f` = c(TRUE, FALSE, NA),
  hmm = c("aaa", "bbb", "ccc"),
  = c(" ", " ", " ")
)

# Select columns with alphabetic names; note that accented characters and
# kanji are recognised as alphabetic here:
df$select(cs$alpha())

# Constrain the definition of "alphabetic" to ASCII characters only:
df$select(cs$alpha(ascii_only = TRUE))
df$select(cs$alpha(ascii_only = TRUE, ignore_spaces = TRUE))

# Select all columns except for those with alphabetic names:
df$select(!cs$alpha())
df$select(!cs$alpha(ignore_spaces = TRUE))
```

---

|                               |  |
|-------------------------------|--|
| <code>cs__alphanumeric</code> | <i>Select all columns with alphanumeric names (e.g. only letters and the digits 0-9)</i> |
|-------------------------------|--|

---

**Description**

Select all columns with alphanumeric names (e.g. only letters and the digits 0-9)

**Usage**

```
cs__alphanumeric(ascii_only = FALSE, ..., ignore_spaces = FALSE)
```

**Arguments**

`ascii_only`      Indicate whether to consider only ASCII alphabetic characters, or the full Unicode range of valid letters (accented, idiographic, etc).

`...`            These dots are for future extensions and must be empty.

`ignore_spaces`   Indicate whether to ignore the presence of spaces in column names; if so, only the other (non-space) characters are considered.

**Details**

Matching column names cannot contain any non-alphabetic characters. Note that the definition of “alphabetic” consists of all valid Unicode alphabetic characters (`p{Alphabetic}`) and digit characters (`d`) by default; this can be changed by setting `ascii_only = TRUE`.

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(
  `1st_col` = c(100, 200, 300),
  flagged = c(TRUE, FALSE, TRUE),
  `00prefix` = c("01:aa", "02:bb", "03:cc"),
  `last_col` = c("x", "y", "z")
)

# Select columns with alphanumeric names:
df$select(cs$alphanumeric())
df$select(cs$alphanumeric(ignore_spaces = TRUE))

# Select all columns except for those with alphanumeric names:
df$select(!cs$alphanumeric())
df$select(!cs$alphanumeric(ignore_spaces = TRUE))
```

---

`cs__binary`

*Select all binary columns*

---

**Description**

Select all binary columns

**Usage**

```
cs__binary()
```



**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(  
  a = charToRaw("hello"),  
  b = "world",  
  c = charToRaw("!"),  
  d = ":"  
)  
  
# Select binary columns:  
df$select(cs$binary())  
  
# Select all columns except for those that are binary:  
df$select(!cs$binary())
```

---

cs\_\_boolean

*Select all boolean columns*

---

**Description**

Select all boolean columns

**Usage**

```
cs__boolean()
```

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(  
  a = 1:4,  
  b = c(FALSE, TRUE, FALSE, TRUE)  
)  
  
# Select and invert boolean columns:  
df$with_columns(inverted = cs$boolean()$not())
```

```
# Select all columns except for those that are boolean:
df$select(!cs$boolean())
```

---

|              |   |
|--------------|---|
| cs__by_dtype | <i>Select all columns matching the given dtypes</i> |
|--------------|---|

---

## Description

Select all columns matching the given dtypes

## Usage

```
cs__by_dtype(...)
```

## Arguments

... [<dynamic-dots>](#) Data types to select.

## Value

A Polars selector

## See Also

[cs](#) for the documentation on operators supported by Polars selectors.

## Examples

```
df <- pl$DataFrame(
  dt = as.Date(c("1999-12-31", "2024-1-1", "2010-7-5")),
  value = c(1234500, 5000555, -4500000),
  other = c("foo", "bar", "foo")
)

# Select all columns with date or string dtypes:
df$select(cs$by_dtype(pl$Date, pl$string))

# Select all columns that are not of date or string dtype:
df$select(!cs$by_dtype(pl$Date, pl$string))

# Group by string columns and sum the numeric columns:
df$group_by(cs$string())$agg(cs$numeric())$sum()$sort("other")
```

---

|              |  |
|--------------|--|
| cs__by_index | Select all columns matching the given indices (or range objects) |
|--------------|--|

---

## Description

Select all columns matching the given indices (or range objects)

## Usage

```
cs__by_index(indices)
```

## Arguments

**indices** One or more column indices (or ranges). Negative indexing is supported.

## Details

Matching columns are returned in the order in which their indexes appear in the selector, not the underlying schema order.

## Value

A Polars selector

## See Also

[cs](#) for the documentation on operators supported by Polars selectors.

## Examples

```
vals <- as.list(0.5 * 0:100)
names(vals) <- paste0("c", 0:100)
df <- pl$DataFrame(!!!vals)
df

# Select columns by index (the two first/last columns):
df$select(cs$by_index(c(0, 1, -2, -1)))

# Use seq()
df$select(cs$by_index(c(0, seq(1, 101, 20))))
df$select(cs$by_index(c(0, seq(101, 0, -25))))

# Select only odd-indexed columns:
df$select(!cs$by_index(seq(0, 100, 2)))
```

---

 cs\_\_by\_name

*Select all columns matching the given names*


---

### Description

Select all columns matching the given names

### Usage

```
cs__by_name(..., require_all = TRUE)
```

### Arguments

...            <dynamic-dots> Column names to select.  
 require\_all    Whether to match all names (the default) or any of the names.

### Details

Matching columns are returned in the order in which their indexes appear in the selector, not the underlying schema order.

### Value

A Polars selector

### See Also

[cs](#) for the documentation on operators supported by Polars selectors.

### Examples

```
df <- pl$DataFrame(
  foo = c("x", "y"),
  bar = c(123, 456),
  baz = c(2.0, 5.5),
  zap = c(FALSE, TRUE)
)

# Select columns by name:
df$select(cs$by_name("foo", "bar"))

# Match any of the given columns by name:
df$select(cs$by_name("baz", "moose", "foo", "bear", require_all = FALSE))

# Match all columns except for those given:
df$select(!cs$by_name("foo", "bar"))
```

---

|                 |                                       |
|-----------------|---------------------------------------|
| cs__categorical | <i>Select all categorical columns</i> |
|-----------------|---------------------------------------|

---

**Description**

Select all categorical columns

**Usage**

```
cs__categorical()
```

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(  
  foo = c("xx", "yy"),  
  bar = c(123, 456),  
  baz = c(2.0, 5.5),  
  .schema_overrides = list(foo = pl$Categorical()),  
)  
  
# Select categorical columns:  
df$select(cs$categorical())  
  
# Select all columns except for those that are categorical:  
df$select(!cs$categorical())
```

---

|              |  |
|--------------|--|
| cs__contains | <i>Select columns whose names contain the given literal substring(s)</i> |
|--------------|--|

---

**Description**

Select columns whose names contain the given literal substring(s)

**Usage**

```
cs__contains(...)
```

**Arguments**

... [<dynamic-dots>](#) Substring(s) that matching column names should contain.

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(
  foo = c("x", "y"),
  bar = c(123, 456),
  baz = c(2.0, 5.5),
  zap = c(FALSE, TRUE)
)

# Select columns that contain the substring "ba":
df$select(cs$contains("ba"))

# Select columns that contain the substring "ba" or the letter "z":
df$select(cs$contains("ba", "z"))

# Select all columns except for those that contain the substring "ba":
df$select(!cs$contains("ba"))
```

---

cs\_\_date

*Select all date columns*

---

**Description**

Select all date columns

**Usage**

```
cs__date()
```

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(
  dtm = as.POSIXct(c("2001-5-7 10:25", "2031-12-31 00:30")),
  dt = as.Date(c("1999-12-31", "2024-8-9"))
)

# Select date columns:
df$select(cs$date())

# Select all columns except for those that are dates:
df$select(!cs$date())
```

---

|              |                                    |
|--------------|------------------------------------|
| cs__datetime | <i>Select all datetime columns</i> |
|--------------|------------------------------------|

---

**Description**

Select all datetime columns

**Usage**

```
cs__datetime(time_unit = c("ms", "us", "ns"), time_zone = list("*", NULL))
```

**Arguments**

|                        |   |
|------------------------|---|
| <code>time_unit</code> | One (or more) of the allowed time unit precision strings, "ms", "us", and "ns". Default is to select columns with any valid timeunit.   |
| <code>time_zone</code> | One of the followings. The value or each element of the vector will be passed to the <code>time_zone</code> argument of the <code>pl\$Datetime()</code> function: <ul style="list-style-type: none"> <li>• A character vector of one or more timezone strings, as defined in <a href="#">OlsonNames()</a>.</li> <li>• <code>NULL</code> to select Datetime columns that do not have a timezone.</li> <li>• <code>"*"</code> to select Datetime columns that have any timezone.</li> <li>• A list of single timezone strings, <code>"*"</code>, and <code>NULL</code> to select Datetime columns that do not have a timezone or have the (specific) timezone. For example, the default value <code>list("*", NULL)</code> selects all Datetime columns.</li> </ul> |

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

## Examples

```
chr_vec <- c("1999-07-21 05:20:16.987654", "2000-05-16 06:21:21.123456")
df <- pl$DataFrame(
  tstamp_tokyo = as.POSIXlt(chr_vec, tz = "Asia/Tokyo"),
  tstamp_utc = as.POSIXct(chr_vec, tz = "UTC"),
  tstamp = as.POSIXct(chr_vec),
  dt = as.Date(chr_vec),
)

# Select all datetime columns:
df$select(cs$datetime())

# Select all datetime columns that have "ms" precision:
df$select(cs$datetime("ms"))

# Select all datetime columns that have any timezone:
df$select(cs$datetime(time_zone = "*"))

# Select all datetime columns that have a specific timezone:
df$select(cs$datetime(time_zone = "UTC"))

# Select all datetime columns that have NO timezone:
df$select(cs$datetime(time_zone = NULL))

# Select all columns except for datetime columns:
df$select(!cs$datetime())
```

---

cs\_\_decimal

*Select all decimal columns*

---

## Description

Select all decimal columns

## Usage

```
cs__decimal()
```

## Value

A Polars selector

## See Also

[cs](#) for the documentation on operators supported by Polars selectors.



**Examples**

```
df <- pl$DataFrame(
  foo = c("x", "y"),
  bar = c(123, 456),
  baz = c("2.0005", "-50.5555"),
  .schema_overrides = list(
    bar = pl$Decimal(),
    baz = pl$Decimal(scale = 5, precision = 10)
  )
)

# Select decimal columns:
df$select(cs$decimal())

# Select all columns except for those that are decimal:
df$select(!cs$decimal())
```

---

cs\_\_digit

*Select all columns having names consisting only of digits*


---

**Description**

Select all columns having names consisting only of digits

**Usage**

```
cs__digit(ascii_only = FALSE)
```

**Arguments**

**ascii\_only**      Indicate whether to consider only ASCII alphabetic characters, or the full Unicode range of valid letters (accented, idiographic, etc).

**Details**

Matching column names cannot contain any non-digit characters. Note that the definition of "digit" consists of all valid Unicode digit characters (d) by default; this can be changed by setting `ascii_only = TRUE`.

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(
  key = c("aaa", "bbb"),
  `2001` = 1:2,
  `2025` = 3:4
)

# Select columns with digit names:
df$select(cs$digit())

# Select all columns except for those with digit names:
df$select(!cs$digit())

# Demonstrate use of ascii_only flag (by default all valid unicode digits
# are considered, but this can be constrained to ascii 0-9):
df <- pl$DataFrame(` ` = 1999, ` ` = 2077, `3000` = 3000)
df$select(cs$digit())
df$select(cs$digit(ascii_only = TRUE))
```

---

**cs\_\_duration**
*Select all duration columns, optionally filtering by time unit*


---

**Description**

Select all duration columns, optionally filtering by time unit

**Usage**

```
cs__duration(time_unit = c("ms", "us", "ns"))
```

**Arguments**

**time\_unit** One (or more) of the allowed time unit precision strings, "ms", "us", and "ns". Default is to select columns with any valid timeunit.

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(
  dtm = as.POSIXct(c("2001-5-7 10:25", "2031-12-31 00:30")),
  dur_ms = clock::duration_milliseconds(1:2),
```

```

dur_us = clock::duration_microseconds(1:2),
dur_ns = clock::duration_nanoseconds(1:2),
)

# Select duration columns:
df$select(cs$duration())

# Select all duration columns that have "ms" precision:
df$select(cs$duration("ms"))

# Select all duration columns that have "ms" OR "ns" precision:
df$select(cs$duration(c("ms", "ns")))

# Select all columns except for those that are duration:
df$select(!cs$duration())

```

---

|               |  |
|---------------|--|
| cs__ends_with | <i>Select columns that end with the given substring(s)</i> |
|---------------|--|

---

## Description

Select columns that end with the given substring(s)

## Usage

```
cs__ends_with(...)
```

## Arguments

... [<dynamic-dots>](#) Substring(s) that matching column names should end with.

## Value

A Polars selector

## See Also

[cs](#) for the documentation on operators supported by Polars selectors.

## Examples

```

df <- pl$DataFrame(
  foo = c("x", "y"),
  bar = c(123, 456),
  baz = c(2.0, 5.5),
  zap = c(FALSE, TRUE)
)

```

```
# Select columns that end with the substring "z":
df$select(cs$ends_with("z"))

# Select columns that end with either the letter "z" or "r":
df$select(cs$ends_with("z", "r"))

# Select all columns except for those that end with the substring "z":
df$select(!cs$ends_with("z"))
```

---

|                          |  |
|--------------------------|--|
| <code>cs__exclude</code> | <i>Select all columns except those matching the given columns, datatypes, or selectors</i> |
|--------------------------|--|

---

## Description

Select all columns except those matching the given columns, datatypes, or selectors

## Usage

```
cs__exclude(...)
```

## Arguments

... [<dynamic-dots>](#) Column names to exclude.

## Details

If excluding a single selector it is simpler to write as `!selector` instead.

## Value

A Polars selector

## See Also

[cs](#) for the documentation on operators supported by Polars selectors.

## Examples

```
df <- pl$DataFrame(
  aa = 1:3,
  ba = c("a", "b", NA),
  cc = c(NA, 2.5, 1.5)
)

# Exclude by column name(s):
df$select(cs$exclude("ba", "xx"))

# Exclude using a column name, a selector, and a dtype:
df$select(cs$exclude("aa", cs$string(), pl$Int32))
```

---

|           |   |
|-----------|---|
| cs__first | <i>Select the first column in the current scope</i> |
|-----------|---|

---

**Description**

Select the first column in the current scope

**Usage**

```
cs__first()
```

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(  
  foo = c("x", "y"),  
  bar = c(123L, 456L),  
  baz = c(2.0, 5.5),  
  zap = c(FALSE, TRUE)  
)  
  
# Select the first column:  
df$select(cs$first())  
  
# Select everything except for the first column:  
df$select(!cs$first())
```

---

|           |                                  |
|-----------|----------------------------------|
| cs__float | <i>Select all float columns.</i> |
|-----------|----------------------------------|

---

**Description**

Select all float columns.

**Usage**

```
cs__float()
```

**Value**

A Polars selector

## See Also

[cs](#) for the documentation on operators supported by Polars selectors.

## Examples

```
df <- pl$DataFrame(  
  foo = c("x", "y"),  
  bar = c(123L, 456L),  
  baz = c(2.0, 5.5),  
  zap = c(FALSE, TRUE),  
  .schema_overrides = list(baz = pl$Float32, zap = pl$Float64),  
)  
  
# Select all float columns:  
df$select(cs$float())  
  
# Select all columns except for those that are float:  
df$select(!cs$float())
```

---

cs\_\_integer

*Select all integer columns.*

---

## Description

Select all integer columns.

## Usage

```
cs__integer()
```

## Value

A Polars selector

## See Also

[cs](#) for the documentation on operators supported by Polars selectors.

## Examples

```
df <- pl$DataFrame(  
  foo = c("x", "y"),  
  bar = c(123L, 456L),  
  baz = c(2.0, 5.5),  
  zap = 0:1  
)  
  
# Select all integer columns:  
df$select(cs$integer())
```

```
# Select all columns except for those that are integer:
df$select(!cs$integer())
```

---

|          |  |
|----------|--|
| cs__last | <i>Select the last column in the current scope</i> |
|----------|--|

---

### Description

Select the last column in the current scope

### Usage

```
cs__last()
```

### Value

A Polars selector

### See Also

[cs](#) for the documentation on operators supported by Polars selectors.

### Examples

```
df <- pl$DataFrame(
  foo = c("x", "y"),
  bar = c(123L, 456L),
  baz = c(2.0, 5.5),
  zap = c(FALSE, TRUE)
)

# Select the last column:
df$select(cs$last())

# Select everything except for the last column:
df$select(!cs$last())
```

---

|             |  |
|-------------|--|
| cs__matches | <i>Select all columns that match the given regex pattern</i> |
|-------------|--|

---

### Description

Select all columns that match the given regex pattern

### Usage

```
cs__matches(pattern)
```

**Arguments**

**pattern** A valid regular expression pattern, compatible with the `regex` crate [crate <https://docs.rs/regex>](https://docs.rs/regex).

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(
  foo = c("x", "y"),
  bar = c(123, 456),
  baz = c(2.0, 5.5),
  zap = c(0, 1)
)

# Match column names containing an "a", preceded by a character that is not
# "z":
df$select(cs$matches("[^z]a"))

# Do not match column names ending in "R" or "z" (case-insensitively):
df$select(!cs$matches(r"((?i)R|z$")))
```

---

|                          |                                    |
|--------------------------|------------------------------------|
| <code>cs__numeric</code> | <i>Select all numeric columns.</i> |
|--------------------------|------------------------------------|

---

**Description**

Select all numeric columns.

**Usage**

```
cs__numeric()
```

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.



## Examples

```
df <- pl$DataFrame(  
  foo = c("x", "y"),  
  bar = c(123L, 456L),  
  baz = c(2.0, 5.5),  
  zap = 0:1,  
  .schema_overrides = list(bar = pl$Int16, baz = pl$Float32, zap = pl$UInt8),  
)  
  
# Select all numeric columns:  
df$select(cs$numeric())  
  
# Select all columns except for those that are numeric:  
df$select(!cs$numeric())
```

---

cs\_\_signed\_integer    *Select all signed integer columns*

---

## Description

Select all signed integer columns

## Usage

```
cs__signed_integer()
```

## Value

A Polars selector

## See Also

[cs](#) for the documentation on operators supported by Polars selectors.

## Examples

```
df <- pl$DataFrame(  
  foo = c(-123L, -456L),  
  bar = c(3456L, 6789L),  
  baz = c(7654L, 4321L),  
  zap = c("ab", "cd"),  
  .schema_overrides = list(bar = pl$UInt32, baz = pl$UInt64),  
)  
  
# Select signed integer columns:  
df$select(cs$signed_integer())  
  
# Select all columns except for those that are signed integer:  
df$select(!cs$signed_integer())
```

```
# Select all integer columns (both signed and unsigned):
df$select(cs$integer())
```

---

cs\_\_starts\_with      *Select columns that start with the given substring(s)*

---

## Description

Select columns that start with the given substring(s)

## Usage

```
cs__starts_with(...)
```

## Arguments

...      `<dynamic-dots>` Substring(s) that matching column names should end with.

## Value

A Polars selector

## See Also

[cs](#) for the documentation on operators supported by Polars selectors.

## Examples

```
df <- pl$DataFrame(
  foo = c("x", "y"),
  bar = c(123, 456),
  baz = c(2.0, 5.5),
  zap = c(FALSE, TRUE)
)

# Select columns that start with the substring "b":
df$select(cs$starts_with("b"))

# Select columns that start with either the letter "b" or "z":
df$select(cs$starts_with("b", "z"))

# Select all columns except for those that start with the substring "b":
df$select(!cs$starts_with("b"))
```

---

|            |   |
|------------|---|
| cs__string | <i>Select all String (and, optionally, Categorical) string columns.</i> |
|------------|---|

---

**Description**

Select all String (and, optionally, Categorical) string columns.

**Usage**

```
cs__string(..., include_categorical = FALSE)
```

**Arguments**

... These dots are for future extensions and must be empty.

include\_categorical  
If TRUE, also select categorical columns.

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(
  w = c("xx", "yy", "xx", "yy", "xx"),
  x = c(1, 2, 1, 4, -2),
  y = c(3.0, 4.5, 1.0, 2.5, -2.0),
  z = c("a", "b", "a", "b", "b")
)$with_columns(
  z = pl$col("z")$cast(pl$Categorical())
)

# Group by all string columns, sum the numeric columns, then sort by the
# string cols:
df$group_by(cs$string())$agg(cs$numeric()$sum())$sort(cs$string())

# Group by all string and categorical columns:
df$
  group_by(cs$string(include_categorical = TRUE))$
  agg(cs$numeric()$sum())$
  sort(cs$string(include_categorical = TRUE))
```

---

|              |                                    |
|--------------|------------------------------------|
| cs__temporal | <i>Select all temporal columns</i> |
|--------------|------------------------------------|

---

### Description

Select all temporal columns

### Usage

```
cs__temporal()
```

### Value

A Polars selector

### See Also

[cs](#) for the documentation on operators supported by Polars selectors.

### Examples

```
df <- pl$DataFrame(  
  dtm = as.POSIXct(c("2001-5-7 10:25", "2031-12-31 00:30")),  
  dt = as.Date(c("1999-12-31", "2024-8-9")),  
  value = 1:2  
)  
  
# Match all temporal columns:  
df$select(cs$temporal())  
  
# Match all temporal columns except for time columns:  
df$select(cs$temporal() - cs$datetime())  
  
# Match all columns except for temporal columns:  
df$select(!cs$temporal())
```

---

|          |                                |
|----------|--------------------------------|
| cs__time | <i>Select all time columns</i> |
|----------|--------------------------------|

---

### Description

Select all time columns

### Usage

```
cs__time()
```

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(
  dtm = as.POSIXct(c("2001-5-7 10:25", "2031-12-31 00:30")),
  dt = as.Date(c("1999-12-31", "2024-8-9")),
  tm = hms::parse_hms(c("0:0:0", "23:59:59"))
)

# Select time columns:
df$select(cs$time())

# Select all columns except for those that are time:
df$select(!cs$time())
```

---

`cs__unsigned_integer` *Select all unsigned integer columns*

---

**Description**

Select all unsigned integer columns

**Usage**

```
cs__unsigned_integer()
```

**Value**

A Polars selector

**See Also**

[cs](#) for the documentation on operators supported by Polars selectors.

**Examples**

```
df <- pl$DataFrame(
  foo = c(-123L, -456L),
  bar = c(3456L, 6789L),
  baz = c(7654L, 4321L),
  zap = c("ab", "cd"),
  .schema_overrides = list(bar = pl$UInt32, baz = pl$UInt64),
```

```

)

# Select unsigned integer columns:
df$select(cs$unsigned_integer())

# Select all columns except for those that are unsigned integer:
df$select(!cs$unsigned_integer())

# Select all integer columns (both unsigned and signed):
df$select(cs$integer())

```

---

dataframe\_\_bottom\_k *Return the k smallest rows*

---

## Description

Non-null elements are always preferred over null elements, regardless of the value of **reverse**. The output is not guaranteed to be in any particular order, call `sort()` after this function if you wish the output to be sorted.

## Usage

```
dataframe__bottom_k(k, ..., by, reverse = FALSE)
```

## Arguments

|                |  |
|----------------|--|
| <b>k</b>       | Number of rows to return.  |
| <b>...</b>     | These dots are for future extensions and must be empty.  |
| <b>by</b>      | Column(s) used to determine the bottom rows. Accepts expression input. Strings are parsed as column names.   |
| <b>reverse</b> | Consider the k largest elements of the by column(s) (instead of the k smallest). This can be specified per column by passing a sequence of booleans. |

## Value

A polars [DataFrame](#)

## Examples

```

df <- pl$DataFrame(
  a = c("a", "b", "a", "b", "b", "c"),
  b = c(2, 1, 1, 3, 2, 1)
)

# Get the rows which contain the 4 smallest values in column b.
df$bottom_k(4, by = "b")

# Get the rows which contain the 4 smallest values when sorting on column a

```

```
# and b$
df$bottom_k(4, by = c("a", "b"))
```

---

|                 |  |
|-----------------|--|
| 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(..., .strict = TRUE)
```

## Arguments

|         |  |
|---------|--|
| ...     | < <a href="#">dynamic-dots</a> > Either a datatype to which all columns will be cast, or a list where the names are column names and the values are the datatypes to convert to. |
| .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 polars [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$cast(foo = pl$Float32, bar = pl$UInt8)

# Cast all columns to the same type
df$cast(pl$String)
```

---

`dataframe__clear`      *Create an empty or n-row null-filled copy of the frame*

---

### Description

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

### Usage

```
dataframe__clear(n = 0)
```

### Arguments

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

### Value

A polars [DataFrame](#)

### 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 = 2)
```

---

`dataframe__clone`      *Clone a DataFrame*

---

### Description

This is a cheap operation that does not copy data. Assigning does not copy the `DataFrame` (environment object). This is because environment objects have reference semantics. Calling `$clone()` creates a new environment, which can be useful when dealing with attributes (see examples).

### Usage

```
dataframe__clone()
```

### Value

A polars [DataFrame](#)



## Examples

```
df1 <- as_polars_df(iris)

# Assigning does not copy the DataFrame (environment object), calling
# $clone() creates a new environment.
df2 <- df1
df3 <- df1$clone()
rlang::env_label(df1)
rlang::env_label(df2)
rlang::env_label(df3)

# Cloning can be useful to add attributes to data used in a function without
# adding those attributes to the original object.

# 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 <- as_polars_df(iris)
df2 <- give_attr(df1)

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

---

dataframe\_\_drop

*Remove columns*

---

## Description

Remove columns

## Usage

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

**Arguments**

`...` **<dynamic-dots>** Names of the columns that should be removed. Accepts column selector input.

`strict` Validate that all column names exist in the current schema, and throw an exception if any do not.

**Value**

A polars [DataFrame](#)

**Examples**

```
# Drop columns by passing the name of those columns
df <- pl$DataFrame(
  foo = 1:3,
  bar = c(6, 7, 8),
  ham = c("a", "b", "c")
)
df$drop("ham")
df$drop("ham", "bar")

# Drop multiple columns by passing a selector
df$drop(cs$all())
```

---

`dataframe__drop_nans` *Drop all rows that contain NaN values*

---

**Description**

The original order of the remaining rows is preserved.

**Usage**

```
dataframe__drop_nans(...)
```

**Arguments**

`...` **<dynamic-dots>** Column name(s) for which null values are considered. If empty (default), use all columns (note that only floating-point columns can contain NaNs).

**Value**

A polars [DataFrame](#)

## Examples

```
df <- pl$DataFrame(
  foo = c(1, NaN, 2.5),
  bar = c(NaN, 110, 25.5),
  ham = c("a", "b", NA)
)

# The default behavior of this method is to drop rows where any single value
# of the row is null.
df$drop_nans()

# This behaviour can be constrained to consider only a subset of columns, as
# defined by name or with a selector. For example, dropping rows if there is
# a null in the "bar" column:
df$drop_nans("bar")

# Dropping a row only if all values are NaN requires a different
# formulation:
df <- pl$DataFrame(
  a = c(NaN, NaN, NaN, NaN),
  b = c(10.0, 2.5, NaN, 5.25),
  c = c(65.75, NaN, NaN, 10.5)
)
df$filter(!pl$all_horizontal(pl$all()$is_nan()))
```

---

dataframe\_\_drop\_nulls

*Drop all rows that contain null values*

---

## Description

The original order of the remaining rows is preserved.

## Usage

```
dataframe__drop_nulls(...)
```

## Arguments

... [<dynamic-dots>](#) Column name(s) for which null values are considered.  
If empty (default), use all columns.

## Value

A polars [DataFrame](#)

## Examples

```
df <- pl$DataFrame(  
  foo = 1:3,  
  bar = c(6L, NA, 8L),  
  ham = c("a", "b", NA)  
)  
  
# The default behavior of this method is to drop rows where any single value  
# of the row is null.  
df$drop_nulls()  
  
# This behaviour can be constrained to consider only a subset of columns, as  
# defined by name or with a selector. For example, dropping rows if there is  
# a null in any of the integer columns:  
df$drop_nulls(cs$integer())
```

---

dataframe\_\_equals      *Check whether the DataFrame is equal to another DataFrame*

---

## Description

Check whether the DataFrame is equal to another DataFrame

## Usage

```
dataframe__equals(other, ..., null_equal = TRUE)
```

## Arguments

**other**              DataFrame to compare with.  
**null\_equal**        Consider null values as equal.

## Value

A logical value

## Examples

```
dat1 <- as_polars_df(iris)  
dat2 <- as_polars_df(iris)  
dat3 <- as_polars_df(mtcars)  
dat1$equals(dat2)  
dat1$equals(dat3)
```

---

`dataframe__explode` *Explode the frame to long format by exploding the given columns*

---

### Description

Explode the frame to long format by exploding the given columns

### Usage

```
dataframe__explode(...)
```

### Arguments

... [<dynamic-dots>](#) Column names, expressions, or a selector defining them. The underlying columns being exploded must be of the `List` or `Array` data type.

### Value

A polars [DataFrame](#)

### Examples

```
df <- pl$DataFrame(  
  letters = c("a", "a", "b", "c"),  
  numbers = list(1, c(2, 3), c(4, 5), c(6, 7, 8))  
)  
  
df$explode("numbers")
```

---

`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

A polars [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 null values using the specified value or strategy*

---

## Description

Fill null values using the specified value or strategy

## Usage

```
dataframe__fill_null(
  value,
  strategy = NULL,
  limit = NULL,
  ...,
  matches_supertype = TRUE
)
```

## Arguments

|                                |   |
|--------------------------------|---|
| <code>value</code>             | Value used to fill null values.   |
| <code>strategy</code>          | Strategy used to fill null values. Must be one of: "forward", "backward", "min", "max", "mean", "zero", "one", or NULL (default). |
| <code>limit</code>             | Number of consecutive null values to fill when using the "forward" or "backward" strategy.  |
| <code>...</code>               | These dots are for future extensions and must be empty.   |
| <code>matches_supertype</code> | Fill all matching supertypes of the fill <code>value</code> literal.  |

## Value

A polars [DataFrame](#)

## Examples

```
df <- pl$DataFrame(
  a = c(1.5, 2, NA, 4),
  b = c(1.5, NA, NA, 4)
)
df$fill_null(99)
```

```
df$fill_null(strategy = "forward")  
df$fill_null(strategy = "max")  
df$fill_null(strategy = "zero")
```

---

dataframe\_\_filter      *Filter rows of a DataFrame*

---

## Description

The original order of the remaining rows is preserved. Rows where the filter does not evaluate to TRUE are discarded, including nulls.

## Usage

```
dataframe__filter(...)
```

## Arguments

...                    <dynamic-dots> Expression that evaluates to a boolean Series.

## Value

A polars [DataFrame](#)

## Examples

```
df <- as_polars_df(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 <- as_polars_df(iris2)  
  
df$filter(pl$col("Species") == "setosa")
```

`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 polars [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 a single column by name*

---

### Description

Get a single column by name

### Usage

```
dataframe__get_column(name)
```

### Arguments

`name` Name of the column to retrieve.

### Value

A polars [Series](#)



## Examples

```
df <- pl$DataFrame(foo = 1:3, bar = 4:6)
df$get_column("foo")

tryCatch(
  df$get_column("baz"),
  error = function(e) print(e)
)
```

---

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

- [as.list\(<polars\\_data\\_frame>\)](#)

## Examples

```
df <- pl$DataFrame(foo = c(1, 2, 3), bar = c(4, 5, 6))
df$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__get_column_index
```

*Find the index of a column by name*

---

### Description

Find the index of a column by name

### Usage

```
dataframe__get_column_index(name)
```

### Arguments

`name` Name of the column to find.

### Value

Numeric value (0-indexed) indicating the index of the column

### Examples

```
df <- pl$DataFrame(foo = 1:3, bar = 4:6, ham = c("a", "b", "c"))
df$get_column_index("ham")

tryCatch(
  df$get_column_index("sandwich"),
  error = function(e) print(e)
)
```

---

```
dataframe__group_by
```

*Group a DataFrame*

---

### Description

Group a DataFrame

### Usage

```
dataframe__group_by(..., .maintain_order = FALSE)
```

### Arguments

`...` [<dynamic-dots>](#) Column(s) to group by. Accepts expression input. Strings 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.

## 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

Time windows are calculated and rows are assigned to windows. Different from a normal group by is that a row can be member of multiple groups. By default, the windows look like:

- [start, start + period)
- [start + every, start + every + period)
- [start + 2every, start + 2every + period)
- ...

## Usage

```
dataframe__group_by_dynamic(
  index_column,
  ...,
  every,
  period = NULL,
  offset = NULL,
  include_boundaries = FALSE,
  closed = c("left", "right", "both", "none"),
  label = c("left", "right", "datapoint"),
  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>group_by</code> is specified, then it must be sorted in ascending order within each group). In case of a dynamic group by on indices, the data type 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>                | These dots are for future extensions and must be empty.  |
| <code>every</code>              | Interval of the window.  |
| <code>period</code>             | Length of the window. If NULL (default), it will equal <code>every</code> .  |
| <code>offset</code>             | Offset of the window, does not take effect if <code>start_by = "datapoint"</code> . Defaults to zero.  |
| <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 interval are closed (inclusive). Default is <code>"left"</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. Can be expressions or objects coercible to expressions.   |

- start\_by**      The strategy to determine the start of the first window by:
- **"window"**: start by taking the earliest timestamp, truncating it with **every**, and then adding **offset**. Note that weekly windows start on Monday.
  - **"datapoint"**: start from the first encountered data point.
  - 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

where **start** is determined by **start\_by**, **offset**, **every**, and the earliest datapoint. See the **start\_by** argument description for details.

The **every**, **period**, and **offset** arguments are created with 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

In case of a **group\_by\_dynamic** on an integer column, the windows are defined by:

- 1i # length 1
- 10i # length 10

## Value

A [GroupByDynamic](#) object

## See Also

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

## Examples

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

```

    n = 0:6
  )
df

# Group by windows of 1 hour.
df$group_by_dynamic("time", every = "1h", closed = "right")$agg(
  vals = pl$col("n")
)

# The window boundaries can also be added to the aggregation result
df$group_by_dynamic(
  "time",
  every = "1h", include_boundaries = TRUE, closed = "right"
)$agg(
  pl$col("n")$mean()
)

# When closed = "left", the window excludes the right end of interval:
# [lower_bound, upper_bound)
df$group_by_dynamic("time", every = "1h", closed = "left")$agg(
  pl$col("n")
)

# When closed = "both" the time values at the window boundaries belong to 2
# groups.
df$group_by_dynamic("time", every = "1h", closed = "both")$agg(
  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$DataFrame(
  idx = 0:5,
  A = c("A", "A", "B", "B", "B", "C")
)$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      *Get the first n rows*

---

## Description

Get the first n rows

## Usage

```
dataframe__head(n = 5)
```

## Arguments

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

## 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\_\_is\_duplicated      *Get a mask of all duplicated rows in this DataFrame.*

---

## Description

Get a mask of all duplicated rows in this DataFrame.

## Usage

```
dataframe__is_duplicated()
```

**Value**

A [polars Series](#)

**Examples**

```
df <- pl$DataFrame(  
  a = c(1, 2, 3, 1),  
  b = c("x", "y", "z", "x")  
)  
df$is_duplicated()  
  
# This mask can be used to visualize the duplicated lines like this:  
df$filter(df$is_duplicated())
```

---

`dataframe__is_empty` *Returns TRUE if the DataFrame contains no rows.*

---

**Description**

Returns TRUE if the DataFrame contains no rows.

**Usage**

```
dataframe__is_empty()
```

**Value**

A logical value

**Examples**

```
df <- pl$DataFrame(  
  a = c(1, 2, 3, 1),  
  b = c("x", "y", "z", "x")  
)  
df$is_empty()  
df$filter(pl$col("a") > 99)$is_empty()
```



---

`dataframe__is_unique` *Get a mask of all unique rows in this DataFrame.*

---

### Description

Get a mask of all unique rows in this DataFrame.

### Usage

```
dataframe__is_unique()
```

### Value

A [polars Series](#)

### Examples

```
df <- pl$DataFrame(  
  a = c(1, 2, 3, 1),  
  b = c("x", "y", "z", "x")  
)  
df$is_unique()  
  
# This mask can be used to visualize the unique lines like this:  
df$filter(df$is_unique())
```

---

`dataframe__join` *Join DataFrames*

---

### 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 = c("inner", "full", "left", "right", "semi", "anti", "cross"),  
  ...,  
  left_on = NULL,  
  right_on = NULL,  
  suffix = "_right",  
  validate = c("m:m", "1:m", "m:1", "1:1"),  
  join_nulls = FALSE,
```

```

maintain_order = c("none", "left", "right", "left_right", "right_left"),
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: <ul style="list-style-type: none"> <li>• "inner": returns rows that have matching values in both tables</li> <li>• "left": returns all rows from the left table, and the matched rows from the right table</li> <li>• "right": returns all rows from the right table, and the matched rows from the left table</li> <li>• "full": returns all rows when there is a match in either left or right table</li> <li>• "cross": returns the Cartesian product of rows from both tables</li> <li>• "semi": returns rows from the left table that have a match in the right table.</li> <li>• "anti": returns rows from the left table that have no match in the right table.</li> </ul> |
| <code>...</code>               | These dots are for future extensions and must be empty.   |
| <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.</p>   |
| <code>join_nulls</code>        | Join on null values. By default null values will never produce matches.   |
| <code>maintain_order</code>    | Which frame row order to preserve, if any. Do not rely on any observed ordering without explicitly setting this parameter, as your code may break in a future release. Not specifying any ordering can improve performance. Supported for inner, left, right and full joins. <ul style="list-style-type: none"> <li>• "none": No specific ordering is desired. The ordering might differ across Polars versions or even between different runs.</li> </ul>  |

- "left": Preserves the order of the left frame.
- "right": Preserves the order of the right frame.
- "left\_right": First preserves the order of the left frame, then the right.
- "right\_left": First preserves the order of the right frame, then the left.

`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).

- NULL: join specific.
- TRUE: Always coalesce join columns.
- FALSE: Never coalesce join columns. Note that joining on any other expressions than `col` will turn off coalescing.

## Value

A polars [DataFrame](#)

## Examples

```
df <- pl$DataFrame(
  foo = 1:3,
  bar = c(6, 7, 8),
  ham = c("a", "b", "c")
)
other_df <- pl$DataFrame(
  apple = c("x", "y", "z"),
  ham = c("a", "b", "d")
)
df$join(other_df, on = "ham")

df$join(other_df, on = "ham", how = "full")

df$join(other_df, on = "ham", how = "left", coalesce = TRUE)

df$join(other_df, on = "ham", how = "semi")

df$join(other_df, on = "ham", how = "anti")
```

---

`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. Both frames must be sorted by the `asof_join` key.

## 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,
  allow_exact_matches = TRUE,
  check_sortedness = TRUE
)
```

## Arguments

|                                |  |
|--------------------------------|--|
| <code>other</code>             | DataFrame to join with.  |
| <code>...</code>               | These dots are for future extensions and must be empty.  |
| <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 LazyFrames.  |
| <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>• <b>"backward"</b> (default): search for the last row in the right table whose on key is less than or equal to the left key.</li> <li>• <b>"forward"</b>: search for the first row in the right table whose on key is greater than or equal to the left key.</li> <li>• <b>"nearest"</b>: 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 (see details).   |
| <code>allow_parallel</code>      | Allow the physical plan to optionally evaluate the computation of both LazyFrames up to the join in parallel.   |
| <code>force_parallel</code>      | Force the physical plan to evaluate the computation of both LazyFrames 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>• <b>TRUE</b>: Always coalesce join columns;</li> <li>• <b>FALSE</b>: Never coalesce join columns. Note that joining on any other expressions than <code>col</code> will turn off coalescing.</li> </ul>  |
| <code>allow_exact_matches</code> | Whether exact matches are valid join predicates. If <b>TRUE</b> (default), allow matching with the same on value (i.e. less-than-or-equal-to / greater-than-or-equal-to). Otherwise, don't match the same on value (i.e., strictly less-than / strictly greater-than).  |
| <code>check_sortedness</code>    | Check the sortedness of the asof keys. If the keys are not sorted, polars will error, or raise a warning if the <code>by</code> argument is provided. This might become a hard error in the future.   |

## Value

A polars [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

```
gdp <- pl$DataFrame(
  date = as.Date(c("2016-1-1", "2017-5-1", "2018-1-1", "2019-1-1", "2020-1-1")),
  gdp = c(4164, 4411, 4566, 4696, 4827)
)

pop <- pl$DataFrame(
  date = as.Date(c("2016-3-1", "2018-8-1", "2019-1-1")),
  population = c(82.19, 82.66, 83.12)
)

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

# Note how the dates don't quite match. If we join them using join_asof and
# strategy = 'backward', then each date from population which doesn't have
# an exact match is matched with the closest earlier date from gdp:
pop$join_asof(gdp, on = "date", strategy = "backward")

# Note how:
# - date 2016-03-01 from population is matched with 2016-01-01 from gdp;
# - date 2018-08-01 from population is matched with 2018-01-01 from gdp.
# You can verify this by passing coalesce = FALSE:
pop$join_asof(
  gdp,
  on = "date", strategy = "backward", coalesce = FALSE
)

# If we instead use strategy = 'forward', then each date from population
# which doesn't have an exact match is matched with the closest later date
# from gdp:
pop$join_asof(gdp, on = "date", strategy = "forward")
```

```

# Note how:
# - date 2016-03-01 from population is matched with 2017-01-01 from gdp;
# - date 2018-08-01 from population is matched with 2019-01-01 from gdp.

# Finally, strategy = 'nearest' gives us a mix of the two results above, as
# each date from population which doesn't have an exact match is matched
# with the closest date from gdp, regardless of whether it's earlier or
# later:
pop$join_asof(gdp, on = "date", strategy = "nearest")

# Note how:
# - date 2016-03-01 from population is matched with 2016-01-01 from gdp;
# - date 2018-08-01 from population is matched with 2019-01-01 from gdp.

# The `by` argument allows joining on another column first, before the asof
# join. In this example we join by country first, then asof join by date, as
# above.
gdp2 <- pl$DataFrame(
  country = rep(c("Germany", "Netherlands"), each = 5),
  date = rep(
    as.Date(c("2016-1-1", "2017-1-1", "2018-1-1", "2019-1-1", "2020-1-1")),
    2
  ),
  gdp = c(4164, 4411, 4566, 4696, 4827, 784, 833, 914, 910, 909)
)$sort("country", "date")
gdp2

pop2 <- pl$DataFrame(
  country = rep(c("Germany", "Netherlands"), each = 3),
  date = rep(as.Date(c("2016-3-1", "2018-8-1", "2019-1-1")), 2),
  population = c(82.19, 82.66, 83.12, 17.11, 17.32, 17.40)
)$sort("country", "date")
pop2

pop2$join_asof(
  gdp2,
  by = "country", on = "date", strategy = "nearest"
)

```

---

dataframe\_\_join\_where

*Perform a join based on one or multiple (in)equality predicates*

---

## Description

### [Experimental]

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**

|                     |  |
|---------------------|--|
| <code>other</code>  | DataFrame to join with.  |
| <code>...</code>    | <dynamic-dots> (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>". |
| <code>suffix</code> | Suffix to append to columns with a duplicate name.   |

**Value**

A polars [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__lazy`

*Convert an existing DataFrame to a LazyFrame*

---

**Description**

Start a new lazy query from a DataFrame.

**Usage**

```
dataframe__lazy()
```



**Value**

A polars [LazyFrame](#)

**Examples**

```
pl$DataFrame(a = 1:2, b = c(NA, "a"))$lazy()
```

---

|                |  |
|----------------|--|
| dataframe__max | <i>Aggregate the columns in the DataFrame to their maximum value</i> |
|----------------|--|

---

**Description**

Aggregate the columns in the DataFrame to their maximum value

**Usage**

```
dataframe__max()
```

**Value**

A polars [DataFrame](#)

**Examples**

```
df <- pl$DataFrame(a = 1:4, b = c(1, 2, 1, 1))
df$max()
```

---

|                           |   |
|---------------------------|---|
| dataframe__max_horizontal | <i>Get the maximum value horizontally across columns.</i> |
|---------------------------|---|

---

**Description**

Get the maximum value horizontally across columns.

**Usage**

```
dataframe__max_horizontal()
```

**Value**

A polars [Series](#)

**Examples**

```
df <- pl$DataFrame(
  foo = c(1, 2, 3),
  bar = c(4.0, 5.0, 6.0),
)
df$max_horizontal()
```

---

|                 |   |
|-----------------|---|
| dataframe__mean | <i>Aggregate the columns in the DataFrame to their mean value</i> |
|-----------------|---|

---

### Description

Aggregate the columns in the DataFrame to their mean value

### Usage

```
dataframe__mean()
```

### Value

A polars [DataFrame](#)

### Examples

```
df <- pl$DataFrame(a = 1:4, b = c(1, 2, 1, 1))
df$mean()
```

---

|                            |   |
|----------------------------|---|
| dataframe__mean_horizontal | <i>Take the mean of all values horizontally across columns.</i> |
|----------------------------|---|

---

### Description

Take the mean of all values horizontally across columns.

### Usage

```
dataframe__mean_horizontal(..., ignore_nulls = TRUE)
```

### Arguments

|              |   |
|--------------|---|
| ...          | These dots are for future extensions and must be empty.   |
| ignore_nulls | Ignore null values (default). If <b>FALSE</b> , any null value in the input will lead to a null output. |

### Value

A polars [Series](#)

### Examples

```
df <- pl$DataFrame(
  foo = c(1, 2, 3),
  bar = c(4.0, 5.0, 6.0),
)
df$mean_horizontal()
```

---

`dataframe__median`     *Aggregate the columns in the DataFrame to their median value*

---

### Description

Aggregate the columns in the DataFrame to their median value

### Usage

```
dataframe__median()
```

### Value

A polars [DataFrame](#)

### Examples

```
df <- pl$DataFrame(a = 1:4, b = c(1, 2, 1, 1))
df$median()
```

---

`dataframe__merge_sorted`     *Take two sorted DataFrames and merge them by the sorted key*

---

### Description

The output of this operation will also be sorted. It is the callers responsibility that the frames are sorted by that key, otherwise the output will not make sense. The schemas of both DataFrames must be equal.

### Usage

```
dataframe__merge_sorted(other, key)
```

### Arguments

`other`             Other DataFrame that must be merged.  
`key`                Key that is sorted.

### Value

A polars [DataFrame](#)

**Examples**

```
df1 <- pl$DataFrame(
  name = c("steve", "elise", "bob"),
  age = c(42, 44, 18)
)$sort("age")

df2 <- pl$DataFrame(
  name = c("anna", "megan", "steve", "thomas"),
  age = c(21, 33, 42, 20)
)$sort("age")

df1$merge_sorted(df2, key = "age")
```

---

|                |  |
|----------------|--|
| dataframe__min | <i>Aggregate the columns in the DataFrame to their minimum value</i> |
|----------------|--|

---

**Description**

Aggregate the columns in the DataFrame to their minimum value

**Usage**

```
dataframe__min()
```

**Value**

A polars [DataFrame](#)

**Examples**

```
df <- pl$DataFrame(a = 1:4, b = c(1, 2, 1, 1))
df$min()
```

---

|                           |   |
|---------------------------|---|
| dataframe__min_horizontal | <i>Get the minimum value horizontally across columns.</i> |
|---------------------------|---|

---

**Description**

Get the minimum value horizontally across columns.

**Usage**

```
dataframe__min_horizontal()
```

**Value**

A polars [Series](#)

## Examples

```
df <- pl$DataFrame(  
  foo = c(1, 2, 3),  
  bar = c(4.0, 5.0, 6.0),  
)  
df$min_horizontal()
```

---

`dataframe__n_chunks` *Get number of chunks used by the ChunkedArrays of this DataFrame*

---

## Description

Get number of chunks used by the ChunkedArrays of this DataFrame

## Usage

```
dataframe__n_chunks(strategy = c("first", "all"))
```

## Arguments

`strategy` Return the number of chunks of the "first" column, or "all" columns in this DataFrame.

## Value

An integer vector.

## Examples

```
df <- pl$DataFrame(  
  a = c(1, 2, 3, 4),  
  b = c(0.5, 4, 10, 13),  
  c = c(TRUE, TRUE, FALSE, TRUE)  
)  
  
df$n_chunks()  
df$n_chunks(strategy = "all")
```

---

dataframe\_\_partition\_by

*Group by the given columns and return the groups as separate dataframes*

---

### Description

Group by the given columns and return the groups as separate dataframes

### Usage

```
dataframe__partition_by(..., maintain_order = TRUE, include_key = TRUE)
```

### Arguments

`...` [<dynamic-dots>](#) Column name(s) to group by.

`maintain_order` Ensure that the order of the groups is consistent with the input data. This is slower than a default partition by operation.

`include_key` Include the columns used to partition the DataFrame in the output.

### Value

A list of polars [DataFrames](#)

### Examples

```
# Pass a single column name to partition by that column.
df <- pl$DataFrame(
  a = c("a", "b", "a", "b", "c"),
  b = c(1, 2, 1, 3, 3),
  c = c(5, 4, 3, 2, 1)
)
df$partition_by("a")

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

---

dataframe\_\_pivot

*Pivot a frame from long to wide format*

---

### Description

Only available in eager mode. See "Examples" section below for how to do a "lazy pivot" if you know the unique column values in advance.

**Usage**

```
dataframe__pivot(
  on,
  ...,
  index = NULL,
  values = NULL,
  aggregate_function = NULL,
  maintain_order = TRUE,
  sort_columns = FALSE,
  separator = "_"
)
```

**Arguments**

|                                 |  |
|---------------------------------|--|
| <code>on</code>                 | The column(s) whose values will be used as the new columns of the output DataFrame.  |
| <code>...</code>                | These dots are for future extensions and must be empty.  |
| <code>index</code>              | The column(s) that remain from the input to the output. The output DataFrame will have one row for each unique combination of the <code>index</code> 's values. If <code>NULL</code> , all remaining columns not specified in <code>on</code> and <code>values</code> will be used. At least one of <code>index</code> and <code>values</code> must be specified.  |
| <code>values</code>             | The existing column(s) of values which will be moved under the new columns from <code>index</code> . If an aggregation is specified, these are the values on which the aggregation will be computed. If <code>NULL</code> , all remaining columns not specified in <code>on</code> and <code>index</code> will be used. At least one of <code>index</code> and <code>values</code> must be specified.  |
| <code>aggregate_function</code> | Choose from: <ul style="list-style-type: none"> <li>• <code>NULL</code>: no aggregation takes place, will raise error if multiple values are in group;</li> <li>• A predefined aggregate function string, one of <code>"min"</code>, <code>"max"</code>, <code>"first"</code>, <code>"last"</code>, <code>"sum"</code>, <code>"mean"</code>, <code>"median"</code>, <code>"len"</code>;</li> <li>• An expression to do the aggregation.</li> </ul> |
| <code>maintain_order</code>     | Ensure the values of <code>index</code> are sorted by discovery order.   |
| <code>sort_columns</code>       | Sort the transposed columns by name. Default is by order of discovery.   |
| <code>separator</code>          | Used as separator/delimiter in generated column names in case of multiple values columns.  |

**Value**

A polars [DataFrame](#)

**Examples**

```
# Suppose we have a dataframe of test scores achieved by some students,
```

```

# where each row represents a distinct test.
df <- pl$DataFrame(
  name = c("Cady", "Cady", "Karen", "Karen"),
  subject = c("maths", "physics", "maths", "physics"),
  test_1 = c(98, 99, 61, 58),
  test_2 = c(100, 100, 60, 60)
)
df

# Using pivot(), we can reshape so we have one row per student, with
# different subjects as columns, and their `test_1` scores as values:
df$pivot("subject", index = "name", values = "test_1")

# If you end up with multiple values per cell, you can specify how to
# aggregate them with `aggregate_function`:
df <- pl$DataFrame(
  ix = c(1, 1, 2, 2, 1, 2),
  col = c("a", "a", "a", "a", "b", "b"),
  foo = c(0, 1, 2, 2, 7, 1),
  bar = c(0, 2, 0, 0, 9, 4)
)
df

df$pivot("col", index = "ix", aggregate_function = "sum")

# You can also pass a custom aggregation function using `pl$element()`:
df <- pl$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$pivot(
  "col2",
  index = "col1",
  values = "col3",
  aggregate_function = pl$element()$tanh()$mean(),
)

# Note that pivot is only available in eager mode. If you know the unique
# column values in advance, you can use `group_by()` on a LazyFrame to get
# the same result as above in lazy mode:
index <- pl$col("col1")
on <- pl$col("col2")
values <- pl$col("col3")
unique_column_values <- c("x", "y")
aggregate_function <- \ (col) col$tanh()$mean()
funs <- lapply(unique_column_values, \ (value) {
  aggregate_function(values$filter(on == value))$alias(value)
})
df$lazy()$group_by(index)$agg(!!!funs)$collect()

```



---

`dataframe__quantile` *Aggregate the columns to a unique quantile value*

---

### Description

Aggregate the columns to a unique quantile value

### Usage

```
dataframe__quantile(  
  quantile,  
  interpolation = c("nearest", "higher", "lower", "midpoint", "linear")  
)  
  
dataframe__quantile(  
  quantile,  
  interpolation = c("nearest", "higher", "lower", "midpoint", "linear")  
)
```

### Arguments

`quantile` Quantile between 0.0 and 1.0.  
`interpolation` Interpolation method.

### Value

A polars [DataFrame](#)

### Examples

```
df <- pl$DataFrame(a = 1:4, b = c(1, 2, 1, 1))  
df$quantile(0.7)  
df <- pl$DataFrame(a = 1:4, b = c(1, 2, 1, 1))  
df$quantile(0.7)
```

---

`dataframe__rechunk` *Rechunk the data in this DataFrame to a contiguous allocation*

---

### Description

This will make sure all subsequent operations have optimal and predictable performance.

### Usage

```
dataframe__rechunk()
```

**Value**

A polars [DataFrame](#)

---

|                   |                            |
|-------------------|----------------------------|
| dataframe__rename | <i>Rename column names</i> |
|-------------------|----------------------------|

---

**Description**

Rename column names

**Usage**

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

**Arguments**

|         |   |
|---------|---|
| ...     | < <a href="#">dynamic-dots</a> > Either a function that takes a character vector as input and returns a character vector as output, or named values where names are old column names and values are the new ones. |
| .strict | Validate that all column names exist in the current schema, and throw an error if any do not. (Note that this parameter is a no-op when passing a function to ...).   |

**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**

A polars [DataFrame](#)

**Examples**

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

---

`dataframe__rolling`    *Create rolling groups based on a date/time or integer column*

---

## Description

Different from `group_by_dynamic()`, the windows are now determined by the individual values and are not of constant intervals. For constant intervals use `group_by_dynamic()`.

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 = c("right", "left", "both", "none"),
  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>group_by</code> is specified, then it must be sorted in ascending order within each group). In case of a dynamic group by on indices, the data type 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>          | These dots are for future extensions and must be empty.  |
| <code>period</code>       | Length of the window - must be non-negative.   |
| <code>offset</code>       | Offset of the window. Default is <code>-period</code> .  |
| <code>closed</code>       | Define which sides of the interval are closed (inclusive). Default is <code>"left"</code> .  |
| <code>group_by</code>     | Also group by this column/these columns. Can be expressions or objects coercible to expressions.   |

**Value**

[RollingGroupBy](#) (a DataFrame with special rolling groupby methods like `$agg()`).

**See Also**

- `<DataFrame>$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$DataFrame(dt = dates, a = c(3, 7, 5, 9, 2, 1))$with_columns(
  pl$col("dt")$str$strptime(pl$Datetime())
)

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

```

---

|                   |                                   |
|-------------------|-----------------------------------|
| dataframe__sample | <i>Sample from this DataFrame</i> |
|-------------------|-----------------------------------|

---

**Description**

Sample from this DataFrame

**Usage**

```

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

```

**Arguments**

|                               |  |
|-------------------------------|--|
| <code>n</code>                | Number of items to return. Cannot be used with <code>fraction</code> . Defaults to 1 if <code>fraction</code> is <code>NULL</code> . |
| <code>...</code>              | These dots are for future extensions and must be empty.  |
| <code>fraction</code>         | Fraction of items to return. Cannot be used with <code>n</code> .  |
| <code>with_replacement</code> | Allow values to be sampled more than once.   |
| <code>shuffle</code>          | Shuffle the order of sampled data points.  |
| <code>seed</code>             | Seed for the random number generator. If <code>NULL</code> (default), a random seed is generated for each sample operation.          |

**Value**

A polars [DataFrame](#)

**Examples**

```
df <- pl$DataFrame(
  foo = 1:3,
  bar = 6:8,
  ham = c("a", "b", "c")
)
df$sample(n = 2, seed = 0)
```

---

`dataframe__select`      *Select and modify columns of a DataFrame*

---

**Description**

Select and perform operations on a subset of columns only. This discards unmentioned columns (like `.` in `data.table` and contrarily to `dplyr::mutate()`).

One cannot use new variables in subsequent expressions in the same `$select()` call. For instance, if you create a variable `x`, you will only be able to use it in another `$select()` or `$with_columns()` call.

**Usage**

```
dataframe__select(...)
```

**Arguments**

`...`      `<dynamic-dots>` Name-value pairs of objects to be converted to polars [expressions](#) by the `as_polars_expr()` function. Characters are parsed as column names, other non-expression inputs are parsed as [literals](#). Each name will be used as the expression name.

**Value**

A polars [DataFrame](#)

**Examples**

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

---

dataframe\_\_set\_sorted

*Indicate that one or multiple columns are sorted*

---

**Description**

This can speed up future operations, but it can lead to incorrect results if the data is **not** sorted! Use with care!

**Usage**

```
dataframe__set_sorted(column, ..., descending = FALSE)
```

**Arguments**

|            |   |
|------------|---|
| column     | Column that is sorted.                                  |
| ...        | These dots are for future extensions and must be empty. |
| descending | Whether the columns are sorted in descending order.     |

**Value**

A polars [DataFrame](#)

**Examples**

```
# We mark the data as sorted by "age" but this is not the case!
# It is up to the user to ensure that the column is actually sorted.
df1 <- pl$DataFrame(
  name = c("steve", "elise", "bob"),
  age = c(42, 44, 18)
)$set_sorted("age")

df1$flags
```

---

`dataframe__shift`      *Shift values by the given number of indices*

---

### Description

Shift values by the given number of indices

### Usage

```
dataframe__shift(n = 1, ..., fill_value = NULL)
```

### Arguments

|                         |  |
|-------------------------|--|
| <code>n</code>          | Number of indices to shift forward. If a negative value is passed, values are shifted in the opposite direction instead. |
| <code>...</code>        | These dots are for future extensions and must be empty.  |
| <code>fill_value</code> | Fill the resulting null values with this value. Accepts expression input. Non-expression inputs are parsed as literals.  |

### Value

A polars [DataFrame](#)

### Examples

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

# By default, values are shifted forward by one index.
df$shift()

# Pass a negative value to shift in the opposite direction instead.
df$shift(-2)

# Specify fill_value to fill the resulting null values.
df$shift(-2, fill_value = 100)
```

---

`dataframe__slice`      *Get a slice of the DataFrame.*

---

### Description

Get a slice of the DataFrame.

### Usage

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

**Arguments**

|                     |  |
|---------------------|--|
| <code>offset</code> | Start index, can be a negative value. This is 0-indexed, so <code>offset = 1</code> skips the first row. |
| <code>length</code> | Length of the slice. If NULL (default), all rows starting at the offset will be selected.                |

**Value**

A polars [DataFrame](#)

**Examples**

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

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

---

|                              |  |
|------------------------------|--|
| <code>dataframe__sort</code> | <i>Sort a DataFrame by the given columns</i> |
|------------------------------|--|

---

**Description**

Sort a DataFrame by the given columns

**Usage**

```
dataframe__sort(
  ...,
  descending = FALSE,
  nulls_last = FALSE,
  multithreaded = TRUE,
  maintain_order = FALSE
)
```

**Arguments**

|                             |  |
|-----------------------------|--|
| <code>...</code>            | < <a href="#">dynamic-dots</a> > Column(s) to sort by. Can be character values indicating column names or Expr(s).   |
| <code>descending</code>     | Sort in descending order. When sorting by multiple columns, this can be specified per column by passing a logical vector.  |
| <code>nulls_last</code>     | Place null values last. When sorting by multiple columns, this can be specified per column by passing a logical vector.  |
| <code>multithreaded</code>  | Sort using multiple threads.   |
| <code>maintain_order</code> | Whether the order should be maintained if elements are equal. If <code>TRUE</code> , streaming is not possible and performance might be worse since this requires a stable search. |



**Value**

A polars [DataFrame](#)

**Examples**

```
df <- pl$DataFrame(
  a = c(1, 2, NA, 4),
  b = c(6, 5, 4, 3),
  c = c("a", "c", "b", "a")
)

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

# Sorting by expressions is also supported
df$sort(pl$col("a") + pl$col("b") * 2, nulls_last = TRUE)

# Sort by multiple columns by passing a vector of columns
df$sort(c("c", "a"), descending = TRUE)

# Or use positional arguments to sort by multiple columns in the same way
df$sort("c", "a", descending = c(FALSE, TRUE))
```

---

|                |   |
|----------------|---|
| dataframe__std | <i>Aggregate the columns of this DataFrame to their standard deviation values</i> |
|----------------|---|

---

**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 polars [DataFrame](#)

**Examples**

```
df <- pl$DataFrame(a = 1:4, b = c(1, 2, 1, 1))
df$std()
df$std(ddof = 0)
```

---

|                |  |
|----------------|--|
| dataframe__sum | <i>Aggregate the columns of this DataFrame to their sum values</i> |
|----------------|--|

---

**Description**

Aggregate the columns of this DataFrame to their sum values

**Usage**

```
dataframe__sum()
```

**Value**

A polars [DataFrame](#)

**Examples**

```
df <- pl$DataFrame(a = 1:4, b = c(1, 2, 1, 1))
df$sum()
```

---

|                           |  |
|---------------------------|--|
| dataframe__sum_horizontal | <i>Sum all values horizontally across columns.</i> |
|---------------------------|--|

---

**Description**

Sum all values horizontally across columns.

**Usage**

```
dataframe__sum_horizontal(..., ignore_nulls = TRUE)
```

**Arguments**

`...` These dots are for future extensions and must be empty.

`ignore_nulls` Ignore null values (default). If `FALSE`, any null value in the input will lead to a null output.

**Value**

A polars [Series](#)

**Examples**

```
df <- pl$DataFrame(
  foo = c(1, 2, 3),
  bar = c(4.0, 5.0, 6.0),
)
df$sum_horizontal()
```

---

dataframe\_\_tail      *Get the last n rows.*

---

### Description

Get the last `n` rows.

### Usage

```
dataframe__tail(n = 5)
```

### 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\_\_top\_k      *Return the k largest rows*

---

### Description

Non-null elements are always preferred over null elements, regardless of the value of `reverse`. The output is not guaranteed to be in any particular order, call `sort()` after this function if you wish the output to be sorted.

### Usage

```
dataframe__top_k(k, ..., by, reverse = FALSE)
```

**Arguments**

|                      |   |
|----------------------|---|
| <code>k</code>       | Number of rows to return.   |
| <code>...</code>     | These dots are for future extensions and must be empty.   |
| <code>by</code>      | Column(s) used to determine the bottom rows. Accepts expression input. Strings are parsed as column names.  |
| <code>reverse</code> | Consider the <code>k</code> smallest elements of the <code>by</code> column(s) (instead of the <code>k</code> largest). This can be specified per column by passing a sequence of booleans. |

**Value**

A polars [DataFrame](#)

**Examples**

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

# Get the rows which contain the 4 largest values in column b.
df$top_k(4, by = "b")

# Get the rows which contain the 4 largest values when sorting on column a
# and b
df$top_k(4, by = c("a", "b"))
```

---

dataframe\_\_to\_dummies

*Convert categorical variables into dummy/indicator variables*

---

**Description**

Convert categorical variables into dummy/indicator variables

**Usage**

```
dataframe__to_dummies(..., separator = "_", drop_first = FALSE)
```

**Arguments**

|                         |   |
|-------------------------|---|
| <code>...</code>        | <dynamic-dots> Column name(s) that should be converted to dummy variables. If empty (default), convert all columns. |
| <code>separator</code>  | Separator/delimiter used when generating column names.  |
| <code>drop_first</code> | Remove the first category from the variables being encoded.   |

**Value**

A polars [DataFrame](#)

**Examples**

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

---

`dataframe__to_series` *Select column as Series at index location*

---

**Description**

Select column as Series at index location

**Usage**

```
dataframe__to_series(index = 0)
```

**Arguments**

|                    |  |
|--------------------|--|
| <code>index</code> | Index of the column to return as Series. Defaults to 0, which is the first column. |
|--------------------|--|

**Value**

Series or NULL

**Examples**

```
df <- as_polars_df(iris[1:10, ])  
  
# default is to extract the first column  
df$to_series()  
  
# Polars is 0-indexed, so we use index = 1 to extract the *2nd* column  
df$to_series(index = 1)
```

`dataframe__to_struct` *Convert a DataFrame to a Series of type Struct*

---

### Description

Convert a DataFrame to a Series of type Struct

### Usage

```
dataframe__to_struct(name = "")
```

### Arguments

`name` A character. Name for the struct [Series](#).

### Value

A [Series](#) of the struct type

### See Also

- [as\\_polars\\_series\(\)](#)

### Examples

```
df <- pl$DataFrame(  
  a = 1:5,  
  b = c("one", "two", "three", "four", "five"),  
)  
df$to_struct("nums")
```

---

`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**

|                             |  |
|-----------------------------|--|
| <code>...</code>            | These dots are for future extensions and must be empty.  |
| <code>include_header</code> | If set, the column names will be added as first column.  |
| <code>header_name</code>    | If <code>include_header</code> is set, this determines the name of the column that will be inserted.   |
| <code>column_names</code>   | Optional string naming an existing column, or a function that takes an integer vector representing the position of value (non-header) columns and returns a character vector of same length. Column position is 0-indexed. |

**Details**

This is a very expensive operation. Perhaps you can do it differently.

**Value**

A polars [DataFrame](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, 2, 3), b = c(4, 5, 6))
df$transpose(include_header = TRUE)

# Replace the auto-generated column names with a list
df$transpose(include_header = FALSE, column_names = c("x", "y", "z"))

# Include the header as a separate column
df$transpose(
  include_header = TRUE, header_name = "foo", column_names = c("x", "y", "z")
)

# Use a function to produce the new column names
name_generator <- function(x) {
  paste0("my_column_", x)
}
df$transpose(include_header = FALSE, column_names = name_generator)

# Use an existing column as the new column names
df <- pl$DataFrame(id = c("i", "j", "k"), a = c(1, 2, 3), b = c(4, 5, 6))
df$transpose(column_names = "id")
df$transpose(include_header = TRUE, header_name = "new_id", column_names = "id")
```

---

`dataframe__unique`      *Drop duplicate rows*

---

**Description**

Drop duplicate rows

## Usage

```
dataframe__unique(
  subset = NULL,
  ...,
  keep = c("any", "none", "first", "last"),
  maintain_order = FALSE
)
```

## Arguments

|                             |   |
|-----------------------------|---|
| <code>subset</code>         | Column name(s) or selector(s), to consider when identifying duplicate rows. If <code>NULL</code> (default), use all columns.  |
| <code>...</code>            | These dots are for future extensions and must be empty.   |
| <code>keep</code>           | Which of the duplicate rows to keep. Must be one of: <ul style="list-style-type: none"> <li>• <code>"any"</code>: does not give any guarantee of which row is kept. This allows more optimizations.</li> <li>• <code>"none"</code>: don't keep duplicate rows.</li> <li>• <code>"first"</code>: keep first unique row.</li> <li>• <code>"last"</code>: keep last unique row.</li> </ul> |
| <code>maintain_order</code> | Keep the same order as the original data. This is more expensive to compute. Setting this to <code>TRUE</code> blocks the possibility to run on the streaming engine.   |

## Value

A polars [DataFrame](#)

## Examples

```
df <- pl$DataFrame(
  foo = c(1, 2, 3, 1),
  bar = c("a", "a", "a", "a"),
  ham = c("b", "b", "b", "b"),
)
df$unique(maintain_order = TRUE)

df$unique(subset = c("bar", "ham"), maintain_order = TRUE)

df$unique(keep = "last", maintain_order = TRUE)
```



---

|                   |  |
|-------------------|--|
| dataframe__unnest | <i>Decompose struct columns into separate columns for each of their fields</i> |
|-------------------|--|

---

### Description

The new columns will be inserted at the location of the struct column.

### Usage

```
dataframe__unnest(...)
```

### Arguments

... <dynamic-dots> Name of the struct column(s) that should be unnested.

### Value

A polars [DataFrame](#)

### 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  
  
df$unnest("a_and_c")  
df$unnest(pl$col("a_and_c"))
```

---

|                    |   |
|--------------------|---|
| dataframe__unpivot | <i>Unpivot a frame from wide to long format</i> |
|--------------------|---|

---

### Description

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

**Usage**

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

**Arguments**

|                            |   |
|----------------------------|---|
| <code>on</code>            | Values to use as identifier variables. If <code>value_vars</code> is empty all columns that are not in <code>id_vars</code> will be used. |
| <code>...</code>           | These dots are for future extensions and must be empty.   |
| <code>index</code>         | Columns to use as identifier variables.   |
| <code>variable_name</code> | Name to give to the new column containing the names of the melted columns. Defaults to "variable".  |
| <code>value_name</code>    | Name to give to the new column containing the values of the melted columns. Defaults to "value".  |

**Value**

A polars [LazyFrame](#)

**Examples**

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

---

|                             |   |
|-----------------------------|---|
| <code>dataframe__var</code> | <i>Aggregate the columns in the DataFrame to their variance value</i> |
|-----------------------------|---|

---

**Description**

Aggregate the columns in the DataFrame to their variance value

**Usage**

```
dataframe__var(ddof = 1)
```

**Arguments**

|                   |  |
|-------------------|--|
| <code>ddof</code> | "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**

A polars [DataFrame](#)

**Examples**

```
df <- pl$DataFrame(a = 1:4, b = c(1, 2, 1, 1))
df$var()
df$var(ddof = 0)
```

---

dataframe\_\_with\_columns

*Modify/append column(s) of a DataFrame*

---

**Description**

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

However, unlike `dplyr::mutate()`, one cannot use new variables in subsequent expressions in the same `$with_columns()` call. For instance, if you create a variable `x`, you will only be able to use it in another `$with_columns()` or `$select()` call.

**Usage**

```
dataframe__with_columns(...)
```

**Arguments**

... [<dynamic-dots>](#) Name-value pairs of objects to be converted to polars [expressions](#) by the `as_polars_expr()` function. Characters are parsed as column names, other non-expression inputs are parsed as [literals](#). Each name will be used as the expression name.

**Value**

A polars [DataFrame](#)

**Examples**

```
# Pass an expression to add it as a new column.
df <- pl$DataFrame(
  a = 1:4,
  b = c(0.5, 4, 10, 13),
  c = c(TRUE, TRUE, FALSE, TRUE),
)
df$with_columns((pl$col("a")^2)$alias("a^2"))

# Added columns will replace existing columns with the same name.
df$with_columns(a = pl$col("a")$cast(pl$Float64))
```

```

# Multiple columns can be added
df$with_columns(
  (pl$col("a")^2)$alias("a^2"),
  (pl$col("b") / 2)$alias("b/2"),
  (pl$col("c")$not())$alias("not c"),
)

# Name expression instead of `$alias()`
df$with_columns(
  `a^2` = pl$col("a")^2,
  `b/2` = pl$col("b") / 2,
  `not c` = pl$col("c")$not(),
)

# Expressions with multiple outputs can automatically be instantiated
# as Structs by enabling the experimental setting `POLARS_AUTO_STRUCTIFY`:
if (requireNamespace("withr", quietly = TRUE)) {
  withr::with_envvar(c(POLARS_AUTO_STRUCTIFY = "1"), {
    df$drop("c")$with_columns(
      diffs = pl$col("a", "b")$diff()$name$suffix("_diff"),
    )
  })
}

```

---

dataframe\_\_with\_columns\_seq

*Modify/append column(s) of a DataFrame*

---

## Description

This will run all expression sequentially instead of in parallel. Use this only when the work per expression is cheap.

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

However, unlike `dplyr::mutate()`, one cannot use new variables in subsequent expressions in the same `$with_columns_seq()` call. For instance, if you create a variable `x`, you will only be able to use it in another `$with_columns_seq()` or `$select()` call.

## Usage

```
dataframe__with_columns_seq(...)
```

## Arguments

... [<dynamic-dots>](#) Name-value pairs of objects to be converted to polars expressions by the `as_polars_expr()` function. Characters are parsed as column names, other non-expression inputs are parsed as [literals](#). Each name will be used as the expression name.

**Value**

A polars [DataFrame](#)

**Examples**

```
# Pass an expression to add it as a new column.
df <- pl$DataFrame(
  a = 1:4,
  b = c(0.5, 4, 10, 13),
  c = c(TRUE, TRUE, FALSE, TRUE),
)
df$with_columns_seq((pl$col("a")^2)$alias("a^2"))

# Added columns will replace existing columns with the same name.
df$with_columns_seq(a = pl$col("a")$cast(pl$Float64))

# Multiple columns can be added
df$with_columns_seq(
  (pl$col("a")^2)$alias("a^2"),
  (pl$col("b") / 2)$alias("b/2"),
  (pl$col("c")$not())$alias("not c"),
)

# Name expression instead of `$alias()`
df$with_columns_seq(
  `a^2` = pl$col("a")^2,
  `b/2` = pl$col("b") / 2,
  `not c` = pl$col("c")$not(),
)

# Expressions with multiple outputs can automatically be instantiated
# as Structs by enabling the experimental setting `POLARS_AUTO_STRUCTIFY`:
if (requireNamespace("withr", quietly = TRUE)) {
  withr::with_envvar(c(POLARS_AUTO_STRUCTIFY = "1"), {
    df$drop("c")$with_columns_seq(
      diffs = pl$col("a", "b")$diff()$name$suffix("_diff"),
    )
  })
}
```

---

dataframe\_\_with\_row\_index

*Add a row index as the first column in the DataFrame*

---

**Description**

Add a row index as the first column in the DataFrame

**Usage**

```
dataframe__with_row_index(name = "index", offset = 0)
```

**Arguments**

`name`            Name of the index column.  
`offset`          Start the index at this offset. Cannot be negative.

**Value**

A polars [DataFrame](#)

**Examples**

```
df <- pl$DataFrame(x = c(1, 3, 5), y = c(2, 4, 6))
df$with_row_index()

df$with_row_index("id", offset = 1000)

# An index column can also be created using the expressions int_range()
# and len()$
df$with_columns(
  index = pl$int_range(pl$len(), dtype = pl$UInt32)
)
```

---

`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_scientific = NULL,
  float_precision = NULL,
  null_value = "",
  quote_style = c("necessary", "always", "never", "non_numeric"),
  storage_options = NULL,
  retries = 2
)
```

**Arguments**

|                               |   |
|-------------------------------|---|
| <code>file</code>             | File path to which the result will be written.  |
| <code>...</code>              | Dots which should be empty.   |
| <code>include_bom</code>      | Logical, whether to include UTF-8 BOM in the CSV output.  |
| <code>include_header</code>   | Logical, whether to include header in the CSV output.   |
| <code>separator</code>        | Separate CSV fields with this symbol.   |
| <code>line_terminator</code>  | String used to end each row.  |
| <code>quote_char</code>       | 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 <code>chrono</code> 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 <code>chrono</code> Rust crate.   |
| <code>time_format</code>      | A format string, with the specifiers defined by the <code>chrono</code> Rust crate.   |
| <code>float_scientific</code> | Whether to use scientific form always ( <code>TRUE</code> ), never ( <code>FALSE</code> ), or automatically ( <code>NULL</code> ) for Float32 and Float64 datatypes.  |
| <code>float_precision</code>  | Number of decimal places to write, applied to both Float32 and Float64 datatypes.   |
| <code>null_value</code>       | A string representing null values (defaulting to the empty string).   |
| <code>quote_style</code>      | Determines the quoting strategy used. Must be one of: <ul style="list-style-type: none"> <li>• <code>"necessary"</code> (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>• <code>"always"</code>: This puts quotes around every field. Always.</li> <li>• <code>"never"</code>: This never puts quotes around fields, even if that results in invalid CSV data (e.g.: by not quoting strings containing the separator).</li> <li>• <code>"non_numeric"</code>: 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> </ul> |
| <code>storage_options</code>  | Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here: <ul style="list-style-type: none"> <li>• <code>aws</code></li> </ul>  |

- `gcp`
- `azure`
- Hugging Face (`hf://`): Accepts an API key under the token parameter `c(token = YOUR_TOKEN)` or by setting the `HF_TOKEN` environment variable.

If `storage_options` is not provided, Polars will try to infer the information from environment variables.

`retries` Number of retries if accessing a cloud instance fails.

### Value

The input `DataFrame` is returned.

### Examples

```
tmpf <- tempfile()
as_polars_df(mtcars)$write_csv(tmpf)
pl$read_csv(tmpf)

as_polars_df(mtcars)$write_csv(tmpf, separator = "|")
pl$read_csv(tmpf, separator = "|")
```

---

`dataframe__write_json`

*Serialize to JSON representation*

---

### Description

Serialize to JSON representation

### Usage

```
dataframe__write_json(file)
```

### Arguments

`file` File path to which the result will be written.

### Value

The input `DataFrame` is returned.



## Examples

```
dat <- as_polars_df(head(mtcars))
destination <- tempfile()

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

---

dataframe\_\_write\_ndjson

*Serialize to newline delimited JSON representation*

---

## Description

Serialize to newline delimited JSON representation

## Usage

```
dataframe__write_ndjson(file)
```

## Value

The input DataFrame is returned.

## Examples

```
dat <- as_polars_df(head(mtcars))
destination <- tempfile()

dat$select(pl$col("drat", "mpg"))$write_ndjson(destination)
jsonlite::stream_in(file(destination))
```

---

dataframe\_\_write\_parquet

*Write to Parquet file*

---

## Description

Write to Parquet file

**Usage**

```
dataframe__write__parquet(
  file,
  ...,
  compression = c("lz4", "uncompressed", "snappy", "gzip", "lzo", "brotli", "zstd"),
  compression_level = NULL,
  statistics = TRUE,
  row_group_size = NULL,
  data_page_size = NULL,
  partition_by = NULL,
  partition_chunk_size_bytes = 4294967296,
  storage_options = NULL,
  retries = 2
)
```

**Arguments**

|                          |   |
|--------------------------|---|
| <b>file</b>              | File path to which the result should be written. This should be a path to a directory if writing a partitioned dataset.   |
| <b>...</b>               | Dots which should be empty.   |
| <b>compression</b>       | The compression method. Must be 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>  |
| <b>compression_level</b> | 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>   |
| <b>statistics</b>        | Whether statistics should be written to the Parquet headers. Possible values: <ul style="list-style-type: none"> <li>• TRUE: enable default set of statistics (default). Some statistics may be disabled.</li> <li>• FALSE: disable all statistics</li> <li>• "full": calculate and write all available statistics</li> <li>• A list created via <code>parquet_statistics()</code> to specify which statistics to include.</li> </ul> |

|   |   |
|---|---|
| <code>row_group_size</code>             | 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.   |
| <code>data_page_size</code>             | Size of the data page in bytes. If NULL (default), it is set to $1024^2$ bytes.   |
| <code>partition_by</code>               | A character vector indicating column(s) to partition by. A partitioned dataset will be written if this is specified.  |
| <code>partition_chunk_size_bytes</code> | 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).   |
| <code>storage_options</code>            | Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here: <ul style="list-style-type: none"> <li>• <code>aws</code></li> <li>• <code>gcp</code></li> <li>• <code>azure</code></li> <li>• Hugging Face (<code>hf://</code>): Accepts an API key under the token parameter <code>c(token = YOUR_TOKEN)</code> or by setting the <code>HF_TOKEN</code> environment variable.</li> </ul> <p>If <code>storage_options</code> is not provided, Polars will try to infer the information from environment variables.</p> |
| <code>retries</code>                    | Number of retries if accessing a cloud instance fails.  |

## Value

The input DataFrame is returned.

## Examples

```
dat = as_polars_df(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)
```

---

|              |   |
|--------------|---|
| expr_arr_all | <i>Evaluate whether all boolean values are true for every sub-array</i> |
|--------------|---|

---

**Description**

Evaluate whether all boolean values are true for every sub-array

**Usage**

```
expr_arr_all()
```

**Value**

A polars [expression](#)

**Examples**

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

---

|              |   |
|--------------|---|
| expr_arr_any | <i>Evaluate whether any boolean value is true for every sub-array</i> |
|--------------|---|

---

**Description**

Evaluate whether any boolean value is true for every sub-array

**Usage**

```
expr_arr_any()
```

**Value**

A polars [expression](#)

**Examples**

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

---

|                  |   |
|------------------|---|
| expr_arr_arg_max | <i>Retrieve the index of the maximum value in every sub-array</i> |
|------------------|---|

---

### Description

Retrieve the index of the maximum value in every sub-array

### Usage

```
expr_arr_arg_max()
```

### Value

A polars [expression](#)

### Examples

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

---

|                  |   |
|------------------|---|
| expr_arr_arg_min | <i>Retrieve the index of the minimum value in every sub-array</i> |
|------------------|---|

---

### Description

Retrieve the index of the minimum value in every sub-array

### Usage

```
expr_arr_arg_min()
```

### Value

A polars [expression](#)

### Examples

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

---

`expr_arr_contains`      *Check if sub-arrays contain the given item*

---

### Description

Check if sub-arrays contain the given item

### Usage

```
expr_arr_contains(item)
```

### Arguments

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

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(
  values = list(0:2, 4:6, c(NA, NA, NA)),
  item = c(0L, 4L, 2L),
)$cast(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)
)
```

---

`expr_arr_count_matches`

*Count how often a value occurs in every sub-array*

---

### Description

Count how often a value occurs in every sub-array

### Usage

```
expr_arr_count_matches(element)
```

### Arguments

`element`                An Expr or something coercible to an Expr that produces a single value.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(  
  values = list(c(1, 2), c(1, 1), c(2, 2))  
)$cast(pl$Array(pl$Int64, 2))  
df$with_columns(number_of_twos = pl$col("values")$arr$count_matches(2))
```

---

expr\_arr\_explode      *Explode array in separate rows*

---

**Description**

Returns a column with a separate row for every array element.

**Usage**

```
expr_arr_explode()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(  
  a = list(c(1, 2, 3), c(4, 5, 6))  
)$cast(pl$Array(pl$Int64, 3))  
df$select(pl$col("a")$arr$explode())
```

---

expr\_arr\_first      *Get the first value of the sub-arrays*

---

**Description**

Get the first value of the sub-arrays

**Usage**

```
expr_arr_first()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = list(c(1, 2, 3), c(4, 5, 6))
)$cast(pl$Array(pl$Int64, 3))
df$with_columns(first = pl$col("a")$arr$first())
```

---

**expr\_arr\_get***Get the value by index in every sub-array*

---

**Description**

This allows to extract one value per array only. Values are 0-indexed (so index 0 would return the first item of every sub-array) and negative values start from the end (so index -1 returns the last item).

**Usage**

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

**Arguments**

|                          |   |
|--------------------------|---|
| <code>index</code>       | An Expr or something coercible to an Expr, that must return a single index.   |
| <code>...</code>         | These dots are for future extensions and must be empty.                       |
| <code>null_on_oob</code> | If TRUE, return null if an index is out of bounds. Otherwise, raise an error. |

**Value**

[Expr](#)

**Examples**

```
df <- pl$DataFrame(
  values = list(c(1, 2), c(3, 4), c(NA, 6)),
  idx = c(1, NA, 3)
)$cast(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)
)
```



---

|               |   |
|---------------|---|
| expr_arr_join | <i>Join elements in every sub-array</i> |
|---------------|---|

---

### Description

Join all string items in a sub-array and place a separator between them. This only works if the inner type of the array is `String`.

### Usage

```
expr_arr_join(separator, ..., ignore_nulls = FALSE)
```

### Arguments

|              |   |
|--------------|---|
| separator    | String to separate the items with. Can be an Expr. Strings are not parsed as columns.                             |
| ...          | These dots are for future extensions and must be empty.   |
| ignore_nulls | If FALSE (default), null values will be propagated, i.e. if the row contains any null values, the output is null. |

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(
  values = list(c("a", "b", "c"), c("x", "y", "z"), c("e", NA, NA)),
  separator = c("-", "+", "/"),
)$cast(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)
)
```

---

|               |   |
|---------------|---|
| expr_arr_last | <i>Get the last value of the sub-arrays</i> |
|---------------|---|

---

### Description

Get the last value of the sub-arrays

### Usage

```
expr_arr_last()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = list(c(1, 2, 3), c(4, 5, 6))
)$cast(pl$Array(pl$Int64, 3))
df$with_columns(last = pl$col("a")$arr$last())
```

---

|              |  |
|--------------|--|
| expr_arr_max | <i>Compute the max value of the sub-arrays</i> |
|--------------|--|

---

**Description**

Compute the max value of the sub-arrays

**Usage**

```
expr_arr_max()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  values = list(c(1, 2), c(3, 4), c(NA, NA))
)$cast(pl$Array(pl$Float64, 2))
df$with_columns(max = pl$col("values")$arr$max())
```

---

|                 |   |
|-----------------|---|
| expr_arr_median | <i>Compute the median value of the sub-arrays</i> |
|-----------------|---|

---

**Description**

Compute the median value of the sub-arrays

**Usage**

```
expr_arr_median()
```

**Value**

A polars [expression](#)

**Examples**

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

---

|              |  |
|--------------|--|
| expr_arr_min | <i>Compute the min value of the sub-arrays</i> |
|--------------|--|

---

**Description**

Compute the min value of the sub-arrays

**Usage**

```
expr_arr_min()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  values = list(c(1, 2), c(3, 4), c(NA, NA))
)$cast(pl$Array(pl$Float64, 2))
df$with_columns(min = pl$col("values")$arr$min())
```

---

|                   |   |
|-------------------|---|
| expr_arr_n_unique | <i>Count the number of unique values in every sub-array</i> |
|-------------------|---|

---

**Description**

Count the number of unique values in every sub-array

**Usage**

```
expr_arr_n_unique()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = list(c(1, 1, 2), c(2, 3, 4))
)$cast(pl$Array(pl$Int64, 3))
df$with_columns(n_unique = pl$col("a")$arr$n_unique())
```

---

`expr_arr_reverse`      *Reverse values in every sub-array*

---

### Description

Reverse values in every sub-array

### Usage

```
expr_arr_reverse()
```

### Value

A polars [expression](#)

### Examples

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

---

`expr_arr_shift`      *Shift values in every sub-array by the given number of indices*

---

### Description

Shift values in every sub-array by the given number of indices

### Usage

```
expr_arr_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

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  values = list(1:3, c(2L, NA, 5L)),
  idx = 1:2,
)$cast(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)
)
```

---

|               |                                       |
|---------------|---------------------------------------|
| expr_arr_sort | <i>Sort values in every sub-array</i> |
|---------------|---------------------------------------|

---

**Description**

Sort values in every sub-array

**Usage**

```
expr_arr_sort(..., descending = FALSE, nulls_last = FALSE)
```

**Arguments**

|            |   |
|------------|---|
| ...        | These dots are for future extensions and must be empty. |
| descending | Sort in descending order.                               |
| nulls_last | Place null values last.                                 |

**Examples**

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

---

|              |   |
|--------------|---|
| expr_arr_std | <i>Compute the standard deviation of the sub-arrays</i> |
|--------------|---|

---

**Description**

Compute the standard deviation of the sub-arrays

**Usage**

```
expr_arr_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 polars [expression](#)

**Examples**

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

---

`expr_arr_sum`

*Compute the sum of the sub-arrays*

---

**Description**

Compute the sum of the sub-arrays

**Usage**

```
expr_arr_sum()
```

**Value**

A polars [expression](#)

**Examples**

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

---

|                  |   |
|------------------|---|
| expr_arr_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

```
expr_arr_to_list()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(  
  a = list(c(1, 2), c(3, 4))  
)$cast(pl$Array(pl$Int8, 2))  
  
df$with_columns(  
  list = pl$col("a")$arr_to_list()  
)
```

---

|                 |   |
|-----------------|---|
| expr_arr_unique | <i>Get the unique values in every sub-array</i> |
|-----------------|---|

---

### Description

Get the unique values in every sub-array

### Usage

```
expr_arr_unique(..., maintain_order = FALSE)
```

### Arguments

... These dots are for future extensions and must be empty.  
maintain\_order Maintain order of data. This requires more work.

### Value

A polars [expression](#)

**Examples**

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

---

expr\_arr\_var                    *Compute the variance of the sub-arrays*

---

**Description**

Compute the variance of the sub-arrays

**Usage**

```
expr_arr_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 polars [expression](#)

**Examples**

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

---

expr\_bin\_contains            *Check if binaries contain a binary substring*

---

**Description**

Check if binaries contain a binary substring

**Usage**

```
expr_bin_contains(literal)
```

**Arguments**

literal                    The binary substring to look for.



**Value**

A polars [expression](#)

**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"))
)
```

---

|                              |  |
|------------------------------|--|
| <code>expr_bin_decode</code> | <i>Decode values using the provided encoding</i> |
|------------------------------|--|

---

**Description**

Decode values using the provided encoding

**Usage**

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

**Arguments**

|                       |   |
|-----------------------|---|
| <code>encoding</code> | A character, "hex" or "base64". The encoding to use.  |
| <code>...</code>      | These dots are for future extensions and must be empty.   |
| <code>strict</code>   | Raise an error if the underlying value cannot be decoded, otherwise mask out with a null value. |

**Value**

A polars [expression](#)

**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))

```

---

|                 |   |
|-----------------|---|
| expr_bin_encode | <i>Encode a value using the provided encoding</i> |
|-----------------|---|

---

## Description

Encode a value using the provided encoding

## Usage

```
expr_bin_encode(encoding)
```

## Arguments

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

## Value

A polars [expression](#)

## 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"))

```

---

`expr_bin_ends_with`     *Check if string values end with a binary substring*

---

### Description

Check if string values end with a binary substring

### Usage

```
expr_bin_ends_with(suffix)
```

### Arguments

`suffix`             Suffix substring.

### Value

A polars [expression](#)

### 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"))
)
```

---

`expr_bin_size`             *Get the size of binary values in the given unit*

---

### Description

Get the size of binary values in the given unit

### Usage

```
expr_bin_size(unit = c("b", "kb", "mb", "gb", "tb"))
```

### Arguments

`unit`                 Scale the returned size to the given unit. Can be "b", "kb", "mb", "gb", "tb".

**Value**

A polars [expression](#)

**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")
)
```

---

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

---

**Description**

Check if values start with a binary substring

**Usage**

```
expr_bin_starts_with(prefix)
```

**Arguments**

prefix            Prefix substring.

**Value**

A polars [expression](#)

**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"))
)
```

---

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

---

**Description**

Get the categories stored in this data type

**Usage**

```
expr_cat_get_categories()
```

**Value**

A polars [expression](#)

**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()  
)
```

---

`expr_cat_set_ordering`*Set Ordering*

---

**Description**

Determine how this categorical series should be sorted.

**Usage**

```
expr_cat_set_ordering(ordering)
```

**Arguments**

`ordering` string either 'physical' or 'lexical'

- "physical": use the physical representation of the categories to determine the order (default).
- "lexical": use the string values to determine the order.

**Value**

A polars [expression](#)

**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")
```

---

expr\_dt\_add\_business\_days

*Offset by n business days.*

---

**Description**

Offset by n business days.

**Usage**

```
expr_dt_add_business_days(
  n,
  ...,
  week_mask = c(TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE),
  holidays = as.Date(integer(0)),
  roll = c("raise", "backward", "forward")
)
```

**Arguments**

|                  |   |
|------------------|---|
| <b>n</b>         | An integer value or a <a href="#">polars expression</a> representing the number of business days to offset by.  |
| <b>...</b>       | These dots are for future extensions and must be empty.   |
| <b>week_mask</b> | Non-NA logical vector of length 7, representing the days of the week to count. The default is Monday to Friday ( <code>c(TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE)</code> ). If you wanted to count only Monday to Thursday, you would pass <code>c(TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE)</code> . |

|          |  |
|----------|--|
| holidays | A <a href="#">Date</a> class vector, representing the holidays to exclude from the count.  |
| roll     | What to do when the start date lands on a non-business day. Options are: <ul style="list-style-type: none"> <li>• "raise": raise an error;</li> <li>• "forward": move to the next business day;</li> <li>• "backward": move to the previous business day.</li> </ul> |

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(start = as.Date(c("2020-1-1", "2020-1-2")))
df$with_columns(result = pl$col("start")$dt$add_business_days(5))

# You can pass a custom weekend - for example, if you only take Sunday off:
week_mask <- c(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE)
df$with_columns(
  result = pl$col("start")$dt$add_business_days(5, week_mask = week_mask)
)

# You can also pass a list of holidays:
holidays <- as.Date(c("2020-1-3", "2020-1-6"))
df$with_columns(
  result = pl$col("start")$dt$add_business_days(5, holidays = holidays)
)

# Roll all dates forwards to the next business day:
df <- pl$DataFrame(start = as.Date(c("2020-1-5", "2020-1-6")))
df$with_columns(
  rolled_forwards = pl$col("start")$dt$add_business_days(0, roll = "forward")
)
```

---

expr\_dt\_base\_utc\_offset

*Base offset from UTC*

---

## Description

This computes the offset between a time zone and UTC. This is usually constant for all datetimes in a given time zone, but may vary in the rare case that a country switches time zone, like Samoa (Apia) did at the end of 2011. Use `$dt$dst_offset()` to take daylight saving time into account.

## Usage

```
expr_dt_base_utc_offset()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  x = as.POSIXct(c("2011-12-29", "2012-01-01"), tz = "Pacific/Apia")
)
df$with_columns(base_utc_offset = pl$col("x")$dt$base_utc_offset())
```

---

```
expr_dt_cast_time_unit
```

*Change time unit*

---

**Description**

Cast the underlying data to another time unit. This may lose precision.

**Usage**

```
expr_dt_cast_time_unit(time_unit)
```

**Arguments**

`time_unit` One of "us" (microseconds), "ns" (nanoseconds) or "ms"(milliseconds). Representing the unit of time.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$select(
  date = pl$datetime_range(
    start = as.Date("2001-1-1"),
    end = as.Date("2001-1-3"),
    interval = "1d1s"
  )
)
df$with_columns(
  cast_time_unit_ms = pl$col("date")$dt$cast_time_unit("ms"),
  cast_time_unit_ns = pl$col("date")$dt$cast_time_unit("ns"),
)
```



---

|                 |   |
|-----------------|---|
| expr_dt_century | <i>Extract the century from underlying representation</i> |
|-----------------|---|

---

### Description

Returns the century number in the calendar date.

### Usage

```
expr_dt_century()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(  
  date = as.Date(  
    c("999-12-31", "1897-05-07", "2000-01-01", "2001-07-05", "3002-10-20")  
  )  
)  
df$with_columns(  
  century = pl$col("date")$dt$century()  
)
```

---

|                 |                              |
|-----------------|------------------------------|
| expr_dt_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

```
expr_dt_combine(time, time_unit = c("us", "ns", "ms"))
```

### Arguments

|                        |   |
|------------------------|---|
| <code>time</code>      | The number of epoch since or before (if negative) the Date. Can be an Expr or a PTime.                        |
| <code>time_unit</code> | One of "us" (default, microseconds), "ns" (nanoseconds) or "ms"(milliseconds). Representing the unit of time. |

### Value

A polars [expression](#)

## 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 = hms::parse_hms(c("1:2:3.456000", "7:8:9.101000"))
)

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(hms::parse_hms("4:5:6"))
)
```

---

```
expr_dt_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

```
expr_dt_convert_time_zone(time_zone)
```

## Arguments

`time_zone` A character time zone from `base::OlsonNames()`.

## Value

A polars [expression](#)

## Examples

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

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

---

|              |                                     |
|--------------|-------------------------------------|
| expr_dt_date | <i>Extract date from date(time)</i> |
|--------------|-------------------------------------|

---

### Description

Extract date from date(time)

### Usage

```
expr_dt_date()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(  
  datetime = as.POSIXct(c("1978-1-1 1:1:1", "1897-5-7 00:00:00"), tz = "UTC")  
)  
df$with_columns(  
  date = pl$col("datetime")$dt$date()  
)
```

---

|             |  |
|-------------|--|
| expr_dt_day | <i>Extract day from underlying Date representation</i> |
|-------------|--|

---

### Description

Returns the day of month starting from 1. The return value ranges from 1 to 31 (the last day of month differs across months).

### Usage

```
expr_dt_day()
```

### Value

A polars [expression](#)

**Examples**

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

---

expr\_dt\_dst\_offset     *Daylight savings offset from UTC*

---

**Description**

This computes the offset between a time zone and UTC, taking into account daylight saving time. Use `$dt$base_utc_offset()` to avoid counting DST.

**Usage**

```
expr_dt_dst_offset()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  x = as.POSIXct(c("2020-10-25", "2020-10-26"), tz = "Europe/London")
)
df$with_columns(dst_offset = pl$col("x")$dt$dst_offset())
```

---

expr\_dt\_epoch     *Get epoch of given Datetime*

---

**Description**

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

**Usage**

```
expr_dt_epoch(time_unit = c("us", "ns", "ms", "s", "d"))
```

**Arguments**

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

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$select(date = pl$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")
)
```

---

|                           |   |
|---------------------------|---|
| <code>expr_dt_hour</code> | <i>Extract hour from underlying Datetime representation</i> |
|---------------------------|---|

---

**Description**

Returns the hour number from 0 to 23.

**Usage**

```
expr_dt_hour()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$select(
  date = pl$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")
)
```

---

`expr_dt_iso_year` *Extract ISO year from underlying Date representation*

---

### Description

Returns the year number in the ISO standard. This may not correspond with the calendar year.

### Usage

```
expr_dt_iso_year()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(  
  date = as.Date(c("1977-01-01", "1978-01-01", "1979-01-01"))  
)  
df$with_columns(  
  year = pl$col("date")$dt$year(),  
  iso_year = pl$col("date")$dt$iso_year()  
)
```

---

`expr_dt_is_leap_year` *Determine whether the year of the underlying date is a leap year*

---

### Description

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

### Usage

```
expr_dt_is_leap_year()
```

### Value

A polars [expression](#)

### 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()  
)
```

---

`expr_dt_microsecond` *Extract microseconds from underlying Datetime representation*

---

### Description

Extract microseconds from underlying Datetime representation

### Usage

```
expr_dt_microsecond()
```

### Value

A polars [expression](#)

### 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()  
)
```

---

`expr_dt_millisecond` *Extract milliseconds from underlying Datetime representation*

---

### Description

Extract milliseconds from underlying Datetime representation

### Usage

```
expr_dt_millisecond()
```

### Value

A polars [expression](#)

**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$with_columns(
  millisecond = pl$col("datetime")$dt$millisecond()
)
```

---

**expr\_dt\_minute**
*Extract minute from underlying Datetime representation*


---

**Description**

Returns the minute number from 0 to 59.

**Usage**

```
expr_dt_minute()
```

**Value**

A polars [expression](#)

**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$with_columns(
  pl$col("datetime")$dt$minute()$alias("minute")
)
```



---

|               |  |
|---------------|--|
| expr_dt_month | <i>Extract month from underlying Date representation</i> |
|---------------|--|

---

### Description

Returns the month number between 1 and 12.

### Usage

```
expr_dt_month()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(  
  date = as.Date(c("2001-01-01", "2001-06-30", "2001-12-27"))  
)  
df$with_columns(  
  month = pl$col("date")$dt$month()  
)
```

---

|                   |  |
|-------------------|--|
| expr_dt_month_end | <i>Roll forward to the last day of the month</i> |
|-------------------|--|

---

### Description

For datetimes, the time of day is preserved.

### Usage

```
expr_dt_month_end()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(date = as.Date(c("2000-01-23", "2001-01-12", "2002-01-01")))  
df$with_columns(  
  month_end = pl$col("date")$dt$month_end()  
)
```

---

`expr_dt_month_start` *Roll backward to the first day of the month*

---

### Description

For datetimes, the time of day is preserved.

### Usage

```
expr_dt_month_start()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(date = as.Date(c("2000-01-23", "2001-01-12", "2002-01-01")))

df$with_columns(
  month_start = pl$col("date")$dt$month_start()
)
```

---

`expr_dt_nanosecond` *Extract nanoseconds from underlying Datetime representation*

---

### Description

Extract nanoseconds from underlying Datetime representation

### Usage

```
expr_dt_nanosecond()
```

### Value

A polars [expression](#)

### 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()
)

```

---

`expr_dt_offset_by`      *Offset a date by a relative time offset*

---

## Description

This differs from `pl$col("foo") + Duration` 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

```
expr_dt_offset_by(by)
```

## Arguments

`by`                    optional string encoding duration see details.

## Details

The `by` are created with 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`

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".

These strings can be combined:

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

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$select(
  dates = pl$date_range(
    as.Date("2000-1-1"),
    as.Date("2005-1-1"),
    "1y"
  )
)
df$with_columns(
  date_plus_1y = pl$col("dates")$dt$offset_by("1y"),
  date_negative_offset = pl$col("dates")$dt$offset_by("-1y2mo")
)

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

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

---

expr\_dt\_ordinal\_day *Extract ordinal day from underlying Date representation*

---

**Description**

Returns the day of year starting from 1. The return value ranges from 1 to 366 (the last day of year differs across years).

**Usage**

```
expr_dt_ordinal_day()
```

**Value**

A polars [expression](#)

## Examples

```
df <- pl$select(  
  date = pl$date_range(  
    as.Date("2020-12-25"),  
    as.Date("2021-1-05"),  
    interval = "1d"  
  )  
)  
df$with_columns(  
  ordinal_day = pl$col("date")$dt$ordinal_day()  
)
```

---

expr\_dt\_quarter

*Extract quarter from underlying Date representation*

---

## Description

Returns the quarter ranging from 1 to 4.

## Usage

```
expr_dt_quarter()
```

## Value

A polars [expression](#)

## Examples

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

---

```
expr_dt_replace_time_zone
```

*Replace time zone for an expression of type Datetime*

---

## Description

Different from `$dt$convert_time_zone()`, this will also modify the underlying timestamp and will ignore the original time zone.

## Usage

```
expr_dt_replace_time_zone(
  time_zone,
  ...,
  ambiguous = c("raise", "earliest", "latest", "null"),
  non_existent = c("raise", "null")
)
```

## Arguments

|                           |  |
|---------------------------|--|
| <code>time_zone</code>    | NULL or a character time zone from <code>base::OlsonNames()</code> . Pass NULL to unset time zone.   |
| <code>...</code>          | These dots are for future extensions and must be empty.  |
| <code>ambiguous</code>    | Determine how to deal with ambiguous datetimes. Character vector or <a href="#">expression</a> containing the followings: <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> |
| <code>non_existent</code> | Determine how to deal with non-existent datetimes. One of the followings: <ul style="list-style-type: none"> <li>• "raise" (default): Throw an error</li> <li>• "null": Return a null value</li> </ul>   |

## Value

A polars [expression](#)

## Examples

```
df <- pl$select(
  london_timezone = pl$datetime_range(
    as.Date("2020-03-01"),
    as.Date("2020-07-01"),
    "1mo",
    time_zone = "UTC"
  )$dt$convert_time_zone(time_zone = "Europe/London")
)
```

```

)
df$with_columns(
  London_to_Amsterdam = pl$col("london_timezone")$dt$replace_time_zone(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"
) |>
  as.POSIXct("UTC")

df2 <- pl$DataFrame(
  ts = as_polars_series(dates),
  ambiguous = c("earliest", "earliest", "latest", "latest"),
)

df2$with_columns(
  ts_localized = pl$col("ts")$dt$replace_time_zone(
    "Europe/Brussels",
    ambiguous = pl$col("ambiguous")
  )
)

```

---

expr\_dt\_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. Ambiguous results are localised using the DST offset of the original timestamp - for example, rounding '2022-11-06 01:20:00 CST' by '1h' results in '2022-11-06 01:00:00 CST', whereas rounding '2022-11-06 01:20:00 CDT' by '1h' results in '2022-11-06 01:00:00 CDT'.

## Usage

```
expr_dt_round(every)
```

## Arguments

|                    |  |
|--------------------|--|
| <code>every</code> | 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

A polars [expression](#)

## Examples

```
df <- pl$select(
  datetime = pl$datetime_range(
    as.Date("2001-01-01"),
    as.Date("2001-01-02"),
    as.difftime("0:25:0")
  )
)
df$with_columns(round = pl$col("datetime")$dt$round("1h"))

df <- pl$select(
  datetime = pl$datetime_range(
    as.POSIXct("2001-01-01 00:00"),
    as.POSIXct("2001-01-01 01:00"),
    as.difftime("0:10:0")
  )
)
df$with_columns(round = pl$col("datetime")$dt$round("1h"))
```



---

|                |  |
|----------------|--|
| expr_dt_second | <i>Extract seconds from underlying Datetime representation</i> |
|----------------|--|

---

### Description

Returns the integer second number from 0 to 59, or a floating point number from 0 to 60 if `fractional = TRUE` that includes any milli/micro/nanosecond component.

### Usage

```
expr_dt_second(fractional = FALSE)
```

### Arguments

`fractional` If TRUE, include the fractional component of the second.

### Value

A polars [expression](#)

### 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)
)
```

---

|                  |  |
|------------------|--|
| expr_dt_strftime | <i>Convert a Date/Time/Datetime/Duration column into a String column with the given format</i> |
|------------------|--|

---

### Description

Similar to `$cast(pl$String)`, but this method allows you to customize the formatting of the resulting string. This is an alias for `$dt$to_string()`.

**Usage**

```
expr_dt_strftime(format)
```

**Arguments**

**format** Single string of format to use, or NULL. NULL will be treated as "iso". Available formats depend on the column [data type](#):

- For [Date/Time/Datetime](#), refer to the [chrono strftime documentation](#) for specification. Example: "%y-%m-%d". Special case "iso" will use the ISO8601 format.
- For [Duration](#), "iso" or "polars" can be used. The "iso" format string results in ISO8601 duration string output, and "polars" results in the same form seen in the polars print representation.

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(
  datetime = c(as.POSIXct(c("2021-01-02 00:00:00", "2021-01-03 00:00:00")))
)$
  with_columns(
    datetime_string = pl$col("datetime")$dt$strftime("%Y/%m/%d %H:%M:%S")
  )
```

---

```
expr_dt_time
```

*Extract time*

---

**Description**

This only works on Datetime columns, it will error on Date columns.

**Usage**

```
expr_dt_time()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$select(dates = pl$datetime_range(
  as.Date("2000-1-1"),
  as.Date("2000-1-2"),
  "1h"
))

df$with_columns(times = pl$col("dates")$dt$time())
```

---

expr\_dt\_timestamp      *Get timestamp in the given time unit*

---

**Description**

Get timestamp in the given time unit

**Usage**

```
expr_dt_timestamp(time_unit = c("us", "ns", "ms"))
```

**Arguments**

time\_unit      Time unit, one of 'ns', 'us', or 'ms'.

**Value**

A polars [expression](#)

**Examples**

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

df$select(
  pl$col("date"),
  pl$col("date")$dt$timestamp()$alias("timestamp_ns"),
  pl$col("date")$dt$timestamp(time_unit = "ms")$alias("timestamp_ms")
)
```

---

`expr_dt_total_days` *Extract the days from a Duration type*

---

### Description

Extract the days from a Duration type

### Usage

```
expr_dt_total_days()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$select(  
  date = pl$datetime_range(  
    start = as.Date("2020-3-1"),  
    end = as.Date("2020-5-1"),  
    interval = "1mo1s"  
  )  
)  
df$with_columns(  
  diff_days = pl$col("date")$diff()$dt$total_days()  
)
```

---

`expr_dt_total_hours` *Extract the hours from a Duration type*

---

### Description

Extract the hours from a Duration type

### Usage

```
expr_dt_total_hours()
```

### Value

A polars [expression](#)

## Examples

```
df <- pl$select(  
  date = pl$date_range(  
    start = as.Date("2020-1-1"),  
    end = as.Date("2020-1-4"),  
    interval = "1d"  
  )  
)  
df$with_columns(  
  diff_hours = pl$col("date")$diff()$dt$total_hours()  
)
```

---

expr\_dt\_total\_microseconds

*Extract the microseconds from a Duration type*

---

## Description

Extract the microseconds from a Duration type

## Usage

```
expr_dt_total_microseconds()
```

## Value

A polars [expression](#)

## Examples

```
df <- pl$select(date = pl$datetime_range(  
  start = as.POSIXct("2020-1-1", tz = "GMT"),  
  end = as.POSIXct("2020-1-1 00:00:01", tz = "GMT"),  
  interval = "1ms"  
)  
)  
df$with_columns(  
  diff_microsec = pl$col("date")$diff()$dt$total_microseconds()  
)
```

expr\_dt\_total\_milliseconds

*Extract the milliseconds from a Duration type*

---

### Description

Extract the milliseconds from a Duration type

### Usage

```
expr_dt_total_milliseconds()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$select(date = pl$datetime_range(  
  start = as.POSIXct("2020-1-1", tz = "GMT"),  
  end = as.POSIXct("2020-1-1 00:00:01", tz = "GMT"),  
  interval = "1ms"  
)  
)  
df$with_columns(  
  diff_millisecc = pl$col("date")$diff()$dt$total_milliseconds()  
)
```

---

expr\_dt\_total\_minutes

*Extract the minutes from a Duration type*

---

### Description

Extract the minutes from a Duration type

### Usage

```
expr_dt_total_minutes()
```

### Value

A polars [expression](#)

## Examples

```
df <- pl$select(  
  date = pl$date_range(  
    start = as.Date("2020-1-1"),  
    end = as.Date("2020-1-4"),  
    interval = "1d"  
  )  
)  
df$with_columns(  
  diff_minutes = pl$col("date")$diff()$dt$total_minutes()  
)
```

---

expr\_dt\_total\_nanoseconds

*Extract the nanoseconds from a Duration type*

---

## Description

Extract the nanoseconds from a Duration type

## Usage

```
expr_dt_total_nanoseconds()
```

## Value

A polars [expression](#)

## Examples

```
df <- pl$select(date = pl$datetime_range(  
  start = as.POSIXct("2020-1-1", tz = "GMT"),  
  end = as.POSIXct("2020-1-1 00:00:01", tz = "GMT"),  
  interval = "1ms"  
)  
)  
df$with_columns(  
  diff_nanosec = pl$col("date")$diff()$dt$total_nanoseconds()  
)
```

---

```
expr_dt_total_seconds
```

*Extract the seconds from a Duration type*

---

### Description

Extract the seconds from a Duration type

### Usage

```
expr_dt_total_seconds()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$select(date = pl$datetime_range(
  start = as.POSIXct("2020-1-1", tz = "GMT"),
  end = as.POSIXct("2020-1-1 00:04:00", tz = "GMT"),
  interval = "1m"
))
df$with_columns(
  diff_sec = pl$col("date")$diff()$dt$total_seconds()
)
```

---

```
expr_dt_to_string
```

*Convert a Date/Time/Datetime/Duration column into a String column with the given format*

---

### Description

Similar to `$cast(pl$String)`, but this method allows you to customize the formatting of the resulting string; if no format is provided, the appropriate ISO format for the underlying data type is used.

### Usage

```
expr_dt_to_string(format = NULL)
```



**Arguments**

- format** Single string of format to use, or NULL (default). NULL will be treated as "iso". Available formats depend on the column [data type](#):
- For [Date/Time/Datetime](#), refer to the [chrono strftime documentation](#) for specification. Example: "%y-%m-%d". Special case "iso" will use the ISO8601 format.
  - For [Duration](#), "iso" or "polars" can be used. The "iso" format string results in ISO8601 duration string output, and "polars" results in the same form seen in the polars print representation.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  dt = as.Date(c("1990-03-01", "2020-05-03", "2077-07-05")),
  dtm = as.POSIXct(c("1980-08-10 00:10:20", "2010-10-20 08:25:35", "2040-12-30 16:40:50")),
  tm = hms::as_hms(c("01:02:03.456789", "23:59:09.101", "00:00:00.000100")),
  dur = clock::duration_days(c(-1, 14, 0)) + clock::duration_hours(c(0, -10, 0)) +
    clock::duration_seconds(c(-42, 0, 0)) + clock::duration_microseconds(c(0, 1001, 0)),
)

# Default format for temporal dtypes is ISO8601:
df$select((cs$date() | cs$datetime())$dt$to_string())$name$prefix("s_")
df$select((cs$time() | cs$duration())$dt$to_string())$name$prefix("s_")

# All temporal types (aside from Duration) support strftime formatting:
df$select(
  pl$col("dtm"),
  s_dtm = pl$col("dtm")$dt$to_string("%Y/%m/%d (%H.%M.%S)"),
)

# The Polars Duration string format is also available:
df$select(pl$col("dur"), s_dur = pl$col("dur")$dt$to_string("polars"))

# If you're interested in extracting the day or month names,
# you can use the '%A' and '%B' strftime specifiers:
df$select(
  pl$col("dt"),
  day_name = pl$col("dt")$dt$to_string("%A"),
  month_name = pl$col("dt")$dt$to_string("%B"),
)
```

---

`expr_dt_truncate`      *Truncate datetime*

---

## Description

Divide the date/datetime range into buckets. Each date/datetime is mapped to the start of its bucket using the corresponding local datetime. Note that weekly buckets start on Monday. Ambiguous results are localised using the DST offset of the original timestamp - for example, truncating '2022-11-06 01:30:00 CST' by '1h' results in '2022-11-06 01:00:00 CST', whereas truncating '2022-11-06 01:30:00 CDT' by '1h' results in '2022-11-06 01:00:00 CDT'.

## Usage

```
expr_dt_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

A polars [expression](#)

**Examples**

```
df <- pl$select(
  datetime = pl$datetime_range(
    as.Date("2001-01-01"),
    as.Date("2001-01-02"),
    as.difftime("0:25:0")
  )
)
df$with_columns(truncated = pl$col("datetime")$dt$truncate("1h"))

df <- pl$select(
  datetime = pl$datetime_range(
    as.POSIXct("2001-01-01 00:00"),
    as.POSIXct("2001-01-01 01:00"),
    as.difftime("0:10:0")
  )
)
df$with_columns(truncated = pl$col("datetime")$dt$truncate("30m"))
```

---

 expr\_dt\_week

*Extract week from underlying Date representation*


---

**Description**

Returns the ISO week number starting from 1. The return value ranges from 1 to 53 (the last week of year differs across years).

**Usage**

```
expr_dt_week()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$select(
  date = pl$date_range(
    as.Date("2020-12-25"),
    as.Date("2021-1-05"),
    interval = "1d"
  )
)
df$with_columns(
  week = pl$col("date")$dt$week()
)
```

---

|                 |  |
|-----------------|--|
| expr_dt_weekday | <i>Extract weekday from underlying Date representation</i> |
|-----------------|--|

---

### Description

Returns the ISO weekday number where Monday = 1 and Sunday = 7.

### Usage

```
expr_dt_weekday()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$select(  
  date = pl$date_range(  
    as.Date("2020-12-25"),  
    as.Date("2021-1-05"),  
    interval = "1d"  
  )  
)  
df$with_columns(  
  weekday = pl$col("date")$dt$weekday()  
)
```

---

|                        |  |
|------------------------|--|
| expr_dt_with_time_unit | <i>Set time unit of a Series of dtype Datetime or Duration</i> |
|------------------------|--|

---

### Description

This is deprecated. Cast to Int64 and then to Datetime instead.

### Usage

```
expr_dt_with_time_unit(time_unit = c("ns", "us", "ms"))
```

### Arguments

time\_unit      Time unit, one of 'ns', 'us', or 'ms'.

### Value

A polars [expression](#)

**Examples**

```
df <- pl$select(
  date = pl$datetime_range(
    start = as.Date("2001-1-1"),
    end = as.Date("2001-1-3"),
    interval = "1d1s"
  )
)
df$with_columns(
  with_time_unit_ns = pl$col("date")$dt$with_time_unit(),
  with_time_unit_ms = pl$col("date")$dt$with_time_unit(time_unit = "ms")
)
```

---

 expr\_dt\_year

*Extract year from underlying Date representation*


---

**Description**

Returns the year number in the calendar date.

**Usage**

```
expr_dt_year()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  date = as.Date(c("1977-01-01", "1978-01-01", "1979-01-01"))
)
df$with_columns(
  year = pl$col("date")$dt$year(),
  iso_year = pl$col("date")$dt$iso_year()
)
```

---

 expr\_list\_all

*Evaluate whether all boolean values in a sub-list are true*


---

**Description**

Evaluate whether all boolean values in a sub-list are true

**Usage**

```
expr_list_all()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = list(c(TRUE, TRUE), c(FALSE, TRUE), c(FALSE, FALSE), NA, c())
)
df$with_columns(all = pl$col("a")$list$all())
```

---

|               |   |
|---------------|---|
| expr_list_any | <i>Evaluate whether any boolean value in a sub-list is true</i> |
|---------------|---|

---

**Description**

Evaluate whether any boolean value in a sub-list is true

**Usage**

```
expr_list_any()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = list(c(TRUE, TRUE), c(FALSE, TRUE), c(FALSE, FALSE), NA, c())
)
df$with_columns(any = pl$col("a")$list$any())
```

---

|                   |  |
|-------------------|--|
| expr_list_arg_max | <i>Retrieve the index of the maximum value in every sub-list</i> |
|-------------------|--|

---

**Description**

Retrieve the index of the maximum value in every sub-list

**Usage**

```
expr_list_arg_max()
```

**Value**

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(s = list(1:2, 2:1))
df$with_columns(
  arg_max = pl$col("s")$list$arg_max()
)
```

---

`expr_list_arg_min`      *Retrieve the index of the minimum value in every sub-list*

---

### Description

Retrieve the index of the minimum value in every sub-list

### Usage

```
expr_list_arg_min()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(s = list(1:2, 2:1))
df$with_columns(
  arg_min = pl$col("s")$list$arg_min()
)
```

---

`expr_list_concat`      *Concat the lists into a new list*

---

### Description

Concat the lists into a new list

### Usage

```
expr_list_concat(other)
```

### Arguments

`other`      Values to concat with. Can be an Expr or something coercible to an Expr.

### Value

A polars [expression](#)

**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"))))
)
```

---

`expr_list_contains`     *Check if sub-lists contains a given value*

---

**Description**

Check if sub-lists contains a given value

**Usage**

```
expr_list_contains(item)
```

**Arguments**

`item`                    Item that will be checked for membership. Can be an Expr or something coercible to an Expr. Strings are not parsed as columns.

**Value**

A polars [expression](#)

**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)
)
```



---

```
expr_list_count_matches
```

*Count how often a value produced occurs*

---

### Description

Count how often a value produced occurs

### Usage

```
expr_list_count_matches(element)
```

### Arguments

`element` An expression that produces a single value.

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = list(0, 1, c(1, 2, 3, 2), c(1, 2, 1), c(4, 4)))

df$with_columns(
  number_of_twos = pl$col("a")$list$count_matches(2)
)
```

---

```
expr_list_diff
```

*Compute difference between sub-list values*

---

### 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. if `n = 2` the output will be the difference between the 1st and the 3rd value, the 2nd and 4th value, etc.

### Usage

```
expr_list_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**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(s = list(1:4, c(10L, 2L, 1L)))
df$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))
```

---

`expr_list_drop_nulls` *Drop all null values in every sub-list*

---

**Description**

Drop all null values in every sub-list

**Usage**

```
expr_list_drop_nulls()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = list(c(NA, 0, NA), c(1, NaN), NA))

df$with_columns(
  without_nulls = pl$col("values")$list$drop_nulls()
)
```

---

`expr_list_eval` *Run any polars expression on the sub-lists' values*

---

**Description**

Run any polars expression on the sub-lists' values

**Usage**

```
expr_list_eval(expr, ..., parallel = FALSE)
```

**Arguments**

|                       |   |
|-----------------------|---|
| <code>expr</code>     | Expression to run. Note that you can select an element with <code>pl\$element()</code> , <code>pl\$first()</code> , and more. See Examples.   |
| <code>...</code>      | These dots are for future extensions and must be empty.   |
| <code>parallel</code> | Run all expressions in 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 <code>\$group_by()</code> context, because groups are already executed in parallel. |

**Value**

A polars [expression](#)

**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()
)
```

---

`expr_list_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**

```
expr_list_explode()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = list(c(1, 2, 3), c(4, 5, 6)))
df$select(pl$col("a")$list$explode())
```

---

|                 |   |
|-----------------|---|
| expr_list_first | <i>Get the first value of the sub-lists</i> |
|-----------------|---|

---

**Description**

Get the first value of the sub-lists

**Usage**

```
expr_list_first()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = list(3:1, NULL, 1:2))
df$with_columns(
  first = pl$col("a")$list$first()
)
```

---

|                  |  |
|------------------|--|
| expr_list_gather | <i>Get several values by index in every sub-list</i> |
|------------------|--|

---

**Description**

This allows to extract several values per list. To extract a single value by index, use [\\$list\\$get\(\)](#). The indices may be defined in a single column, or by sub-lists in another column of dtype List.

**Usage**

```
expr_list_gather(index, ..., null_on_oob = FALSE)
```

**Arguments**

|                          |  |
|--------------------------|--|
| <code>index</code>       | An Expr or something coercible to an Expr, that can return several indices. Values are 0-indexed (so index 0 would return the first item of every sub-list) 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. |
| <code>...</code>         | These dots are for future extensions and must be empty.  |
| <code>null_on_oob</code> | If TRUE, return null if an index is out of bounds. Otherwise, raise an error.  |

**Value**

A polars [expression](#)

**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)
)
```

---

`expr_list_gather_every`

*Take every n-th value starting from offset in sub-lists*

---

**Description**

Take every n-th value starting from offset in sub-lists

**Usage**

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

**Arguments**

|                     |                        |
|---------------------|------------------------|
| <code>n</code>      | Gather every n-th row. |
| <code>offset</code> | Starting index.        |

**Value**

A polars [expression](#)

**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"))
)
```

---

|                            |   |
|----------------------------|---|
| <code>expr_list_get</code> | <i>Get the value by index in every sub-list</i> |
|----------------------------|---|

---

**Description**

This allows to extract one value per list only. To extract several values by index, use [\\$list\\$gather\(\)](#).

**Usage**

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

**Arguments**

|                          |  |
|--------------------------|--|
| <code>index</code>       | 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-list) and negative values start from the end (index -1 returns the last item). |
| <code>...</code>         | These dots are for future extensions and must be empty.  |
| <code>null_on_oob</code> | 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, 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)
)

```

---

expr\_list\_head      *Slice the first n values of every sub-list*

---

### Description

Slice the first n values of every sub-list

### Usage

```
expr_list_head(n = 5L)
```

### Arguments

n                      Number of values to return for each sub-list. Can be an Expr. Strings are parsed as column names.

### Value

A polars [expression](#)

### 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)
)

```

---

expr\_list\_join      *Join elements of every sub-list*

---

### Description

Join all string items in a sub-list and place a separator between them. This only works if the inner dtype is `String`.

### Usage

```
expr_list_join(separator, ..., ignore_nulls = FALSE)
```

**Arguments**

- separator** String to separate the items with. Can be an Expr. Strings are *not* parsed as columns.
- ...** [<dynamic-dots>](#) Columns to concatenate into a single string column. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals. Non-String columns are cast to String.
- ignore\_nulls** If FALSE (default), null values will be propagated, i.e. if the row contains any null values, the output is null.

**Value**

A polars [expression](#)

**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)
)
```

---

expr\_list\_last

*Get the last value of the sub-lists*

---

**Description**

Get the last value of the sub-lists

**Usage**

```
expr_list_last()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = list(3:1, NULL, 1:2))
df$with_columns(
  last = pl$col("a")$list$last()
)
```



---

|               |   |
|---------------|---|
| expr_list_len | <i>Return the number of elements in each sub-list</i> |
|---------------|---|

---

### Description

Null values are counted in the total.

### Usage

```
expr_list_len()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(list_of_strs = list(c("a", "b", NA), "c"))
df$with_columns(len_list = pl$col("list_of_strs")$list$len())
```

---

|               |  |
|---------------|--|
| expr_list_max | <i>Compute the maximum value in every sub-list</i> |
|---------------|--|

---

### Description

Compute the maximum value in every sub-list

### Usage

```
expr_list_max()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA))
df$with_columns(max = pl$col("values")$list$max())
```

---

`expr_list_mean`      *Compute the mean value in every sub-list*

---

**Description**

Compute the mean value in every sub-list

**Usage**

```
expr_list_mean()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA))
df$with_columns(mean = pl$col("values")$list$mean())
```

---

`expr_list_median`      *Compute the median in every sub-list*

---

**Description**

Compute the median in every sub-list

**Usage**

```
expr_list_median()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = list(c(-1, 0, 1), c(1, 10)))

df$with_columns(
  median = pl$col("values")$list$median()
)
```

---

|               |  |
|---------------|--|
| expr_list_min | <i>Compute the minimum value in every sub-list</i> |
|---------------|--|

---

**Description**

Compute the minimum value in every sub-list

**Usage**

```
expr_list_min()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA))
df$with_columns(min = pl$col("values")$list$min())
```

---

|                    |   |
|--------------------|---|
| expr_list_n_unique | <i>Count the number of unique values in every sub-lists</i> |
|--------------------|---|

---

**Description**

Count the number of unique values in every sub-lists

**Usage**

```
expr_list_n_unique()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = list(c(2, 2, NA), c(1, 2, 3), NA))
df$with_columns(unique = pl$col("values")$list$n_unique())
```

---

`expr_list_reverse`      *Reverse values in every sub-list*

---

### Description

Reverse values in every sub-list

### Usage

```
expr_list_reverse()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA))
df$with_columns(reverse = pl$col("values")$list$reverse())
```

---

`expr_list_sample`      *Sample values from every sub-list*

---

### Description

Sample values from every sub-list

### Usage

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

### Arguments

`n`                      Number of items to return. Cannot be used with `fraction`. Defaults to 1 if `fraction` is `NULL`.

`...`                    These dots are for future extensions and must be empty.

`fraction`                Fraction of items to return. Cannot be used with `n`.

`with_replacement`      Allow values to be sampled more than once.

|                      |  |
|----------------------|--|
| <code>shuffle</code> | Shuffle the order of sampled data points.  |
| <code>seed</code>    | Seed for the random number generator. If NULL (default), a random seed is generated for each sample operation. |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  values = list(1:3, NA, c(NA, 3L), 5:7),
  n = c(1, 1, 1, 2)
)

df$with_columns(
  sample = pl$col("values")$list$sample(n = pl$col("n"), seed = 1)
)
```

---

`expr_list_set_difference`

*Compute the set difference between elements of a list and other elements*

---

**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**

```
expr_list_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**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = list(1:3, NA, c(NA, 3L), 5:7),
  b = list(2:4, 3L, c(3L, 4L, NA), c(6L, 8L))
)

df$with_columns(difference = pl$col("a")$list$set_difference("b"))
```

---

```
expr_list_set_intersection
```

*Compute the intersection between elements of a list and other elements*

---

**Description**

Compute the intersection between elements of a list and other elements

**Usage**

```
expr_list_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**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = list(1:3, NA, c(NA, 3L), 5:7),
  b = list(2:4, 3L, c(3L, 4L, NA), c(6L, 8L))
)

df$with_columns(intersection = pl$col("a")$list$set_intersection("b"))
```

---

`expr_list_set_symmetric_difference`

*Compute the set symmetric difference between elements of a list and other elements*

---

## 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

```
expr_list_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

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(  
  a = list(1:3, NA, c(NA, 3L), 5:7),  
  b = list(2:4, 3L, c(3L, 4L, NA), c(6L, 8L))  
)  
  
df$with_columns(  
  symmetric_difference = pl$col("a")$list$set_symmetric_difference("b")  
)
```

---

`expr_list_set_union` *Compute the union of elements of a list and other elements*

---

### Description

Compute the union of elements of a list and other elements

### Usage

```
expr_list_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

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(
  a = list(1:3, NA, c(NA, 3L), 5:7),
  b = list(2:4, 3L, c(3L, 4L, NA), c(6L, 8L))
)

df$with_columns(union = pl$col("a")$list$set_union("b"))
```

---

`expr_list_shift` *Shift list values by the given number of indices*

---

### Description

Shift list values by the given number of indices

### Usage

```
expr_list_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**

A polars [expression](#)

**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$shift(2),
  shift_by_negative_lit = pl$col("s")$list$shift(-2)
)
```

---

|                 |                             |
|-----------------|-----------------------------|
| expr_list_slice | <i>Slice every sub-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**

```
expr_list_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**

A polars [expression](#)

**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)
)
```

---

|                |                                      |
|----------------|--------------------------------------|
| expr_list_sort | <i>Sort values in every sub-list</i> |
|----------------|--------------------------------------|

---

**Description**

Sort values in every sub-list

**Usage**

```
expr_list_sort(..., descending = FALSE, nulls_last = FALSE)
```

**Arguments**

|            |   |
|------------|---|
| ...        | These dots are for future extensions and must be empty. |
| descending | Sort values in descending order.                        |
| nulls_last | Place null values last.                                 |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = list(c(NA, 2, 1, 3), c(Inf, 2, 3, NaN), NA))
df$with_columns(sort = pl$col("values")$list$sort())
```

---

|               |   |
|---------------|---|
| expr_list_std | <i>Compute the standard deviation in every sub-list</i> |
|---------------|---|

---

**Description**

Compute the standard deviation in every sub-list

**Usage**

```
expr_list_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 polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = list(c(-1, 0, 1), c(1, 10)))

df$with_columns(
  std = pl$col("values")$list$std()
)
```

---

|               |   |
|---------------|---|
| expr_list_sum | <i>Sum all elements in every sub-list</i> |
|---------------|---|

---

**Description**

Sum all elements in every sub-list

**Usage**

```
expr_list_sum()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA))
df$with_columns(sum = pl$col("values")$list$sum())
```



**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = list(c(-1, 0), c(1, 10)))

df$with_columns(
  array = pl$col("values")$list$to_array(2)
)
```

---

expr\_list\_unique      *Get unique values in a list*

---

**Description**

Get unique values in a list

**Usage**

```
expr_list_unique(..., maintain_order = FALSE)
```

**Arguments**

...                    These dots are for future extensions and must be empty.

maintain\_order        Maintain order of data. This requires more work.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = list(c(2, 2, NA), c(1, 2, 3), NA))
df$with_columns(unique = pl$col("values")$list$unique())
```

---

`expr_list_var`            *Compute the variance in every sub-list*

---

### Description

Compute the variance in every sub-list

### Usage

```
expr_list_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 polars [expression](#)

### Examples

```
df <- pl$DataFrame(values = list(c(-1, 0, 1), c(1, 10)))

df$with_columns(
  var = pl$col("values")$list$var()
)
```

---

`expr_meta_eq`            *Indicate if this expression is the same as another expression*

---

### Description

Indicate if this expression is the same as another expression

### Usage

```
expr_meta_eq(other)
```

### Arguments

`other`            Expression to compare with.

### Value

A polars [expression](#)

### Examples

```
foo_bar <- pl$col("foo")$alias("bar")
foo <- pl$col("foo")
foo_bar$meta$eq(foo)

foo_bar2 <- pl$col("foo")$alias("bar")
foo_bar$meta$eq(foo_bar2)
```

---

expr\_meta\_has\_multiple\_outputs

*Indicate if this expression expands into multiple expressions*

---

### Description

Indicate if this expression expands into multiple expressions

### Usage

```
expr_meta_has_multiple_outputs()
```

### Value

A polars [expression](#)

### Examples

```
e <- pl$col("a", "b")$name$suffix("_foo")
e$meta$has_multiple_outputs()
```

---

expr\_meta\_is\_column *Indicate if this expression is a basic (non-regex) unaliased column*

---

### Description

Indicate if this expression is a basic (non-regex) unaliased column

### Usage

```
expr_meta_is_column()
```

### Value

A logical value.

**Examples**

```
e <- pl$col("foo")
e$meta$is_column()

e <- pl$col("foo") * pl$col("bar")
e$meta$is_column()

e <- pl$col("^col\\.\\.\\.\\d+$")
e$meta$is_column()
```

---

```
expr_meta_is_column_selection
```

*Indicate if this expression only selects columns (optionally with aliasing)*

---

**Description**

This can include bare columns, column matches by regex or dtype, selectors and exclude ops, and (optionally) column/expression aliasing.

**Usage**

```
expr_meta_is_column_selection(..., allow_aliasing = FALSE)
```

**Arguments**

... These dots are for future extensions and must be empty.

allow\_aliasing If FALSE (default), any aliasing is not considered pure column selection. Set TRUE to allow for column selection that also includes aliasing.

**Value**

A logical value.

**Examples**

```
e <- pl$col("foo")
e$meta$is_column_selection()

e <- pl$col("foo")$alias("bar")
e$meta$is_column_selection()

e$meta$is_column_selection(allow_aliasing = TRUE)

e <- pl$col("foo") * pl$col("bar")
e$meta$is_column_selection()

e <- cs$starts_with("foo")
e$meta$is_column_selection()
```



---

```
expr_meta_is_regex_projection
```

*Indicate if this expression expands to columns that match a regex pattern*

---

### Description

Indicate if this expression expands to columns that match a regex pattern

### Usage

```
expr_meta_is_regex_projection()
```

### Value

A logical value.

### Examples

```
e <- pl$col("^.*$")$name$prefix("foo_")
e$meta$is_regex_projection()
```

---

```
expr_meta_ne
```

*Indicate if this expression is not the same as another expression*

---

### Description

Indicate if this expression is not the same as another expression

### Usage

```
expr_meta_ne(other)
```

### Arguments

`other` Expression to compare with.

### Value

A logical [expression](#)

### Examples

```
foo_bar <- pl$col("foo")$alias("bar")
foo <- pl$col("foo")
foo_bar$meta$ne(foo)

foo_bar2 <- pl$col("foo")$alias("bar")
foo_bar$meta$ne(foo_bar2)
```

---

expr\_meta\_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 = TRUE` (the default), and return `NA` otherwise.

## Usage

```
expr_meta_output_name(..., raise_if_undetermined = TRUE)
```

## Arguments

`...` These dots are for future extensions and must be empty.

`raise_if_undetermined`  
If `TRUE` (default), raise an error if the output name cannot be determined. Otherwise return `NA`.

## Value

A polars [expression](#)

## 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$col("foo")$sum()$over("groups")
e_sum_over$meta$output_name()

e_sum_slice <- pl$col("foo")$sum()$slice(pl$len() - 10, pl$col("bar"))
e_sum_slice$meta$output_name()

pl$len()$meta$output_name()
```

---

|               |   |
|---------------|---|
| expr_meta_pop | <i>Pop the latest expression and return the input(s) of the popped expression</i> |
|---------------|---|

---

### Description

Pop the latest expression and return the input(s) of the popped expression

### Usage

```
expr_meta_pop()
```

### Value

A polars [expression](#)

### Examples

```
e <- pl$col("foo")$alias("bar")
pop <- e$meta$pop()
pop

pop[[1]]$meta$eq(pl$col("foo"))
pop[[1]]$meta$eq(pl$col("foo"))
```

---

|                      |   |
|----------------------|---|
| expr_meta_root_names | <i>Get a list with the root column name</i> |
|----------------------|---|

---

### Description

Get a list with the root column name

### Usage

```
expr_meta_root_names()
```

### Value

A polars [expression](#)

## Examples

```
e <- pl$col("foo") * pl$col("bar")
e$meta$root_names()

e_filter <- pl$col("foo")$filter(pl$col("bar") == 13)
e_filter$meta$root_names()

e_sum_over <- pl$sum("foo")$over("groups")
e_sum_over$meta$root_names()

e_sum_slice <- pl$sum("foo")$slice(pl$len() - 10, pl$col("bar"))
e_sum_slice$meta$root_names()
```

---

`expr_meta_serialize` *Serialize this expression to a string in binary or JSON format*

---

## Description

Serialize this expression to a string in binary or JSON format

## Usage

```
expr_meta_serialize(..., format = c("binary", "json"))
```

## Arguments

`...` These dots are for future extensions and must be empty.

`format` The format in which to serialize. Must be one of:

- "binary" (default): serialize to binary format (bytes).
- "json": serialize to JSON format (string).

## Details

Serialization is not stable across Polars versions: a LazyFrame serialized in one Polars version may not be deserializable in another Polars version.

## Value

A polars [expression](#)

## Examples

```
# Serialize the expression into a binary representation.
expr <- pl$col("foo")$sum()$over("bar")
bytes <- expr$meta$serialize()
rawToChar(bytes)
```

```
pl$deserialize_expr(bytes)

# Serialize into json
expr$meta$serialize(format = "json") |>
  jsonlite::prettify()
```

---

expr\_meta\_tree\_format

*Format the expression as a tree*

---

## Description

Format the expression as a tree

## Usage

```
expr_meta_tree_format()
```

## Value

A character vector

## Examples

```
my_expr <- (pl$col("foo") * pl$col("bar"))$sum()$over(pl$col("ham")) / 2
my_expr$meta$tree_format() |>
  cat()
```

---

expr\_meta\_undo\_aliases

*Undo any renaming operation like alias or name\$keep*

---

## Description

Undo any renaming operation like `alias` or `name$keep`

## Usage

```
expr_meta_undo_aliases()
```

## Value

A polars [expression](#)

**Examples**

```
e <- pl$col("foo")$alias("bar")
e$meta$undo_aliases()$meta$eq(pl$col("foo"))

e <- pl$col("foo")$sum()$over("bar")
e$name$keep()$meta$undo_aliases()$meta$eq(e)
```

---

`expr_struct_field`      *Retrieve one or multiple Struct field(s) as a new Series*

---

**Description**

Retrieve one or multiple Struct field(s) as a new Series

**Usage**

```
expr_struct_field(...)
```

**Arguments**

...                    [<dynamic-dots>](#) Names of struct fields to retrieve.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  aaa = c(1, 2),
  bbb = c("ab", "cd"),
  ccc = c(TRUE, NA),
  ddd = list(1:2, 3)
)$select(struct_col = pl$struct("aaa", "bbb", "ccc", "ddd"))
df

# Retrieve struct field(s) as Series:
df$select(pl$col("struct_col")$struct$field("bbb"))

df$select(
  pl$col("struct_col")$struct$field("bbb"),
  pl$col("struct_col")$struct$field("ddd")
)

# Use wildcard expansion:
df$select(pl$col("struct_col")$struct$field("*"))

# Retrieve multiple fields by name:
df$select(pl$col("struct_col")$struct$field("aaa", "bbb"))
```

```
# Retrieve multiple fields by regex expansion:
df$select(pl$col("struct_col")$struct$field("^a.*|b.*$"))
```

---

expr\_struct\_json\_encode

*Convert this struct to a string column with json values*

---

## Description

Convert this struct to a string column with json values

## Usage

```
expr_struct_json_encode()
```

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(
  a = list(1:2, c(9, 1, 3)),
  b = list(45, NA)
)$select(a = pl$struct("a", "b"))

df

df$with_columns(encoded = pl$col("a")$struct$json_encode())
```

---

expr\_struct\_rename\_fields

*Rename the fields of the struct*

---

## Description

Rename the fields of the struct

## Usage

```
expr_struct_rename_fields(names)
```

## Arguments

**names**            New names, given in the same order as the struct's fields.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  aaa = c(1, 2),
  bbb = c("ab", "cd"),
  ccc = c(TRUE, NA),
  ddd = list(1:2, 3)
)$select(struct_col = pl$struct("aaa", "bbb", "ccc", "ddd"))
df

df <- df$select(
  pl$col("struct_col")$struct$rename_fields(c("www", "xxx", "yyy", "zzz"))
)
df$select(pl$col("struct_col")$struct$field("*"))

# Following a rename, the previous field names cannot be referenced:
tryCatch(
  {
    df$select(pl$col("struct_col")$struct$field("aaa"))
  },
  error = function(e) print(e)
)
```

---

`expr_struct_unnest`    *Expand the struct into its individual fields*

---

**Description**

This is an alias for `Expr$struct$field("*")`.

**Usage**

```
expr_struct_unnest()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  aaa = c(1, 2),
  bbb = c("ab", "cd"),
  ccc = c(TRUE, NA),
  ddd = list(1:2, 3)
)$select(struct_col = pl$struct("aaa", "bbb", "ccc", "ddd"))
df
```



```
df$select(pl$col("struct_col")$struct$unnest())
```

---

```
expr_struct_with_fields
```

*Add or overwrite fields of this struct*

---

## Description

This is similar to `with_columns()` on `DataFrame` and `LazyFrame`.

## Usage

```
expr_struct_with_fields(...)
```

## Arguments

... [<dynamic-dots>](#) Field(s) to add. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals.

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(
  x = c(1, 4, 9),
  y = c(4, 9, 16),
  multiply = c(10, 2, 3)
)$select(coords = pl$struct("x", "y"), "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$select(pl$col("coords")$struct$field("*"))
```

---

`expr_str_contains`      *Check if string contains a substring that matches a pattern*

---

## Description

Check if string contains a substring that matches a pattern

## Usage

```
expr_str_contains(pattern, ..., literal = FALSE, strict = TRUE)
```

## Arguments

|                      |  |
|----------------------|--|
| <code>pattern</code> | 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> . |
| <code>...</code>     | These dots are for future extensions and must be empty.  |
| <code>literal</code> | Logical. If <code>TRUE</code> (default), treat <code>pattern</code> as a literal string, not as a regular expression.                                |
| <code>strict</code>  | Logical. If <code>TRUE</code> (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

A polars [expression](#)

## 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 `(?)` 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)
)

```

---

```
expr_str_contains_any
```

*Use the Aho-Corasick algorithm to find matches*

---

## Description

This function determines if any of the patterns find a match.

## Usage

```
expr_str_contains_any(patterns, ..., ascii_case_insensitive = FALSE)
```

## Arguments

**patterns** Character vector or something can be coerced to strings [Expr](#) of a valid regex pattern, compatible with the [regex crate](#).

**...** These dots are for future extensions and must be empty.

**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

A polars [expression](#)

## 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"))
)

```

---

**expr\_str\_count\_matches**
*Count all successive non-overlapping regex matches*


---

### Description

Count all successive non-overlapping regex matches

### Usage

```
expr_str_count_matches(pattern, ..., literal = FALSE)
```

### Arguments

|                |  |
|----------------|--|
| <b>pattern</b> | 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> . |
| <b>...</b>     | These dots are for future extensions and must be empty.  |
| <b>literal</b> | Logical. If <b>TRUE</b> (default), treat <b>pattern</b> as a literal string, not as a regular expression.  |

### Value

A polars [expression](#)

### 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)
)
```

---

**expr\_str\_decode**
*Decode a value using the provided encoding*


---

### Description

Decode a value using the provided encoding

### Usage

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

**Arguments**

|          |   |
|----------|---|
| encoding | Either 'hex' or 'base64'.   |
| ...      | These dots are for future extensions and must be empty.   |
| strict   | If TRUE (default), raise an error if the underlying value cannot be decoded. Otherwise, replace it with a null value. |

**Value**

A polars [expression](#)

**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)
)
```

---

|                 |   |
|-----------------|---|
| expr_str_encode | <i>Encode a value using the provided encoding</i> |
|-----------------|---|

---

**Description**

Encode a value using the provided encoding

**Usage**

```
expr_str_encode(encoding)
```

**Arguments**

|          |                           |
|----------|---------------------------|
| encoding | Either 'hex' or 'base64'. |
|----------|---------------------------|

**Value**

A polars [expression](#)

## 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)
)
```

---

expr\_str\_ends\_with      *Check if string ends with a regex*

---

## Description

Check if string values end with a substring.

## Usage

```
expr_str_ends_with(suffix)
```

## Arguments

suffix                  Suffix substring or Expr.

## Details

See also `$str$starts_with()` and `$str$contains()`.

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(fruits = c("apple", "mango", NA))
df$select(
  pl$col("fruits"),
  pl$col("fruits")$str$ends_with("go")$alias("has_suffix")
)
```

---

`expr_str_extract`      *Extract the target capture group from provided patterns*

---

### Description

Extract the target capture group from provided patterns

### Usage

```
expr_str_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

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(
  a = c(
    "http://vote.com/ballon_dor?candidate=messi&ref=polars",
    "http://vote.com/ballon_dor?candidat=jorginho&ref=polars",
    "http://vote.com/ballon_dor?candidate=ronaldo&ref=polars"
  )
)
df$with_columns(
  extracted = pl$col("a")$str$extract(pl$lit(r"(candidate=(\w+))"), 1)
)
```

---

`expr_str_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

```
expr_str_extract_all(pattern)
```

**Arguments**

pattern            A valid regex pattern

**Value**

A polars [expression](#)

**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")
)
```

---

expr\_str\_extract\_groups

*Extract all capture groups for the given regex pattern*

---

**Description**

Extract all capture groups for the given regex pattern

**Usage**

```
expr_str_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**

A polars [expression](#)

**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")

```

---

```
expr_str_extract_many
```

*Use the Aho-Corasick algorithm to extract matches*

---

## Description

Use the Aho-Corasick algorithm to extract matches

## Usage

```

expr_str_extract_many(
  patterns,
  ...,
  ascii_case_insensitive = FALSE,
  overlapping = FALSE
)

```

## Arguments

|                               |  |
|-------------------------------|--|
| <b>patterns</b>               | String patterns to search. This can be an Expr or something coercible to an Expr. Strings are parsed as column names.  |
| <b>...</b>                    | These dots are for future extensions and must be empty.  |
| <b>ascii_case_insensitive</b> | 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. |
| <b>overlapping</b>            | Whether matches can overlap.   |

## Value

A polars [expression](#)

## 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"))
```

---

|               |  |
|---------------|--|
| expr_str_find | <i>Return the index position of the first substring matching a pattern</i> |
|---------------|--|

---

## Description

Return the index position of the first substring matching a pattern

## Usage

```
expr_str_find(pattern, ..., literal = FALSE, strict = TRUE)
```

## Arguments

|                |  |
|----------------|--|
| <b>pattern</b> | 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> . |
| <b>...</b>     | These dots are for future extensions and must be empty.  |
| <b>literal</b> | Logical. If <b>TRUE</b> (default), treat <b>pattern</b> as a literal string, not as a regular expression.  |
| <b>strict</b>  | Logical. If <b>TRUE</b> (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

A polars [expression](#)

**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")
)
```

---

**expr\_str\_head**
*Return the first n characters of each string*


---

**Description**

Return the first n characters of each string

**Usage**

```
expr_str_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 **n**th 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**

A polars [expression](#)

**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")
)
```

---

|                            |   |
|----------------------------|---|
| <code>expr_str_join</code> | <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**

```
expr_str_join(delimiter = "", ..., ignore_nulls = TRUE)
```

**Arguments**

|                           |   |
|---------------------------|---|
| <code>delimiter</code>    | The delimiter to insert between consecutive string values.  |
| <code>...</code>          | These dots are for future extensions and must be empty.   |
| <code>ignore_nulls</code> | Ignore null values (default). If <code>FALSE</code> , null values will be propagated: if the column contains any null values, the output is null. |

**Value**

A polars [expression](#)

**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))
```

---

`expr_str_json_decode` *Parse string values as JSON.*

---

### Description

Parse string values as JSON.

### Usage

```
expr_str_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.

`...` These dots are for future extensions and must be empty.

`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

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(
  json_val = c('{\"a\":1, \"b\": true}', NA, '{\"a\":2, \"b\": false}')
)
dtype <- pl$Struct(a = pl$Int64, b = pl$Boolean)
df$select(pl$col("json_val")$str$json_decode(dtype))
```

---

`expr_str_json_path_match`

*Extract the first match of JSON string with the provided JSON-Path expression*

---

### Description

Extract the first match of JSON string with the provided JSONPath expression

**Usage**

```
expr_str_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**

A polars [expression](#)

**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"))
```

---

`expr_str_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**

```
expr_str_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**

A polars [expression](#)

## 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")  
)
```

---

expr\_str\_len\_chars *Get the number of characters in strings*

---

## Description

Get length of the strings as UInt32 (as number of characters). Use `$str$len_bytes()` to get the number of bytes.

## Usage

```
expr_str_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

A polars [expression](#)

## 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")  
)
```

expr\_str\_pad\_end      *Left justify strings*

---

### Description

Return the string left justified in a string of length `width`.

### Usage

```
expr_str_pad_end(length, fill_char = " ")
```

### Arguments

`length`            Justify left to this length.  
`fill_char`        Fill with this ASCII character.

### Details

Padding is done using the specified `fill_char`. The original string is returned if `length` is less than or equal to `len(s)`.

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = c("cow", "monkey", NA, "hippopotamus"))  
df$select(pl$col("a")$str$pad_end(8, "*"))
```

---

expr\_str\_pad\_start      *Right justify strings*

---

### Description

Return the string right justified in a string of length `length`.

### Usage

```
expr_str_pad_start(length, fill_char = " ")
```

### Arguments

`length`            Justify right to this length.  
`fill_char`        Fill with this ASCII character.



**Details**

Padding is done using the specified `fill_char`. The original string is returned if `length` is less than or equal to `len(s)`.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c("cow", "monkey", NA, "hippopotamus"))
df$select(pl$col("a")$str$pad_start(8, "*"))
```

---

|                               |   |
|-------------------------------|---|
| <code>expr_str_replace</code> | <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**

```
expr_str_replace(pattern, value, ..., literal = FALSE, n = 1L)
```

**Arguments**

|                      |   |
|----------------------|---|
| <code>pattern</code> | 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> .  |
| <code>value</code>   | A character or an <a href="#">Expr</a> of string that will replace the matched substring.   |
| <code>...</code>     | These dots are for future extensions and must be empty.   |
| <code>literal</code> | Logical. If <code>TRUE</code> (default), treat <code>pattern</code> as a literal string, not as a regular expression.   |
| <code>n</code>       | A number of matches to replace. Note that regex replacement with <code>n &gt; 1</code> not yet supported, so raise an error if <code>n &gt; 1</code> and <code>pattern</code> 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**

A polars [expression](#)

## 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$DataFrame(
  city = rep("Philadelphia", 4),
  season = c("Spring", "Summer", "Autumn", "Winter"),
  weather = c("Rainy", "Sunny", "Cloudy", "Snowy")
)
df$with_columns(
  pl$col("weather")$str$replace("(?i)foggy|rainy|cloudy|snowy", "Sunny")
)
```

---

`expr_str_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

```
expr_str_replace_all(pattern, value, ..., literal = FALSE)
```

**Arguments**

|                      |  |
|----------------------|--|
| <code>pattern</code> | 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> . |
| <code>value</code>   | A character or an <a href="#">Expr</a> of string that will replace the matched substring.  |
| <code>...</code>     | These dots are for future extensions and must be empty.  |
| <code>literal</code> | Logical. If TRUE (default), treat <code>pattern</code> 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**

A polars [expression](#)

**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$DataFrame(
  city = rep("Philadelphia", 4),
  season = c("Spring", "Summer", "Autumn", "Winter"),
  weather = c("Rainy", "Sunny", "Cloudy", "Snowy")
)
df$with_columns(
  pl$col("weather")$str$replace_all(
```

```

    "(?i)foggy|rainy|cloudy|snowy", "Sunny"
  )
)

```

---

```
expr_str_replace_many
```

*Use the Aho-Corasick algorithm to replace many matches*

---

## Description

This function replaces several matches at once.

## Usage

```
expr_str_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

A polars [expression](#)

## 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"))
)
```

---

`expr_str_reverse` *Returns string values in reversed order*

---

### Description

Returns string values in reversed order

### Usage

```
expr_str_reverse()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(text = c("foo", "bar", NA))
df$with_columns(reversed = pl$col("text")$str$reverse())
```

---

`expr_str_slice` *Create subslices of the string values of a String Series*

---

### Description

Create subslices of the string values of a String Series

### Usage

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

### Arguments

|                     |  |
|---------------------|--|
| <code>offset</code> | Start index. Negative indexing is supported.   |
| <code>length</code> | Length of the slice. If NULL (default), the slice is taken to the end of the string. |

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(s = c("pear", NA, "papaya", "dragonfruit"))
df$with_columns(
  pl$col("s")$str$slice(-3)$alias("s_sliced")
)
```

---

|                |  |
|----------------|--|
| expr_str_split | <i>Split the string by a substring</i> |
|----------------|--|

---

**Description**

Split the string by a substring

**Usage**

```
expr_str_split(by, ..., inclusive = FALSE)
```

**Arguments**

|           |   |
|-----------|---|
| by        | Substring to split by. Can be an Expr.                      |
| ...       | These dots are for future extensions and must be empty.     |
| inclusive | If TRUE, include the split character/string in the results. |

**Value**

A polars [expression](#)

**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(split = pl$col("s")$str$split(by = pl$col("by")))
```

---

|                 |   |
|-----------------|---|
| expr_str_splitn | <i>Split the string by a substring, restricted to returning at most n items</i> |
|-----------------|---|

---

**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**

```
expr_str_splitn(by, n)
```

**Arguments**

**by** Substring to split by. Can be an Expr.  
**n** Number of splits to make.

**Value**

A polars [expression](#)

**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)
)
```

---

`expr_str_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**

```
expr_str_split_exact(by, n, ..., inclusive = FALSE)
```

**Arguments**

**by** Substring to split by. Can be an Expr.  
**n** Number of splits to make.  
**...** These dots are for future extensions and must be empty.  
**inclusive** If TRUE, include the split character/string in the results.

**Value**

A polars [expression](#)

**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)
)
```

---

`expr_str_starts_with` *Check if string starts with a regex*

---

### Description

Check if string values starts with a substring.

### Usage

```
expr_str_starts_with(prefix)
```

### Arguments

`prefix` Prefix substring or Expr.

### Details

See also `$str$contains()` and `$str$ends_with()`.

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(fruits = c("apple", "mango", NA))
df$select(
  pl$col("fruits"),
  pl$col("fruits")$str$starts_with("app")$alias("has_suffix")
)
```

---

`expr_str_strip_chars` *Strip leading and trailing characters*

---

### Description

Remove leading and trailing characters.

### Usage

```
expr_str_strip_chars(characters = NULL)
```

### Arguments

`characters` 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**

A polars [expression](#)

**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"))
```

---

expr\_str\_strip\_chars\_end

*Strip trailing characters*

---

**Description**

Remove trailing characters.

**Usage**

```
expr_str_strip_chars_end(characters = NULL)
```

**Arguments**

**characters** 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**

A polars [expression](#)

**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 "))
```

---

```
expr_str_strip_chars_start
    Strip leading characters
```

---

### Description

Remove leading characters.

### Usage

```
expr_str_strip_chars_start(characters = NULL)
```

### Arguments

**characters** 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

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(foo = c(" hello", "\tworld"))
df$select(pl$col("foo")$str$strip_chars_start(" hel rld"))
```

---

```
expr_str_strip_prefix
    Strip prefix
```

---

### Description

The prefix will be removed from the string exactly once, if found.

### Usage

```
expr_str_strip_prefix(prefix = NULL)
```

### Arguments

**prefix** The prefix to be removed.

**Details**

This method strips the exact character sequence provided in `prefix` from the start of the input. To strip a set of characters in any order, use `$strip_chars_start()` instead.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c("foobar", "foofoobar", "foo", "bar"))
df$with_columns(
  stripped = pl$col("a")$str$strip_prefix("foo")
)
```

---

```
expr_str_strip_suffix
      Strip suffix
```

---

**Description**

The suffix will be removed from the string exactly once, if found.

**Usage**

```
expr_str_strip_suffix(suffix = NULL)
```

**Arguments**

`suffix`            The suffix to be removed.

**Details**

This method strips the exact character sequence provided in `suffix` from the end of the input. To strip a set of characters in any order, use `$strip_chars_end()` instead.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c("foobar", "foobarbar", "foo", "bar"))
df$with_columns(
  stripped = pl$col("a")$str$strip_suffix("bar")
)
```

---

`expr_str_strptime`      *Convert a String column into a Date/Datetime/Time column.*

---

## Description

Similar to the `strptime()` function.

## Usage

```
expr_str_strptime(
  dtype,
  format = NULL,
  ...,
  strict = TRUE,
  exact = TRUE,
  cache = TRUE,
  ambiguous = c("raise", "earliest", "latest", "null")
)
```

## 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>       | These dots are for future extensions and must be empty.  |
| <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 <code>null</code> .   |
| <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 <code>Time</code> 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. Character vector or <a href="#">expression</a> containing the followings: <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

A polars [expression](#)

## See Also

- `<Expr>$str$to_date()`
- `<Expr>$str$to_datetime()`
- `<Expr>$str$to_time()`

## Examples

```
# Dealing with a consistent format
df <- pl$DataFrame(x = c("2020-01-01 01:00Z", "2020-01-01 02:00Z"))

df$select(pl$col("x")$str$strptime(pl$Datetime(), "%Y-%m-%d %H:%M%#z"))

# Auto infer format
df$select(pl$col("x")$str$strptime(pl$Datetime()))

# Datetime with timezone is interpreted as UTC timezone
df <- pl$DataFrame(x = c("2020-01-01T01:00:00+09:00"))
df$select(pl$col("x")$str$strptime(pl$Datetime()))

# Dealing with different formats.
df <- pl$DataFrame(
  date = c(
    "2021-04-22",
    "2022-01-04 00:00:00",
    "01/31/22",
    "Sun Jul 8 00:34:60 2001"
  )
)

df$select(
  pl$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
df <- pl$DataFrame(
  x = c(
```

```

      "2023-01-01 11:22:33 -0100",
      "2023-01-01 11:22:33 +0300",
      "invalid time"
    )
  )

df$select(pl$col("x")$str$strptime(
  pl$Datetime("ns"),
  format = "%Y-%m-%d %H:%M:%S %z",
  strict = FALSE
))

```

---

`expr_str_tail`      *Return the last  $n$  characters of each string*

---

## Description

Return the last  $n$  characters of each string

## Usage

```
expr_str_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  $n$ th 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

A polars [expression](#)

## 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")
)
```

---

|                  |   |
|------------------|---|
| expr_str_to_date | <i>Convert a String column into a Date column</i> |
|------------------|---|

---

## Description

Convert a String column into a Date column

## Usage

```
expr_str_to_date(format = NULL, ..., strict = TRUE, exact = TRUE, cache = TRUE)
```

## Arguments

|               |   |
|---------------|---|
| <b>format</b> | 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. |
| <b>...</b>    | These dots are for future extensions and must be empty.   |
| <b>strict</b> | If TRUE (default), raise an error if a single string cannot be parsed. If FALSE, produce a polars null.   |
| <b>exact</b>  | 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 <b>exact = FALSE</b> introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.                   |
| <b>cache</b>  | Use a cache of unique, converted dates to apply the datetime conversion.  |

## Value

A polars [expression](#)

## See Also

- [<Expr>\\$str\\$strptime\(\)](#)

## Examples

```
df <- pl$DataFrame(x = c("2020/01/01", "2020/02/01", "2020/03/01"))

df$select(pl$col("x")$str$to_date())

# by default, this errors if some values cannot be converted
df <- pl$DataFrame(x = c("2020/01/01", "2020 02 01", "2020-03-01"))
try(df$select(pl$col("x")$str$to_date()))
df$select(pl$col("x")$str$to_date(strict = FALSE))
```

---

expr\_str\_to\_datetime *Convert a String column into a Datetime column*

---

## Description

Convert a String column into a Datetime column

## Usage

```
expr_str_to_datetime(
  format = NULL,
  ...,
  time_unit = NULL,
  time_zone = NULL,
  strict = TRUE,
  exact = TRUE,
  cache = TRUE,
  ambiguous = c("raise", "earliest", "latest", "null")
)
```

## Arguments

|                        |   |
|------------------------|---|
| <code>format</code>    | 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. |
| <code>...</code>       | These dots are for future extensions and must be empty.   |
| <code>time_unit</code> | 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).   |
| <code>time_zone</code> | for the resulting <a href="#">Datetime</a> column.  |
| <code>strict</code>    | If TRUE (default), raise an error if a single string cannot be parsed. If FALSE, produce a polars null.   |



|                        |  |
|------------------------|--|
| <code>exact</code>     | If TRUE (default), require an exact format match. If FALSE, allow the format to match anywhere in the target string. 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. Character vector or <a href="#">expression</a> containing the followings: <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> |

**Value**

A polars [expression](#)

**See Also**

- `<Expr>$str$strptime()`

**Examples**

```
df <- pl$DataFrame(x = c("2020-01-01 01:00Z", "2020-01-01 02:00Z"))

df$select(pl$col("x")$str$to_datetime("%Y-%m-%d %H:%M%#z"))
df$select(pl$col("x")$str$to_datetime(time_unit = "ms"))
```

---

`expr_str_to_decimal` *Convert a String column into a Decimal column*

---

**Description**

This method infers the needed parameters `precision` and `scale`.

**Usage**

```
expr_str_to_decimal(..., inference_length = 100)
```

**Arguments**

`...` These dots are for future extensions and must be empty.

`inference_length` Number of elements to parse to determine the `precision` and `scale`.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  numbers = c(
    "40.12", "3420.13", "120134.19", "3212.98",
    "12.90", "143.09", "143.9"
  )
)
df$with_columns(numbers_decimal = pl$col("numbers")$str$to_decimal())
```

---

`expr_str_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**

```
expr_str_to_integer(..., base = 10L, strict = TRUE)
```

**Arguments**

|                     |   |
|---------------------|---|
| <code>...</code>    | These dots are for future extensions and must be empty.   |
| <code>base</code>   | A positive integer or expression which is the base of the string we are parsing. Characters are parsed as column names. Default: 10L. |
| <code>strict</code> | A logical. If TRUE (default), parsing errors or integer overflow will raise an error. If FALSE, silently convert to null.             |

**Value**

A polars [expression](#)

**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)
)
```

---

```
expr_str_to_lowercase
```

*Convert a string to lowercase*

---

### Description

Transform to lowercase variant.

### Usage

```
expr_str_to_lowercase()
```

### Value

A polars [expression](#)

### Examples

```
pl$select(
  pl$lit(c("A", "b", "c", "1", NA))$str$to_lowercase()
)$to_series()
```

---

```
expr_str_to_time
```

*Convert a String column into a Time column*

---

### Description

Convert a String column into a Time column

### Usage

```
expr_str_to_time(format = NULL, ..., strict = TRUE, cache = TRUE)
```

### Arguments

|                     |   |
|---------------------|---|
| <code>format</code> | 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. |
| <code>...</code>    | These dots are for future extensions and must be empty.   |
| <code>strict</code> | If TRUE (default), raise an error if a single string cannot be parsed. If FALSE, produce a polars null.   |
| <code>cache</code>  | Use a cache of unique, converted dates to apply the datetime conversion.  |

**Value**

A polars [expression](#)

**See Also**

- `<Expr>$str$strptime()`

**Examples**

```
df <- pl$DataFrame(x = c("01:00", "02:00", "03:00"))
df$select(pl$col("x")$str$to_time("%H:%M"))
```

---

`expr_str_to_uppercase`

*Convert a string to uppercase*

---

**Description**

Transform to uppercase variant.

**Usage**

```
expr_str_to_uppercase()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$select(
  pl$lit(c("A", "b", "c", "1", NA))$str$to_uppercase()
)$to_series()
```

---

`expr_str_zfill`

*Fills the string with zeroes.*

---

**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**

```
expr_str_zfill(length)
```

## Arguments

**length** Pad the string until it reaches this length. Strings with length equal to or greater than this value are returned as-is. 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

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(a = c(-1L, 123L, 999999L, NA))
df$with_columns(zfill = pl$col("a")$cast(pl$String)$str$zfill(4))
```

---

expr\_\_abs

*Compute absolute values*

---

## Description

Compute absolute values

## Usage

```
expr__abs()
```

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(a = -1:2)
df$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` Element to add. Can be a string (only if `expr` is a string), a numeric value or an other expression.

## Value

A polars [expression](#)

## 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`      *Get the group indexes of the group by operation*

---

### Description

Should be used in aggregation context only.

### Usage

```
expr__agg_groups()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(  
  group = rep(c("one", "two"), each = 3),  
  value = c(94, 95, 96, 97, 97, 99)  
)  
  
df$group_by("group", maintain_order = TRUE)$agg(pl$col("value")$agg_groups())
```

---

`expr__alias`      *Rename the expression*

---

### Description

Rename the expression

### Usage

```
expr__alias(name)
```

### Arguments

`name`      The new name.

### Value

A polars [expression](#)

## Examples

```
# Rename an expression to avoid overwriting an existing column
df <- pl$DataFrame(a = 1:3, b = c("x", "y", "z"))
df$with_columns(
  pl$col("a") + 10,
  pl$col("b")$str$to_uppercase()$alias("c")
)

# Overwrite the default name of literal columns to prevent errors due to
# duplicate column names.
df$with_columns(
  pl$lit(TRUE)$alias("c"),
  pl$lit(4)$alias("d")
)
```

---

**expr\_\_all***Check if all boolean values in a column are true*

---

## Description

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

`...` These dots are for future extensions and must be empty.

`ignore_nulls` If `TRUE` (default), ignore null values. If `FALSE`, **Kleene logic** is used to deal with nulls: if the column contains any null values and no `TRUE` values, the output is null.

## Value

A polars [expression](#)

## 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** Element to add. Can be a string (only if **expr** is a string), a numeric value or an other expression.

### Value

A polars [expression](#)

### Examples

```
pl$lit(TRUE) & TRUE
pl$lit(TRUE)$and(pl$lit(TRUE))
```

---

 expr\_\_any

*Check if any boolean value in a column is true*


---

### Description

Check if any boolean value in a column is true

### Usage

```
expr__any(..., ignore_nulls = TRUE)
```

### Arguments

**...** These dots are for future extensions and must be empty.

**ignore\_nulls** If TRUE (default), ignore null values. If FALSE, **Kleene logic** is used to deal with nulls: if the column contains any null values and no TRUE values, the output is null.

**Value**

A polars [expression](#)

**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**

Append expressions

**Usage**

```
expr__append(other, ..., upcast = TRUE)
```

**Arguments**

|        |  |
|--------|--|
| other  | Expression to append.                                      |
| ...    | These dots are for future extensions and must be empty.    |
| upcast | If TRUE (default), cast both Series to the same supertype. |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 8:10, b = c(NA, 4, 4))
df$select(pl$all()$head(1)$append(pl$all()$tail(1)))
```

---

`expr__approx_n_unique`*Approximate count of unique values*

---

### Description

This is done using the HyperLogLog++ algorithm for cardinality estimation.

### Usage

```
expr__approx_n_unique()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(n = c(1, 1, 2))
df$select(pl$col("n")$approx_n_unique())

df <- pl$DataFrame(n = 0:1000)
df$select(
  exact = pl$col("n")$n_unique(),
  approx = pl$col("n")$approx_n_unique()
)
```

---

`expr__arccos`*Compute inverse cosine*

---

### Description

Compute inverse cosine

### Usage

```
expr__arccos()
```

### Value

A polars [expression](#)

### Examples

```
pl$DataFrame(a = c(-1, cos(0.5), 0, 1, NA))$
  with_columns(arccos = pl$col("a")$arccos())
```

---

|               |  |
|---------------|--|
| expr__arccosh | <i>Compute inverse hyperbolic cosine</i> |
|---------------|--|

---

**Description**

Compute inverse hyperbolic cosine

**Usage**

```
expr__arccosh()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(-1, cosh(0.5), 0, 1, NA))$  
  with_columns(arccosh = pl$col("a")$arccosh())
```

---

|              |                             |
|--------------|-----------------------------|
| expr__arcsin | <i>Compute inverse sine</i> |
|--------------|-----------------------------|

---

**Description**

Compute inverse sine

**Usage**

```
expr__arcsin()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(-1, sin(0.5), 0, 1, NA))$  
  with_columns(arcsin = pl$col("a")$arcsin())
```

---

|               |  |
|---------------|--|
| expr__arcsinh | <i>Compute inverse hyperbolic sine</i> |
|---------------|--|

---

**Description**

Compute inverse hyperbolic sine

**Usage**

```
expr__arcsinh()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(-1, sinh(0.5), 0, 1, NA))$  
  with_columns(arcsinh = pl$col("a")$arcsinh())
```

---

|              |                                |
|--------------|--------------------------------|
| expr__arctan | <i>Compute inverse tangent</i> |
|--------------|--------------------------------|

---

**Description**

Compute inverse tangent

**Usage**

```
expr__arctan()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(-1, tan(0.5), 0, 1, NA_real_))$  
  with_columns(arctan = pl$col("a")$arctan())
```

---

`expr__arctanh`      *Compute inverse hyperbolic tangent*

---

### Description

Compute inverse hyperbolic tangent

### Usage

```
expr__arctanh()
```

### Value

A polars [expression](#)

### Examples

```
pl$DataFrame(a = c(-1, tanh(0.5), 0, 1, NA))$  
  with_columns(arctanh = pl$col("a")$arctanh())
```

---

`expr__arg_max`      *Get the index of the maximal value*

---

### Description

Get the index of the maximal value

### Usage

```
expr__arg_max()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = c(20, 10, 30))  
df$select(pl$col("a")$arg_max())
```

---

|               |   |
|---------------|---|
| expr__arg_min | <i>Get the index of the minimal value</i> |
|---------------|---|

---

**Description**

Get the index of the minimal value

**Usage**

```
expr__arg_min()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(20, 10, 30))
df$select(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**

|            |   |
|------------|---|
| ...        | These dots are for future extensions and must be empty. |
| descending | Sort in descending order.                               |
| nulls_last | Place null values last.                                 |

**Value**

A polars [expression](#)

**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())
```

---

|                             |  |
|-----------------------------|--|
| <code>expr__arg_true</code> | <i>Return indices where expression is true</i> |
|-----------------------------|--|

---

**Description**

Return indices where expression is true

**Usage**

```
expr__arg_true()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, 1, 2, 1))
df$select((pl$col("a") == 1)$arg_true())
```

---

|                               |  |
|-------------------------------|--|
| <code>expr__arg_unique</code> | <i>Get the index of the first unique value</i> |
|-------------------------------|--|

---

**Description**

Get the index of the first unique value

**Usage**

```
expr__arg_unique()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:3, b = c(NA, 4, 4))
df$select(pl$col("a")$arg_unique())
df$select(pl$col("b")$arg_unique())
```



---

`expr__backward_fill` *Fill missing values with the next non-null value*

---

### Description

Fill missing values with the next non-null value

### Usage

```
expr__backward_fill(limit = NULL)
```

### Arguments

`limit`            The number of consecutive null values to backward fill.

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(  
  a = c(1, 2, NA),  
  b = c(4, NA, 6),  
  c = c(NA, NA, 2)  
)  
df$select(pl$all().backward_fill())  
df$select(pl$all().backward_fill(limit = 1))
```

---

`expr__bitwise_and` *Perform an aggregation of bitwise ANDs.*

---

### Description

Perform an aggregation of bitwise ANDs.

### Usage

```
expr__bitwise_and()
```

### Value

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(n = -1:1)
df$select(pl$col("n")$bitwise_and())

df <- pl$DataFrame(
  grouper = c("a", "a", "a", "b", "b"),
  n = c(-1L, 0L, 1L, -1L, 1L)
)
df$group_by("grouper", .maintain_order = TRUE)$agg(pl$col("n")$bitwise_and())
```

---

```
expr__bitwise_count_ones
```

*Evaluate the number of set bits.*

---

**Description**

Evaluate the number of set bits.

**Usage**

```
expr__bitwise_count_ones()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(n = c(-1L, 0L, 2L, 1L))
df$with_columns(set_bits = pl$col("n")$bitwise_count_ones())
```

---

```
expr__bitwise_count_zeros
```

*Evaluate the number of unset bits.*

---

**Description**

Evaluate the number of unset bits.

**Usage**

```
expr__bitwise_count_zeros()
```

**Value**

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(n = c(-1L, 0L, 2L, 1L))
df$with_columns(unset_bits = pl$col("n")$bitwise_count_zeros())
```

---

expr\_\_bitwise\_leading\_ones

*Evaluate the number most-significant set bits before seeing an unset bit.*

---

### Description

Evaluate the number most-significant set bits before seeing an unset bit.

### Usage

```
expr__bitwise_leading_ones()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(n = c(-1L, 0L, 2L, 1L))
df$with_columns(leading_ones = pl$col("n")$bitwise_leading_ones())
```

---

expr\_\_bitwise\_leading\_zeros

*Evaluate the number most-significant unset bits before seeing a set bit.*

---

### Description

Evaluate the number most-significant unset bits before seeing a set bit.

### Usage

```
expr__bitwise_leading_zeros()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(n = c(-1L, 0L, 2L, 1L))
df$with_columns(leading_zeros = pl$col("n")$bitwise_leading_zeros())
```

---

`expr__bitwise_or`      *Perform an aggregation of bitwise ORs.*

---

### Description

Perform an aggregation of bitwise ORs.

### Usage

```
expr__bitwise_or()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(n = -1:1)
df$select(pl$col("n")$bitwise_or())

df <- pl$DataFrame(
  grouper = c("a", "a", "a", "b", "b"),
  n = c(-1L, 0L, 1L, -1L, 1L)
)
df$group_by("grouper", .maintain_order = TRUE)$agg(pl$col("n")$bitwise_or())
```

---

`expr__bitwise_trailing_ones`  
*Evaluate the number least-significant set bits before seeing an unset bit.*

---

### Description

Evaluate the number least-significant set bits before seeing an unset bit.

### Usage

```
expr__bitwise_trailing_ones()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(n = c(-1L, 0L, 2L, 1L))
df$with_columns(trailing_ones = pl$col("n")$bitwise_trailing_ones())
```

---

```
expr__bitwise_trailing_zeros
```

*Evaluate the number least-significant unset bits before seeing a set bit.*

---

### Description

Evaluate the number least-significant unset bits before seeing a set bit.

### Usage

```
expr__bitwise_trailing_zeros()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(n = c(-1L, 0L, 2L, 1L))
df$with_columns(trailing_zeros = pl$col("n")$bitwise_trailing_zeros())
```

---

```
expr__bitwise_xor
```

*Perform an aggregation of bitwise XORs.*

---

### Description

Perform an aggregation of bitwise XORs.

### Usage

```
expr__bitwise_xor()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(n = -1:1)
df$select(pl$col("n")$bitwise_xor())

df <- pl$DataFrame(
  grouper = c("a", "a", "a", "b", "b"),
  n = c(-1L, 0L, 1L, -1L, 1L)
)
df$group_by("grouper", .maintain_order = TRUE)$agg(pl$col("n")$bitwise_xor())
```

---

`expr__bottom_k`      *Return the k smallest elements*

---

### Description

Non-null elements are always preferred over null elements. The output is not guaranteed to be in any particular order, call `$sort()` after this function if you wish the output to be sorted. This has time complexity  $O(n)$ .

### Usage

```
expr__bottom_k(k = 5)
```

### Arguments

`k`                      Number of elements to return.

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(value = c(1, 98, 2, 3, 99, 4))
df$select(
  top_k = pl$col("value")$top_k(k = 3),
  bottom_k = pl$col("value")$bottom_k(k = 3)
)
```

---

`expr__bottom_k_by`      *Return the elements corresponding to the k smallest elements of the by column(s)*

---

### Description

Non-null elements are always preferred over null elements. The output is not guaranteed to be in any particular order, call `$sort()` after this function if you wish the output to be sorted. This has time complexity  $O(n)$ .

### Usage

```
expr__bottom_k_by(by, k = 5, ..., reverse = FALSE)
```

**Arguments**

|                      |   |
|----------------------|---|
| <code>by</code>      | Column(s) used to determine the smallest elements. Accepts expression input. Strings are parsed as column names.  |
| <code>k</code>       | Number of elements to return.   |
| <code>...</code>     | These dots are for future extensions and must be empty.   |
| <code>reverse</code> | Consider the <code>k</code> largest elements of the <code>by</code> column(s) (instead of the <code>k</code> smallest). This can be specified per column by passing a sequence of booleans. |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = 1:6,
  b = 6:1,
  c = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")
)

# Get the bottom 2 rows by column a or b:
df$select(
  pl$all()$bottom_k_by("a", 2)$name$suffix("_btm_by_a"),
  pl$all()$bottom_k_by("b", 2)$name$suffix("_btm_by_b")
)

# Get the bottom 2 rows by multiple columns with given order.
df$select(
  pl$all()$
  bottom_k_by(c("c", "a"), 2, reverse = c(FALSE, TRUE))$
  name$suffix("_btm_by_ca"),
  pl$all()$
  bottom_k_by(c("c", "b"), 2, reverse = c(FALSE, TRUE))$
  name$suffix("_btm_by_cb"),
)

# Get the bottom 2 rows by column a in each group
df$group_by("c", .maintain_order = TRUE)$agg(
  pl$all()$bottom_k_by("a", 2)
)$explode(pl$all()$exclude("c"))
```

---

`expr__cast`

*Cast between DataType*

---

**Description**

Cast between DataType

**Usage**

```
expr__cast(dtype, ..., strict = TRUE, wrap_numerical = FALSE)
```

**Arguments**

**dtype**            DataType to cast to.

**...**            These dots are for future extensions and must be empty.

**strict**           If TRUE (default), an error will be thrown if cast failed at resolve time.

**wrap\_numerical**   If TRUE, numeric casts wrap overflowing values instead of marking the cast as invalid.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:3, b = c(1, 2, 3))
df$with_columns(
  pl$col("a")$cast(pl$Float64),
  pl$col("b")$cast(pl$Int32)
)

# strict FALSE, inserts null for any cast failure
pl$select(
  pl$lit(c(100, 200, 300))$cast(pl$UInt8, strict = FALSE)
)$to_series()

# strict TRUE, raise any failure as an error when query is executed.
tryCatch(
  {
    pl$select(
      pl$lit("a")$cast(pl$Float64, strict = TRUE)
    )$to_series()
  },
  error = function(e) e
)
```

---

```
expr__cbrt
```

```
  Compute cube root
```

---

**Description**

Compute cube root

**Usage**

```
expr__cbrt()
```



**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(1, 2, 4))$
  with_columns(cbirt = pl$col("a")$cbirt())
```

---

expr\_\_ceil

*Rounds up to the nearest integer value*

---

**Description**

This only works on floating point Series.

**Usage**

```
expr__ceil()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(0.3, 0.5, 1.0, 1.1))
df$with_columns(
  ceil = pl$col("a")$ceil()
)
```

---

expr\_\_clip

*Set values outside the given boundaries to the boundary value*

---

**Description**

This method only works for numeric and temporal columns. To clip other data types, consider writing a when-then-otherwise expression.

**Usage**

```
expr__clip(lower_bound = NULL, upper_bound = NULL)
```

**Arguments**

**lower\_bound** Lower bound. Accepts expression input. Non-expression inputs are parsed as literals.

**upper\_bound** Upper bound. Accepts expression input. Non-expression inputs are parsed as literals.

## Details

This method only works for numeric and temporal columns. To clip other data types, consider writing a when-then-otherwise expression.

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(a = c(-50, 5, 50, NA))

# Specifying both a lower and upper bound:
df$with_columns(
  clip = pl$col("a")$clip(1, 10)
)

# Specifying only a single bound:
df$with_columns(
  clip = pl$col("a")$clip(upper_bound = 10)
)
```

---

expr\_\_cos

*Compute cosine*

---

## Description

Compute cosine

## Usage

```
expr__cos()
```

## Value

A polars [expression](#)

## Examples

```
pl$DataFrame(a = c(0, pi / 2, pi, NA))$
  with_columns(cosine = pl$col("a")$cos())
```

---

|            |                                  |
|------------|----------------------------------|
| expr__cosh | <i>Compute hyperbolic cosine</i> |
|------------|----------------------------------|

---

**Description**

Compute hyperbolic cosine

**Usage**

```
expr__cosh()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(-1, acosh(2), 0, 1, NA))$  
  with_columns(cosh = pl$col("a")$cosh())
```

---

|           |                          |
|-----------|--------------------------|
| expr__cot | <i>Compute cotangent</i> |
|-----------|--------------------------|

---

**Description**

Compute cotangent

**Usage**

```
expr__cot()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(0, pi / 2, -5, NA))$  
  with_columns(cotangent = pl$col("a")$cot())
```

---

|                          |  |
|--------------------------|--|
| <code>expr__count</code> | <i>Get the number of non-null elements in the column</i> |
|--------------------------|--|

---

**Description**

Get the number of non-null elements in the column

**Usage**

```
expr__count()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:3, b = c(NA, 4, 4))
df$select(pl$all())$count()
```

---

|                                    |   |
|------------------------------------|---|
| <code>expr__cumulative_eval</code> | <i>Return the cumulative count of the non-null values in the column</i> |
|------------------------------------|---|

---

**Description**

Return the cumulative count of the non-null values in the column

**Usage**

```
expr__cumulative_eval(expr, ..., min_periods = 1, parallel = FALSE)
```

**Arguments**

|                          |   |
|--------------------------|---|
| <code>expr</code>        | Expression to evaluate.   |
| <code>...</code>         | These dots are for future extensions and must be empty.   |
| <code>min_periods</code> | Number of valid values (i.e. <code>length - null_count</code> ) there should be in the window before the expression is evaluated. |
| <code>parallel</code>    | Run in parallel. Don't do this in a group by 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**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = 1:5)
df$with_columns(
  pl$col("values")$cumulative_eval(
    pl$element()$first() - pl$element()$last()**2
  )
)
```

---

|                              |   |
|------------------------------|---|
| <code>expr__cum_count</code> | <i>Return the cumulative count of the non-null values in the column</i> |
|------------------------------|---|

---

**Description**

Return the cumulative count of the non-null values in the column

**Usage**

```
expr__cum_count(..., reverse = FALSE)
```

**Arguments**

`...` These dots are for future extensions and must be empty.

`reverse` If TRUE, reverse the count.

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = 1:4)$with_columns(
  cum_count = pl$col("a")$cum_count(),
  cum_count_reversed = pl$col("a")$cum_count(reverse = TRUE)
)
```

---

`expr__cum_max`      *Return the cumulative max computed at every element.*

---

### Description

Return the cumulative max computed at every element.

### Usage

```
expr__cum_max(..., reverse = FALSE)
```

### Arguments

`...`      These dots are for future extensions and must be empty.  
`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

A polars [expression](#)

### Examples

```
pl$DataFrame(a = c(1:4, 2L))$with_columns(
  cum_max = pl$col("a")$cum_max(),
  cum_max_reversed = pl$col("a")$cum_max(reverse = TRUE)
)
```

---

`expr__cum_min`      *Return the cumulative min computed at every element.*

---

### Description

Return the cumulative min computed at every element.

### Usage

```
expr__cum_min(..., reverse = FALSE)
```

### Arguments

`...`      These dots are for future extensions and must be empty.  
`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**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(1:4, 2L))$with_columns(
  cum_min = pl$col("a")$cum_min(),
  cum_min_reversed = pl$col("a")$cum_min(reverse = TRUE)
)
```

---

|                |   |
|----------------|---|
| expr__cum_prod | <i>Return the cumulative product computed at every element.</i> |
|----------------|---|

---

**Description**

Return the cumulative product computed at every element.

**Usage**

```
expr__cum_prod(..., reverse = FALSE)
```

**Arguments**

|         |   |
|---------|---|
| ...     | These dots are for future extensions and must be empty.                           |
| 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**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = 1:4)$with_columns(
  cum_prod = pl$col("a")$cum_prod(),
  cum_prod_reversed = pl$col("a")$cum_prod(reverse = TRUE)
)
```

---

`expr__cum_sum`      *Return the cumulative sum computed at every element.*

---

### Description

Return the cumulative sum computed at every element.

### Usage

```
expr__cum_sum(..., reverse = FALSE)
```

### Arguments

`...`      These dots are for future extensions and must be empty.

`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

A polars [expression](#)

### Examples

```
pl$DataFrame(a = 1:4)$with_columns(
  cum_sum = pl$col("a")$cum_sum(),
  cum_sum_reversed = pl$col("a")$cum_sum(reverse = TRUE)
)
```

---

`expr__cut`      *Bin continuous values into discrete categories*

---

### Description

[Experimental]

### Usage

```
expr__cut(
  breaks,
  ...,
  labels = NULL,
  left_closed = FALSE,
  include_breaks = FALSE
)
```



**Arguments**

|                             |   |
|-----------------------------|---|
| <code>breaks</code>         | List of unique cut points.  |
| <code>...</code>            | These dots are for future extensions and must be empty.   |
| <code>labels</code>         | Names of the categories. The number of labels must be equal to the number of cut points plus one.   |
| <code>left_closed</code>    | Set the intervals to be left-closed instead of right-closed.  |
| <code>include_breaks</code> | 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**

A polars [expression](#)

**Examples**

```
# Divide a column into three categories.
df <- pl$DataFrame(foo = -2:2)
df$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")
```

---

|                            |  |
|----------------------------|--|
| <code>expr__degrees</code> | <i>Convert from radians to degrees</i> |
|----------------------------|--|

---

**Description**

Convert from radians to degrees

**Usage**

```
expr__degrees()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(1, 2, 4) * pi)$
  with_columns(degrees = pl$col("a")$degrees())
```

---

|                         |  |
|-------------------------|--|
| <code>expr__diff</code> | <i>Calculate the n-th discrete difference between elements</i> |
|-------------------------|--|

---

### Description

Calculate the n-th discrete difference between elements

### Usage

```
expr__diff(n = 1, null_behavior = c("ignore", "drop"))
```

### Arguments

`n` Integer indicating the number of slots to shift.  
`null_behavior` How to handle null values. Must be "ignore" (default), or "drop".

### Value

A polars [expression](#)

### Examples

```
pl$DataFrame(a = c(20, 10, 30, 25, 35))$with_columns(  
  diff_default = pl$col("a")$diff(),  
  diff_2_ignore = pl$col("a")$diff(2, "ignore")  
)
```

---

|                        |  |
|------------------------|--|
| <code>expr__dot</code> | <i>Compute the dot/inner product between two Expressions</i> |
|------------------------|--|

---

### Description

Compute the dot/inner product between two Expressions

### Usage

```
expr__dot(other)
```

### Arguments

`other` Expression to compute dot product with.

### Value

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, 3, 5), b = c(2, 4, 6))
df$select(pl$col("a")$dot(pl$col("b")))
```

---

|                 |   |
|-----------------|---|
| expr__drop_nans | <i>Drop all floating point NaN values</i> |
|-----------------|---|

---

**Description**

The original order of the remaining elements is preserved. A NaN value is not the same as a null value. To drop null values, use [\\$drop\\_nulls\(\)](#).

**Usage**

```
expr__drop_nans()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, NA, 3, NaN))
df$select(pl$col("a")$drop_nans())
```

---

|                  |  |
|------------------|--|
| expr__drop_nulls | <i>Drop all floating point null values</i> |
|------------------|--|

---

**Description**

The original order of the remaining elements is preserved. A null value is not the same as a NaN value. To drop NaN values, use [\\$drop\\_nans\(\)](#).

**Usage**

```
expr__drop_nulls()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, NA, 3, NaN))
df$select(pl$col("a")$drop_nulls())
```

---

|                            |                        |
|----------------------------|------------------------|
| <code>expr__entropy</code> | <i>Compute entropy</i> |
|----------------------------|------------------------|

---

**Description**

Uses the formula  $-\sum(pk * \log(pk))$  where `pk` are discrete probabilities.

**Usage**

```
expr__entropy(base = exp(1), ..., normalize = TRUE)
```

**Arguments**

|                        |   |
|------------------------|---|
| <code>base</code>      | Numeric value used as base, defaults to <code>exp(1)</code> . |
| <code>...</code>       | These dots are for future extensions and must be empty.       |
| <code>normalize</code> | Normalize <code>pk</code> if it doesn't sum to 1.             |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a").entropy(base = 2))
df$select(pl$col("a").entropy(base = 2, normalize = FALSE))
```

---

|                       |                       |
|-----------------------|-----------------------|
| <code>expr__eq</code> | <i>Check equality</i> |
|-----------------------|-----------------------|

---

**Description**

This propagates null values, i.e. any comparison involving `null` will return `null`. Use [\\$eq\\_missing\(\)](#) to consider null values as equal.

**Usage**

```
expr__eq(other)
```

**Arguments**

|                    |  |
|--------------------|--|
| <code>other</code> | A literal or expression value to compare with. |
|--------------------|--|

**Value**

A polars [expression](#)

**See Also**[expr\\_\\_eq\\_missing](#)**Examples**

```
df <- pl$DataFrame(x = c(NA, FALSE, TRUE), y = c(TRUE, TRUE, TRUE))
df$with_columns(
  eq = pl$col("x")$eq(pl$col("y")),
  eq_missing = pl$col("x")$eq_missing(pl$col("y"))
)
```

---

|                               |  |
|-------------------------------|--|
| <code>expr__eq_missing</code> | <i>Check equality without null propagation</i> |
|-------------------------------|--|

---

**Description**

This considers that null values are equal. It differs from `$eq()` where null values are propagated.

**Usage**

```
expr__eq_missing(other)
```

**Arguments**

`other` A literal or expression value to compare with.

**Value**

A polars [expression](#)

**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      *Compute exponentially-weighted moving mean*

---

### Description

Compute exponentially-weighted moving mean

### Usage

```
expr__ewm__mean(
    ...,
    com,
    span,
    half_life,
    alpha,
    adjust = TRUE,
    min_periods = 1,
    ignore_nulls = FALSE
)
```

### Arguments

...      These dots are for future extensions and must be empty.

com      Specify decay in terms of center of mass,  $\gamma$ , with

$$\alpha = \frac{1}{1 + \gamma} \quad \forall \gamma \geq 0$$

span      Specify decay in terms of span,  $\theta$ , with

$$\alpha = \frac{2}{\theta + 1} \quad \forall \theta \geq 1$$

half\_life      Specify decay in terms of half-life,  $\lambda$ , with

$$\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \quad \forall \lambda > 0$$

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:

- when **TRUE** (default), the EW function is calculated using weights  $w_i = (1 - \alpha)^i$ ;
- when **FALSE**, the EW function is calculated recursively by

$$y_0 = x_0$$

$$y_t = (1 - \alpha)y_{t-1} + \alpha x_t$$

- min\_periods** The number of values in the window that should be non-null before computing a result. If NULL (default), it will be set equal to `window_size`.
- ignore\_nulls** Ignore missing values when calculating weights.
- when `FALSE` (default), weights are based on absolute positions. For example, the weights of  $x_0$  and  $x_2$  used in calculating the final weighted average of  $(x_0, \text{null}, x_2)$  are  $(1 - \alpha)^2$  and 1 if `adjust = TRUE`, and  $(1 - \alpha)^2$  and  $\alpha$  if `adjust = FALSE`.
  - when `TRUE`, weights are based on relative positions. For example, the weights of  $x_0$  and  $x_2$  used in calculating the final weighted average of  $(x_0, \text{null}, x_2)$  are  $1 - \alpha$  and 1 if `adjust = TRUE`, and  $1 - \alpha$  and  $\alpha$  if `adjust = FALSE`.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a").ewm_mean(com = 1, ignore_nulls = FALSE))
```

---

`expr__ewm_mean_by`      *Compute time-based exponentially weighted moving average*

---

**Description**

Given observations  $x_0, x_1, \dots, x_{n-1}$  at times  $t_0, t_1, \dots, t_{n-1}$ , the EWMA is calculated as

$$y_0 = x_0$$

$$\alpha_i = 1 - \exp\left\{-\frac{\ln(2)(t_i - t_{i-1})}{\tau}\right\}$$

$$y_i = \alpha_i x_i + (1 - \alpha_i) y_{i-1}; \quad i > 0$$

where  $\tau$  is the `half_life`.

**Usage**

```
expr__ewm_mean_by(by, ..., half_life)
```

**Arguments**

- by** Times to calculate average by. Should be `DateTime`, `Date`, `UInt64`, `UInt32`, `Int64`, or `Int32` data type.
- ...** These dots are for future extensions and must be empty.
- half\_life** Unit over which observation decays to half its value. Can be created either from a `timedelta`, or by using the following string language:

- 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".

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(
  values = c(0, 1, 2, NA, 4),
  times = as.Date(
    c("2020-01-01", "2020-01-03", "2020-01-10", "2020-01-15", "2020-01-17")
  )
)
df$with_columns(
  result = pl$col("values")$ewm_mean_by("times", half_life = "4d")
)
```

---

expr\_\_ewm\_std

*Compute exponentially-weighted moving standard deviation*

---

## Description

Compute exponentially-weighted moving standard deviation

## Usage

```
expr__ewm_std(
  ...,
  com,
  span,
```



```

    half_life,
    alpha,
    adjust = TRUE,
    bias = FALSE,
    min_periods = 1,
    ignore_nulls = FALSE
)

```

**Arguments**

... These dots are for future extensions and must be empty.  
 com Specify decay in terms of center of mass,  $\gamma$ , with

$$\alpha = \frac{1}{1 + \gamma} \quad \forall \gamma \geq 0$$

span Specify decay in terms of span,  $\theta$ , with

$$\alpha = \frac{2}{\theta + 1} \quad \forall \theta \geq 1$$

half\_life Specify decay in terms of half-life,  $\lambda$ , with

$$\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \quad \forall \lambda > 0$$

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:

- when TRUE (default), the EW function is calculated using weights  $w_i = (1 - \alpha)^i$ ;
- when FALSE, the EW function is calculated recursively by

$$y_0 = x_0$$

$$y_t = (1 - \alpha)y_{t-1} + \alpha x_t$$

bias If FALSE (default), apply a correction to make the estimate statistically unbiased.

min\_periods The number of values in the window that should be non-null before computing a result. If NULL (default), it will be set equal to `window_size`.

ignore\_nulls Ignore missing values when calculating weights.

- when FALSE (default), weights are based on absolute positions. For example, the weights of  $x_0$  and  $x_2$  used in calculating the final weighted average of  $(x_0, \text{null}, x_2)$  are  $(1 - \alpha)^2$  and 1 if `adjust = TRUE`, and  $(1 - \alpha)^2$  and  $\alpha$  if `adjust = FALSE`.
- when TRUE, weights are based on relative positions. For example, the weights of  $x_0$  and  $x_2$  used in calculating the final weighted average of  $(x_0, \text{null}, x_2)$  are  $1 - \alpha$  and 1 if `adjust = TRUE`, and  $1 - \alpha$  and  $\alpha$  if `adjust = FALSE`.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a")$ewm_std(com = 1, ignore_nulls = FALSE))
```

---

|                            |   |
|----------------------------|---|
| <code>expr__ewm_var</code> | <i>Compute exponentially-weighted moving variance</i> |
|----------------------------|---|

---

**Description**

Compute exponentially-weighted moving variance

**Usage**

```
expr__ewm_var(
  ...,
  com,
  span,
  half_life,
  alpha,
  adjust = TRUE,
  bias = FALSE,
  min_periods = 1,
  ignore_nulls = FALSE
)
```

**Arguments**

`...` These dots are for future extensions and must be empty.  
`com` Specify decay in terms of center of mass,  $\gamma$ , with

$$\alpha = \frac{1}{1 + \gamma} \quad \forall \gamma \geq 0$$

`span` Specify decay in terms of span,  $\theta$ , with

$$\alpha = \frac{2}{\theta + 1} \quad \forall \theta \geq 1$$

`half_life` Specify decay in terms of half-life,  $\lambda$ , with

$$\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \quad \forall \lambda > 0$$

`alpha` Specify smoothing factor alpha directly,  $0 < \alpha \leq 1$ .

|                     |  |
|---------------------|--|
| <b>adjust</b>       | <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 <b>TRUE</b> (default), the EW function is calculated using weights <math>w_i = (1 - \alpha)^i</math>;</li> <li>when <b>FALSE</b>, the EW function is calculated recursively by           <math display="block">y_0 = x_0</math> <math display="block">y_t = (1 - \alpha)y_{t-1} + \alpha x_t</math> </li> </ul>  |
| <b>bias</b>         | If <b>FALSE</b> (default), apply a correction to make the estimate statistically unbiased.   |
| <b>min_periods</b>  | The number of values in the window that should be non-null before computing a result. If <b>NULL</b> (default), it will be set equal to <b>window_size</b> .   |
| <b>ignore_nulls</b> | <p>Ignore missing values when calculating weights.</p> <ul style="list-style-type: none"> <li>when <b>FALSE</b> (default), 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{null}, x_2)</math> are <math>(1 - \alpha)^2</math> and 1 if <b>adjust = TRUE</b>, and <math>(1 - \alpha)^2</math> and <math>\alpha</math> if <b>adjust = FALSE</b>.</li> <li>when <b>TRUE</b>, 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{null}, x_2)</math> are <math>1 - \alpha</math> and 1 if <b>adjust = TRUE</b>, and <math>1 - \alpha</math> and <math>\alpha</math> if <b>adjust = FALSE</b>.</li> </ul> |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a")$ewm_var(com = 1, ignore_nulls = FALSE))
```

---

|                      |  |
|----------------------|--|
| <b>expr__exclude</b> | <i>Exclude columns from a multi-column expression.</i> |
|----------------------|--|

---

**Description**

Exclude columns from a multi-column expression.

**Usage**

```
expr__exclude(...)
```

**Arguments**

... The name or datatype of the column(s) to exclude. Accepts regular expression input. Regular expressions should start with  $\wedge$  and end with  $\$$ .

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(aa = 1:2, ba = c("a", NA), cc = c(NA, 2.5))
df

# Exclude by column name(s):
df$select(pl$all().$exclude("ba"))

# Exclude by regex, e.g. removing all columns whose names end with the
# letter "a":
df$select(pl$all().$exclude("^.*a$"))

# Exclude by dtype(s), e.g. removing all columns of type Int64 or Float64:
df$select(pl$all().$exclude(pl$Int64, pl$Float64))
```

---

expr\_\_exp

*Compute the exponential*

---

**Description**

Compute the exponential

**Usage**

```
expr__exp()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(1, 2, 4))$
  with_columns(exp = pl$col("a")$exp())
```

---

|               |                                  |
|---------------|----------------------------------|
| expr__explode | <i>Explode a list expression</i> |
|---------------|----------------------------------|

---

### Description

This means that every item is expanded to a new row.

### Usage

```
expr__explode()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(  
  groups = c("a", "b"),  
  values = list(1:2, 3:4)  
)  
  
df$select(pl$col("values")$explode())
```

---

|                       |   |
|-----------------------|---|
| expr__extend_constant | <i>Extend the Series with n copies of a value</i> |
|-----------------------|---|

---

### Description

Extend the Series with n copies of a value

### Usage

```
expr__extend_constant(value, n)
```

### Arguments

|       |   |
|-------|---|
| value | A constant literal value or a unit expression with which to extend the expression result Series. This can be NA to extend with nulls. |
| n     | The number of additional values that will be added.   |

### Value

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = 1:3)
df$select(pl$col("values")$extend_constant(99, n = 2))
```

---

`expr__fill_nan`      *Fill floating point NaN value with a fill value*

---

**Description**

Fill floating point NaN value with a fill value

**Usage**

```
expr__fill_nan(value)
```

**Arguments**

`value`      Value used to fill NaN values.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, NA, 2, NaN))
df$with_columns(
  filled_nan = pl$col("a")$fill_nan(99)
)
```

---

`expr__fill_null`      *Fill floating point null value with a fill value*

---

**Description**

Fill floating point null value with a fill value

**Usage**

```
expr__fill_null(value, strategy = NULL, limit = NULL)
```

**Arguments**

`value`      Value used to fill null values. Can be missing if `strategy` is specified. Accepts expression input, strings are parsed as column names.

`strategy`      Strategy used to fill null values. Must be one of "forward", "backward", "min", "max", "mean", "zero", "one".

`limit`      Number of consecutive null values to fill when using the "forward" or "backward" strategy.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, NA, 2, NaN))
df$with_columns(
  filled_null_zero = pl$col("a")$fill_null(strategy = "zero"),
  filled_null_99 = pl$col("a")$fill_null(99),
  filled_null_forward = pl$col("a")$fill_null(strategy = "forward"),
  filled_null_expr = pl$col("a")$fill_null(pl$col("a")$median())
)
```

---

**expr\_\_filter**
*Filter the expression based on one or more predicate expressions*


---

**Description**

Elements where the filter does not evaluate to TRUE are discarded, including nulls. This is mostly useful in an aggregation context. If you want to filter on a DataFrame level, use [DataFrame\\$filter\(\)](#) or [LazyFrame\\$filter\(\)](#).

**Usage**

```
expr__filter(...)
```

**Arguments**

... [<dynamic-dots>](#) Expression(s) that evaluate to a boolean Series.

**Value**

A polars [expression](#)

**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**

A polars [expression](#)

**Examples**

```
pl$DataFrame(x = 3:1)$with_columns(first = pl$col("x")$first())
```

---

|               |  |
|---------------|--|
| expr__flatten | <i>Flatten a list or string column</i> |
|---------------|--|

---

**Description**

This is an alias for [\\$explode\(\)](#).

**Usage**

```
expr__flatten()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  group = c("a", "b", "b"),
  values = list(1:2, 2:3, 4)
)

df$group_by("group")$agg(pl$col("values")$flatten())
```



---

|             |   |
|-------------|---|
| expr__floor | <i>Rounds down to the nearest integer value</i> |
|-------------|---|

---

### Description

This only works on floating point Series.

### Usage

```
expr__floor()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = c(0.3, 0.5, 1.0, 1.1))
df$with_columns(
  floor = pl$col("a")$floor()
)
```

---

|                 |   |
|-----------------|---|
| expr__floor_div | <i>Floor divide using two expressions</i> |
|-----------------|---|

---

### Description

Method equivalent of floor division operator `expr %/% other`. `$floordiv()` is an alias for `$floor_div()`, which exists for compatibility with Python Polars.

### Usage

```
expr__floor_div(other)
```

```
expr__floordiv(other)
```

### Arguments

`other` Numeric literal or expression value.

### Value

A polars [expression](#)

**See Also**

- [Arithmetic operators](#)
- `<Expr>$true_div()`
- `<Expr>$mod()`

**Examples**

```
df <- pl$DataFrame(x = 1:5)

df$with_columns(
  `x/2` = pl$col("x")$true_div(2),
  `x/%2` = pl$col("x")$floor_div(2)
)
```

---

`expr__forward_fill`     *Fill missing values with the last non-null value*

---

**Description**

Fill missing values with the last non-null value

**Usage**

```
expr__forward_fill(limit = NULL)
```

**Arguments**

`limit`             The number of consecutive null values to forward fill.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = c(1, 2, NA),
  b = c(4, NA, 6),
  c = c(2, NA, NA)
)
df$select(pl$all())$forward_fill()
df$select(pl$all())$forward_fill(limit = 1)
```

---

|              |                             |
|--------------|-----------------------------|
| expr__gather | <i>Take values by index</i> |
|--------------|-----------------------------|

---

**Description**

Take values by index

**Usage**

```
expr__gather(indices)
```

**Arguments**

`indices` An expression that leads to a UInt32 dtyped Series.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  group = c("one", "one", "one", "two", "two", "two"),
  value = c(1, 98, 2, 3, 99, 4)
)
df$group_by("group", maintain_order = TRUE)$agg(
  pl$col("value")$gather(c(2, 1))
)
```

---

|                    |   |
|--------------------|---|
| expr__gather_every | <i>Take every n-th value in the Series and return as a new Series</i> |
|--------------------|---|

---

**Description**

Take every n-th value in the Series and return as a new Series

**Usage**

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

**Arguments**

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

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(foo = 1:9)
df$select(pl$col("foo")$gather_every(3))
df$select(pl$col("foo")$gather_every(3, offset = 1))
```

---

**expr\_\_ge***Check greater or equal inequality*

---

**Description**

Check greater or equal inequality

**Usage**

```
expr__ge(other)
```

**Arguments**

**other** A literal or expression value to compare with.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(x = 1:3)
df$with_columns(
  with_ge = pl$col("x")$ge(pl$lit(2)),
  with_symbol = pl$col("x") >= pl$lit(2)
)
```

---

**expr\_\_get***Return a single value by index*

---

**Description**

Return a single value by index

**Usage**

```
expr__get(index)
```

**Arguments**

**index** An expression that leads to a UInt32 dtyped Series.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(  
  group = c("one", "one", "one", "two", "two", "two"),  
  value = c(1, 98, 2, 3, 99, 4)  
)  
df$group_by("group", maintain_order = TRUE)$agg(  
  pl$col("value")$get(1)  
)
```

---

expr\_\_gt

*Check greater or equal inequality*

---

**Description**

Check greater or equal inequality

**Usage**

```
expr__gt(other)
```

**Arguments**

**other** A literal or expression value to compare with.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(x = 1:3)  
df$with_columns(  
  with_gt = pl$col("x")$gt(pl$lit(2)),  
  with_symbol = pl$col("x") > pl$lit(2)  
)
```

---

|            |                      |
|------------|----------------------|
| expr__hash | <i>Hash elements</i> |
|------------|----------------------|

---

**Description**

Hash elements

**Usage**

```
expr__hash(seed = 0, seed_1 = NULL, seed_2 = NULL, seed_3 = NULL)
```

**Arguments**

**seed** Integer, random seed parameter. Defaults to 0.  
**seed\_1, seed\_2, seed\_3** Integer, random seed parameters. Default to **seed** if not set.

**Details**

This implementation of hash does not guarantee stable results across different Polars versions. Its stability is only guaranteed within a single version.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, 2, NA), b = c("x", NA, "z"))
df$with_columns(pl$all()$hash(10, 20, 30, 40))
```

---

|                 |  |
|-----------------|--|
| 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**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(  
  a = c(NA, 1, NA),  
  b = c(10, NA, 300),  
  c = c(350, 650, 850)  
)  
df$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**

A polars [expression](#)

**Examples**

```
pl$DataFrame(x = 1:11)$select(pl$col("x")$head(3))
```

---

|            |  |
|------------|--|
| expr__hist | <i>Bin values into buckets and count their occurrences</i> |
|------------|--|

---

**Description**

[Experimental]

**Usage**

```
expr__hist(  
  bins = NULL,  
  ...,  
  bin_count = NULL,  
  include_category = FALSE,  
  include_breakpoint = FALSE  
)
```

**Arguments**

|                                 |   |
|---------------------------------|---|
| <code>bins</code>               | Discretizations to make. If <code>NULL</code> (default), we determine the boundaries based on the data. |
| <code>...</code>                | These dots are for future extensions and must be empty.   |
| <code>bin_count</code>          | If no bins provided, this will be used to determine the distance of the bins.                           |
| <code>include_category</code>   | Include a column that shows the intervals as categories.  |
| <code>include_breakpoint</code> | Include a column that indicates the upper breakpoint.   |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, 3, 8, 8, 2, 1, 3))
df$select(pl$col("a")$hist(bins = 1:3))
df$select(
  pl$col("a")$hist(
    bins = 1:3, include_category = TRUE, include_breakpoint = TRUE
  )
)
```

---

|                            |                                     |
|----------------------------|-------------------------------------|
| <code>expr__implode</code> | <i>Aggregate values into a list</i> |
|----------------------------|-------------------------------------|

---

**Description**

Aggregate values into a list

**Usage**

```
expr__implode()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:3, b = 4:6)
df$with_columns(pl$col("a")$implode())
```



---

`expr__interpolate` *Fill null values using interpolation*

---

### Description

Fill null values using interpolation

### Usage

```
expr__interpolate(method = c("linear", "nearest"))
```

### Arguments

`method` Interpolation method. Must be one of "linear" or "nearest".

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = c(1, NA, 3), b = c(1, NaN, 3))
df$with_columns(
  a_interpolated = pl$col("a")$interpolate(),
  b_interpolated = pl$col("b")$interpolate()
)
```

---

`expr__interpolate_by` *Fill null values using interpolation based on another column*

---

### Description

Fill null values using interpolation based on another column

### Usage

```
expr__interpolate_by(by)
```

### Arguments

`by` Column to interpolate values based on.

### Value

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, NA, NA, 3), b = c(1, 2, 7, 8))
df$with_columns(
  a_interpolated = pl$col("a")$interpolate_by("b")
)
```

---

|                               |   |
|-------------------------------|---|
| <code>expr__is_between</code> | <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 = c("both", "left", "right", "none")
)
```

**Arguments**

|                          |  |
|--------------------------|--|
| <code>lower_bound</code> | Lower bound value. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals. |
| <code>upper_bound</code> | Upper bound value. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals. |
| <code>closed</code>      | Define which sides of the interval are closed (inclusive). Must be one of "left", "right", "both" or "none".                         |

**Details**

If the value of the `lower_bound` is greater than that of the `upper_bound` then the result will be `FALSE`, as no value can satisfy the condition.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(num = 1:5)
df$with_columns(
  is_between = pl$col("num")$is_between(2, 4)
)
```

# Use the closed argument to include or exclude the values at the bounds:

```
df$with_columns(
  is_between = pl$col("num")$is_between(2, 4, closed = "left")
)

# You can also use strings as well as numeric/temporal values (note: ensure
# that string literals are wrapped with lit so as not to conflate them with
# column names):
df <- pl$DataFrame(a = letters[1:5])
df$with_columns(
  is_between = pl$col("a")$is_between(pl$lit("a"), pl$lit("c"))
)

# Use column expressions as lower/upper bounds, comparing to a literal value:
df <- pl$DataFrame(a = 1:5, b = 5:1)
df$with_columns(
  between_ab = pl$lit(3)$is_between(pl$col("a"), pl$col("b"))
)
```

---

`expr__is_duplicated` *Return a boolean mask indicating duplicated values*

---

## Description

Return a boolean mask indicating duplicated values

## Usage

```
expr__is_duplicated()
```

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(a = c(1, 1, 2, 3, 2))
df$select(pl$col("a")$is_duplicated())
```

---

`expr__is_finite` *Check if elements are finite*

---

## Description

Check if elements are finite

## Usage

```
expr__is_finite()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, 2), b = c(3, Inf))
df$with_columns(
  a_finite = pl$col("a")$is_finite(),
  b_finite = pl$col("b")$is_finite()
)
```

---

`expr__is_first_distinct`

*Return a boolean mask indicating the first occurrence of each distinct value*

---

**Description**

Return a boolean mask indicating the first occurrence of each distinct value

**Usage**

```
expr__is_first_distinct()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, 1, 2, 3, 2))
df$with_columns(
  is_first_distinct = pl$col("a")$is_first_distinct()
)
```

---

`expr__is_in`

*Check if elements of an expression are present in another expression*

---

**Description**

Check if elements of an expression are present in another expression

**Usage**

```
expr__is_in(other)
```

**Arguments**

`other` Accepts expression input. Strings are parsed as column names.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(  
  sets = list(1:3, 1:2, 9:10),  
  optional_members = 1:3  
)  
df$with_columns(  
  contains = pl$col("optional_members")$is_in("sets")  
)
```

---

`expr__is_infinite` *Check if elements are infinite*

---

**Description**

Check if elements are infinite

**Usage**

```
expr__is_infinite()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, 2), b = c(3, Inf))  
df$with_columns(  
  a_infinite = pl$col("a")$is_infinite(),  
  b_infinite = pl$col("b")$is_infinite()  
)
```

---

`expr__is_last_distinct`

*Return a boolean mask indicating the last occurrence of each distinct value*

---

### Description

Return a boolean mask indicating the last occurrence of each distinct value

### Usage

```
expr__is_last_distinct()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = c(1, 1, 2, 3, 2))
df$with_columns(
  is_last_distinct = pl$col("a")$is_last_distinct()
)
```

---

`expr__is_nan`

*Check if elements are NaN*

---

### Description

Floating point NaN (Not A Number) should not be confused with missing data represented as NA (in R) or null (in Polars).

### Usage

```
expr__is_nan()
```

### Value

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = c(1, 2, NA, 1, 5),
  b = c(1, 2, NaN, 1, 5)
)
df$with_columns(
  a_nan = pl$col("a")$is_nan(),
  b_nan = pl$col("b")$is_nan()
)
```

---

expr\_\_is\_not\_nan      *Check if elements are not NaN*

---

**Description**

Floating point NaN (Not A Number) should not be confused with missing data represented as NA (in R) or null (in Polars).

**Usage**

```
expr__is_not_nan()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = c(1, 2, NA, 1, 5),
  b = c(1, 2, NaN, 1, 5)
)
df$with_columns(
  a_not_nan = pl$col("a")$is_not_nan(),
  b_not_nan = pl$col("b")$is_not_nan()
)
```

---

expr\_\_is\_not\_null      *Check if elements are not NULL*

---

**Description**

Check if elements are not NULL

**Usage**

```
expr__is_not_null()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(  
  a = c(1, 2, NA, 1, 5),  
  b = c(1, 2, NaN, 1, 5)  
)  
df$with_columns(  
  a_not_null = pl$col("a")$is_not_null(),  
  b_not_null = pl$col("b")$is_not_null()  
)
```

---

|               |                                   |
|---------------|-----------------------------------|
| expr__is_null | <i>Check if elements are NULL</i> |
|---------------|-----------------------------------|

---

**Description**

Check if elements are NULL

**Usage**

```
expr__is_null()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(  
  a = c(1, 2, NA, 1, 5),  
  b = c(1, 2, NaN, 1, 5)  
)  
df$with_columns(  
  a_null = pl$col("a")$is_null(),  
  b_null = pl$col("b")$is_null()  
)
```



---

|                 |   |
|-----------------|---|
| expr__is_unique | <i>Return a boolean mask indicating unique values</i> |
|-----------------|---|

---

### Description

Return a boolean mask indicating unique values

### Usage

```
expr__is_unique()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = c(1, 1, 2, 3, 2))
df$select(pl$col("a")$is_unique())
```

---

|                |   |
|----------------|---|
| expr__kurtosis | <i>Compute the kurtosis (Fisher or Pearson)</i> |
|----------------|---|

---

### Description

Kurtosis is the fourth central moment divided by the square of the variance. If Fisher's definition is used, then 3.0 is subtracted from the result to give 0.0 for a normal distribution. If `bias` is `FALSE` then the kurtosis is calculated using `k` statistics to eliminate bias coming from biased moment estimators.

### Usage

```
expr__kurtosis(..., fisher = TRUE, bias = TRUE)
```

### Arguments

|                     |  |
|---------------------|--|
| <code>...</code>    | These dots are for future extensions and must be empty.  |
| <code>fisher</code> | If <code>TRUE</code> (default), Fisher's definition is used (normal ==> 0.0). If <code>FALSE</code> , Pearson's definition is used (normal ==> 3.0). |
| <code>bias</code>   | If <code>FALSE</code> , the calculations are corrected for statistical bias.   |

### Value

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(x = c(1, 2, 3, 2, 1))
df$select(pl$col("x")$kurtosis())
```

---

|            |                           |
|------------|---------------------------|
| expr__last | <i>Get the last value</i> |
|------------|---------------------------|

---

**Description**

Get the last value

**Usage**

```
expr__last()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(x = 3:1)$with_columns(last = pl$col("x")$last())
```

---

|          |  |
|----------|--|
| expr__le | <i>Check lower or equal inequality</i> |
|----------|--|

---

**Description**

Check lower or equal inequality

**Usage**

```
expr__le(other)
```

**Arguments**

**other** A literal or expression value to compare with.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(x = 1:3)
df$with_columns(
  with_le = pl$col("x")$le(pl$lit(2)),
  with_symbol = pl$col("x") <= pl$lit(2)
)
```

---

|           |  |
|-----------|--|
| expr__len | <i>Return the number of elements in the column</i> |
|-----------|--|

---

**Description**

Null values are counted in the total.

**Usage**

```
expr__len()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:3, b = c(NA, 4, 4))
df$select(pl$all()$len())
```

---

|             |  |
|-------------|--|
| expr__limit | <i>Get the first <math>n</math> rows</i> |
|-------------|--|

---

**Description**

This is an alias for [\\$head\(\)](#).

**Usage**

```
expr__limit(n = 10)
```

**Arguments**

**n**                      Number of rows to return.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:9)
df$select(pl$col("a")$limit(3))
```

---

|           |                              |
|-----------|------------------------------|
| expr__log | <i>Compute the logarithm</i> |
|-----------|------------------------------|

---

**Description**

Compute the logarithm

**Usage**

```
expr__log(base = exp(1))
```

**Arguments**

**base** Numeric value used as base, defaults to `exp(1)`.

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(1, 2, 4))$  
  with_columns(  
    log = pl$col("a")$log(),  
    log_base_2 = pl$col("a")$log(base = 2)  
  )
```

---

|             |                                      |
|-------------|--------------------------------------|
| expr__log10 | <i>Compute the base-10 logarithm</i> |
|-------------|--------------------------------------|

---

**Description**

Compute the base-10 logarithm

**Usage**

```
expr__log10()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(1, 2, 4))$  
  with_columns(log10 = pl$col("a")$log10())
```

---

|             |   |
|-------------|---|
| expr__log1p | <i>Compute the natural logarithm plus one</i> |
|-------------|---|

---

### Description

This computes  $\log(1 + x)$  but is more numerically stable for  $x$  close to zero.

### Usage

```
expr__log1p()
```

### Value

A polars [expression](#)

### Examples

```
pl$DataFrame(a = c(1, 2, 4))$  
  with_columns(log1p = pl$col("a")$log1p())
```

---

|                   |                                  |
|-------------------|----------------------------------|
| expr__lower_bound | <i>Calculate the lower bound</i> |
|-------------------|----------------------------------|

---

### Description

Returns a unit Series with the lowest value possible for the dtype of this expression.

### Usage

```
expr__lower_bound()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = 1:3)  
df$select(pl$col("a")$lower_bound())
```

---

|          |  |
|----------|--|
| expr__lt | <i>Check strictly lower inequality</i> |
|----------|--|

---

**Description**

Check strictly lower inequality

**Usage**

```
expr__lt(other)
```

**Arguments**

`other` A literal or expression value to compare with.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(x = 1:3)
df$with_columns(
  with_lt = pl$col("x")$lt(pl$lit(2)),
  with_symbol = pl$col("x") < pl$lit(2)
)
```

---

|                   |   |
|-------------------|---|
| expr__map_batches | <i>Apply a custom R function to a whole Series or sequence of Series.</i> |
|-------------------|---|

---

**Description****[Experimental]**

The output of this custom function is presumed to be either a Series, or a scalar that will be converted into a Series. If the result is a scalar and you want it to stay as a scalar, pass in `returns_scalar = TRUE`.

If you want to apply a custom function elementwise over single values, see [map\\_elements\(\)](#). A reasonable use case for map functions is transforming the values represented by an expression using a third-party package.

**Usage**

```
expr__map_batches(lambda, return_dtype = NULL, ..., agg_list = FALSE)
```

**Arguments**

|                           |  |
|---------------------------|--|
| <code>lambda</code>       | Function to apply.   |
| <code>return_dtype</code> | Dtype of the output Series. If NULL (default), the dtype will be inferred based on the first non-null value that is returned by the function. This can lead to unexpected results, so it is recommended to provide the return dtype. |
| <code>...</code>          | These dots are for future extensions and must be empty.  |
| <code>agg_list</code>     | Aggregate the values of the expression into a list before applying the function. This parameter only works in a group-by context. The function will be invoked only once on a list of groups, rather than once per group.            |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  sine = c(0.0, 1.0, 0.0, -1.0),
  cosine = c(1.0, 0.0, -1.0, 0.0)
)
df$select(pl$all().$map_batches(\(x) {
  elems <- as.vector(x)
  which.max(elems)
}))

# In a group-by context, the `agg_list` parameter can improve performance if
# used correctly. The following example has `agg_list = FALSE`, which causes
# the function to be applied once per group. The input of the function is a
# Series of type Int64. This is less efficient.
df <- pl$DataFrame(
  a = c(0, 1, 0, 1),
  b = c(1, 2, 3, 4)
)
system.time({
  print(
    df$group_by("a")$agg(
      pl$col("b")$map_batches(\(x) x + 2, agg_list = FALSE)
    )
  )
})

# Using `agg_list = TRUE` would be more efficient. In this example, the input
# of the function is a Series of type List(Int64).
system.time({
  print(
    df$group_by("a")$agg(
      pl$col("b")$map_batches(
        \(\(x) x$list$eval(pl$element() + 2),
        agg_list = TRUE
      )
    )
  )
})
```

```

    )
  )
}

# Call a function that takes multiple arguments by creating a struct and
# referencing its fields inside the function call.
df <- pl$DataFrame(
  a = c(5, 1, 0, 3),
  b = c(4, 2, 3, 4),
)
df$with_columns(
  a_times_b = pl$struct("a", "b")$map_batches(
    \(x) x$struct$field("a") * x$struct$field("b")
  )
)

```

---

`expr__max` *Get the maximum value*

---

### Description

Get the maximum value

### Usage

```
expr__max()
```

### Value

A polars [expression](#)

### Examples

```
pl$DataFrame(x = c(1, NaN, 3))$
  with_columns(max = pl$col("x")$max())

```

---

`expr__mean` *Get mean value*

---

### Description

Get mean value

### Usage

```
expr__mean()
```



**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(x = c(1, 3, 4, NA))$  
  with_columns(mean = pl$col("x")$mean())
```

---

|              |                         |
|--------------|-------------------------|
| expr__median | <i>Get median value</i> |
|--------------|-------------------------|

---

**Description**

Get median value

**Usage**

```
expr__median()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(x = c(1, 3, 4, NA))$  
  with_columns(median = pl$col("x")$median())
```

---

|           |                              |
|-----------|------------------------------|
| expr__min | <i>Get the minimum value</i> |
|-----------|------------------------------|

---

**Description**

Get the minimum value

**Usage**

```
expr__min()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(x = c(1, NaN, 3))$  
  with_columns(min = pl$col("x")$min())
```

---

|           |                                     |
|-----------|-------------------------------------|
| expr__mod | <i>Modulo using two expressions</i> |
|-----------|-------------------------------------|

---

**Description**

Method equivalent of modulus operator `expr %% other`.

**Usage**

```
expr__mod(other)
```

**Arguments**

`other` Numeric literal or expression value.

**Value**

A polars [expression](#)

**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 | <i>Compute the most occurring value(s)</i> |
|------------|--|

---

**Description**

Compute the most occurring value(s)

**Usage**

```
expr__mode()
```

**Value**

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(a = c(1, 1, 2, 3), b = c(1, 1, 2, 2))
df$select(pl$col("a")$mode())
df$select(pl$col("b")$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

A polars [expression](#)

## 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 the maximum value with NaN</i> |
|---------------|---------------------------------------|

---

### Description

This returns NaN if there are any.

### Usage

```
expr__nan_max()
```

### Value

A polars [expression](#)

### Examples

```
pl$DataFrame(x = c(1, NA, 3, NaN, Inf))$  
  with_columns(nan_max = pl$col("x")$nan_max())
```

---

|               |                                       |
|---------------|---------------------------------------|
| expr__nan_min | <i>Get the minimum value with NaN</i> |
|---------------|---------------------------------------|

---

### Description

This returns NaN if there are any.

### Usage

```
expr__nan_min()
```

### Value

A polars [expression](#)

### Examples

```
pl$DataFrame(x = c(1, NA, 3, NaN, Inf))$  
  with_columns(nan_min = pl$col("x")$nan_min())
```

---

|          |                         |
|----------|-------------------------|
| expr__ne | <i>Check inequality</i> |
|----------|-------------------------|

---

**Description**

This propagates null values, i.e. any comparison involving `null` will return `null`. Use `$ne_missing()` to consider null values as equal.

**Usage**

```
expr__ne(other)
```

**Arguments**

`other` A literal or expression value to compare with.

**Value**

A polars [expression](#)

**See Also**

[expr\\_\\_ne\\_missing](#)

**Examples**

```
df <- pl$DataFrame(x = c(NA, FALSE, TRUE), y = c(TRUE, TRUE, TRUE))
df$with_columns(
  ne = pl$col("x")$ne(pl$col("y")),
  ne_missing = pl$col("x")$ne_missing(pl$col("y"))
)
```

---

|                  |  |
|------------------|--|
| expr__ne_missing | <i>Check inequality without null propagation</i> |
|------------------|--|

---

**Description**

Method equivalent of addition operator `expr + other`.

**Usage**

```
expr__ne_missing(other)
```

**Arguments**

`other` Element to add. Can be a string (only if `expr` is a string), a numeric value or an other expression.

**Value**

A polars [expression](#)

**See Also**

[expr\\_\\_ne](#)

**Examples**

```
df <- pl$DataFrame(x = c(NA, FALSE, TRUE), y = c(TRUE, TRUE, TRUE))
df$with_columns(
  ne = pl$col("x")$ne("y"),
  ne_missing = pl$col("x")$ne_missing("y")
)
```

---

expr\_\_not

*Negate a boolean expression*

---

**Description**

Negate a boolean expression

**Usage**

```
expr__not()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(TRUE, FALSE, FALSE, NA))

df$with_columns(a_not = pl$col("a")$not())

# Same result with "!"
df$with_columns(a_not = !pl$col("a"))
```

---

|                  |                          |
|------------------|--------------------------|
| expr__null_count | <i>Count null values</i> |
|------------------|--------------------------|

---

**Description**

Count null values

**Usage**

```
expr__null_count()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(  
  a = c(NA, 1, NA),  
  b = c(10, NA, 300),  
  c = c(1, 2, 2)  
)  
df$select(pl$all().$null_count())
```

---

|                |                            |
|----------------|----------------------------|
| expr__n_unique | <i>Count unique values</i> |
|----------------|----------------------------|

---

**Description**

null is considered to be a unique value for the purposes of this operation.

**Usage**

```
expr__n_unique()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(  
  x = c(1, 1, 2, 2, 3),  
  y = c(1, 1, 1, NA, NA)  
)  
df$select(  
  x_unique = pl$col("x").$n_unique(),  
  y_unique = pl$col("y").$n_unique()  
)
```

---

`expr__or` *Apply logical OR on two expressions*

---

### Description

Combine two boolean expressions with OR.

### Usage

```
expr__or(other)
```

### Arguments

`other` Element to add. Can be a string (only if `expr` is a string), a numeric value or an other expression.

### Value

A polars [expression](#)

### Examples

```
p1$lit(TRUE) | FALSE
p1$lit(TRUE)$or(p1$lit(TRUE))
```

---

`expr__over` *Compute expressions over the given groups*

---

### 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 = c("group_to_rows", "join", "explode")
)
```



**Arguments**

- ... [dynamic-dots](#)> 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`. Accepts expression input. 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**

A polars [expression](#)

**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 several column names a or list of
# expressions.
df$with_columns(
  pl$col("c")$min()$over("a", "b")$name$suffix("_min")
)

group_vars <- list(pl$col("a"), pl$col("b"))
df$with_columns(
  pl$col("c")$min()$over(!!!group_vars)$name$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>Computes percentage change between values</i> |
|------------------|--|

---

## Description

Computes the percentage change (as fraction) between current element and most-recent non-null element at least *n* period(s) before the current element. By default it computes the change from the previous row.

## Usage

```
expr__pct_change(n = 1)
```

## Arguments

*n* Integer or Expr indicating the number of periods to shift for forming percent change.

## Value

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(10:12, NA, 12))
df$with_columns(
  pct_change = pl$col("a")$pct_change()
)
```

---

expr\_\_peak\_max            *Get a boolean mask of the local maximum peaks*

---

**Description**

Get a boolean mask of the local maximum peaks

**Usage**

```
expr__peak_max()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(x = c(1, 2, 3, 2, 3, 4, 5, 2))
df$with_columns(peak_max = pl$col("x")$peak_max())
```

---

expr\_\_peak\_min            *Get a boolean mask of the local minimum peaks*

---

**Description**

Get a boolean mask of the local minimum peaks

**Usage**

```
expr__peak_min()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(x = c(1, 2, 3, 2, 3, 4, 5, 2))
df$with_columns(peak_min = pl$col("x")$peak_min())
```

---

|           |   |
|-----------|---|
| expr__pow | <i>Exponentiation using two expressions</i> |
|-----------|---|

---

**Description**

Method equivalent of exponentiation operator `expr ^ exponent`.

**Usage**

```
expr__pow(exponent)
```

**Arguments**

`exponent` Numeric literal or expression value.

**Value**

A polars [expression](#)

**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>Compute the product of an expression.</i> |
|---------------|--|

---

**Description**

Compute the product of an expression.

**Usage**

```
expr__product()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = 1:3, b = c(NA, 4, 4))$
  select(pl$all()$product())
```

---

|            |  |
|------------|--|
| expr__qcut | <i>Bin continuous values into discrete categories based on their quantiles</i> |
|------------|--|

---

**Description**

[Experimental]

**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.                         |
| ...              | These dots are for future extensions and must be empty.  |
| labels           | Names of the categories. The number of labels must be equal to the number of categories.   |
| left_closed      | Set the intervals to be left-closed instead of right-closed.   |
| allow_duplicates | If 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**

A polars [expression](#)

## Examples

```
# Divide a column into three categories according to pre-defined quantile
# probabilities.
df <- pl$DataFrame(foo = -2:2)
df$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 | <i>Get quantile value(s)</i> |
|----------------|------------------------------|

---

## Description

Get quantile value(s)

## Usage

```
expr__quantile(
  quantile,
  interpolation = c("nearest", "higher", "lower", "midpoint", "linear")
)
```

## Arguments

**quantile** Quantile between 0.0 and 1.0.

**interpolation** Interpolation method. Must be one of "nearest", "higher", "lower", "midpoint", "linear".

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(a = 0:5)
df$select(pl$col("a")$quantile(0.3))
df$select(pl$col("a")$quantile(0.3, interpolation = "higher"))
df$select(pl$col("a")$quantile(0.3, interpolation = "lower"))
df$select(pl$col("a")$quantile(0.3, interpolation = "midpoint"))
df$select(pl$col("a")$quantile(0.3, interpolation = "linear"))
```

---

|               |  |
|---------------|--|
| expr__radians | <i>Convert from degrees to radians</i> |
|---------------|--|

---

**Description**

Convert from degrees to radians

**Usage**

```
expr__radians()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(-720, -540, -360, -180, 0, 180, 360, 540, 720))$
  with_columns(radians = pl$col("a")$radians())
```

---

|            |  |
|------------|--|
| expr__rank | <i>Assign ranks to data, dealing with ties appropriately</i> |
|------------|--|

---

**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 | The method used to assign ranks to tied elements. Must be one of the following: <ul style="list-style-type: none"> <li>• "average" (default): The average of the ranks that would have been assigned to all the tied values is assigned to each value.</li> <li>• "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.)</li> </ul> |
|--------|---|

- "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.
- "ordinal" : All values are given a distinct rank, corresponding to the order that the values occur in the Series.
- "random" : Like 'ordinal', but the rank for ties is not dependent on the order that the values occur in the Series.

... These dots are for future extensions and must be empty.

descending Rank in descending order.

seed Integer. Only used if `method = "random"`.

### Value

A polars [expression](#)

### Examples

```
# Default is to use the "average" method to break ties
df <- pl$DataFrame(a = c(3, 6, 1, 1, 6))
df$with_columns(rank = pl$col("a")$rank())

# Ordinal method
df$with_columns(rank = pl$col("a")$rank("ordinal"))

# Use "rank" with "over" to rank within groups:
df <- pl$DataFrame(
  a = c(1, 1, 2, 2, 2),
  b = c(6, 7, 5, 14, 11)
)
df$with_columns(
  rank = pl$col("b")$rank()$over("a")
)
```

---

expr\_\_rechunk

*Create a single chunk of memory for this Series*

---

### Description

Create a single chunk of memory for this Series

### Usage

```
expr__rechunk()
```

### Value

A polars [expression](#)



**Examples**

```
df <- pl$DataFrame(a = c(1, 1, 2))

# Create a Series with 3 nulls, append column a then rechunk
df$select(pl$repeat_(NA, 3)$append(pl$col("a"))$rechunk())
```

---

`expr__reinterpret`      *Reinterpret the underlying bits as a signed/unsigned integer*

---

**Description**

This operation is only allowed for 64-bit integers. For lower bits integers, you can safely use the `$cast()` operation.

**Usage**

```
expr__reinterpret(..., signed = TRUE)
```

**Arguments**

`...`                    These dots are for future extensions and must be empty.

`signed`                 If TRUE (default), reinterpret as `pl$Int64`. Otherwise, reinterpret as `pl$UInt64`.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, 1, 2))$cast(pl$UInt64)

# Create a Series with 3 nulls, append column a then rechunk
df$with_columns(
  reinterpreted = pl$col("a")$reinterpret()
)
```

---

`expr__repeat_by`      *Repeat the elements in this Series as specified in the given expression*

---

**Description**

The repeated elements are expanded into a List dtype.

**Usage**

```
expr__repeat_by(by)
```

**Arguments**

**by** Numeric column that determines how often the values will be repeated. The column will be coerced to UInt32. Give this dtype to make the coercion a no-op. Accepts expression input, strings are parsed as column names.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c("x", "y", "z"), n = 1:3)

df$with_columns(
  repeated = pl$col("a")$repeat_by("n")
)
```

---

**expr\_\_replace**      *Replace the given values by different values of the same data type.*

---

**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** Value or vector of values to replace. Accepts expression input. Vectors are parsed as Series, other non-expression inputs are parsed as literals. Also accepts a list of values like `list(old = new)`.

**new** Value or vector of values to replace by. Accepts expression input. Vectors are parsed as Series, other non-expression inputs are parsed as literals. Length must match the length of `old` or have length 1.

**Details**

The global string cache must be enabled when replacing categorical values.

**Value**

A polars [expression](#)

**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))

# The original data type is preserved when replacing by values of a
# different data type. Use $replace_strict() to replace and change the
# return data type.
df <- pl$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**

|            |   |
|------------|---|
| <b>old</b> | Value or vector of values to replace. Accepts expression input. Vectors are parsed as Series, other non-expression inputs are parsed as literals. Also accepts a list of values like <code>list(old = new)</code> .     |
| <b>new</b> | Value or vector of values to replace by. Accepts expression input. Vectors are parsed as Series, other non-expression inputs are parsed as literals. Length must match the length of <code>old</code> or have length 1. |

|              |   |
|--------------|---|
| ...          | These dots are for future extensions and must be empty.   |
| default      | Set values that were not replaced to this value. If NULL (default), an error is raised if any values were not replaced. Accepts expression input. Non-expression inputs are parsed as literals. |
| return_dtype | The data type of the resulting expression. If NULL (default), the data type is determined automatically based on the other inputs.  |

## Details

The global string cache must be enabled when replacing categorical values.

## Value

A polars [expression](#)

## 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))

# By default, an error is raised if any non-null values were not replaced.
# Specify a default to set all values that were not matched.
tryCatch(
  df$with_columns(replaced = pl$col("a")$replace_strict(mapping)),
  error = function(e) print(e)
)

# 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)
```

## Arguments

`dimensions`      A integer vector of length of the dimension size. If -1 is used in any of the dimensions, that dimension is inferred.

## Details

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.

## Value

A polars [expression](#)

## 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)))

# We can have more than 2 dimensions
df <- pl$DataFrame(foo = 1:8)
df$select(pl$col("foo")$reshape(c(2, 2, 2)))

```

---

|               |                              |
|---------------|------------------------------|
| expr__reverse | <i>Reverse an expression</i> |
|---------------|------------------------------|

---

### Description

Reverse an expression

### Usage

```
expr__reverse()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(  
  a = 1:5,  
  fruits = c("banana", "banana", "apple", "apple", "banana"),  
  b = 5:1  
)  
  
df$with_columns(  
  pl$all().$reverse().$name$.suffix("_reverse")  
)
```

---

|           |   |
|-----------|---|
| expr__rle | <i>Compress the column data using run-length encoding</i> |
|-----------|---|

---

### Description

Run-length encoding (RLE) encodes data by storing each run of identical values as a single value and its length.

### Usage

```
expr__rle()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = c(1, 1, 2, 1, NA, 1, 3, 3))  
  
df$select(pl$col("a").$rle()).$unnest("a")
```

---

|              |   |
|--------------|---|
| expr__rle_id | <i>Get a distinct integer ID for each run of identical values</i> |
|--------------|---|

---

### Description

The ID starts at 0 and increases by one each time the value of the column changes.

### Usage

```
expr__rle_id()
```

### Details

This functionality is especially useful for defining a new group for every time a column's value changes, rather than for every distinct value of that column.

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(
  a = c(1, 2, 1, 1, 1),
  b = c("x", "x", NA, "y", "y")
)

df$with_columns(
  rle_id_a = pl$col("a")$rle_id(),
  rle_id_ab = pl$struct("a", "b")$rle_id()
)
```

---

|               |  |
|---------------|--|
| expr__rolling | <i>Create rolling groups based on a temporal or integer column</i> |
|---------------|--|

---

### 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 + offset, t\_0 + offset + period]
- (t\_1 + offset, t\_1 + offset + period]
- ...
- (t\_n + offset, t\_n + offset + period]

## Usage

```
expr__rolling(index_column, ..., period, offset = NULL, closed = "right")
```

## Arguments

|                           |  |
|---------------------------|--|
| <code>index_column</code> | Character. Name of the column used to group based on the time window. Often of type Date/Datetime. This column must be sorted in ascending order. In case of a rolling group by on indices, dtype needs to be one of UInt32, UInt64, Int32, Int64. Note that the first three get cast to Int64, so if performance matters use an Int64 column. |
| <code>...</code>          | These dots are for future extensions and must be empty.  |
| <code>period</code>       | Length of the window - must be non-negative.   |
| <code>offset</code>       | Offset of the window. Default is <code>-period</code> .  |
| <code>closed</code>       | Define which sides of the range are closed (inclusive). One of the following: "both" (default), "left", "right", "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

A polars [expression](#)

## Examples

```
dates <- as.POSIXct(
  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))

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



---

expr\_\_rolling\_max      *Apply a rolling max over values*

---

## Description

### [Experimental]

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 `weights` vector. The resulting values will be aggregated.

The window at a given row will include the row itself, and the `window_size - 1` elements before it.

## Usage

```
expr__rolling_max(
    window_size,
    weights = NULL,
    ...,
    min_periods = NULL,
    center = FALSE
)
```

## Arguments

|                          |  |
|--------------------------|--|
| <code>window_size</code> | The length of the window in number of elements.  |
| <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>...</code>         | These dots are for future extensions and must be empty.  |
| <code>min_periods</code> | The number of values in the window that should be non-null before computing a result. If <code>NULL</code> (default), it will be set equal to <code>window_size</code> . |
| <code>center</code>      | If <code>TRUE</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

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_max = pl$col("a")$rolling_max(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_max = pl$col("a")$rolling_max(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

# Center the values in the window
df$with_columns(
  rolling_max = pl$col("a")$rolling_max(window_size = 3, center = TRUE)
)
```

---

`expr__rolling_max_by` *Apply a rolling max based on another column*

---

**Description****[Experimental]**

Given a by column  $\langle t_0, t_1, \dots, t_n \rangle$ , then `closed = "right"` (the default) means the windows will be:

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

**Usage**

```
expr__rolling_max_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = c("right", "both", "left", "none")
)
```

**Arguments**

**by** Should be `DateTime`, `Date`, `UInt64`, `UInt32`, `Int64`, or `Int32` data type after conversion by `as_polars_expr()`. Note that the integer ones require using `"i"` in `window_size`. Accepts expression input. Strings are parsed as column names.

|                          |  |
|--------------------------|--|
| <code>window_size</code> | <p>The length of the window. Can be a dynamic temporal size indicated by a timedelta or the following string language:</p> <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 calendar day)</li> <li>• 1w (1 calendar week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1q (1 calendar quarter)</li> <li>• 1y (1 calendar year)</li> </ul> <p>Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds</p> <p>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".</p> <p>...</p> <p>These dots are for future extensions and must be empty.</p> |
| <code>min_periods</code> | The number of values in the window that should be non-null before computing a result. If NULL (default), it will be set equal to <code>window_size</code> .  |
| <code>closed</code>      | Define which sides of the interval are closed (inclusive). Default is "right".   |

## 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

A polars [expression](#)

## Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# Compute the rolling max with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_max = pl$col("index")$rolling_max_by(
```

```

    "date",
    window_size = "2h"
  )
)

# Compute the rolling max with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_max = pl$col("index")$rolling_max_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)

```

---

`expr__rolling_mean`     *Apply a rolling mean over values*

---

## Description

### [Experimental]

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 `weights` vector. The resulting values will be aggregated.

The window at a given row will include the row itself, and the `window_size - 1` elements before it.

## Usage

```

expr__rolling_mean(
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE
)

```

## Arguments

|                          |  |
|--------------------------|--|
| <code>window_size</code> | The length of the window in number of elements.  |
| <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>...</code>         | These dots are for future extensions and must be empty.  |
| <code>min_periods</code> | The number of values in the window that should be non-null before computing a result. If <code>NULL</code> (default), it will be set equal to <code>window_size</code> . |
| <code>center</code>      | If <code>TRUE</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**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_mean = pl$col("a")$rolling_mean(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_mean = pl$col("a")$rolling_mean(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

# Center the values in the window
df$with_columns(
  rolling_mean = pl$col("a")$rolling_mean(window_size = 3, center = TRUE)
)
```

---

```
expr__rolling_mean_by
```

*Apply a rolling mean based on another column*

---

**Description****[Experimental]**

Given a by column  $\langle t_0, t_1, \dots, t_n \rangle$ , then `closed = "right"` (the default) means the windows will be:

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

**Usage**

```
expr__rolling_mean_by(
  by,
  window_size,
  ...,
```

```

    min_periods = 1,
    closed = c("right", "both", "left", "none")
)

```

## Arguments

|                    |  |
|--------------------|--|
| <b>by</b>          | Should be <code>DateTime</code> , <code>Date</code> , <code>UInt64</code> , <code>UInt32</code> , <code>Int64</code> , or <code>Int32</code> data type after conversion by <code>as_polars_expr()</code> . Note that the integer ones require using "i" in <code>window_size</code> . Accepts expression input. Strings are parsed as column names.  |
| <b>window_size</b> | The length of the window. Can be a dynamic temporal size indicated by a <code>timedelta</code> or 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 calendar day)</li> <li>• 1w (1 calendar week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1q (1 calendar quarter)</li> <li>• 1y (1 calendar year)</li> </ul> <p>Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds</p> <p>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".</p> <p>...</p> <p>These dots are for future extensions and must be empty.</p> |
| <b>min_periods</b> | The number of values in the window that should be non-null before computing a result. If <code>NULL</code> (default), it will be set equal to <code>window_size</code> .   |
| <b>closed</b>      | Define which sides of the interval are closed (inclusive). Default is "right".   |

## 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

A polars [expression](#)

**Examples**

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# Compute the rolling mean with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_mean = pl$col("index")$rolling_mean_by(
    "date",
    window_size = "2h"
  )
)

# Compute the rolling mean with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_mean = pl$col("index")$rolling_mean_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)
```

---

`expr__rolling_median` *Apply a rolling median over values*

---

**Description****[Experimental]**

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 `weights` vector. The resulting values will be aggregated.

The window at a given row will include the row itself, and the `window_size - 1` elements before it.

**Usage**

```
expr__rolling_median(
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE
)
```

**Arguments**

|                          |   |
|--------------------------|---|
| <code>window_size</code> | The length of the window in number of elements.   |
| <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>...</code>         | These dots are for future extensions and must be empty.   |
| <code>min_periods</code> | The number of values in the window that should be non-null before computing a result. If NULL (default), it will be set equal to <code>window_size</code> . |
| <code>center</code>      | If TRUE, 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**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_median = pl$col("a")$rolling_median(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_median = pl$col("a")$rolling_median(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

# Center the values in the window
df$with_columns(
  rolling_median = pl$col("a")$rolling_median(window_size = 3, center = TRUE)
)
```

---

```
expr__rolling_median_by
```

*Apply a rolling median based on another column*

---

**Description****[Experimental]**

Given a by column `<t_0, t_1, ..., t_n>`, then `closed = "right"` (the default) means the windows will be:

- `(t_0 - window_size, t_0]`



- (t<sub>1</sub> - window\_size, t<sub>1</sub>]
- ...
- (t<sub>n</sub> - window\_size, t<sub>n</sub>]

## Usage

```
expr__rolling_median_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = c("right", "both", "left", "none")
)
```

## Arguments

|                    |  |
|--------------------|--|
| <b>by</b>          | Should be DateTime, Date, UInt64, UInt32, Int64, or Int32 data type after conversion by <code>as_polars_expr()</code> . Note that the integer ones require using "i" in <code>window_size</code> . Accepts expression input. Strings are parsed as column names.   |
| <b>window_size</b> | The length of the window. Can be a dynamic temporal size indicated by a timedelta or 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 calendar day)</li> <li>• 1w (1 calendar week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1q (1 calendar quarter)</li> <li>• 1y (1 calendar year)</li> </ul> Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds<br>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". |
| <b>...</b>         | These dots are for future extensions and must be empty.  |
| <b>min_periods</b> | The number of values in the window that should be non-null before computing a result. If NULL (default), it will be set equal to <code>window_size</code> .  |
| <b>closed</b>      | Define which sides of the interval are closed (inclusive). Default is "right".   |

## 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**

A polars [expression](#)

**Examples**

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# Compute the rolling median with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_median = pl$col("index")$rolling_median_by(
    "date",
    window_size = "2h"
  )
)

# Compute the rolling median with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_median = pl$col("index")$rolling_median_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)
```

---

expr\_\_rolling\_min      *Apply a rolling min over values*

---

**Description****[Experimental]**

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 `weights` vector. The resulting values will be aggregated.

The window at a given row will include the row itself, and the `window_size - 1` elements before it.

**Usage**

```
expr__rolling_min(
  window_size,
```

```

    weights = NULL,
    ...,
    min_periods = NULL,
    center = FALSE
  )

```

### Arguments

|                          |   |
|--------------------------|---|
| <code>window_size</code> | The length of the window in number of elements.   |
| <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>...</code>         | These dots are for future extensions and must be empty.   |
| <code>min_periods</code> | The number of values in the window that should be non-null before computing a result. If NULL (default), it will be set equal to <code>window_size</code> . |
| <code>center</code>      | If TRUE, 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

A polars [expression](#)

### Examples

```

df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_min = pl$col("a")$rolling_min(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_min = pl$col("a")$rolling_min(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

# Center the values in the window
df$with_columns(
  rolling_min = pl$col("a")$rolling_min(window_size = 3, center = TRUE)
)

```

---

`expr__rolling_min_by` *Apply a rolling min based on another column*

---

## Description

### [Experimental]

Given a by column `<t_0, t_1, ..., t_n>`, then `closed = "right"` (the default) means the windows will be:

- `(t_0 - window_size, t_0]`
- `(t_1 - window_size, t_1]`
- ...
- `(t_n - window_size, t_n]`

## Usage

```
expr__rolling_min_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = c("right", "both", "left", "none")
)
```

## Arguments

- |                          |   |
|--------------------------|---|
| <code>by</code>          | Should be <code>DateTime</code> , <code>Date</code> , <code>UInt64</code> , <code>UInt32</code> , <code>Int64</code> , or <code>Int32</code> data type after conversion by <code>as_polars_expr()</code> . Note that the integer ones require using "i" in <code>window_size</code> . Accepts expression input. Strings are parsed as column names.   |
| <code>window_size</code> | The length of the window. Can be a dynamic temporal size indicated by a <code>timedelta</code> or the following string language: <ul style="list-style-type: none"> <li>• <code>1ns</code> (1 nanosecond)</li> <li>• <code>1us</code> (1 microsecond)</li> <li>• <code>1ms</code> (1 millisecond)</li> <li>• <code>1s</code> (1 second)</li> <li>• <code>1m</code> (1 minute)</li> <li>• <code>1h</code> (1 hour)</li> <li>• <code>1d</code> (1 calendar day)</li> <li>• <code>1w</code> (1 calendar week)</li> <li>• <code>1mo</code> (1 calendar month)</li> <li>• <code>1q</code> (1 calendar quarter)</li> <li>• <code>1y</code> (1 calendar year)</li> </ul> |

|             |   |
|-------------|---|
|             | 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". |
| ...         | These dots are for future extensions and must be empty.   |
| min_periods | The number of values in the window that should be non-null before computing a result. If NULL (default), it will be set equal to <code>window_size</code> .   |
| closed      | Define which sides of the interval are closed (inclusive). Default is "right".  |

### 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

A polars [expression](#)

### Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# Compute the rolling min with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_min = pl$col("index")$rolling_min_by(
    "date",
    window_size = "2h"
  )
)

# Compute the rolling min with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_min = pl$col("index")$rolling_min_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)
```

---

```
expr__rolling_quantile
```

*Apply a rolling quantile over values*

---

## Description

### [Experimental]

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 `weights` vector. The resulting values will be aggregated.

The window at a given row will include the row itself, and the `window_size - 1` elements before it.

## Usage

```
expr__rolling_quantile(
  quantile,
  interpolation = c("nearest", "higher", "lower", "midpoint", "linear"),
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE
)
```

## Arguments

|                            |  |
|----------------------------|--|
| <code>quantile</code>      | Quantile between 0.0 and 1.0.  |
| <code>interpolation</code> | Interpolation method. Must be one of "nearest", "higher", "lower", "midpoint", "linear".   |
| <code>window_size</code>   | The length of the window in number of elements.  |
| <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>...</code>           | These dots are for future extensions and must be empty.  |
| <code>min_periods</code>   | The number of values in the window that should be non-null before computing a result. If <code>NULL</code> (default), it will be set equal to <code>window_size</code> . |
| <code>center</code>        | If <code>TRUE</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

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_quantile = pl$col("a")$rolling_quantile(
    quantile = 0.25, window_size = 4
  )
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_quantile = pl$col("a")$rolling_quantile(
    quantile = 0.25, window_size = 4, weights = c(0.2, 0.4, 0.4, 0.2)
  )
)

# Specify weights and interpolation method:
df$with_columns(
  rolling_quantile = pl$col("a")$rolling_quantile(
    quantile = 0.25, window_size = 4, weights = c(0.2, 0.4, 0.4, 0.2),
    interpolation = "linear"
  )
)

# Center the values in the window
df$with_columns(
  rolling_quantile = pl$col("a")$rolling_quantile(
    quantile = 0.25, window_size = 5, center = TRUE
  )
)
```

---

```
expr__rolling_quantile_by
```

*Apply a rolling quantile based on another column*

---

**Description****[Experimental]**

Given a by column  $\langle t_0, t_1, \dots, t_n \rangle$ , then `closed = "right"` (the default) means the windows will be:

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

**Usage**

```

expr__rolling_quantile_by(
  by,
  window_size,
  ...,
  quantile,
  interpolation = c("nearest", "higher", "lower", "midpoint", "linear"),
  min_periods = 1,
  closed = c("right", "both", "left", "none")
)

```

**Arguments**

|                            |  |
|----------------------------|--|
| <code>by</code>            | Should be <code>DateTime</code> , <code>Date</code> , <code>UInt64</code> , <code>UInt32</code> , <code>Int64</code> , or <code>Int32</code> data type after conversion by <code>as_polars_expr()</code> . Note that the integer ones require using "i" in <code>window_size</code> . Accepts expression input. Strings are parsed as column names.  |
| <code>window_size</code>   | The length of the window. Can be a dynamic temporal size indicated by a <code>timedelta</code> or 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 calendar day)</li> <li>• 1w (1 calendar week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1q (1 calendar quarter)</li> <li>• 1y (1 calendar year)</li> </ul> <p>Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds</p> <p>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".</p> |
| <code>...</code>           | These dots are for future extensions and must be empty.  |
| <code>quantile</code>      | Quantile between 0.0 and 1.0.  |
| <code>interpolation</code> | Interpolation method. Must be one of "nearest", "higher", "lower", "midpoint", "linear".   |
| <code>min_periods</code>   | The number of values in the window that should be non-null before computing a result. If <code>NULL</code> (default), it will be set equal to <code>window_size</code> .   |
| <code>closed</code>        | Define which sides of the interval are closed (inclusive). Default is "right".   |



## 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

A polars [expression](#)

## Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# Compute the rolling quantile with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_quantile = pl$col("index")$rolling_quantile_by(
    "date",
    window_size = "2h",
    quantile = 0.3
  )
)

# Compute the rolling quantile with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_quantile = pl$col("index")$rolling_quantile_by(
    "date",
    window_size = "2h",
    quantile = 0.3,
    closed = "both"
  )
)
```

---

`expr__rolling_skew`    *Apply a rolling skew over values*

---

## Description

### [Experimental]

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 `weights` vector. The resulting values will be aggregated.

The window at a given row will include the row itself, and the `window_size - 1` elements before it.

**Usage**

```
expr__rolling_skew(window_size, ..., bias = TRUE)
```

**Arguments**

**window\_size** The length of the window in number of elements.  
**...** These dots are for future extensions and must be empty.  
**bias** If **FALSE**, the calculations are corrected for statistical bias.

**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**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(1, 4, 2, 9))
df$with_columns(
  rolling_skew = pl$col("a")$rolling_skew(3)
)
```

---

`expr__rolling_std`      *Apply a rolling standard deviation over values*

---

**Description****[Experimental]**

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 `weights` vector. The resulting values will be aggregated.

The window at a given row will include the row itself, and the `window_size - 1` elements before it.

**Usage**

```
expr__rolling_std(
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE,
  ddof = 1
)
```

**Arguments**

|                          |  |
|--------------------------|--|
| <code>window_size</code> | The length of the window in number of elements.  |
| <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>...</code>         | These dots are for future extensions and must be empty.  |
| <code>min_periods</code> | The number of values in the window that should be non-null before computing a result. If <code>NULL</code> (default), it will be set equal to <code>window_size</code> . |
| <code>center</code>      | If <code>TRUE</code> , set the labels at the center of the window.   |
| <code>ddof</code>        | "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**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_std = pl$col("a")$rolling_std(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_std = pl$col("a")$rolling_std(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

# Center the values in the window
df$with_columns(
  rolling_std = pl$col("a")$rolling_std(window_size = 3, center = TRUE)
)
```

## Description

### [Experimental]

Given a by column  $\langle t_0, t_1, \dots, t_n \rangle$ , then `closed = "right"` (the default) means the windows will be:

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

## Usage

```
expr__rolling_std_by(
    by,
    window_size,
    ...,
    min_periods = 1,
    closed = c("right", "both", "left", "none"),
    ddof = 1
)
```

## Arguments

**by** Should be `DateTime`, `Date`, `UInt64`, `UInt32`, `Int64`, or `Int32` data type after conversion by `as_polars_expr()`. Note that the integer ones require using "i" in `window_size`. Accepts expression input. Strings are parsed as column names.

**window\_size** The length of the window. Can be a dynamic temporal size indicated by a `timedelta` or the following string language:

- 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".

|             |  |
|-------------|--|
| ...         | These dots are for future extensions and must be empty.  |
| min_periods | The number of values in the window that should be non-null before computing a result. If NULL (default), it will be set equal to <code>window_size</code> .            |
| closed      | Define which sides of the interval are closed (inclusive). Default is "right".   |
| ddof        | "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

A polars [expression](#)

## Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# 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      *Apply a rolling sum over values*

---

## Description

### [Experimental]

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 `weights` vector. The resulting values will be aggregated.

The window at a given row will include the row itself, and the `window_size - 1` elements before it.

## Usage

```
expr__rolling_sum(
    window_size,
    weights = NULL,
    ...,
    min_periods = NULL,
    center = FALSE
)
```

## Arguments

|                          |  |
|--------------------------|--|
| <code>window_size</code> | The length of the window in number of elements.  |
| <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>...</code>         | These dots are for future extensions and must be empty.  |
| <code>min_periods</code> | The number of values in the window that should be non-null before computing a result. If <code>NULL</code> (default), it will be set equal to <code>window_size</code> . |
| <code>center</code>      | If <code>TRUE</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

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_sum = pl$col("a")$rolling_sum(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_sum = pl$col("a")$rolling_sum(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

# Center the values in the window
df$with_columns(
  rolling_sum = pl$col("a")$rolling_sum(window_size = 3, center = TRUE)
)
```

---

expr\_\_rolling\_sum\_by *Apply a rolling sum based on another column*

---

**Description****[Experimental]**

Given a by column  $\langle t_0, t_1, \dots, t_n \rangle$ , then `closed = "right"` (the default) means the windows will be:

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

**Usage**

```
expr__rolling_sum_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = c("right", "both", "left", "none")
)
```

**Arguments**

**by** Should be `DateTime`, `Date`, `UInt64`, `UInt32`, `Int64`, or `Int32` data type after conversion by `as_polars_expr()`. Note that the integer ones require using `"i"` in `window_size`. Accepts expression input. Strings are parsed as column names.

|                          |  |
|--------------------------|--|
| <code>window_size</code> | <p>The length of the window. Can be a dynamic temporal size indicated by a timedelta or the following string language:</p> <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 calendar day)</li> <li>• 1w (1 calendar week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1q (1 calendar quarter)</li> <li>• 1y (1 calendar year)</li> </ul> <p>Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds</p> <p>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".</p> <p>...</p> <p>These dots are for future extensions and must be empty.</p> |
| <code>min_periods</code> | The number of values in the window that should be non-null before computing a result. If NULL (default), it will be set equal to <code>window_size</code> .  |
| <code>closed</code>      | Define which sides of the interval are closed (inclusive). Default is "right".   |

## 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

A polars [expression](#)

## Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# Compute the rolling sum with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_sum = pl$col("index")$rolling_sum_by(
```



```

    "date",
    window_size = "2h"
  )
)

# Compute the rolling sum with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_sum = pl$col("index")$rolling_sum_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)

```

---

expr\_\_rolling\_var      *Apply a rolling variance over values*

---

## Description

### [Experimental]

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 `weights` vector. The resulting values will be aggregated.

The window at a given row will include the row itself, and the `window_size - 1` elements before it.

## Usage

```

expr__rolling_var(
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE,
  ddof = 1
)

```

## Arguments

|                          |  |
|--------------------------|--|
| <code>window_size</code> | The length of the window in number of elements.  |
| <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>...</code>         | These dots are for future extensions and must be empty.  |
| <code>min_periods</code> | The number of values in the window that should be non-null before computing a result. If <code>NULL</code> (default), it will be set equal to <code>window_size</code> . |
| <code>center</code>      | If <code>TRUE</code> , set the labels at the center of the window.   |
| <code>ddof</code>        | "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

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_var = pl$col("a")$rolling_var(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_var = pl$col("a")$rolling_var(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

# Center the values in the window
df$with_columns(
  rolling_var = pl$col("a")$rolling_var(window_size = 3, center = TRUE)
)
```

---

`expr__rolling_var_by` *Apply a rolling variance based on another column*

---

## Description

### [Experimental]

Given a by column  $\langle t_0, t_1, \dots, t_n \rangle$ , then `closed = "right"` (the default) means the windows will be:

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

## Usage

```
expr__rolling_var_by(
  by,
  window_size,
  ...,
  min_periods = 1,
```

```

    closed = c("right", "both", "left", "none"),
    ddof = 1
)

```

## Arguments

|                    |  |
|--------------------|--|
| <b>by</b>          | Should be <code>DateTime</code> , <code>Date</code> , <code>UInt64</code> , <code>UInt32</code> , <code>Int64</code> , or <code>Int32</code> data type after conversion by <code>as_polars_expr()</code> . Note that the integer ones require using "i" in <code>window_size</code> . Accepts expression input. Strings are parsed as column names.  |
| <b>window_size</b> | The length of the window. Can be a dynamic temporal size indicated by a <code>timedelta</code> or 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 calendar day)</li> <li>• 1w (1 calendar week)</li> <li>• 1mo (1 calendar month)</li> <li>• 1q (1 calendar quarter)</li> <li>• 1y (1 calendar year)</li> </ul> <p>Or combine them: "3d12h4m25s" # 3 days, 12 hours, 4 minutes, and 25 seconds</p> <p>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".</p> |
| <b>...</b>         | These dots are for future extensions and must be empty.  |
| <b>min_periods</b> | The number of values in the window that should be non-null before computing a result. If <code>NULL</code> (default), it will be set equal to <code>window_size</code> .   |
| <b>closed</b>      | Define which sides of the interval are closed (inclusive). Default is "right".   |
| <b>ddof</b>        | "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

A polars [expression](#)

**Examples**

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# 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 underlying floating point data by decimals digits*

---

**Description**

Round underlying floating point data by decimals digits

**Usage**

```
expr__round(decimals)
```

**Arguments**

**decimals**      Number of decimals to round by.

**Value**

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = c(0.33, 0.52, 1.02, 1.17))

df$with_columns(
  rounded = pl$col("a")$round(1)
)
```

---

`expr__round_sig_figs` *Round to a number of significant figures*

---

### Description

Round to a number of significant figures

### Usage

```
expr__round_sig_figs(digits)
```

### Arguments

`digits`            Number of significant figures to round to.

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = c(0.01234, 3.333, 1234))

df$with_columns(
  rounded = pl$col("a")$round_sig_figs(2)
)
```

---

`expr__sample`            *Sample from this expression*

---

### Description

Sample from this expression

**Usage**

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

**Arguments**

|                         |   |
|-------------------------|---|
| <b>n</b>                | Number of items to return. Cannot be used with <b>fraction</b> . Defaults to 1 if <b>fraction</b> is <b>NULL</b> .    |
| <b>...</b>              | These dots are for future extensions and must be empty.   |
| <b>fraction</b>         | Fraction of items to return. Cannot be used with <b>n</b> .   |
| <b>with_replacement</b> | Allow values to be sampled more than once.  |
| <b>shuffle</b>          | Shuffle the order of sampled data points.   |
| <b>seed</b>             | Seed for the random number generator. If <b>NULL</b> (default), a random seed is generated for each sample operation. |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a")$sample(
  fraction = 1, with_replacement = TRUE, seed = 1
))
```

---

**expr\_\_search\_sorted** *Find indices where elements should be inserted to maintain order*

---

**Description**

This returns -1 if x is lower than 0, 0 if x == 0, and 1 if x is greater than 0.

**Usage**

```
expr__search_sorted(element, side = c("any", "left", "right"))
```

**Arguments**

|                      |  |
|----------------------|--|
| <code>element</code> | Expression or scalar value.  |
| <code>side</code>    | Must be one of the following: <ul style="list-style-type: none"> <li>• "any": the index of the first suitable location found is given;</li> <li>• "left": the index of the leftmost suitable location found is given;</li> <li>• "right": the index the rightmost suitable location found is given.</li> </ul> |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(values = c(1, 2, 3, 5))
df$select(
  zero = pl$col("values")$search_sorted(0),
  three = pl$col("values")$search_sorted(3),
  six = pl$col("values")$search_sorted(6),
)
```

---

`expr__set_sorted`      *Flags the expression as "sorted"*

---

**Description**

Enables downstream code to user fast paths for sorted arrays.

**Warning:** This can lead to incorrect results if the data is NOT sorted!! Use with care!

**Usage**

```
expr__set_sorted(..., descending = FALSE)
```

**Arguments**

|                         |   |
|-------------------------|---|
| <code>...</code>        | These dots are for future extensions and must be empty. |
| <code>descending</code> | Whether the Series order is descending.                 |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a")$set_sorted()$max())
```

---

`expr__shift`                    *Shift values by the given number of indices*

---

### 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.

`...`                These dots are for future extensions and must be empty.

`fill_value`        Fill the resulting null values with this value.

### Value

A polars [expression](#)

### Examples

```
# By default, values are shifted forward by one index.
df <- pl$DataFrame(a = 1:4)
df$with_columns(shift = pl$col("a")$shift())

# Pass a negative value to shift in the opposite direction instead.
df$with_columns(shift = pl$col("a")$shift(-2))

# Specify fill_value to fill the resulting null values.
df$with_columns(shift = pl$col("a")$shift(-2, fill_value = 100))
```

---

`expr__shrink_dtype`           *Shrink numeric columns to the minimal required datatype*

---

### 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

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(a = c(-112, 2, 112))$cast(pl$Int64)
df$with_columns(
  shrunk = pl$col("a")$shrink_dtype()
)
```

---

expr\_\_shuffle

*Shuffle the contents of this expression*

---

## Description

Note this is shuffled independently of any other column or Expression. If you want each row to stay the same use `df$sample(shuffle = TRUE)`.

## Usage

```
expr__shuffle(seed = NULL)
```

## Arguments

**seed** Integer indicating the seed for the random number generator. If `NULL` (default), a random seed is generated each time the shuffle is called.

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(a = 1:3)
df$with_columns(
  shuffled = pl$col("a")$shuffle(seed = 1)
)
```

---

`expr__sign`*Compute the sign*

---

**Description**

This returns -1 if x is lower than 0, 0 if x == 0, and 1 if x is greater than 0.

**Usage**

```
expr__sign()
```

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(-9, 0, 0, 4, NA))
df$with_columns(sign = pl$col("a")$sign())
```

---

`expr__sin`*Compute sine*

---

**Description**

Compute sine

**Usage**

```
expr__sin()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(0, pi / 2, pi, NA))$
  with_columns(sine = pl$col("a")$sin())
```

---

|            |                                |
|------------|--------------------------------|
| expr__sinh | <i>Compute hyperbolic sine</i> |
|------------|--------------------------------|

---

### Description

Compute hyperbolic sine

### Usage

```
expr__sinh()
```

### Value

A polars [expression](#)

### Examples

```
pl$DataFrame(a = c(-1, asinh(0.5), 0, 1, NA))$  
  with_columns(sinh = pl$col("a")$sinh())
```

---

|            |                             |
|------------|-----------------------------|
| expr__skew | <i>Compute the skewness</i> |
|------------|-----------------------------|

---

### Description

For normally distributed data, the skewness should be about zero. For unimodal continuous distributions, a skewness value greater than zero means that there is more weight in the right tail of the distribution.

### Usage

```
expr__skew(..., bias = TRUE)
```

### Arguments

|      |  |
|------|--|
| ...  | These dots are for future extensions and must be empty.        |
| bias | If FALSE, the calculations are corrected for statistical bias. |

**Details**

The sample skewness is computed as the Fisher-Pearson coefficient of skewness, i.e.

$$g_1 = \frac{m_3}{m_2^{3/2}}$$

where

$$m_i = \frac{1}{N} \sum_{n=1}^N (x[n] - \bar{x})^i$$

is the biased sample  $i$ th central moment, and  $\bar{x}$  is the sample mean. If `bias = FALSE`, the calculations are corrected for bias and the value computed is the adjusted Fisher-Pearson standardized moment coefficient, i.e.

$$G_1 = \frac{k_3}{k_2^{3/2}} = \frac{\sqrt{N(N-1)}}{N-2} \frac{m_3}{m_2^{3/2}}$$

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(x = c(1, 2, 3, 2, 1))
df$select(pl$col("x")$skew())
```

---

|                          |                                       |
|--------------------------|---------------------------------------|
| <code>expr__slice</code> | <i>Get a slice of this expression</i> |
|--------------------------|---------------------------------------|

---

**Description**

Get a slice of this expression

**Usage**

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

**Arguments**

|                     |  |
|---------------------|--|
| <code>offset</code> | Numeric or expression, zero-indexed. Indicates where to start the slice. A negative value is one-indexed and starts from the end.    |
| <code>length</code> | Maximum number of elements contained in the slice. If <code>NULL</code> (default), all rows starting at the offset will be selected. |

**Value**

A polars [expression](#)

**Examples**

```
# as head
pl$DataFrame(a = 0:100)$select(
  pl$all()$slice(0, 6)
)

# as tail
pl$DataFrame(a = 0:100)$select(
  pl$all()$slice(-6, 6)
)

pl$DataFrame(a = 0:100)$select(
  pl$all()$slice(80)
)
```

---

```
expr__sort
```

```
Sort this expression
```

---

**Description**

If used in a groupby context, values within each group are sorted.

**Usage**

```
expr__sort(..., descending = FALSE, nulls_last = FALSE)
```

**Arguments**

... These dots are for future extensions and must be empty.

descending Sort in descending order.

nulls\_last Place null values last.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = c(6, 1, 0, NA, Inf, NaN))

df$with_columns(
  sorted = pl$col("a")$sort(),
  sorted_desc = pl$col("a")$sort(descending = TRUE),
  sorted_nulls_last = pl$col("a")$sort(nulls_last = TRUE)
)

# When sorting in a group by context, values in each group are sorted.
df <- pl$DataFrame(
  group = c("one", "one", "one", "two", "two", "two"),
```

```

    value = c(1, 98, 2, 3, 99, 4)
  )

df$group_by("group")$agg(pl$col("value")$sort())

```

---

|                            |   |
|----------------------------|---|
| <code>expr__sort_by</code> | <i>Sort this column by the ordering of another column, or multiple other columns.</i> |
|----------------------------|---|

---

### Description

If used in a `groupby` context, values within each group are sorted.

### Usage

```

expr__sort_by(
  ...,
  descending = FALSE,
  nulls_last = FALSE,
  multithreaded = TRUE,
  maintain_order = FALSE
)

```

### Arguments

|                             |  |
|-----------------------------|--|
| <code>...</code>            | < <a href="#">dynamic-dots</a> > Column(s) to sort by. Accepts expression input. Strings are parsed as column names.           |
| <code>descending</code>     | Sort in descending order. When sorting by multiple columns, can be specified per column by passing a sequence of booleans.     |
| <code>nulls_last</code>     | Place null values last; can specify a single boolean applying to all columns or a sequence of booleans for per-column control. |
| <code>multithreaded</code>  | Sort using multiple threads.   |
| <code>maintain_order</code> | Whether the order should be maintained if elements are equal.  |

### Value

A polars [expression](#)

### Examples

```

df <- pl$DataFrame(
  group = c("a", "a", "b", "b"),
  value1 = c(1, 3, 4, 2),
  value2 = c(8, 7, 6, 5)
)

# 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")$sort_by(
    "value2", pl$col("value1"),
    descending = c(TRUE, FALSE)
  )
)

# by some expression
df$with_columns(
  sorted = pl$col("group")$sort_by(pl$col("value1") + pl$col("value2"))
)

# in an aggregation context, values are sorted within groups
df$group_by("group")$agg(
  pl$col("value1")$sort_by("value2")
)

```

---

 expr\_\_sqrt

*Compute square root*


---

## Description

Compute square root

## Usage

```
expr__sqrt()
```

## Value

A polars [expression](#)

## Examples

```

pl$DataFrame(a = c(1, 2, 4))$
  with_columns(sqrt = pl$col("a")$sqrt())

```

---

|           |                                       |
|-----------|---------------------------------------|
| expr__std | <i>Compute the standard deviation</i> |
|-----------|---------------------------------------|

---

### Description

Compute the standard deviation

### Usage

```
expr__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 polars [expression](#)

### Examples

```
pl$DataFrame(a = c(1, 3, 5, 6))$  
  select(pl$all()$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

A polars [expression](#)

### 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**

A polars [expression](#)

**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**

A polars [expression](#)

**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**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(0, pi / 2, pi, NA))$  
  with_columns(tangent = pl$col("a")$tan())
```

---

|            |                                   |
|------------|-----------------------------------|
| expr__tanh | <i>Compute hyperbolic tangent</i> |
|------------|-----------------------------------|

---

**Description**

Compute hyperbolic tangent

**Usage**

```
expr__tanh()
```

**Value**

A polars [expression](#)

**Examples**

```
pl$DataFrame(a = c(-1, atanh(0.5), 0, 1, NA))$  
  with_columns(tanh = pl$col("a")$tanh())
```

---

`expr__top_k`                      *Return the k largest elements*

---

### Description

Non-null elements are always preferred over null elements. The output is not guaranteed to be in any particular order, call `$sort()` after this function if you wish the output to be sorted. This has time complexity  $O(n)$ .

### Usage

```
expr__top_k(k = 5)
```

### Arguments

`k`                      Number of elements to return.

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(value = c(1, 98, 2, 3, 99, 4))
df$select(
  top_k = pl$col("value")$top_k(k = 3),
  bottom_k = pl$col("value")$bottom_k(k = 3)
)
```

---

`expr__top_k_by`                      *Return the elements corresponding to the k largest elements of the by column(s)*

---

### Description

Non-null elements are always preferred over null elements. The output is not guaranteed to be in any particular order, call `$sort()` after this function if you wish the output to be sorted. This has time complexity  $O(n)$ .

### Usage

```
expr__top_k_by(by, k = 5, ..., reverse = FALSE)
```

**Arguments**

|                |   |
|----------------|---|
| <b>by</b>      | Column(s) used to determine the smallest elements. Accepts expression input. Strings are parsed as column names.  |
| <b>k</b>       | Number of elements to return.   |
| <b>...</b>     | These dots are for future extensions and must be empty.   |
| <b>reverse</b> | Consider the <b>k</b> smallest elements of the <b>by</b> column(s) (instead of the <b>k</b> largest). This can be specified per column by passing a sequence of booleans. |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = 1:6,
  b = 6:1,
  c = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")
)

# Get the top 2 rows by column a or b:
df$select(
  pl$all().$top_k_by("a", 2)$name$suffix("_btm_by_a"),
  pl$all().$top_k_by("b", 2)$name$suffix("_btm_by_b")
)

# Get the top 2 rows by multiple columns with given order.
df$select(
  pl$all().$
    top_k_by(c("c", "a"), 2, reverse = c(FALSE, TRUE))$
    name$suffix("_btm_by_ca"),
  pl$all().$
    top_k_by(c("c", "b"), 2, reverse = c(FALSE, TRUE))$
    name$suffix("_btm_by_cb"),
)

# Get the top 2 rows by column a in each group
df$group_by("c", .maintain_order = TRUE)$agg(
  pl$all().$top_k_by("a", 2)
)$explode(pl$all().$exclude("c"))
```

**Description**

The following data types will be changed:

- 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**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = factor(c("a", "x", NA, "a")))
df$with_columns(
  phys = pl$col("a")$to_physical()
)
```

---

|                |                               |
|----------------|-------------------------------|
| expr__true_div | <i>Divide two expressions</i> |
|----------------|-------------------------------|

---

**Description**

Method equivalent of float division operator `expr / other`. `$truediv()` is an alias for `$true_div()`, which exists for compatibility with Python Polars.

**Usage**

```
expr__true_div(other)
```

```
expr__truediv(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.:  $\pm$ infinity.

**Value**

A polars [expression](#)

**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")$true_div(2),
  `x/y` = pl$col("x")$true_div(pl$col("y"))
)
```

---

expr\_\_unique

*Get unique values*

---

**Description**

This method differs from `$value_counts()` in that it does not return the values, only the counts and might be faster.

**Usage**

```
expr__unique(..., maintain_order = FALSE)
```

**Arguments**

... These dots are for future extensions and must be empty.

maintain\_order Maintain order of data. This requires more work.

**Value**

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = c(1, 1, 2))
df$select(pl$col("a")$unique())
```

---

expr\_\_unique\_counts    *Count unique values in the order of appearance*

---

### Description

This method differs from `$value_counts()` in that it does not return the values, only the counts and might be faster.

### Usage

```
expr__unique_counts()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(id = c("a", "b", "b", "c", "c", "c"))
df$select(pl$col("id")$unique_counts())
```

---

expr\_\_upper\_bound    *Calculate the upper bound*

---

### Description

Returns a unit Series with the highest value possible for the dtype of this expression.

### Usage

```
expr__upper_bound()
```

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a")$upper_bound())
```

---

`expr__value_counts`     *Count the occurrences of unique values*

---

## Description

Count the occurrences of unique values

## Usage

```
expr__value_counts(
  ...,
  sort = FALSE,
  parallel = FALSE,
  name = NULL,
  normalize = FALSE
)
```

## Arguments

|                        |   |
|------------------------|---|
| <code>...</code>       | These dots are for future extensions and must be empty.   |
| <code>sort</code>      | Sort the output by count in descending order. If <code>FALSE</code> (default), the order of the output is random.   |
| <code>parallel</code>  | Execute the computation in parallel. This option should likely not be enabled in a group by context, as the computation is already parallelized per group.                                |
| <code>name</code>      | Give the resulting count field a specific name. If <code>normalize</code> is <code>TRUE</code> it defaults to <code>"proportion"</code> , otherwise it defaults to <code>"count"</code> . |
| <code>normalize</code> | If <code>TRUE</code> , gives relative frequencies of the unique values.   |

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(color = c("red", "blue", "red", "green", "blue", "blue"))
df$select(pl$col("color")$value_counts())

# Sort the output by (descending) count and customize the count field name.
df <- df$select(pl$col("color")$value_counts(sort = TRUE, name = "n"))
df

df$unnest("color")
```



---

expr\_\_var                    *Compute the variance*

---

### Description

Compute the variance

### Usage

```
expr__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 polars [expression](#)

### Examples

```
pl$DataFrame(a = c(1, 3, 5, 6))$  
  select(pl$all()$var())
```

---

expr\_\_xor                    *Apply logical XOR on two expressions*

---

### Description

Combine two boolean expressions with XOR.

### Usage

```
expr__xor(other)
```

### Arguments

other                    Element to add. Can be a string (only if `expr` is a string), a numeric value or an other expression.

### Value

A polars [expression](#)

### Examples

```
pl$lit(TRUE)$xor(pl$lit(FALSE))
```

---

infer\_polars\_dtype     *Infer Polars DataType corresponding to a given R object*

---

## Description

`infer_polars_dtype()` is a helper function used to quickly find the `DataType` corresponding to an R object, in other words, it infers the type of the Polars `Series` that would be constructed from the object. In many cases, this function simply performs something like `head(x, 0) |> as_polars_series()`. It is much faster than actually constructing a `Series` using the entire object. This function is similar to `nanoarrow::infer_nanoarrow_schema()`. `is_convertible_to_polars_series()` and `is_convertible_to_polars_expr()` are helper functions that check if the object can be converted to a `Series` or `Expr` respectively. These functions call `infer_polars_dtype()` internally and return `TRUE` if the type can be inferred without error. (Or, that object is already a Polars `Expr` for `is_convertible_to_polars_expr()`.)

## Usage

```
infer_polars_dtype(x, ...)

is_convertible_to_polars_series(x, ...)

is_convertible_to_polars_expr(x, ...)

## Default S3 method:
infer_polars_dtype(x, ...)

## S3 method for class 'polars_series'
infer_polars_dtype(x, ...)

## S3 method for class 'polars_data_frame'
infer_polars_dtype(x, ...)

## S3 method for class 'polars_lazy_frame'
infer_polars_dtype(x, ...)

## S3 method for class '`NULL`'
infer_polars_dtype(x, ...)

## S3 method for class 'list'
infer_polars_dtype(x, ..., strict = FALSE, infer_dtype_length = 10L)

## S3 method for class 'AsIs'
infer_polars_dtype(x, ...)

## S3 method for class 'data.frame'
infer_polars_dtype(x, ...)
```

```
## S3 method for class 'nanoarrow_array_stream'
infer_polars_dtype(x, ...)

## S3 method for class 'nanoarrow_array'
infer_polars_dtype(x, ...)

## S3 method for class 'RecordBatchReader'
infer_polars_dtype(x, ...)

## S3 method for class 'ArrowTabular'
infer_polars_dtype(x, ...)

## S3 method for class 'vctrs_vctr'
infer_polars_dtype(x, ...)
```

### Arguments

|                                 |   |
|---------------------------------|---|
| <code>x</code>                  | An R object.  |
| <code>...</code>                | Additional arguments passed to the methods.   |
| <code>strict</code>             | A logical value to indicate whether throwing an error when the input <a href="#">list</a> 's elements have different data types. If <code>FALSE</code> (default), all elements are automatically cast to the super type, or, casting to the super type is failed, the value will be <code>null</code> . If <code>TRUE</code> , the first non-NULL element's data type is used as the data type of the inner Series. |
| <code>infer_dtype_length</code> | The number of non-NULL elements to use for type inference. Must be a single positive integer-ish value. The default is 10. If you want to infer the type of the entire list, set this to <code>Inf</code> , but be aware that it may be slow.   |

### Details

S3 objects based on atomic vectors or classes built on [the vctrs package](#) will work accurately if the S3 method of the `as_polars_series()` function is defined.

### Value

A [polars DataType](#)

### See Also

- [as\\_polars\\_series\(\)](#)
- [check\\_polars](#): Functions to check if the object is a polars object.

### Examples

```
infer_polars_dtype(1:10)

# The type inference is also fast for objects
```

```

# that would take a long time to construct a Series.
infer_polars_dtype(1:100000000)

# For lists, it is not possible to infer the type
# without inspecting all elements.
# However, this function can be configured to inspect only a few elements
# via the `infer_dtype_length` argument.
# If a sufficient length is specified, the correct type can be inferred.
# (By default, the length is set to 10.)
mixed_list <- list(1, NULL, "foo")
infer_polars_dtype(mixed_list)
infer_polars_dtype(mixed_list, infer_dtype_length = 2)

# But if the length is too short, an incorrect type may be inferred.
infer_polars_dtype(mixed_list, infer_dtype_length = 1)

# is_convertible_to_polars_* functions are useful for checking if
# the object can be converted to a Series or Expr quickly.
try(infer_polars_dtype(1i))
is_convertible_to_polars_series(1i)
is_convertible_to_polars_expr(1i)

# For polars Expr objects, infer_polars_dtype() will raise an error
# because Expr can't be converted to a Series by `as_polars_series()`.
try(infer_polars_dtype(pl$lit(1)))
is_convertible_to_polars_series(pl$lit(1))
is_convertible_to_polars_expr(pl$lit(1))

```

---

lazyframe\_\_bottom\_k *Return the k smallest rows*

---

## Description

Non-null elements are always preferred over null elements, regardless of the value of **reverse**. The output is not guaranteed to be in any particular order, call **sort()** after this function if you wish the output to be sorted.

## Usage

```
lazyframe__bottom_k(k, ..., by, reverse = FALSE)
```

## Arguments

|                |  |
|----------------|--|
| <b>k</b>       | Number of rows to return.  |
| <b>...</b>     | These dots are for future extensions and must be empty.  |
| <b>by</b>      | Column(s) used to determine the bottom rows. Accepts expression input. Strings are parsed as column names.   |
| <b>reverse</b> | Consider the k largest elements of the by column(s) (instead of the k smallest). This can be specified per column by passing a sequence of booleans. |

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(
  a = c("a", "b", "a", "b", "b", "c"),
  b = c(2, 1, 1, 3, 2, 1)
)

# Get the rows which contain the 4 smallest values in column b.
lf$bottom_k(4, by = "b")$collect()

# Get the rows which contain the 4 smallest values when sorting on column a
# and b$
lf$bottom_k(4, by = c("a", "b"))$collect()
```

---

|                              |   |
|------------------------------|---|
| <code>lazyframe__cast</code> | <i>Cast LazyFrame column(s) to the specified dtype(s)</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(..., .strict = TRUE)
```

**Arguments**

|                      |  |
|----------------------|--|
| <code>...</code>     | < <a href="#">dynamic-dots</a> > Either a datatype to which all columns will be cast, or a list where the names are column names and the values are the datatypes to convert to. |
| <code>.strict</code> | If TRUE (default), throw an error if a cast could not be done (for instance, due to an overflow). Otherwise, return <code>null</code> .  |

**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$cast(foo = pl$Float32, bar = pl$UInt8)$collect()  
  
# Cast all columns to the same type  
lf$cast(pl$string)$collect()
```

---

|                  |  |
|------------------|--|
| lazyframe__clear | Create an empty or n-row null-filled copy of the frame |
|------------------|--|

---

## Description

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

## Usage

```
lazyframe__clear(n = 0)
```

## Arguments

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

## Value

A polars [LazyFrame](#)

## 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()$collect()  
  
df$clear(n = 2)$collect()
```

---

|                  |                          |
|------------------|--------------------------|
| lazyframe__clone | <i>Clone a LazyFrame</i> |
|------------------|--------------------------|

---

## 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 polars [LazyFrame](#)

## Examples

```
df1 <- as_polars_lf(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 <- as_polars_lf(iris)
df2 <- give_attr(df1)

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

---

`lazyframe__collect`     *Materialize this LazyFrame into a DataFrame*

---

## Description

By default, all query optimizations are enabled. Individual optimizations may be disabled by setting the corresponding parameter to `FALSE`.

## Usage

```
lazyframe__collect(
  ...,
  type_coercion = TRUE,
  `_type_check` = 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,
  collapse_joins = TRUE,
  no_optimization = FALSE,
  streaming = FALSE,
  `_check_order` = TRUE,
  `_eager` = FALSE
)
```

## Arguments

`...`                    These dots are for future extensions and must be empty.

`type_coercion`     A logical, indicates type coercion optimization.

`predicate_pushdown`  
                      A logical, indicates predicate pushdown optimization.

`projection_pushdown`  
                      A logical, indicates projection pushdown optimization.

`simplify_expression`  
                      A logical, indicates simplify expression optimization.

`slice_pushdown`  
                      A logical, indicates slice pushdown optimization.

`comm_subplan_elim`  
                      A logical, indicates try to cache branching subplans that occur on self-joins or unions.

`comm_subexpr_elim`  
                      A logical, indicates try to cache common subexpressions.



|  |  |
|--|--|
| <code>cluster_with_columns</code>      | A logical, indicates to combine sequential independent calls to <code>with_columns</code> .  |
| <code>collapse_joins</code>            | Collapse a join and filters into a faster join.  |
| <code>no_optimization</code>           | A logical. If <code>TRUE</code> , turn off (certain) optimizations.  |
| <code>streaming</code>                 | A logical. If <code>TRUE</code> , process the query in batches to handle larger-than-memory data. If <code>FALSE</code> (default), the entire query is processed in a single batch. Note that streaming mode is considered unstable. It may be changed at any point without it being considered a breaking change. |
| <code>_check_order, _type_check</code> | For internal use only.   |
| <code>_eager</code>                    | A logical, indicates to turn off multi-node optimizations and the other optimizations. This option is intended for internal use only.  |

## Value

A polars [DataFrame](#)

## See Also

- [\\$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

```
lf <- pl$LazyFrame(
  a = c("a", "b", "a", "b", "b", "c"),
  b = 1:6,
  c = 6:1,
)
lf$group_by("a")$agg(pl$all()$sum())$collect()

# Collect in streaming mode
lf$group_by("a")$agg(pl$all()$sum())$collect(
  streaming = TRUE
)
```

---

```
lazyframe__collect_schema
```

*Resolve the schema of this LazyFrame*

---

### Description

This resolves the query plan but does not trigger computations.

### Usage

```
lazyframe__collect_schema()
```

### Value

A named list with names indicating column names and values indicating column data types.

### Examples

```
lf <- pl$LazyFrame(
  foo = 1:3,
  bar = 6:8,
  ham = c("a", "b", "c")
)

lf$collect_schema()

lf$with_columns(
  baz = (pl$col("foo") + pl$col("bar"))$cast(pl$String),
  pl$col("bar")$cast(pl$Int64)
)$collect_schema()
```

---

```
lazyframe__count
```

*Return the number of non-null elements for each column*

---

### Description

Return the number of non-null elements for each column

### Usage

```
lazyframe__count()
```

### Value

A polars [LazyFrame](#)

## Examples

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, NA), c = rep(NA, 4))
lf$count().collect()
```

---

|                 |                       |
|-----------------|-----------------------|
| lazyframe__drop | <i>Remove columns</i> |
|-----------------|-----------------------|

---

## Description

Remove columns

## Usage

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

## Arguments

|        |  |
|--------|--|
| ...    | < <a href="#">dynamic-dots</a> > Names of the columns that should be removed. Accepts column selector input. |
| strict | Validate that all column names exist in the current schema, and throw an exception if any do not.            |

## Value

A polars [LazyFrame](#)

## Examples

```
# Drop columns by passing the name of those columns
lf <- pl$LazyFrame(
  foo = 1:3,
  bar = c(6, 7, 8),
  ham = c("a", "b", "c")
)
lf$drop("ham").collect()
lf$drop("ham", "bar").collect()

# Drop multiple columns by passing a selector
lf$drop(cs$all()).collect()
```

---

lazyframe\_\_drop\_nans *Drop all rows that contain NaN values*

---

### Description

The original order of the remaining rows is preserved.

### Usage

```
lazyframe__drop_nans(...)
```

### Arguments

... `<dynamic-dots>` Column name(s) for which null values are considered. If empty (default), use all columns (note that only floating-point columns can contain NaNs).

### Value

A polars [LazyFrame](#)

### Examples

```
lf <- pl$LazyFrame(
  foo = c(1, NaN, 2.5),
  bar = c(NaN, 110, 25.5),
  ham = c("a", "b", NA)
)

# The default behavior of this method is to drop rows where any single value
# of the row is null.
lf$drop_nans()$collect()

# This behaviour can be constrained to consider only a subset of columns, as
# defined by name or with a selector. For example, dropping rows if there is
# a null in the "bar" column:
lf$drop_nans("bar")$collect()

# Dropping a row only if *all* values are NaN requires a different
# formulation:
df <- pl$LazyFrame(
  a = c(NaN, NaN, NaN, NaN),
  b = c(10.0, 2.5, NaN, 5.25),
  c = c(65.75, NaN, NaN, 10.5)
)
df$filter(!pl$all_horizontal(pl$all()$is_nan()))$collect()
```

---

 lazyframe\_\_drop\_nulls

*Drop all rows that contain null values*


---

### Description

The original order of the remaining rows is preserved.

### Usage

```
lazyframe__drop_nulls(...)
```

### Arguments

... [<dynamic-dots>](#) Column name(s) for which null values are considered. If empty (default), use all columns.

### Value

A polars [LazyFrame](#)

### Examples

```
lf <- pl$LazyFrame(
  foo = 1:3,
  bar = c(6L, NA, 8L),
  ham = c("a", "b", NA)
)

# The default behavior of this method is to drop rows where any single value
# of the row is null.
lf$drop_nulls()$collect()

# This behaviour can be constrained to consider only a subset of columns, as
# defined by name or with a selector. For example, dropping rows if there is
# a null in any of the integer columns:
lf$drop_nulls(cs$integer())$collect()
```

---

 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 = c("plain", "tree"),
  optimized = TRUE,
  type_coercion = TRUE,
  `_type_check` = 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,
  collapse_joins = TRUE,
  streaming = FALSE,
  `_check_order` = TRUE
)

```

**Arguments**

|                                   |  |
|-----------------------------------|--|
| <code>...</code>                  | These dots are for future extensions and must be empty.  |
| <code>format</code>               | The format to use for displaying the logical plan. Must be either <b>"plain"</b> (default) or <b>"tree"</b> .  |
| <code>optimized</code>            | Return an optimized query plan. If <b>TRUE</b> (default), the subsequent optimization flags control which optimizations run.                                     |
| <code>type_coercion</code>        | A logical, indicates type coercion optimization.   |
| <code>predicate_pushdown</code>   | A logical, indicates predicate pushdown optimization.  |
| <code>projection_pushdown</code>  | A logical, indicates projection pushdown optimization.   |
| <code>simplify_expression</code>  | A logical, indicates simplify expression optimization.   |
| <code>slice_pushdown</code>       | A logical, indicates slice pushdown optimization.  |
| <code>comm_subplan_elim</code>    | A logical, indicates try to cache branching subplans that occur on self-joins or unions.   |
| <code>comm_subexpr_elim</code>    | A logical, indicates try to cache common subexpressions.   |
| <code>cluster_with_columns</code> | A logical, indicates to combine sequential independent calls to <code>with_columns</code> .  |
| <code>collapse_joins</code>       | Collapse a join and filters into a faster join.  |
| <code>streaming</code>            | A logical. If <b>TRUE</b> , process the query in batches to handle larger-than-memory data. If <b>FALSE</b> (default), the entire query is processed in a single |

batch. Note that streaming mode is considered unstable. It may be changed at any point without it being considered a breaking change.

`_check_order`, `_type_check`

For internal use only.

## Value

A character value containing the query plan.

## Examples

```
lazy_frame <- as_polars_lf(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` *Explode the frame to long format by exploding the given columns*

---

## Description

Explode the frame to long format by exploding the given columns

## Usage

```
lazyframe__explode(...)
```

## Arguments

... [<dynamic-dots>](#) Column names, expressions, or a selector defining them. The underlying columns being exploded must be of the `List` or `Array` data type.

## Value

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(
  letters = c("a", "a", "b", "c"),
  numbers = list(1, c(2, 3), c(4, 5), c(6, 7, 8))
)

lf$explode("numbers")$collect()
```

---

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**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(
  a = c(1.5, 2, NaN, 4),
  b = c(1.5, NaN, NaN, 4)
)
lf$fill_nan(99)$collect()
```

---

lazyframe\_\_fill\_null *Fill null values using the specified value or strategy*

---

**Description**

Fill null values using the specified value or strategy



**Usage**

```
lazyframe__fill_null(
  value,
  strategy = NULL,
  limit = NULL,
  ...,
  matches_supertype = TRUE
)
```

**Arguments**

|                                |   |
|--------------------------------|---|
| <code>value</code>             | Value used to fill null values.   |
| <code>strategy</code>          | Strategy used to fill null values. Must be one of: "forward", "backward", "min", "max", "mean", "zero", "one", or NULL (default). |
| <code>limit</code>             | Number of consecutive null values to fill when using the "forward" or "backward" strategy.  |
| <code>...</code>               | These dots are for future extensions and must be empty.   |
| <code>matches_supertype</code> | Fill all matching supertypes of the fill <code>value</code> literal.  |

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(
  a = c(1.5, 2, NA, 4),
  b = c(1.5, NA, NA, 4)
)
lf$fill_null(99)$collect()

lf$fill_null(strategy = "forward")$collect()

lf$fill_null(strategy = "max")$collect()

lf$fill_null(strategy = "zero")$collect()
```

---

|                                |   |
|--------------------------------|---|
| <code>lazyframe__filter</code> | <i>Filter the rows in the LazyFrame based on a predicate expression</i> |
|--------------------------------|---|

---

**Description**

The original order of the remaining rows is preserved. Rows where the filter does not evaluate to TRUE are discarded, including nulls.

**Usage**

```
lazyframe__filter(...)
```

**Arguments**

... [<dynamic-dots>](#) Expression that evaluates to a boolean Series.

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(
  foo = c(1, 2, 3, NA, 4, NA, 0),
  bar = c(6, 7, 8, NA, NA, 9, 0),
  ham = c("a", "b", "c", NA, "d", "e", "f")
)

# Filter on one condition
lf$filter(pl$col("foo") > 1)$collect()

# Filter on multiple conditions
lf$filter((pl$col("foo") < 3) & (pl$col("ham") == "a"))$collect()

# Filter on an OR condition
lf$filter((pl$col("foo") == 1) | (pl$col("ham") == "c"))$collect()

# Filter by comparing two columns against each other
lf$filter(pl$col("foo") == pl$col("bar"))$collect()
lf$filter(pl$col("foo") != pl$col("bar"))$collect()

# Notice how the row with null values is filtered out$ In order to keep the
# rows with nulls, use:
lf$filter(pl$col("foo")$ne_missing(pl$col("bar")))$collect()
```

---

lazyframe\_\_first      *Get the first row of the LazyFrame*

---

**Description**

Get the first row of the LazyFrame

**Usage**

```
lazyframe__first()
```

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, 1))
lf$first().$collect()
```

---

lazyframe\_\_gather\_every

*Take every nth row in the LazyFrame*

---

**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 polars [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 *Start a group by operation*

---

**Description**

Start a group by operation

**Usage**

```
lazyframe__group_by(..., .maintain_order = FALSE)
```

**Arguments**

... <dynamic-dots> Column(s) to group by. Accepts expression input. Strings 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.

**Value**

A lazy groupby

**Examples**

```
# Group by one column and call agg() to compute the grouped sum of another
# column.
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$col("b")$sum())$collect()

# Group by multiple columns by passing a vector of column names.
lf$group_by(c("a", "b"))$agg(pl$col("c")$max())$collect()

# Or use positional 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()
```

---

lazyframe\_\_group\_by\_dynamic

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

---

**Description**

Time windows are calculated and rows are assigned to windows. Different from a normal group by is that a row can be member of multiple groups. By default, the windows look like:

- [start, start + period)
- [start + every, start + every + period)
- [start + 2every, start + 2every + period)
- ...

**Usage**

```

lazyframe__group_by_dynamic(
  index_column,
  ...,
  every,
  period = NULL,
  offset = NULL,
  include_boundaries = FALSE,
  closed = c("left", "right", "both", "none"),
  label = c("left", "right", "datapoint"),
  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>group_by</code> is specified, then it must be sorted in ascending order within each group). In case of a dynamic group by on indices, the data type 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>                | These dots are for future extensions and must be empty.  |
| <code>every</code>              | Interval of the window.  |
| <code>period</code>             | Length of the window. If NULL (default), it will equal <code>every</code> .  |
| <code>offset</code>             | Offset of the window, does not take effect if <code>start_by = "datapoint"</code> . Defaults to zero.  |
| <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 interval are closed (inclusive). Default is " <code>left</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. Can be expressions or objects coercible to expressions.   |
| <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

where `start` is determined by `start_by`, `offset`, `every`, and the earliest datapoint. See the `start_by` argument description for details.

The `every`, `period`, and `offset` arguments are created with 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

In case of a `group_by_dynamic` 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$select(
  time = pl$datetime_range(
    start = as.POSIXct(strptime("2021-12-16 00:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC")),
    end = as.POSIXct(strptime("2021-12-16 03:00:00", format = "%Y-%m-%d %H:%M:%S", tz = "UTC")),
    interval = "30m"
  ),
  n = 0:6
)$lazy()
lf$collect()

# Group by windows of 1 hour.
lf$group_by_dynamic("time", every = "1h", closed = "right")$agg(
```

```

    vals = pl$col("n")
  )$collect()

# The window boundaries can also be added to the aggregation result
lf$group_by_dynamic(
  "time",
  every = "1h", include_boundaries = TRUE, closed = "right"
)$agg(
  pl$col("n")$mean()
)$collect()

# When closed = "left", the window excludes the right end of interval:
# [lower_bound, upper_bound)
lf$group_by_dynamic("time", every = "1h", closed = "left")$agg(
  pl$col("n")
)$collect()

# When closed = "both" the time values at the window boundaries belong to 2
# groups.
lf$group_by_dynamic("time", every = "1h", closed = "both")$agg(
  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**

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

**Usage**

```
lazyframe__head(n = 5)
```

```
lazyframe__limit(n = 5)
```

**Arguments**

`n`                      Number of rows to return.

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(a = 1:6, b = 7:12)
lf$head()$collect()
lf$head(2)$collect()
```

---

lazyframe\_\_interpolate  
*Interpolate intermediate values*

---

**Description**

The interpolation method is linear.

**Usage**

```
lazyframe__interpolate()
```

**Value**

A polars [LazyFrame](#)



## Examples

```
lf <- pl$LazyFrame(
  foo = c(1, NA, 9, 10),
  bar = c(6, 7, 9, NA),
  ham = c(1, NA, NA, 9)
)

lf$interpolate()$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 = c("inner", "full", "left", "right", "semi", "anti", "cross"),
  ...,
  left_on = NULL,
  right_on = NULL,
  suffix = "_right",
  validate = c("m:m", "1:m", "m:1", "1:1"),
  join_nulls = FALSE,
  maintain_order = c("none", "left", "right", "left_right", "right_left"),
  allow_parallel = TRUE,
  force_parallel = FALSE,
  coalesce = NULL
)
```

## Arguments

|              |   |
|--------------|---|
| <b>other</b> | LazyFrame to join with.   |
| <b>on</b>    | 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 LazyFrames.   |
| <b>how</b>   | One of the following methods: <ul style="list-style-type: none"> <li>• "inner": returns rows that have matching values in both tables</li> <li>• "left": returns all rows from the left table, and the matched rows from the right table</li> </ul> |

- "right": returns all rows from the right table, and the matched rows from the left table
- "full": returns all rows when there is a match in either left or right table
- "cross": returns the Cartesian product of rows from both tables
- "semi": returns rows from the left table that have a match in the right table.
- "anti": returns rows from the left table that have no match in the right table.

... These dots are for future extensions and must be empty.

**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:

- "m:m" (default): many-to-many, doesn't perform any checks;
- "1:1": one-to-one, check if join keys are unique in both left and right datasets;
- "1:m": one-to-many, check if join keys are unique in left dataset
- "m:1": many-to-one, check if join keys are unique in right dataset

Note that this is currently not supported by the streaming engine.

**join\_nulls** Join on null values. By default null values will never produce matches.

**maintain\_order** Which frame row order to preserve, if any. Do not rely on any observed ordering without explicitly setting this parameter, as your code may break in a future release. Not specifying any ordering can improve performance. Supported for inner, left, right and full joins.

- "none": No specific ordering is desired. The ordering might differ across Polars versions or even between different runs.
- "left": Preserves the order of the left frame.
- "right": Preserves the order of the right frame.
- "left\_right": First preserves the order of the left frame, then the right.
- "right\_left": First preserves the order of the right frame, then the left.

**allow\_parallel** Allow the physical plan to optionally evaluate the computation of both LazyFrames up to the join in parallel.

**force\_parallel** Force the physical plan to evaluate the computation of both LazyFrames up to the join in parallel.

**coalesce** Coalescing behavior (merging of join columns).

- NULL: join specific.
- TRUE: Always coalesce join columns.
- FALSE: Never coalesce join columns. Note that joining on any other expressions than col will turn off coalescing.

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(
  foo = 1:3,
  bar = c(6, 7, 8),
  ham = c("a", "b", "c")
)
other_lf <- pl$LazyFrame(
  apple = c("x", "y", "z"),
  ham = c("a", "b", "d")
)
lf$join(other_lf, on = "ham")$collect()

lf$join(other_lf, on = "ham", how = "full")$collect()

lf$join(other_lf, on = "ham", how = "left", coalesce = TRUE)$collect()

lf$join(other_lf, on = "ham", how = "semi")$collect()

lf$join(other_lf, on = "ham", how = "anti")$collect()
```

---

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. Both frames must be sorted by the `asof_join` key.

**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,
    allow_exact_matches = TRUE,
    check_sortedness = TRUE
)

```

### Arguments

|                                |   |
|--------------------------------|---|
| <code>other</code>             | LazyFrame to join with.   |
| <code>...</code>               | These dots are for future extensions and must be empty.   |
| <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 LazyFrames.   |
| <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>• <code>"backward"</code> (default): search for the last row in the right table whose <code>on</code> key is less than or equal to the left key.</li> <li>• <code>"forward"</code>: search for the first row in the right table whose <code>on</code> key is greater than or equal to the left key.</li> <li>• <code>"nearest"</code>: 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 <code>"Date"</code> , <code>"Datetime"</code> , <code>"Duration"</code> or <code>"Time"</code> , use the Polars duration string language (see details).  |
| <code>allow_parallel</code>    | Allow the physical plan to optionally evaluate the computation of both LazyFrames up to the join in parallel.   |
| <code>force_parallel</code>    | Force the physical plan to evaluate the computation of both LazyFrames 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>• <code>TRUE</code>: Always coalesce join columns;</li> <li>• <code>FALSE</code>: Never coalesce join columns. Note that joining on any other expressions than <code>col</code> will turn off coalescing.</li> </ul>  |

**allow\_exact\_matches**

Whether exact matches are valid join predicates. If **TRUE** (default), allow matching with the same on value (i.e. less-than-or-equal-to / greater-than-or-equal-to). Otherwise, don't match the same on value (i.e., strictly less-than / strictly greater-than).

**check\_sortedness**

Check the sortedness of the asof keys. If the keys are not sorted, polars will error, or raise a warning if the **by** argument is provided. This might become a hard error in the future.

**Value**

A polars [LazyFrame](#)

**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**

```
gdp <- pl$LazyFrame(
  date = as.Date(c("2016-1-1", "2017-5-1", "2018-1-1", "2019-1-1", "2020-1-1")),
  gdp = c(4164, 4411, 4566, 4696, 4827)
)
```

```
pop <- pl$LazyFrame(
  date = as.Date(c("2016-3-1", "2018-8-1", "2019-1-1")),
  population = c(82.19, 82.66, 83.12)
)
```

```

)

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

# Note how the dates don't quite match. If we join them using join_asof and
# strategy = 'backward', then each date from population which doesn't have
# an exact match is matched with the closest earlier date from gdp:
pop$join_asof(gdp, on = "date", strategy = "backward")$collect()

# Note how:
# - date 2016-03-01 from population is matched with 2016-01-01 from gdp;
# - date 2018-08-01 from population is matched with 2018-01-01 from gdp.
# You can verify this by passing coalesce = FALSE:
pop$join_asof(
  gdp,
  on = "date", strategy = "backward", coalesce = FALSE
)$collect()

# If we instead use strategy = 'forward', then each date from population
# which doesn't have an exact match is matched with the closest later date
# from gdp:
pop$join_asof(gdp, on = "date", strategy = "forward")$collect()

# Note how:
# - date 2016-03-01 from population is matched with 2017-01-01 from gdp;
# - date 2018-08-01 from population is matched with 2019-01-01 from gdp.

# Finally, strategy = 'nearest' gives us a mix of the two results above, as
# each date from population which doesn't have an exact match is matched
# with the closest date from gdp, regardless of whether it's earlier or
# later:
pop$join_asof(gdp, on = "date", strategy = "nearest")$collect()

# Note how:
# - date 2016-03-01 from population is matched with 2016-01-01 from gdp;
# - date 2018-08-01 from population is matched with 2019-01-01 from gdp.

# The `by` argument allows joining on another column first, before the asof
# join. In this example we join by country first, then asof join by date, as
# above.
gdp2 <- pl$LazyFrame(
  country = rep(c("Germany", "Netherlands"), each = 5),
  date = rep(
    as.Date(c("2016-1-1", "2017-1-1", "2018-1-1", "2019-1-1", "2020-1-1")),
    2
  ),
  gdp = c(4164, 4411, 4566, 4696, 4827, 784, 833, 914, 910, 909)
)$sort("country", "date")
gdp2$collect()

```

```

pop2 <- pl$LazyFrame(
  country = rep(c("Germany", "Netherlands"), each = 3),
  date = rep(as.Date(c("2016-3-1", "2018-8-1", "2019-1-1")), 2),
  population = c(82.19, 82.66, 83.12, 17.11, 17.32, 17.40)
)$sort("country", "date")
pop2$collect()

pop2$join_asof(
  gdp2,
  by = "country", on = "date", strategy = "nearest"
)$collect()

```

---

lazyframe\_\_join\_where

*Perform a join based on one or multiple (in)equality predicates*

---

## Description

### [Experimental]

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

|                     |  |
|---------------------|--|
| <code>other</code>  | LazyFrame to join with.  |
| <code>...</code>    | <dynamic-dots> (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>". |
| <code>suffix</code> | Suffix to append to columns with a duplicate name.   |

## Value

A polars [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

*Get the last row of the LazyFrame*

---

### Description

Get the last row of the LazyFrame

### Usage

```
lazyframe__last()
```

### Value

A polars [LazyFrame](#)

### Examples

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, 1))
lf$last()$collect()
```

---

lazyframe\_\_max

*Aggregate the columns in the LazyFrame to their maximum value*

---

### Description

Aggregate the columns in the LazyFrame to their maximum value

### Usage

```
lazyframe__max()
```



**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, 1))
lf$max()$collect()
```

---

|                 |   |
|-----------------|---|
| lazyframe__mean | <i>Aggregate the columns in the LazyFrame to their mean value</i> |
|-----------------|---|

---

**Description**

Aggregate the columns in the LazyFrame to their mean value

**Usage**

```
lazyframe__mean()
```

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, 1))
lf$mean()$collect()
```

---

|                   |   |
|-------------------|---|
| lazyframe__median | <i>Aggregate the columns in the LazyFrame to their median value</i> |
|-------------------|---|

---

**Description**

Aggregate the columns in the LazyFrame to their median value

**Usage**

```
lazyframe__median()
```

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, 1))
lf$median()$collect()
```

---

lazyframe\_\_merge\_sorted

*Take two sorted LazyFrames and merge them by the sorted key*

---

### Description

The output of this operation will also be sorted. It is the callers responsibility that the frames are sorted by that key, otherwise the output will not make sense. The schemas of both LazyFrames must be equal.

### Usage

```
lazyframe__merge_sorted(other, key)
```

### Arguments

|                    |                                      |
|--------------------|--------------------------------------|
| <code>other</code> | Other LazyFrame that must be merged. |
| <code>key</code>   | Key that is sorted.                  |

### Value

A polars [LazyFrame](#)

### Examples

```
lf1 <- pl$LazyFrame(
  name = c("steve", "elise", "bob"),
  age = c(42, 44, 18)
)$sort("age")

lf2 <- pl$LazyFrame(
  name = c("anna", "megan", "steve", "thomas"),
  age = c(21, 33, 42, 20)
)$sort("age")

lf1$merge_sorted(lf2, key = "age")$collect()
```

---

lazyframe\_\_min

*Aggregate the columns in the LazyFrame to their minimum value*

---

### Description

Aggregate the columns in the LazyFrame to their minimum value

### Usage

```
lazyframe__min()
```

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, 1))
lf$min()$collect()
```

---

lazyframe\_\_null\_count

*Return the number of null elements for each column*

---

**Description**

Return the number of null elements for each column

**Usage**

```
lazyframe__null_count()
```

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, NA), c = rep(NA, 4))
lf>null_count()$collect()
```

---

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,
  `__type_check` = 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,
  collapse_joins = TRUE,
  streaming = FALSE,
  no_optimization = FALSE,
  `__check_order` = TRUE,
  show_plot = FALSE,
  truncate_nodes = 0
)

```

**Arguments**

... These dots are for future extensions and must be empty.

**type\_coercion** A logical, indicates type coercion optimization.

**predicate\_pushdown** A logical, indicates predicate pushdown optimization.

**projection\_pushdown** A logical, indicates projection pushdown optimization.

**simplify\_expression** A logical, indicates simplify expression optimization.

**slice\_pushdown** A logical, indicates slice pushdown optimization.

**comm\_subplan\_elim** A logical, indicates try to cache branching subplans that occur on self-joins or unions.

**comm\_subexpr\_elim** A logical, indicates try to cache common subexpressions.

**cluster\_with\_columns** A logical, indicates to combine sequential independent calls to `with_columns`.

**collapse\_joins** Collapse a join and filters into a faster join.

**streaming** A logical. If `TRUE`, process the query in batches to handle larger-than-memory data. If `FALSE` (default), the entire query is processed in a single batch. Note that streaming mode is considered unstable. It may be changed at any point without it being considered a breaking change.

**no\_optimization** A logical. If `TRUE`, turn off (certain) optimizations.

|  |  |
|--|--|
| <code>_check_order</code> , <code>_type_check</code> | For internal use only.   |
| <code>show_plot</code>                               | Show a Gantt chart of the profiling result   |
| <code>truncate_nodes</code>                          | 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

- `$collect()` - regular collect.
- `$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
as_polars_lf(iris)$
  sort("Sepal.Length")$
  group_by("Species", maintain_order = TRUE)$
  agg(pl$col(pl$Float64)$first() + 5)$
  profile()
## some R function, prints `.` for each time called by polars
# cat(".")
# s$to_r()[1] + 5
# sort("Sepal.Length")$
# group_by("Species", maintain_order = TRUE)$
# agg(pl$col(pl$Float64)$map_elements(r_func))$
# profile()
```

---

lazyframe\_\_quantile    *Aggregate the columns to a unique quantile value*

---

### Description

Aggregate the columns to a unique quantile value

### Usage

```
lazyframe__quantile(
  quantile,
  interpolation = c("nearest", "higher", "lower", "midpoint", "linear")
)
```

### Arguments

`quantile`        Quantile between 0.0 and 1.0.  
`interpolation`    Interpolation method.

### Value

A polars [LazyFrame](#)

### Examples

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, 1))
lf$quantile(0.7)$collect()
```

---

lazyframe\_\_rename        *Rename column names*

---

### Description

Rename column names

### Usage

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

### Arguments

`...`            [<dynamic-dots>](#) Either a function that takes a character vector as input and returns a character vector as output, or named values where names are old column names and values are the new ones.

`.strict`        Validate that all column names exist in the current schema, and throw an error if any do not. (Note that this parameter is a no-op when passing a function to `...`).

## 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

A polars [LazyFrame](#)

## Examples

```
lf <- pl$LazyFrame(  
  foo = 1:3,  
  bar = 6:8,  
  ham = letters[1:3]  
)  
  
lf$rename(foo = "apple")$collect()
```

---

lazyframe\_\_reverse    *Reverse the LazyFrame*

---

## Description

Reverse the LazyFrame

## Usage

```
lazyframe__reverse()
```

## Value

A polars [LazyFrame](#)

## Examples

```
lf <- pl$LazyFrame(key = c("a", "b", "c"), val = 1:3)  
lf$reverse()$collect()
```

---

`lazyframe__rolling`    *Create rolling groups based on a date/time or integer column*

---

### Description

Different from `group_by_dynamic()`, the windows are now determined by the individual values and are not of constant intervals. For constant intervals use `group_by_dynamic()`.

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 = c("right", "left", "both", "none"),
  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>group_by</code> is specified, then it must be sorted in ascending order within each group). In case of a dynamic group by on indices, the data type 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>          | These dots are for future extensions and must be empty.  |
| <code>period</code>       | Length of the window - must be non-negative.   |
| <code>offset</code>       | Offset of the window. Default is <code>-period</code> .  |
| <code>closed</code>       | Define which sides of the interval are closed (inclusive). Default is <code>"left"</code> .  |
| <code>group_by</code>     | Also group by this column/these columns. Can be expressions or objects coercible to expressions.   |



## 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

A `LazyGroupBy` object

## 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())
)

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

---

`lazyframe__select`      *Select and modify columns of a LazyFrame*

---

## Description

Select and perform operations on a subset of columns only. This discards unmentioned columns (like `.` in `data.table` and contrarily to `dplyr::mutate()`).

One cannot use new variables in subsequent expressions in the same `$select()` call. For instance, if you create a variable `x`, you will only be able to use it in another `$select()` or `$with_columns()` call.

## Usage

```
lazyframe__select(...)
```

**Arguments**

... [<dynamic-dots>](#) Name-value pairs of objects to be converted to polars expressions by the `as_polars_expr()` function. Characters are parsed as column names, other non-expression inputs are parsed as [literals](#). Each name will be used as the expression name.

**Value**

A polars [LazyFrame](#)

**Examples**

```
# Pass the name of a column to select that column.
lf <- pl$LazyFrame(
  foo = 1:3,
  bar = 6:8,
  ham = letters[1:3]
)
lf$select("foo")$collect()

# Multiple columns can be selected by passing a list of column names.
lf$select("foo", "bar")$collect()

# Expressions are also accepted.
lf$select(pl$col("foo"), pl$col("bar") + 1)$collect()

# Name expression (used as the column name of the output DataFrame)
lf$select(
  threshold = pl$when(pl$col("foo") > 2)$then(10)$otherwise(0)
)$collect()

# Expressions with multiple outputs can be automatically instantiated
# as Structs by setting the `POLARS_AUTO_STRUCTIFY` environment variable.
# (Experimental)
if (requireNamespace("withr", quietly = TRUE)) {
  withr::with_envvar(c(POLARS_AUTO_STRUCTIFY = "1"), {
    lf$select(
      is_odd = ((pl$col(pl$Int32) %% 2) == 1)$name$suffix("_is_odd"),
    )$collect()
  })
}
```

---

lazyframe\_\_select\_seq

*Select columns from this LazyFrame*

---

**Description**

This will run all expression sequentially instead of in parallel. Use this when the work per expression is cheap.

## Usage

```
lazyframe__select_seq(...)
```

## Arguments

... [<dynamic-dots>](#) Name-value pairs of objects to be converted to polars [expressions](#) by the `as_polars_expr()` function. Characters are parsed as column names, other non-expression inputs are parsed as [literals](#). Each name will be used as the expression name.

## Value

A polars [LazyFrame](#)

## Examples

```
lf <- pl$LazyFrame(  
  foo = 1:3,  
  bar = 6:8,  
  ham = letters[1:3]  
)  
lf$select_seq("foo")$collect()
```

---

`lazyframe__serialize` *Serialize the logical plan of this LazyFrame to a string in JSON format*

---

## Description

Serialize the logical plan of this LazyFrame to a string in JSON format

## Usage

```
lazyframe__serialize()
```

## Value

A character value

## Examples

```
lf <- pl$LazyFrame(a = 1:3)$sum()  
lf$serialize()
```

---

lazyframe\_\_set\_sorted

*Indicate that one or multiple columns are sorted*

---

### Description

This can speed up future operations, but it can lead to incorrect results if the data is **not** sorted! Use with care!

### Usage

```
lazyframe__set_sorted(column, ..., descending = FALSE)
```

### Arguments

|            |   |
|------------|---|
| column     | Column that is sorted.                                  |
| ...        | These dots are for future extensions and must be empty. |
| descending | Whether the columns are sorted in descending order.     |

### Value

A polars [LazyFrame](#)

---

lazyframe\_\_shift

*Shift values by the given number of indices*

---

### Description

Shift values by the given number of indices

### 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. |
| ...        | These dots are for future extensions and must be empty.  |
| fill_value | Fill the resulting null values with this value. Accepts expression input. Non-expression inputs are parsed as literals.  |

### Value

A polars [LazyFrame](#)

## Examples

```
lf <- pl$LazyFrame(a = 1:4, b = 5:8)

# By default, values are shifted forward by one index.
lf$shift()$collect()

# Pass a negative value to shift in the opposite direction instead.
lf$shift(-2)$collect()

# Specify fill_value to fill the resulting null values.
lf$shift(-2, fill_value = 100)$collect()
```

---

lazyframe\_\_sink\_csv *Evaluate the query in streaming mode and write to a CSV file*

---

## Description

### [Experimental]

This allows streaming results that are larger than RAM to be written to disk.

## 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_scientific = NULL,
  float_precision = NULL,
  null_value = "",
  quote_style = c("necessary", "always", "never", "non_numeric"),
  maintain_order = TRUE,
  type_coercion = TRUE,
  `_type_check` = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  collapse_joins = TRUE,
  no_optimization = FALSE,
```

```

    storage_options = NULL,
    retries = 2
)

```

## Arguments

|                               |   |
|-------------------------------|---|
| <code>path</code>             | A character. File path to which the file should be written.   |
| <code>...</code>              | Dots which should be empty.   |
| <code>include_bom</code>      | Logical, whether to include UTF-8 BOM in the CSV output.  |
| <code>include_header</code>   | Logical, whether to include header in the CSV output.   |
| <code>separator</code>        | Separate CSV fields with this symbol.   |
| <code>line_terminator</code>  | String used to end each row.  |
| <code>quote_char</code>       | 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 <code>chrono</code> 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 <code>chrono</code> Rust crate.   |
| <code>time_format</code>      | A format string, with the specifiers defined by the <code>chrono</code> Rust crate.   |
| <code>float_scientific</code> | Whether to use scientific form always ( <code>TRUE</code> ), never ( <code>FALSE</code> ), or automatically ( <code>NULL</code> ) for Float32 and Float64 datatypes.  |
| <code>float_precision</code>  | Number of decimal places to write, applied to both Float32 and Float64 datatypes.   |
| <code>null_value</code>       | A string representing null values (defaulting to the empty string).   |
| <code>quote_style</code>      | Determines the quoting strategy used. Must be one of: <ul style="list-style-type: none"> <li>• <code>"necessary"</code> (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>• <code>"always"</code>: This puts quotes around every field. Always.</li> <li>• <code>"never"</code>: This never puts quotes around fields, even if that results in invalid CSV data (e.g.: by not quoting strings containing the separator).</li> <li>• <code>"non_numeric"</code>: 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> </ul> |

|                                  |   |
|----------------------------------|---|
| <code>maintain_order</code>      | Maintain the order in which data is processed. Setting this to <code>FALSE</code> will be slightly faster.  |
| <code>type_coercion</code>       | A logical, indicates type coercion optimization.  |
| <code>predicate_pushdown</code>  | A logical, indicates predicate pushdown optimization.   |
| <code>projection_pushdown</code> | A logical, indicates projection pushdown optimization.  |
| <code>simplify_expression</code> | A logical, indicates simplify expression optimization.  |
| <code>slice_pushdown</code>      | A logical, indicates slice pushdown optimization.   |
| <code>collapse_joins</code>      | Collapse a join and filters into a faster join.   |
| <code>no_optimization</code>     | A logical. If <code>TRUE</code> , turn off (certain) optimizations.   |
| <code>storage_options</code>     | Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here: <ul style="list-style-type: none"> <li>• <code>aws</code></li> <li>• <code>gcp</code></li> <li>• <code>azure</code></li> <li>• Hugging Face (<code>hf://</code>): Accepts an API key under the token parameter <code>c(token = YOUR_TOKEN)</code> or by setting the <code>HF_TOKEN</code> environment variable.</li> </ul> <p>If <code>storage_options</code> is not provided, Polars will try to infer the information from environment variables.</p> |
| <code>retries</code>             | Number of retries if accessing a cloud instance fails.  |

## Value

Invisibly returns the input LazyFrame

## Examples

```
# sink table 'mtcars' from mem to CSV
tmpf <- tempfile(fileext = ".csv")
as_polars_lf(mtcars)$sink_csv(tmpf)

# stream a query end-to-end
tmpf2 <- tempfile(fileext = ".csv")
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\_\_slice      *Get a slice of the LazyFrame.*

---

### Description

Get a slice of the LazyFrame.

### Usage

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

### Arguments

|                     |   |
|---------------------|---|
| <code>offset</code> | Start index. Negative indexing is supported.  |
| <code>length</code> | Length of the slice. If NULL (default), all rows starting at the offset will be selected. |

### Value

A [LazyFrame](#)

### Examples

```
lf <- pl$LazyFrame(x = c("a", "b", "c"), y = 1:3, z = 4:6)
lf$slice(1, 2)$collect()
```

---

lazyframe\_\_sort      *Sort the LazyFrame by the given columns*

---

### Description

Sort the LazyFrame by the given columns

### Usage

```
lazyframe__sort(
  ...,
  descending = FALSE,
  nulls_last = FALSE,
  multithreaded = TRUE,
  maintain_order = FALSE
)
```



**Arguments**

|                |  |
|----------------|--|
| ...            | < <a href="#">dynamic-dots</a> > Column(s) to sort by. Can be character values indicating column names or Expr(s).   |
| descending     | Sort in descending order. When sorting by multiple columns, this can be specified per column by passing a logical vector.  |
| nulls_last     | Place null values last. When sorting by multiple columns, this can be specified per column by passing a logical vector.  |
| multithreaded  | Sort using multiple threads.   |
| maintain_order | Whether the order should be maintained if elements are equal. If <b>TRUE</b> , streaming is not possible and performance might be worse since this requires a stable search. |

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(
  a = c(1, 2, NA, 4),
  b = c(6, 5, 4, 3),
  c = c("a", "c", "b", "a")
)

# Pass a single column name to sort by that column.
lf$sort("a")$collect()

# Sorting by expressions is also supported
lf$sort(pl$col("a") + pl$col("b") * 2, nulls_last = TRUE)$collect()

# Sort by multiple columns by passing a vector of columns
lf$sort(c("c", "a"), descending = TRUE)$collect()

# Or use positional arguments to sort by multiple columns in the same way
lf$sort("c", "a", descending = c(FALSE, TRUE))$collect()
```

---

|                |   |
|----------------|---|
| lazyframe__std | <i>Aggregate the columns of this LazyFrame to their standard deviation values</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 polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, 1))
lf$std()$collect()
lf$std(ddof = 0)$collect()
```

---

|                             |  |
|-----------------------------|--|
| <code>lazyframe__sum</code> | <i>Aggregate the columns of this LazyFrame to their sum values</i> |
|-----------------------------|--|

---

**Description**

Aggregate the columns of this LazyFrame to their sum values

**Usage**

```
lazyframe__sum()
```

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, 1))
lf$sum()$collect()
```

---

|                              |                             |
|------------------------------|-----------------------------|
| <code>lazyframe__tail</code> | <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 polars [LazyFrame](#)

**See Also**

[<LazyFrame>\\$head\(\)](#)

**Examples**

```
lf <- pl$LazyFrame(a = 1:6, b = 7:12)
lf$tail()$collect()
lf$tail(2)$collect()
```

---

lazyframe\_\_top\_k      *Return the k largest rows*

---

**Description**

Non-null elements are always preferred over null elements, regardless of the value of **reverse**. The output is not guaranteed to be in any particular order, call `sort()` after this function if you wish the output to be sorted.

**Usage**

```
lazyframe__top_k(k, ..., by, reverse = FALSE)
```

**Arguments**

|                |  |
|----------------|--|
| <b>k</b>       | Number of rows to return.  |
| <b>...</b>     | These dots are for future extensions and must be empty.  |
| <b>by</b>      | Column(s) used to determine the bottom rows. Accepts expression input. Strings are parsed as column names.   |
| <b>reverse</b> | Consider the k smallest elements of the by column(s) (instead of the k largest). This can be specified per column by passing a sequence of booleans. |

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(
  a = c("a", "b", "a", "b", "b", "c"),
  b = c(2, 1, 1, 3, 2, 1)
)

# Get the rows which contain the 4 largest values in column b.
lf$top_k(4, by = "b")$collect()

# Get the rows which contain the 4 largest values when sorting on column a
# and b
lf$top_k(4, by = c("a", "b"))$collect()
```

---

```
lazyframe__to_dot    Plot the query plan
```

---

**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,
  `_type_check` = 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,
  collapse_joins = TRUE,
  streaming = FALSE,
  `_check_order` = TRUE
)
```

**Arguments**

```
...          Not used..
optimized    Optimize the query plan.
type_coercion A logical, indicates type coercion optimization.
predicate_pushdown
              A logical, indicates predicate pushdown optimization.
```

|  |  |
|--|--|
| <code>projection_pushdown</code>       | A logical, indicates projection pushdown optimization.   |
| <code>simplify_expression</code>       | A logical, indicates simplify expression optimization.   |
| <code>slice_pushdown</code>            | A logical, indicates slice pushdown optimization.  |
| <code>comm_subplan_elim</code>         | A logical, indicates trying to cache branching subplans that occur on self-joins or unions.  |
| <code>comm_subexpr_elim</code>         | A logical, indicates trying to cache common subexpressions.  |
| <code>cluster_with_columns</code>      | A logical, indicates to combine sequential independent calls to <code>with_columns</code> .  |
| <code>collapse_joins</code>            | Collapse a join and filters into a faster join.  |
| <code>streaming</code>                 | A logical. If <code>TRUE</code> , process the query in batches to handle larger-than-memory data. If <code>FALSE</code> (default), the entire query is processed in a single batch. Note that streaming mode is considered unstable. It may be changed at any point without it being considered a breaking change. |
| <code>_check_order, _type_check</code> | For internal use only.   |

## 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 duplicate rows*

---

### Description

Drop duplicate rows

### Usage

```
lazyframe__unique(
  subset = NULL,
  ...,
  keep = c("any", "none", "first", "last"),
  maintain_order = FALSE
)
```

### Arguments

|                |   |
|----------------|---|
| subset         | Column name(s) or selector(s), to consider when identifying duplicate rows. If <code>NULL</code> (default), use all columns.  |
| ...            | These dots are for future extensions and must be empty.   |
| keep           | Which of the duplicate rows to keep. Must be one of: <ul style="list-style-type: none"> <li>• "any": does not give any guarantee of which row is kept. This allows more optimizations.</li> <li>• "none": don't keep duplicate rows.</li> <li>• "first": keep first unique row.</li> <li>• "last": keep last unique row.</li> </ul> |
| maintain_order | Keep the same order as the original data. This is more expensive to compute. Setting this to <code>TRUE</code> blocks the possibility to run on the streaming engine.   |

### Value

A polars [LazyFrame](#)

### Examples

```
lf <- pl$LazyFrame(
  foo = c(1, 2, 3, 1),
  bar = c("a", "a", "a", "a"),
  ham = c("b", "b", "b", "b"),
)
lf$unique(maintain_order = TRUE)$collect()

lf$unique(subset = c("bar", "ham"), maintain_order = TRUE)$collect()

lf$unique(keep = "last", maintain_order = TRUE)$collect()
```

---

|                   |  |
|-------------------|--|
| lazyframe__unnest | <i>Decompose struct columns into separate columns for each of their fields</i> |
|-------------------|--|

---

### Description

The new columns will be inserted at the location of the struct column.

### Usage

```
lazyframe__unnest(...)
```

### Arguments

...                    <dynamic-dots> Name of the struct column(s) that should be unnested.

### Value

A polars [LazyFrame](#)

### 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()

lf$unnest("a_and_c")$collect()
lf$unnest(pl$col("a_and_c"))$collect()
```

---

|                    |   |
|--------------------|---|
| lazyframe__unpivot | <i>Unpivot a frame from wide to long format</i> |
|--------------------|---|

---

### Description

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

**Usage**

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

**Arguments**

|                            |   |
|----------------------------|---|
| <code>on</code>            | Values to use as identifier variables. If <code>value_vars</code> is empty all columns that are not in <code>id_vars</code> will be used. |
| <code>...</code>           | These dots are for future extensions and must be empty.   |
| <code>index</code>         | Columns to use as identifier variables.   |
| <code>variable_name</code> | Name to give to the new column containing the names of the melted columns. Defaults to "variable".  |
| <code>value_name</code>    | Name to give to the new column containing the values of the melted columns. Defaults to "value".  |

**Value**

A polars [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()
```

---

|                             |   |
|-----------------------------|---|
| <code>lazyframe__var</code> | <i>Aggregate the columns in the LazyFrame to their variance value</i> |
|-----------------------------|---|

---

**Description**

Aggregate the columns in the LazyFrame to their variance value

**Usage**

```
lazyframe__var(ddof = 1)
```

**Arguments**

|                   |  |
|-------------------|--|
| <code>ddof</code> | "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**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(a = 1:4, b = c(1, 2, 1, 1))
lf$var()$collect()
lf$var(ddof = 0)$collect()
```

---

lazyframe\_\_with\_columns

*Modify/append column(s) of a LazyFrame*

---

**Description**

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

However, unlike `dplyr::mutate()`, one cannot use new variables in subsequent expressions in the same `$with_columns()` call. For instance, if you create a variable `x`, you will only be able to use it in another `$with_columns()` or `$select()` call.

**Usage**

```
lazyframe__with_columns(...)
```

**Arguments**

... [<dynamic-dots>](#) Name-value pairs of objects to be converted to polars [expressions](#) by the `as_polars_expr()` function. Characters are parsed as column names, other non-expression inputs are parsed as [literals](#). Each name will be used as the expression name.

**Value**

A polars [LazyFrame](#)

**Examples**

```
# Pass an expression to add it as a new column.
lf <- pl$LazyFrame(
  a = 1:4,
  b = c(0.5, 4, 10, 13),
  c = c(TRUE, TRUE, FALSE, TRUE),
)
lf$with_columns((pl$col("a")^2)$alias("a^2"))$collect()

# Added columns will replace existing columns with the same name.
lf$with_columns(a = pl$col("a")$cast(pl$Float64))$collect()
```

```

# Multiple columns can be added
lf$with_columns(
  (pl$col("a")^2)$alias("a^2"),
  (pl$col("b") / 2)$alias("b/2"),
  (pl$col("c")$not())$alias("not c"),
)$collect()

# Name expression instead of `$alias()`
lf$with_columns(
  `a^2` = pl$col("a")^2,
  `b/2` = pl$col("b") / 2,
  `not c` = pl$col("c")$not(),
)$collect()

# Expressions with multiple outputs can automatically be instantiated
# as Structs by enabling the experimental setting `POLARS_AUTO_STRUCTIFY`:
if (requireNamespace("withr", quietly = TRUE)) {
  withr::with_envvar(c(POLARS_AUTO_STRUCTIFY = "1"), {
    lf$drop("c")$with_columns(
      diffs = pl$col("a", "b")$diff()$name$suffix("_diff"),
    )$collect()
  })
}

```

---

```
lazyframe__with_columns_seq
```

*Modify/append column(s) of a LazyFrame*

---

## Description

This will run all expression sequentially instead of in parallel. Use this only when the work per expression is cheap.

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

However, unlike `dplyr::mutate()`, one cannot use new variables in subsequent expressions in the same `$with_columns_seq()` call. For instance, if you create a variable `x`, you will only be able to use it in another `$with_columns_seq()` or `$select()` call.

## Usage

```
lazyframe__with_columns_seq(...)
```

## Arguments

... [<dynamic-dots>](#) Name-value pairs of objects to be converted to polars expressions by the `as_polars_expr()` function. Characters are parsed as column names, other non-expression inputs are parsed as [literals](#). Each name will be used as the expression name.

**Value**

A polars [LazyFrame](#)

**Examples**

```
# Pass an expression to add it as a new column.
lf <- pl$LazyFrame(
  a = 1:4,
  b = c(0.5, 4, 10, 13),
  c = c(TRUE, TRUE, FALSE, TRUE),
)
lf$with_columns_seq((pl$col("a")^2)$alias("a^2"))$collect()

# Added columns will replace existing columns with the same name.
lf$with_columns_seq(a = pl$col("a")$cast(pl$Float64))$collect()

# Multiple columns can be added
lf$with_columns_seq(
  (pl$col("a")^2)$alias("a^2"),
  (pl$col("b") / 2)$alias("b/2"),
  (pl$col("c")$not())$alias("not c"),
)$collect()

# Name expression instead of `alias()`
lf$with_columns_seq(
  `a^2` = pl$col("a")^2,
  `b/2` = pl$col("b") / 2,
  `not c` = pl$col("c")$not(),
)$collect()

# Expressions with multiple outputs can automatically be instantiated
# as Structs by enabling the experimental setting `POLARS_AUTO_STRUCTIFY`:
if (requireNamespace("withr", quietly = TRUE)) {
  withr::with_envvar(c(POLARS_AUTO_STRUCTIFY = "1"), {
    lf$drop("c")$with_columns_seq(
      diffs = pl$col("a", "b")$diff()$name$suffix("_diff"),
    )$collect()
  })
}
```

---

lazyframe\_\_with\_row\_index

*Add a row index as the first column in the LazyFrame*

---

**Description**

Using this function can have a negative effect on query performance. This may, for instance, block predicate pushdown optimization.

**Usage**

```
lazyframe__with_row_index(name = "index", offset = 0)
```

**Arguments**

|        |   |
|--------|---|
| name   | Name of the index column.                           |
| offset | Start the index at this offset. Cannot be negative. |

**Value**

A polars [LazyFrame](#)

**Examples**

```
lf <- pl$LazyFrame(x = c(1, 3, 5), y = c(2, 4, 6))
lf$with_row_index()$collect()

lf$with_row_index("id", offset = 1000)$collect()

# An index column can also be created using the expressions int_range()
# and len()$
lf$with_columns(
  index = pl$int_range(pl$len(), dtype = pl$UInt32)
)$collect()
```

---

parquet\_statistics     *Evaluate the query in streaming mode and write to a Parquet file*

---

**Description****[Experimental]**

This allows streaming results that are larger than RAM to be written to disk.

**Usage**

```
parquet_statistics(
  ...,
  min = TRUE,
  max = TRUE,
  distinct_count = TRUE,
  null_count = TRUE
)

lazyframe__sink_parquet(
  path,
  ...,
  compression = c("lz4", "uncompressed", "snappy", "gzip", "lzo", "brotli", "zstd"),
  compression_level = NULL,
```

```

    statistics = TRUE,
    row_group_size = NULL,
    data_page_size = NULL,
    maintain_order = TRUE,
    type_coercion = TRUE,
    `_type_check` = TRUE,
    predicate_pushdown = TRUE,
    projection_pushdown = TRUE,
    simplify_expression = TRUE,
    slice_pushdown = TRUE,
    collapse_joins = TRUE,
    no_optimization = FALSE,
    storage_options = NULL,
    retries = 2
  )

```

### Arguments

|                                |  |
|--------------------------------|--|
| <code>...</code>               | Dots which should be empty.  |
| <code>min</code>               | Include stats on the minimum values in the column.   |
| <code>max</code>               | Include stats on the maximum values in the column.   |
| <code>distinct_count</code>    | Include stats on the number of distinct values in the column.  |
| <code>null_count</code>        | Include stats on the number of null values in the column.  |
| <code>path</code>              | A character. File path to which the file should be written.  |
| <code>compression</code>       | The compression method. Must be 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> | 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>                        |
| <code>statistics</code>        | Whether statistics should be written to the Parquet headers. Possible values: <ul style="list-style-type: none"> <li>• TRUE: enable default set of statistics (default). Some statistics may be disabled.</li> </ul>   |

- `FALSE`: disable all statistics
- `"full"`: calculate and write all available statistics
- A list created via `parquet_statistics()` to specify which statistics to include.

**row\_group\_size** 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.

**maintain\_order** Maintain the order in which data is processed. Setting this to `FALSE` will be slightly faster.

**type\_coercion** A logical, indicates type coercion optimization.

**predicate\_pushdown** A logical, indicates predicate pushdown optimization.

**projection\_pushdown** A logical, indicates projection pushdown optimization.

**simplify\_expression** A logical, indicates simplify expression optimization.

**slice\_pushdown** A logical, indicates slice pushdown optimization.

**collapse\_joins** Collapse a join and filters into a faster join.

**no\_optimization** A logical. If `TRUE`, turn off (certain) optimizations.

**storage\_options** Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here:

- `aws`
- `gcp`
- `azure`
- Hugging Face (`hf://`): Accepts an API key under the token parameter `c(token = YOUR_TOKEN)` or by setting the `HF_TOKEN` environment variable.

If `storage_options` is not provided, Polars will try to infer the information from environment variables.

**retries** Number of retries if accessing a cloud instance fails.

## Value

Invisibly returns the input LazyFrame

## Examples

```
# sink table 'mtcars' from mem to parquet
tmpf <- tempfile()
as_polars_lf(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()
```

---

|    |  |
|----|--|
| pl | <i>Polars top-level function namespace</i> |
|----|--|

---

## Description

pl is an [environment class](#) object that stores all the top-level functions of the R Polars API which mimics the Python Polars API. It is intended to work the same way in Python as if you had imported Python Polars with `import polars as pl`.

## Usage

```
pl
```

## Format

An object of class `polars_object` of length 79.

## Examples

```
pl

# How many members are in the `pl` environment?
length(pl)

# Create a polars DataFrame
# In Python:
# ```python
# >>> import polars as pl
# >>> df = pl.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
# ...
# In R:
df <- pl$DataFrame(a = c(1, 2, 3), b = c(4, 5, 6))
df
```

---

```
pl_api_register_series_namespace
```

*Registering custom functionality with a polars Series*

---

## Description

Registering custom functionality with a polars Series

## Usage

```
pl_api_register_series_namespace(name, ns_fn)
```

## Arguments

|                    |   |
|--------------------|---|
| <code>name</code>  | Name under which the functionality will be accessed.  |
| <code>ns_fn</code> | A function returns a new <a href="#">environment</a> with the custom functionality. See examples for details. |

## Value

NULL invisibly.

## Examples

```
# s: polars series
math_shortcuts <- function(s) {
  # Create a new environment to store the methods
  self <- new.env(parent = emptyenv())

  # Store the series
  self$`_s` <- s

  # Add methods
  self$`square` <- function() self$`_s` * self$`_s`
  self$`cube` <- function() self$`_s` * self$`_s` * self$`_s`

  # Set the class
  class(self) <- c("polars_namespace_series", "polars_object")

  # Return the environment
  self
}

pl_api_register_series_namespace("math", math_shortcuts)

s <- as_polars_series(c(1.5, 31, 42, 64.5))
s$math$`square`() $rename("s^2")

s <- as_polars_series(1:5)
s$math$`cube`() $rename("s^3")
```



---

|        |  |
|--------|--|
| pl_all | <i>Either return an expression representing all columns, or evaluate a bitwise AND operation</i> |
|--------|--|

---

### Description

If no arguments are passed, this function is syntactic sugar for `col("*")`. Otherwise, this function is syntactic sugar for `col(names)$all()`.

### Usage

```
pl_all(..., ignore_nulls = TRUE)
```

### Arguments

|                           |   |
|---------------------------|---|
| <code>...</code>          | Name(s) of the columns to use in the aggregation.   |
| <code>ignore_nulls</code> | If <code>TRUE</code> (default), ignore null values. If <code>FALSE</code> , <b>Kleene logic</b> is used to deal with nulls: if the column contains any null values and no <code>TRUE</code> values, the output is null. |

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(
  a = c(TRUE, FALSE, TRUE),
  b = c(FALSE, FALSE, FALSE)
)

# Selecting all columns
df$select(pl$all())$sum()

# Evaluate bitwise AND for a column.
df$select(pl$all("a"))
```

---

|                   |  |
|-------------------|--|
| pl_all_horizontal | <i>Apply the AND logical horizontally across columns</i> |
|-------------------|--|

---

### Description

Apply the AND logical horizontally across columns

### Usage

```
pl_all_horizontal(...)
```

**Arguments**

... `<dynamic-dots>` Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

**Details**

**Kleene logic** is used to deal with nulls: if the column contains any null values and no **FALSE** values, the output is null.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = c(FALSE, FALSE, TRUE, TRUE, FALSE, NA),
  b = c(FALSE, TRUE, TRUE, NA, NA, NA),
  c = c("u", "v", "w", "x", "y", "z")
)

df$with_columns(
  all = pl$all_horizontal("a", "b")
)
```

---

pl\_\_any

*Evaluate a bitwise OR operation*

---

**Description**

This function is syntactic sugar for `col(names)$any()`.

**Usage**

```
pl__any(..., ignore_nulls = TRUE)
```

**Arguments**

... Name(s) of the columns to use in the aggregation.

`ignore_nulls` If **TRUE** (default), ignore null values. If **FALSE**, **Kleene logic** is used to deal with nulls: if the column contains any null values and no **TRUE** values, the output is null.

**Value**

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(  
  a = c(TRUE, FALSE, TRUE),  
  b = c(FALSE, FALSE, FALSE)  
)  
  
df$select(pl$any("a"))
```

---

pl\_\_any\_horizontal    *Apply the OR logical horizontally across columns*

---

## Description

Apply the OR logical horizontally across columns

## Usage

```
pl__any_horizontal(...)
```

## Arguments

...            <dynamic-dots> Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

## Details

**Kleene logic** is used to deal with nulls: if the column contains any null values and no **FALSE** values, the output is null.

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(  
  a = c(FALSE, FALSE, TRUE, TRUE, FALSE, NA),  
  b = c(FALSE, TRUE, TRUE, NA, NA, NA),  
  c = c("u", "v", "w", "x", "y", "z")  
)  
  
df$with_columns(  
  any = pl$any_horizontal("a", "b")  
)
```

---

|               |   |
|---------------|---|
| pl__arg_where | <i>Return indices where condition evaluates to TRUE</i> |
|---------------|---|

---

**Description**

Return indices where `condition` evaluates to `TRUE`

**Usage**

```
pl__arg_where(condition)
```

**Arguments**

`condition` Boolean expression to evaluate.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = 1:5)
df$select(
  pl$arg_where(pl$col("a") %% 2 == 0)
)
```

---

|              |   |
|--------------|---|
| pl__coalesce | <i>Folds the columns from left to right, keeping the first non-null value</i> |
|--------------|---|

---

**Description**

Folds the columns from left to right, keeping the first non-null value

**Usage**

```
pl__coalesce(...)
```

**Arguments**

... [<dynamic-dots>](#) Non-named objects can be referenced as columns. Each object will be converted to [expression](#) by `as_polars_expr()`. Strings are parsed as column names, other non-expression inputs are parsed as literals.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = c(1, NA, NA, NA),
  b = c(1, 2, NA, NA),
  c = c(5, NA, 3, NA)
)

df$with_columns(d = pl$coalesce("a", "b", "c", 10))

df$with_columns(d = pl$coalesce(pl$col("a", "b", "c"), 10))
```

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**

... [<dynamic-dots>](#) The name or [data type](#) of the column(s) to represent. Unnamed objects one of the following:

- Single string(s) representing column names
  - Regular expressions starting with `^` and ending with `$` are allowed.
  - Single wildcard `"*"` has a special meaning: check the examples.
- [Polars DataType\(s\)](#)

**Value**

A polars [expression](#)

**Examples**

```
# a single column by a character
pl$col("foo")

# multiple columns by characters
pl$col("foo", "bar")

# multiple columns by polars data types
pl$col(pl$Float64, pl$String)

# Single `"*"` is converted to a wildcard expression
pl$col("*")
```

```

# Character vectors with length > 1 should be used with `!!!`
pl$col(!!!c("foo", "bar"), "baz")
pl$col("foo", !!!c("bar", "baz"))

# 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$col("^ba.*$"))

```

---

|            |  |
|------------|--|
| pl__concat | <i>Combine multiple DataFrames, LazyFrames, or Series into a single object</i> |
|------------|--|

---

## Description

Combine multiple DataFrames, LazyFrames, or Series into a single object

## Usage

```

pl__concat(
  ...,
  how = c("vertical", "vertical_relaxed", "diagonal", "diagonal_relaxed", "horizontal",
    "align"),
  rechunk = FALSE,
  parallel = TRUE
)

```

## Arguments

... [<dynamic-dots>](#) [DataFrames](#), [LazyFrames](#), [Series](#). All elements must have the same class.

how Strategy to concatenate items. Must be one of:

- "vertical": applies multiple vstack operations;
- "vertical\_relaxed": same as "vertical", but additionally coerces columns to their common supertype if they are mismatched (eg: Int32 to Int64);
- "diagonal": finds a union between the column schemas and fills missing column values with null;
- "diagonal\_relaxed": same as "diagonal", but additionally coerces columns to their common supertype if they are mismatched (eg: Int32 to Int64);

- "horizontal": stacks Series from DataFrames horizontally and fills with null if the lengths don't match;
- "align": Combines frames horizontally, auto-determining the common key columns and aligning rows using the same logic as `align_frames`; this behaviour is patterned after a full outer join, but does not handle column-name collision. (If you need more control, you should use a suitable join method instead).

`Series` only support the "vertical" strategy.

|                       |  |
|-----------------------|--|
| <code>rechunk</code>  | Make sure that the result data is in contiguous memory.  |
| <code>parallel</code> | Only relevant for <code>LazyFrames</code> . This determines if the concatenated lazy computations may be executed in parallel. |

## Value

The same class (`polars_data_frame`, `polars_lazy_frame` or `polars_series`) as the input.

## Examples

```
# default is 'vertical' strategy
df1 <- pl$DataFrame(a = 1L, b = 3L)
df2 <- pl$DataFrame(a = 2L, b = 4L)
pl$concat(df1, df2)

# 'a' is coerced to float64
df1 <- pl$DataFrame(a = 1L, b = 3L)
df2 <- pl$DataFrame(a = 2, b = 4L)
pl$concat(df1, df2, how = "vertical_relaxed")

df_h1 <- pl$DataFrame(l1 = 1:2, l2 = 3:4)
df_h2 <- pl$DataFrame(r1 = 5:6, r2 = 7:8, r3 = 9:10)
pl$concat(df_h1, df_h2, how = "horizontal")

# use 'diagonal' strategy to fill empty column values with nulls
df1 <- pl$DataFrame(a = 1L, b = 3L)
df2 <- pl$DataFrame(a = 2L, c = 4L)
pl$concat(df1, df2, how = "diagonal")
```

---

|                              |   |
|------------------------------|---|
| <code>pl__concat_list</code> | <i>Horizontally concatenate columns into a single list column</i> |
|------------------------------|---|

---

## Description

Horizontally concatenate columns into a single list column

## Usage

```
pl__concat_list(...)
```

**Arguments**

... `<dynamic-dots>` Columns to concatenate into a single list column. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(a = list(1:2, 3, 4:5), b = list(4, integer(0), NULL))

# Concatenate two existing list columns. Null values are propagated.
df$with_columns(concat_list = pl$concat_list("a", "b"))

# Non-list columns are cast to a list before concatenation. The output data
# type is the supertype of the concatenated columns.
df$select("a", concat_list = pl$concat_list("a", pl$lit("x")))

# Create lagged columns and collect them into a list. This mimics a rolling
# window.
df <- pl$DataFrame(A = c(1, 2, 9, 2, 13))
df <- df$select(
  A_lag_1 = pl$col("A")$shift(1),
  A_lag_2 = pl$col("A")$shift(2),
  A_lag_3 = pl$col("A")$shift(3)
)
df$select(A_rolling = pl$concat_list("A_lag_1", "A_lag_2", "A_lag_3"))
```

---

pl\_\_concat\_str

*Horizontally concatenate columns into a single string column*

---

**Description**

Operates in linear time.

**Usage**

```
pl__concat_str(..., separator = "", ignore_nulls = FALSE)
```

**Arguments**

... `<dynamic-dots>` Columns to concatenate into a single string column. Accepts expression input. 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 will be propagated, i.e. if the row contains any null values, the output is null.



**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = 1:3,
  b = c("dogs", "cats", NA),
  c = c("play", "swim", "walk")
)
df$with_columns(
  full_sentence = pl$concat_str(
    pl$col("a") * 2L,
    pl$col("b"),
    pl$col("c"),
    separator = " ",
  )
)
```

---

|                          |                                    |
|--------------------------|------------------------------------|
| <code>pl__cum_sum</code> | <i>Cumulatively sum all values</i> |
|--------------------------|------------------------------------|

---

**Description**

This function is syntactic sugar for `col(names)$cum_sum()`.

**Usage**

```
pl__cum_sum(...)
```

**Arguments**

... Name(s) of the columns to use in the aggregation.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

# Get the cum_sum of a column
df$select(pl$cum_sum("a"))

# Get the cum_sum of multiple columns
df$select(pl$cum_sum("a", "b"))
```

---

pl\_DataFrame      *Polars DataFrame class (polars\_data\_frame)*

---

## Description

DataFrames are two-dimensional data structure representing data as a table with rows and columns. Polars DataFrames are similar to [R Data Frames](#). R Data Frame's columns are [R vectors](#), while Polars DataFrame's columns are [Polars Series](#).

## Usage

```
pl_DataFrame(..., .schema_overrides = NULL, .strict = TRUE)
```

## Arguments

...      [<dynamic-dots>](#) Name-value pairs of objects to be converted to polars [Series](#) by the [as\\_polars\\_series\(\)](#) function. Each [Series](#) will be used as a column of the [DataFrame](#). All values must be the same length. Each name will be used as the column name. If the name is empty, the original name of the [Series](#) will be used.

.schema\_overrides      **[Experimental]** A list of polars data types or NULL (default). Passed to the [\\$cast\(\)](#) method as dynamic-dots.

.strict      **[Experimental]** A logical value. Passed to the [\\$cast\(\)](#) method's `.strict` argument.

## Details

The `pl$DataFrame()` function mimics the constructor of the `DataFrame` class of Python Polars. This function is basically a shortcut for `as_polars_df(list(...))$cast(!!!.schema_overrides, .strict = .strict)`, so each argument in ... is converted to a Polars Series by [as\\_polars\\_series\(\)](#) and then passed to [as\\_polars\\_df\(\)](#).

## Value

A polars [DataFrame](#)

## Active bindings

- `columns`: `$columns` returns a character vector with the names of the columns.
- `dtypes`: `$dtypes` returns a nameless list of the data type of each column.
- `schema`: `$schema` returns a named list with the column names as names and the data types as values.
- `shape`: `$shape` returns a integer vector of length two with the number of rows and columns of the `DataFrame`.
- `height`: `$height` returns a integer with the number of rows of the `DataFrame`.

- `width`: `$width` returns an integer with the number of columns of the DataFrame.
- `flags`: `$flags` returns a list with column names as names and a named logical vector with the flags as values.

## Flags

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.

## Examples

```
# Constructing a DataFrame from vectors:
pl$DataFrame(a = 1:2, b = 3:4)

# Constructing a DataFrame from Series:
pl$DataFrame(pl$Series("a", 1:2), pl$Series("b", 3:4))

# Constructing a DataFrame from a list:
data <- list(a = 1:2, b = 3:4)

## Using the as_polars_df function (recommended)
as_polars_df(data)

## Using dynamic dots feature
pl$DataFrame(!!!data)

# Active bindings:
df <- pl$DataFrame(a = 1:3, b = c("foo", "bar", "baz"))

df$columns
df$dtypes
df$schema
df$shape
df$height
df$width
```

---

pl\_datetime

*Create a Polars literal expression of type Datetime*

---

## Description

Create a Polars literal expression of type Datetime

**Usage**

```
pl__datetime(
  year,
  month,
  day,
  hour = NULL,
  minute = NULL,
  second = NULL,
  microsecond = NULL,
  ...,
  time_unit = c("us", "ns", "ms"),
  time_zone = NULL,
  ambiguous = c("raise", "earliest", "latest", "null")
)
```

**Arguments**

|                          |  |
|--------------------------|--|
| <code>year</code>        | An <a href="#">polars expression</a> or something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> , which represents a column or literal number of year.                         |
| <code>month</code>       | An <a href="#">polars expression</a> or something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> , which represents a column or literal number of month. Range: 1-12.           |
| <code>day</code>         | An <a href="#">polars expression</a> or something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> , which represents a column or literal number of day. Range: 1-31.             |
| <code>hour</code>        | An <a href="#">polars expression</a> or something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> , which represents a column or literal number of hour. Range: 0-23.            |
| <code>minute</code>      | An <a href="#">polars expression</a> or something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> , which represents a column or literal number of minute. Range: 0-59.          |
| <code>second</code>      | An <a href="#">polars expression</a> or something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> , which represents a column or literal number of second. Range: 0-59.          |
| <code>microsecond</code> | An <a href="#">polars expression</a> or something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> , which represents a column or literal number of microsecond. Range: 0-999999. |
| <code>...</code>         | These dots are for future extensions and must be empty.  |
| <code>time_unit</code>   | One of "us" (default, microseconds), "ns" (nanoseconds) or "ms"(milliseconds). Representing the unit of time.  |
| <code>time_zone</code>   | A string or NULL (default). Representing the timezone.   |
| <code>ambiguous</code>   | Determine how to deal with ambiguous datetimes. Character vector or <a href="#">expression</a> containing the followings: <ul style="list-style-type: none"> <li>• "raise" (default): Throw an error</li> </ul>          |

- "earliest": Use the earliest datetime
- "latest": Use the latest datetime
- "null": Return a null value

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(  
  month = c(1, 2, 3),  
  day = c(4, 5, 6),  
  hour = c(12, 13, 14),  
  minute = c(15, 30, 45)  
)  
  
df$with_columns(  
  pl$datetime(  
    2024,  
    pl$col("month"),  
    pl$col("day"),  
    pl$col("hour"),  
    pl$col("minute"),  
    time_zone = "Australia/Sydney"  
  )  
)  
  
# We can also use `pl$datetime()` for filtering:  
df <- pl$select(  
  start = ISOdatetime(2024, 1, 1, 0, 0, 0),  
  end = c(  
    ISOdatetime(2024, 5, 1, 20, 15, 10),  
    ISOdatetime(2024, 7, 1, 21, 25, 20),  
    ISOdatetime(2024, 9, 1, 22, 35, 30)  
  )  
)  
  
df$filter(pl$col("end") > pl$datetime(2024, 6, 1))
```

---

pl\_\_datetime\_range      *Generate a datetime range*

---

## Description

Generate a datetime range

**Usage**

```
pl__datetime_range(
  start,
  end,
  interval = "1d",
  ...,
  closed = c("both", "left", "none", "right"),
  time_unit = NULL,
  time_zone = NULL
)
```

**Arguments**

|                  |   |
|------------------|---|
| <b>start</b>     | 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.  |
| <b>end</b>       | 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.  |
| <b>interval</b>  | 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. Must consist of full days. |
| <b>...</b>       | These dots are for future extensions and must be empty.   |
| <b>closed</b>    | Define which sides of the range are closed (inclusive). One of the following: "both" (default), "left", "right", "none".  |
| <b>time_unit</b> | Time unit of the resulting the <a href="#">Datetime</a> data type. One of "ns", "us", "ms" or NULL  |
| <b>time_zone</b> | Time zone of the resulting <a href="#">Datetime</a> data type.  |

**Value**

A polars [expression](#)

**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$select(
  datetime = pl$datetime_range(as.Date("2022-01-01"), as.Date("2022-03-01"), "1mo")
)

# Using `difftime` object to specify the interval:
pl$select(
  datetime = pl$datetime_range(
    as.Date("1985-01-01"),
    as.Date("1985-01-10"),
    as.difftime(1, units = "days") + as.difftime(12, units = "hours")
  )
)

# Specifying a time zone:
pl$select(
  datetime = pl$datetime_range(
    as.Date("2022-01-01"),
    as.Date("2022-03-01"),
    "1mo",
    time_zone = "America/New_York"
  )
)
```

---

`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 = c("both", "left", "none", "right"),
  time_unit = NULL,
  time_zone = NULL
)
```

**Arguments**

|                  |   |
|------------------|---|
| <b>start</b>     | 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.  |
| <b>end</b>       | 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.  |
| <b>interval</b>  | 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. Must consist of full days. |
| <b>...</b>       | These dots are for future extensions and must be empty.   |
| <b>closed</b>    | Define which sides of the range are closed (inclusive). One of the following: "both" (default), "left", "right", "none".  |
| <b>time_unit</b> | Time unit of the resulting the <a href="#">Datetime</a> data type. One of "ns", "us", "ms" or NULL  |
| <b>time_zone</b> | Time zone of the resulting <a href="#">Datetime</a> data type.  |

**Value**

A polars [expression](#)

**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 = rep(as.POSIXct("2022-01-01 12:00"), 3)
)

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      *Generate a date range*

---

### 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 = c("both", "left", "none", "right")
)

```

### Arguments

|                 |   |
|-----------------|---|
| <b>start</b>    | 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.  |
| <b>end</b>      | 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.  |
| <b>interval</b> | 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. Must consist of full days. |
| <b>...</b>      | These dots are for future extensions and must be empty.   |
| <b>closed</b>   | Define which sides of the range are closed (inclusive). One of the following: "both" (default), "left", "right", "none".  |

### Value

A polars [expression](#)

### 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$select(
  date = pl$date_range(as.Date("2022-01-01"), as.Date("2022-03-01"), "1mo")
)

# Using `difftime` object to specify the interval:
pl$select(
  date = pl$date_range(
    as.Date("1985-01-01"),
    as.Date("1985-01-10"),
    as.difftime(2, units = "days")
  )
)
```

---

|                 |                                       |
|-----------------|---------------------------------------|
| pl__date_ranges | <i>Create a column of date ranges</i> |
|-----------------|---------------------------------------|

---

**Description**

If both `start` and `end` are passed as 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 = c("both", "left", "none", "right")
)
```

**Arguments**

|                       |   |
|-----------------------|---|
| <code>start</code>    | 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.  |
| <code>end</code>      | 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.  |
| <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. Must consist of full days. |

... These dots are for future extensions and must be empty.

`closed` Define which sides of the range are closed (inclusive). One of the following: "both" (default), "left", "right", "none".

### Value

A polars [expression](#)

### 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 = rep(as.Date("2022-01-03"), 3)
)

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\_\_duration**
*Create polars Duration from distinct time components*


---

## Description

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.

## Usage

```
pl__duration(
  ...,
  weeks = NULL,
  days = NULL,
  hours = NULL,
  minutes = NULL,
  seconds = NULL,
  milliseconds = NULL,
  microseconds = NULL,
  nanoseconds = NULL,
  time_unit = NULL
)
```

## Arguments

|                           |  |
|---------------------------|--|
| <code>...</code>          | These dots are for future extensions and must be empty.  |
| <code>weeks</code>        | Something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> which represents a column or literal number of weeks, or <code>NULL</code> (default).        |
| <code>days</code>         | Something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> which represents a column or literal number of days, or <code>NULL</code> (default).         |
| <code>hours</code>        | Something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> which represents a column or literal number of hours, or <code>NULL</code> (default).        |
| <code>minutes</code>      | Something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> which represents a column or literal number of minutes, or <code>NULL</code> (default).      |
| <code>seconds</code>      | Something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> which represents a column or literal number of seconds, or <code>NULL</code> (default).      |
| <code>milliseconds</code> | Something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> which represents a column or literal number of milliseconds, or <code>NULL</code> (default). |

|                           |  |
|---------------------------|--|
| <code>microseconds</code> | Something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> which represents a column or literal number of microseconds, or NULL (default).  |
| <code>nanoseconds</code>  | Something can be coerced to an <a href="#">polars expression</a> by <code>as_polars_expr()</code> which represents a column or literal number of nanoseconds, or NULL (default).   |
| <code>time_unit</code>    | One of NULL, "us" (microseconds), "ns" (nanoseconds) or "ms" (milliseconds). Representing the unit of time. If NULL (default), the time unit will be inferred from the other inputs: "ns" if <code>nanoseconds</code> was specified, "us" otherwise. |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  dt = as.POSIXct(c("2022-01-01", "2022-01-02")),
  add = c(1, 2)
)
df

df$select(
  add_weeks = pl$col("dt") + pl$duration(weeks = pl$col("add")),
  add_days = pl$col("dt") + pl$duration(days = pl$col("add")),
  add_seconds = pl$col("dt") + pl$duration(seconds = pl$col("add")),
  add_millis = pl$col("dt") + pl$duration(milliseconds = pl$col("add")),
  add_hours = pl$col("dt") + pl$duration(hours = pl$col("add"))
)
```

---

`pl__element`

*Alias for an element being evaluated in an eval expression*

---

**Description**

Alias for an element being evaluated in an eval expression

**Usage**

```
pl__element()
```

**Value**

A polars [expression](#)

## Examples

```
# A horizontal rank computation by taking the elements of a list:
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2)
)
df$with_columns(
  rank = pl$concat_list(c("a", "b"))$list$eval(pl$element())$rank()
)

# A mathematical operation on array elements:
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2)
)
df$with_columns(
  a_b_doubled = pl$concat_list(c("a", "b"))$list$eval(pl$element() * 2)
)
```

---

pl\_\_first

*Get the first column of the context*

---

## Description

Get the first column of the context

## Usage

```
pl__first()
```

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "baz")
)

df$select(pl$first())
```

---

|               |                                     |
|---------------|-------------------------------------|
| 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

|                    |   |
|--------------------|---|
| <code>start</code> | Start of the range (inclusive). Defaults to 0.  |
| <code>end</code>   | End of the range (exclusive). If <code>NULL</code> (default), the value of <code>start</code> is used and <code>start</code> is set to 0. |
| <code>step</code>  | Step size of the range.   |
| <code>...</code>   | These dots are for future extensions and must be empty.   |
| <code>dtype</code> | Data type of the range.   |

## Value

A polars [expression](#)

## Examples

```
pl$select(int = pl$int_range(0, 3))

# end can be omitted for a shorter syntax.
pl$select(int = pl$int_range(3))

# Generate an index column by using int_range in conjunction with len().
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 | <i>Generate a range of integers for each row of the input columns</i> |
|----------------|---|

---

**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**

|              |  |
|--------------|--|
| <b>start</b> | Start of the range (inclusive). Defaults to 0.   |
| <b>end</b>   | End of the range (exclusive). If NULL (default), the value of <b>start</b> is used and <b>start</b> is set to 0. |
| <b>step</b>  | Step size of the range.  |
| <b>...</b>   | These dots are for future extensions and must be empty.  |
| <b>dtype</b> | Data type of the range.  |

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(start = c(1, -1), end = c(3, 2))
df$with_columns(int_range = pl$int_ranges("start", "end"))

# end can be omitted for a shorter syntax$
df$select("end", int_range = pl$int_ranges("end"))
```

---

|          |   |
|----------|---|
| pl__last | <i>Get the last column of the context</i> |
|----------|---|

---

**Description**

Get the last column of the context

**Usage**

```
pl__last()
```

**Value**

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "baz")
)

df$select(pl$last())
```

---

pl\_\_LazyFrame      *Polars LazyFrame class (polars\_lazy\_frame)*

---

## Description

Representation of a Lazy computation graph/query against a [DataFrame](#). This allows for whole-query optimisation in addition to parallelism, and is the preferred (and highest-performance) mode of operation for polars.

## Usage

```
pl__LazyFrame(..., .schema_overrides = NULL, .strict = TRUE)
```

## Arguments

...      <dynamic-dots> Name-value pairs of objects to be converted to polars [Series](#) by the [as\\_polars\\_series\(\)](#) function. Each [Series](#) will be used as a column of the [DataFrame](#). All values must be the same length. Each name will be used as the column name. If the name is empty, the original name of the [Series](#) will be used.

.schema\_overrides      **[Experimental]** A list of polars data types or NULL (default). Passed to the [\\$cast\(\)](#) method as dynamic-dots.

.strict      **[Experimental]** A logical value. Passed to the [\\$cast\(\)](#) method's `.strict` argument.

## Details

The `pl$LazyFrame(...)` function is a shortcut for `pl$DataFrame(...)$lazy()`.

## Value

A polars [LazyFrame](#)

## See Also

- [<LazyFrame>\\$collect\(\)](#): Materialize a [LazyFrame](#) into a [DataFrame](#).

## Examples

```
# Constructing a LazyFrame from vectors:
pl$LazyFrame(a = 1:2, b = 3:4)

# Constructing a LazyFrame from Series:
pl$LazyFrame(pl$Series("a", 1:2), pl$Series("b", 3:4))

# Constructing a LazyFrame from a list:
data <- list(a = 1:2, b = 3:4)

## Using dynamic dots feature
pl$LazyFrame(!!!data)
```

---

pl\_\_len

*Return the number of rows in the context\$*

---

## Description

This is similar to `COUNT(*)` in SQL.

## Usage

```
pl__len()
```

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(
  a = c(1, 2, NA),
  b = c(3, NA, NA),
  c = c("foo", "bar", "foo"),
)
df$select(pl$len())

# Generate an index column by using len in conjunction with $int_range()
df$with_columns(
  pl$int_range(pl$len(), dtype = pl$UInt32)$alias("index")
)
```

---

|         |  |
|---------|--|
| pl__lit | <i>Return an expression representing a literal value</i> |
|---------|--|

---

### Description

This function is a shorthand for `as_polars_expr(x, as_lit = TRUE)` and in most cases, the actual conversion is done by `as_polars_series()`.

### Usage

```
pl__lit(value, dtype = NULL)
```

### Arguments

|       |  |
|-------|--|
| value | An R object. Passed as the <code>x</code> param of <code>as_polars_expr()</code> .                               |
| dtype | A polars data type or <code>NULL</code> (default). If not <code>NULL</code> , casted to the specified data type. |

### Value

A polars [expression](#)

### Literal scalar mapping

Since R has no scalar class, each of the following types of length 1 cases is specially converted to a scalar literal.

- character: String
- logical: Boolean
- integer: Int32
- double: Float64

These types' NA is converted to a `null` literal with casting to the corresponding Polars type. The `raw` type vector is converted to a Binary scalar.

- raw: Binary

`NULL` is converted to a Null type `null` literal.

- `NULL`: Null

For other R class, the default S3 method is called and R object will be converted via `as_polars_series()`. So the type mapping is defined by `as_polars_series()`.

### See Also

- `as_polars_series()`: R -> Polars type mapping is mostly defined by this function.
- `as_polars_expr()`: Internal implementation of `pl$lit()`.

**Examples**

```

# Literal scalar values
pl$lit(1L)
pl$lit(5.5)
pl$lit(NULL)
pl$lit("foo_bar")

## Generally, for a vector (an R object) becomes a Series with length 1,
## it is converted to a Series and then get the first value to become a scalar literal.
pl$lit(as.Date("2021-01-20"))
pl$lit(as.POSIXct("2023-03-31 10:30:45"))
pl$lit(data.frame(a = 1, b = "foo"))

# Literal Series data
pl$lit(1:3)
pl$lit(pl$Series("x", 1:3))

```

---

pl\_\_max

*Get the maximum value*


---

**Description**

This function is syntactic sugar for `col(names)$max()`.

**Usage**

```
pl__max(...)
```

**Arguments**

... Name(s) of the columns to use in the aggregation.

**Value**

A polars [expression](#)

**Examples**

```

df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

# Get the maximum value of a column
df$select(pl$max("a"))

# Get the maximum value of multiple columns
df$select(pl$max("a", "b"))

```

---

`pl__max_horizontal` *Get the maximum value horizontally across columns*

---

### Description

Get the maximum value horizontally across columns

### Usage

```
pl__max_horizontal(...)
```

### Arguments

... [<dynamic-dots>](#) Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(  
  a = c(1, 8, 3),  
  b = c(4, 5, NA),  
  c = c("x", "y", "z")  
)  
df$with_columns(  
  max = pl$max_horizontal("a", "b")  
)
```

---

`pl__mean_horizontal` *Compute the mean horizontally across columns*

---

### Description

Compute the mean horizontally across columns

### Usage

```
pl__mean_horizontal(..., ignore_nulls = TRUE)
```

**Arguments**

- ... [<dynamic-dots>](#) Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.
- ignore\_nulls A logical. If TRUE, ignore null values (default). If FALSE, any null value in the input will lead to a null output.

**Value**

A polars [expression](#)

**Examples**

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, NA),
  c = c("x", "y", "z")
)

df$with_columns(
  mean = pl$mean_horizontal("a", "b")
)
```

---

pl\_\_min

*Get the minimum value*

---

**Description**

This function is syntactic sugar for `col(names)$min()`.

**Usage**

```
pl__min(...)
```

**Arguments**

- ... Name(s) of the columns to use in the aggregation.

**Value**

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(  
  a = c(1, 8, 3),  
  b = c(4, 5, 2),  
  c = c("foo", "bar", "foo")  
)  
  
# Get the minimum value of a column  
df$select(pl$min("a"))  
  
# Get the minimum value of multiple columns  
df$select(pl$min("a", "b"))
```

---

pl\_\_min\_horizontal    *Get the minimum value horizontally across columns*

---

## Description

Get the minimum value horizontally across columns

## Usage

```
pl__min_horizontal(...)
```

## Arguments

...            <dynamic-dots> Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

## Value

A polars [expression](#)

## Examples

```
df <- pl$DataFrame(  
  a = c(1, 8, 3),  
  b = c(4, 5, NA),  
  c = c("x", "y", "z")  
)  
df$with_columns(  
  min = pl$min_horizontal("a", "b")  
)
```



---

|         |   |
|---------|---|
| pl__nth | <i>Get the nth column(s) of the context</i> |
|---------|---|

---

### Description

Get the nth column(s) of the context

### Usage

```
pl__nth(indices)
```

### Arguments

`indices` One or more indices representing the columns to retrieve.

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(  
  a = c(1, 8, 3),  
  b = c(4, 5, 2),  
  c = c("foo", "bar", "baz")  
)  
  
df$select(pl$nth(1))  
df$select(pl$nth(c(2, 0)))
```

---

|              |                               |
|--------------|-------------------------------|
| pl__read_csv | <i>New DataFrame from CSV</i> |
|--------------|-------------------------------|

---

### Description

New DataFrame from CSV

### Usage

```
pl__read_csv(  
  source,  
  ...,  
  has_header = TRUE,  
  separator = ",",  
  comment_prefix = NULL,  
  quote_char = "\"",  
  skip_rows = 0,
```

```

schema = NULL,
schema_overrides = NULL,
null_values = NULL,
missing_utf8_is_empty_string = FALSE,
ignore_errors = FALSE,
cache = FALSE,
infer_schema = TRUE,
infer_schema_length = 100,
n_rows = NULL,
encoding = c("utf8", "utf8-lossy"),
low_memory = FALSE,
rechunk = FALSE,
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,
decimal_comma = FALSE,
glob = TRUE,
storage_options = NULL,
retries = 2,
file_cache_ttl = NULL,
include_file_paths = NULL
)

```

### Arguments

|                             |  |
|-----------------------------|--|
| <code>source</code>         | 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.  |
| <code>...</code>            | Dots which should be empty.  |
| <code>has_header</code>     | Indicate if the first row of dataset is a header or not. If <code>FALSE</code> , column names will be autogenerated in the following format: " <code>column_x</code> " with <code>x</code> being an enumeration over every column in the dataset starting at 1.                          |
| <code>separator</code>      | Single byte character to use as separator in the file.   |
| <code>comment_prefix</code> | 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 <code>#</code> or <code>//</code> .   |
| <code>quote_char</code>     | Single byte character used for quoting. Set to <code>NULL</code> to turn off special handling and escaping of quotes.  |
| <code>skip_rows</code>      | Start reading after a particular number of rows. The header will be parsed at this offset.   |
| <code>schema</code>         | Provide the schema. This means that polars doesn't do schema inference. This argument expects the complete schema, whereas <code>schema_overrides</code> can be used to partially overwrite a schema. This must be a list. Names of list elements are used to match to inferred columns. |

|   |   |
|---|---|
| <code>schema_overrides</code>             | Overwrite dtypes during inference. This must be a list. Names of list elements are used to match to inferred columns.   |
| <code>null_values</code>                  | Character vector specifying the values to interpret as NA values. It can be named, in which case names specify the columns in which this replacement must be made (e.g. <code>c(col1 = "a")</code> ).   |
| <code>missing_utf8_is_empty_string</code> | By default, a missing value is considered to be NA. Setting this parameter to <code>TRUE</code> will consider missing UTF8 values as an empty character.  |
| <code>ignore_errors</code>                | Keep reading the file even if some lines yield errors. You can also use <code>infer_schema = FALSE</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</code>                 | If <code>TRUE</code> (default), the schema is inferred from the data using the first <code>infer_schema_length</code> rows. When <code>FALSE</code> , the schema is not inferred and will be <code>pl\$String</code> if not specified in <code>schema</code> or <code>schema_overrides</code> . |
| <code>infer_schema_length</code>          | The maximum number of rows to scan for schema inference. If <code>NULL</code> , the full data may be scanned (this is slow). Set <code>infer_schema = FALSE</code> to read all columns as <code>pl\$String</code> .   |
| <code>n_rows</code>                       | Stop reading from CSV file after reading <code>n_rows</code> .  |
| <code>encoding</code>                     | Either <code>"utf8"</code> or <code>"utf8-lossy"</code> . Lossy means that invalid UTF8 values are replaced with <code>"?"</code> characters.   |
| <code>low_memory</code>                   | Reduce memory pressure at the expense of performance.   |
| <code>rechunk</code>                      | Reallocate to contiguous memory when all chunks / files are parsed.   |
| <code>skip_rows_after_header</code>       | Skip this number of rows when the header is parsed.   |
| <code>row_index_name</code>               | If not <code>NULL</code> , 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 <code>FALSE</code> , parsing an empty file returns an empty DataFrame or LazyFrame.  |
| <code>truncate_ragged_lines</code>        | Truncate lines that are longer than the schema.   |
| <code>decimal_comma</code>                | Parse floats using a comma as the decimal separator instead of a period.  |

|                                 |  |
|---------------------------------|--|
| <code>glob</code>               | Expand path given via globbing rules.  |
| <code>storage_options</code>    | <p>Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here:</p> <ul style="list-style-type: none"> <li>• <code>aws</code></li> <li>• <code>gcp</code></li> <li>• <code>azure</code></li> <li>• Hugging Face (<code>hf://</code>): Accepts an API key under the token parameter <code>c(token = YOUR_TOKEN)</code> or by setting the <code>HF_TOKEN</code> environment variable.</li> </ul> <p>If <code>storage_options</code> is not provided, Polars will try to infer the information from environment variables.</p> |
| <code>retries</code>            | Number of retries if accessing a cloud instance fails.   |
| <code>file_cache_ttl</code>     | Amount of time to keep downloaded cloud files since their last access time, in seconds. Uses the <code>POLARS_FILE_CACHE_TTL</code> environment variable (which defaults to 1 hour) if not given.  |
| <code>include_file_paths</code> | Include the path of the source file(s) as a column with this name.   |

**Value**

A polars [DataFrame](#)

**Examples**

```
my_file <- tempfile()
write.csv(iris, my_file)
pl$read_csv(my_file)
unlink(my_file)
```

---

`pl__read_ipc`

*Read into a DataFrame from Arrow IPC (Feather v2) file*

---

**Description**

Read into a DataFrame from Arrow IPC (Feather v2) file

**Usage**

```
pl__read_ipc(
  source,
  ...,
  n_rows = NULL,
  cache = TRUE,
```

```

    rechunk = FALSE,
    row_index_name = NULL,
    row_index_offset = 0L,
    storage_options = NULL,
    retries = 2,
    file_cache_ttl = NULL,
    hive_partitioning = NULL,
    hive_schema = NULL,
    try_parse_hive_dates = TRUE,
    include_file_paths = NULL
  )

```

### Arguments

|                                |   |
|--------------------------------|---|
| <code>source</code>            | Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the <code>storage_options</code> parameter.  |
| <code>...</code>               | These dots are for future extensions and must be empty.   |
| <code>n_rows</code>            | Stop reading from parquet file after reading <code>n_rows</code> .  |
| <code>cache</code>             | Cache the result after reading.   |
| <code>rechunk</code>           | In case of reading multiple files via a glob pattern rechunk the final DataFrame into contiguous memory chunks.   |
| <code>row_index_name</code>    | If not <code>NULL</code> , 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>storage_options</code>   | Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here: <ul style="list-style-type: none"> <li>• <code>aws</code></li> <li>• <code>gcp</code></li> <li>• <code>azure</code></li> <li>• Hugging Face (<code>hf://</code>): Accepts an API key under the token parameter <code>c(token = YOUR_TOKEN)</code> or by setting the <code>HF_TOKEN</code> environment variable.</li> </ul> <p>If <code>storage_options</code> is not provided, Polars will try to infer the information from environment variables.</p> |
| <code>retries</code>           | Number of retries if accessing a cloud instance fails.  |
| <code>hive_partitioning</code> | Infer statistics and schema from Hive partitioned sources and use them to prune reads. If <code>NULL</code> (default), it is automatically enabled when a single directory is passed, and otherwise disabled.   |
| <code>hive_schema</code>       | 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 <code>NULL</code> (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).

**Value**

A polars [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)

# If the path is a folder, Polars automatically tries to detect partitions
# and includes them in the output
pl$read_ipc(temp_dir)

# We can also impose a schema to the partition
pl$read_ipc(temp_dir, hive_schema = list(cyl = pl$String, gear = pl$Int32))
```

---

`pl__read_ipc_stream` *Read into a DataFrame from Arrow IPC stream format*

---

**Description**

Read into a DataFrame from Arrow IPC stream format

**Usage**

```
pl__read_ipc_stream(
  source,
  ...,
  columns = NULL,
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  rechunk = TRUE
)
```

**Arguments**

|                               |  |
|-------------------------------|--|
| <code>source</code>           | A character of the path to an Arrow IPC stream file.                                     |
| <code>...</code>              | These dots are for future extensions and must be empty.                                  |
| <code>columns</code>          | A character vector of column names to read.  |
| <code>n_rows</code>           | Stop reading from parquet file after reading <code>n_rows</code> .                       |
| <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>rechunk</code>          | A logical value to indicate whether to make sure that all data is contiguous.            |

**Value**

A polars [DataFrame](#)

**Examples**

```
temp_file <- tempfile(fileext = ".arrows")

mtcars |>
  nanoarrow::write_nanoarrow(temp_file)

pl$read_ipc_stream(temp_file, columns = c("cyl", "am"))
```

---

`pl__read_ndjson`      *Read into a DataFrame from NDJSON file*

---

**Description**

Read into a DataFrame from NDJSON file

**Usage**

```
pl__read_ndjson(
  source,
  ...,
  schema = NULL,
  schema_overrides = NULL,
  infer_schema_length = 100,
  batch_size = 1024,
  n_rows = NULL,
  low_memory = FALSE,
```

```

rechunk = FALSE,
row_index_name = NULL,
row_index_offset = 0L,
ignore_errors = FALSE,
storage_options = NULL,
retries = 2,
file_cache_ttl = NULL,
include_file_paths = NULL
)

```

### Arguments

|                                  |  |
|----------------------------------|--|
| <code>source</code>              | Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the <code>storage_options</code> parameter.   |
| <code>...</code>                 | These dots are for future extensions and must be empty.  |
| <code>schema</code>              | Specify the datatypes of the columns. The datatypes must match the datatypes in the file(s). If there are extra columns that are not in the file(s), consider also enabling <code>allow_missing_columns</code> .   |
| <code>schema_overrides</code>    | Overwrite dtypes during inference. This must be a list. Names of list elements are used to match to inferred columns.  |
| <code>infer_schema_length</code> | The maximum number of rows to scan for schema inference. If <code>NULL</code> , the full data may be scanned (this is slow). Set <code>infer_schema = FALSE</code> to read all columns as <code>pl\$String</code> .  |
| <code>n_rows</code>              | Stop reading from parquet file after reading <code>n_rows</code> .   |
| <code>low_memory</code>          | Reduce memory pressure at the expense of performance   |
| <code>rechunk</code>             | In case of reading multiple files via a glob pattern rechunk the final DataFrame into contiguous memory chunks.  |
| <code>row_index_name</code>      | If not <code>NULL</code> , 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>ignore_errors</code>       | Keep reading the file even if some lines yield errors. You can also use <code>infer_schema = FALSE</code> to read all columns as UTF8 to check which values might cause an issue.  |
| <code>storage_options</code>     | Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here: <ul style="list-style-type: none"> <li>• <code>aws</code></li> <li>• <code>gcp</code></li> <li>• <code>azure</code></li> <li>• Hugging Face (<code>hf://</code>): Accepts an API key under the token parameter <code>c(token = YOUR_TOKEN)</code> or by setting the <code>HF_TOKEN</code> environment variable.</li> </ul> |



|                                 |   |
|---------------------------------|---|
|                                 | If <code>storage_options</code> is not provided, Polars will try to infer the information from environment variables.   |
| <code>retries</code>            | Number of retries if accessing a cloud instance fails.  |
| <code>file_cache_ttl</code>     | Amount of time to keep downloaded cloud files since their last access time, in seconds. Uses the <code>POLARS_FILE_CACHE_TTL</code> environment variable (which defaults to 1 hour) if not given. |
| <code>include_file_paths</code> | Character value indicating the column name that will include the path of the source file(s).  |

**Value**

A polars [DataFrame](#)

**Examples**

```
ndjson_filename <- tempfile()
jsonlite::stream_out(iris, file(ndjson_filename), verbose = FALSE)
pl$read_ndjson(ndjson_filename)
```

---

|                               |  |
|-------------------------------|--|
| <code>pl__read_parquet</code> | <i>Read into a DataFrame from Parquet file</i> |
|-------------------------------|--|

---

**Description**

Read into a DataFrame from Parquet file

**Usage**

```
pl__read_parquet(
  source,
  ...,
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  parallel = c("auto", "columns", "row_groups", "prefiltered", "none"),
  use_statistics = TRUE,
  hive_partitioning = NULL,
  glob = TRUE,
  schema = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  rechunk = FALSE,
  low_memory = FALSE,
```

```

    cache = TRUE,
    storage_options = NULL,
    retries = 2,
    include_file_paths = NULL,
    allow_missing_columns = FALSE
  )

```

### Arguments

|                                   |  |
|-----------------------------------|--|
| <code>source</code>               | Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the <code>storage_options</code> parameter.   |
| <code>...</code>                  | These dots are for future extensions and must be empty.  |
| <code>n_rows</code>               | Stop reading from parquet file after reading <code>n_rows</code> .   |
| <code>row_index_name</code>       | If not <code>NULL</code> , 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>parallel</code>             | This determines the direction and strategy of parallelism. "auto" will try to determine the optimal direction. 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. |
| <code>use_statistics</code>       | Use statistics in the parquet to determine if pages can be skipped from reading.   |
| <code>hive_partitioning</code>    | Infer statistics and schema from Hive partitioned sources and use them to prune reads.   |
| <code>glob</code>                 | Expand path given via globbing rules.  |
| <code>schema</code>               | Specify the datatypes of the columns. The datatypes must match the datatypes in the file(s). If there are extra columns that are not in the file(s), consider also enabling <code>allow_missing_columns</code> .   |
| <code>hive_schema</code>          | The column names and data types of the columns by which the data is partitioned. If <code>NULL</code> (default), the schema of the hive partitions is inferred.  |
| <code>try_parse_hive_dates</code> | Whether to try parsing hive values as date / datetime types.   |
| <code>rechunk</code>              | In case of reading multiple files via a glob pattern rechunk the final DataFrame into contiguous memory chunks.  |
| <code>low_memory</code>           | Reduce memory pressure at the expense of performance   |
| <code>cache</code>                | Cache the result after reading.  |

**storage\_options**

Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here:

- `aws`
- `gcp`
- `azure`
- Hugging Face (`hf://`): Accepts an API key under the token parameter `c(token = YOUR_TOKEN)` or by setting the `HF_TOKEN` environment variable.

If `storage_options` is not provided, Polars will try to infer the information from environment variables.

**retries** Number of retries if accessing a cloud instance fails.

**include\_file\_paths**

Character value indicating the column name that will include the path of the source file(s).

**allow\_missing\_columns**

When reading a list of parquet files, if a column existing in the first file cannot be found in subsequent files, the default behavior is to raise an error. However, if `allow_missing_columns` is set to `TRUE`, a full-NULL column is returned instead of erroring for the files that do not contain the column.

**Value**

A polars [DataFrame](#)

**Examples**

```
# Write a Parquet file than we can then import as DataFrame
temp_file <- withr::local_tempfile(fileext = ".parquet")
as_polars_df(mtcars)$write_parquet(temp_file)
pl$read_parquet(temp_file)

# Write a hive-style partitioned parquet dataset
temp_dir <- withr::local_tempdir()
as_polars_df(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)
```

---

|                          |  |
|--------------------------|--|
| <code>pl__repeat_</code> | <i>Construct a column of length n“ filled with the given value</i> |
|--------------------------|--|

---

## Description

Construct a column of length n“ filled with the given value

## Usage

```
pl__repeat_(value, n, ..., dtype = NULL)
```

## Arguments

|                    |  |
|--------------------|--|
| <code>value</code> | Value to repeat.   |
| <code>n</code>     | Length of the resulting column   |
| <code>...</code>   | These dots are for future extensions and must be empty.  |
| <code>dtype</code> | Data type of the resulting column. If NULL (default), data type is inferred from the given value. Defaults to Int32 for integer values, unless Int64 is required to fit the given value. Defaults to Float64 for float values. |

## Details

If you want to construct a column in lazy mode and do not need a pre-determined length, use `pl$lit()` instead.

## Value

A polars [expression](#)

## Examples

```
# Construct a column with a repeated value in a lazy context.
pl$select(pl$repeat_("z", n = 3))

# Specify an output dtype
pl$select(pl$repeat_(3, n = 3, dtype = pl$Int8))
```

---

|              |  |
|--------------|--|
| pl__scan_csv | <i>Lazily read from a CSV 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_csv(  
    source,  
    ...,  
    has_header = TRUE,  
    separator = ",",  
    comment_prefix = NULL,  
    quote_char = "\"",  
    skip_rows = 0,  
    schema = NULL,  
    schema_overrides = NULL,  
    null_values = NULL,  
    missing_utf8_is_empty_string = FALSE,  
    ignore_errors = FALSE,  
    cache = FALSE,  
    infer_schema = TRUE,  
    infer_schema_length = 100,  
    n_rows = NULL,  
    encoding = c("utf8", "utf8-lossy"),  
    low_memory = FALSE,  
    rechunk = FALSE,  
    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,  
    decimal_comma = FALSE,  
    glob = TRUE,  
    storage_options = NULL,  
    retries = 2,  
    file_cache_ttl = NULL,  
    include_file_paths = NULL  
)
```

**Arguments**

|   |   |
|---|---|
| <code>source</code>                       | 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.   |
| <code>...</code>                          | Dots which should be empty.   |
| <code>has_header</code>                   | Indicate if the first row of dataset is a header or not. If <code>FALSE</code> , column names will be autogenerated in the following format: "column_x" with x being an enumeration over every column in the dataset starting at 1.   |
| <code>separator</code>                    | Single byte character to use as separator in the file.  |
| <code>comment_prefix</code>               | 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 //.   |
| <code>quote_char</code>                   | Single byte character used for quoting. Set to <code>NULL</code> to turn off special handling and escaping of quotes.   |
| <code>skip_rows</code>                    | Start reading after a particular number of rows. The header will be parsed at this offset.  |
| <code>schema</code>                       | Provide the schema. This means that polars doesn't do schema inference. This argument expects the complete schema, whereas <code>schema_overrides</code> can be used to partially overwrite a schema. This must be a list. Names of list elements are used to match to inferred columns.        |
| <code>schema_overrides</code>             | Overwrite dtypes during inference. This must be a list. Names of list elements are used to match to inferred columns.   |
| <code>null_values</code>                  | Character vector specifying the values to interpret as NA values. It can be named, in which case names specify the columns in which this replacement must be made (e.g. <code>c(col1 = "a")</code> ).   |
| <code>missing_utf8_is_empty_string</code> | By default, a missing value is considered to be NA. Setting this parameter to <code>TRUE</code> will consider missing UTF8 values as an empty character.  |
| <code>ignore_errors</code>                | Keep reading the file even if some lines yield errors. You can also use <code>infer_schema = FALSE</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</code>                 | If <code>TRUE</code> (default), the schema is inferred from the data using the first <code>infer_schema_length</code> rows. When <code>FALSE</code> , the schema is not inferred and will be <code>pl\$String</code> if not specified in <code>schema</code> or <code>schema_overrides</code> . |
| <code>infer_schema_length</code>          | The maximum number of rows to scan for schema inference. If <code>NULL</code> , the full data may be scanned (this is slow). Set <code>infer_schema = FALSE</code> to read all columns as <code>pl\$String</code> .   |
| <code>n_rows</code>                       | Stop reading from CSV file after reading <code>n_rows</code> .  |
| <code>encoding</code>                     | Either "utf8" or "utf8-lossy". Lossy means that invalid UTF8 values are replaced with "?" characters.   |
| <code>low_memory</code>                   | Reduce memory pressure at the expense of performance.   |

|                                     |   |
|-------------------------------------|---|
| <code>rechunk</code>                | Reallocate to contiguous memory when all chunks / files are parsed.   |
| <code>skip_rows_after_header</code> | Skip this number of rows when the header is parsed.   |
| <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>decimal_comma</code>          | Parse floats using a comma as the decimal separator instead of a period.  |
| <code>glob</code>                   | Expand path given via globbing rules.   |
| <code>storage_options</code>        | Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here: <ul style="list-style-type: none"> <li>• <code>aws</code></li> <li>• <code>gcp</code></li> <li>• <code>azure</code></li> <li>• Hugging Face (<code>hf://</code>): Accepts an API key under the token parameter <code>c(token = YOUR_TOKEN)</code> or by setting the <code>HF_TOKEN</code> environment variable.</li> </ul> <p>If <code>storage_options</code> is not provided, Polars will try to infer the information from environment variables.</p> |
| <code>retries</code>                | Number of retries if accessing a cloud instance fails.  |
| <code>file_cache_ttl</code>         | Amount of time to keep downloaded cloud files since their last access time, in seconds. Uses the <code>POLARS_FILE_CACHE_TTL</code> environment variable (which defaults to 1 hour) if not given.   |
| <code>include_file_paths</code>     | Include the path of the source file(s) as a column with this name.  |
| <code>credential_provider</code>    | Provide a function that can be called to provide cloud storage credentials. The function is expected to return a dictionary of credential keys along with an optional credential expiry time.   |

**Value**

A polars [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,
  cache = TRUE,
  rechunk = FALSE,
  row_index_name = NULL,
  row_index_offset = 0L,
  storage_options = NULL,
  retries = 2,
  file_cache_ttl = NULL,
  hive_partitioning = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  include_file_paths = NULL
)
```

**Arguments**

|               |  |
|---------------|--|
| <b>source</b> | Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the <b>storage_options</b> parameter. |
| <b>...</b>    | These dots are for future extensions and must be empty.  |
| <b>n_rows</b> | Stop reading from parquet file after reading <b>n_rows</b> .   |
| <b>cache</b>  | Cache the result after reading.  |



|                                   |   |
|-----------------------------------|---|
| <code>rechunk</code>              | In case of reading multiple files via a glob pattern <code>rechunk</code> the final <code>DataFrame</code> into contiguous memory chunks.   |
| <code>row_index_name</code>       | If not <code>NULL</code> , this will insert a row index column with the given name into the <code>DataFrame</code> .  |
| <code>row_index_offset</code>     | Offset to start the row index column (only used if the name is set).  |
| <code>storage_options</code>      | Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here: <ul style="list-style-type: none"> <li>• <code>aws</code></li> <li>• <code>gcp</code></li> <li>• <code>azure</code></li> <li>• Hugging Face (<code>hf://</code>): Accepts an API key under the token parameter <code>c(token = YOUR_TOKEN)</code> or by setting the <code>HF_TOKEN</code> environment variable.</li> </ul> <p>If <code>storage_options</code> is not provided, Polars will try to infer the information from environment variables.</p> |
| <code>retries</code>              | Number of retries if accessing a cloud instance fails.  |
| <code>hive_partitioning</code>    | Infer statistics and schema from Hive partitioned sources and use them to prune reads. If <code>NULL</code> (default), it is automatically enabled when a single directory is passed, and otherwise disabled.   |
| <code>hive_schema</code>          | 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 <code>NULL</code> (default), the schema of the Hive partitions is inferred.   |
| <code>try_parse_hive_dates</code> | Whether to try parsing hive values as date / datetime types.  |
| <code>include_file_paths</code>   | Character value indicating the column name that will include the path of the source file(s).  |

## Value

A polars [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 | <i>Lazily read from a local or cloud-hosted NDJSON file(s)</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_ndjson(
  source,
  ...,
  schema = NULL,
  schema_overrides = NULL,
  infer_schema_length = 100,
  batch_size = 1024,
  n_rows = NULL,
  low_memory = FALSE,
  rechunk = FALSE,
  row_index_name = NULL,
  row_index_offset = 0L,
  ignore_errors = FALSE,
  storage_options = NULL,
  retries = 2,
  file_cache_ttl = NULL,
  include_file_paths = NULL
)

```

## Arguments

|        |  |
|--------|--|
| source | Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the <code>storage_options</code> parameter. |
| ...    | These dots are for future extensions and must be empty.  |

|                                  |  |
|----------------------------------|--|
| <code>schema</code>              | Specify the datatypes of the columns. The datatypes must match the datatypes in the file(s). If there are extra columns that are not in the file(s), consider also enabling <code>allow_missing_columns</code> .   |
| <code>schema_overrides</code>    | Overwrite dtypes during inference. This must be a list. Names of list elements are used to match to inferred columns.  |
| <code>infer_schema_length</code> | The maximum number of rows to scan for schema inference. If NULL, the full data may be scanned (this is slow). Set <code>infer_schema = FALSE</code> to read all columns as <code>pl\$string</code> .  |
| <code>n_rows</code>              | Stop reading from parquet file after reading <code>n_rows</code> .   |
| <code>low_memory</code>          | Reduce memory pressure at the expense of performance   |
| <code>rechunk</code>             | In case of reading multiple files via a glob pattern rechunk the final DataFrame into contiguous memory chunks.  |
| <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>ignore_errors</code>       | Keep reading the file even if some lines yield errors. You can also use <code>infer_schema = FALSE</code> to read all columns as UTF8 to check which values might cause an issue.  |
| <code>storage_options</code>     | <p>Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here:</p> <ul style="list-style-type: none"> <li>• <code>aws</code></li> <li>• <code>gcp</code></li> <li>• <code>azure</code></li> <li>• Hugging Face (<code>hf://</code>): Accepts an API key under the token parameter <code>c(token = YOUR_TOKEN)</code> or by setting the <code>HF_TOKEN</code> environment variable.</li> </ul> <p>If <code>storage_options</code> is not provided, Polars will try to infer the information from environment variables.</p> |
| <code>retries</code>             | Number of retries if accessing a cloud instance fails.   |
| <code>file_cache_ttl</code>      | Amount of time to keep downloaded cloud files since their last access time, in seconds. Uses the <code>POLARS_FILE_CACHE_TTL</code> environment variable (which defaults to 1 hour) if not given.  |
| <code>include_file_paths</code>  | Character value indicating the column name that will include the path of the source file(s).   |

**Value**

A polars [LazyFrame](#)

## Examples

```
ndjson_filename <- tempfile()
jsonlite::stream_out(iris, file(ndjson_filename), verbose = FALSE)
pl$scan_ndjson(ndjson_filename)$collect()
```

---

|                   |   |
|-------------------|---|
| pl__scan__parquet | <i>Lazily read from a local or cloud-hosted parquet file (or files)</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__parquet(
  source,
  ...,
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  parallel = c("auto", "columns", "row_groups", "prefiltered", "none"),
  use_statistics = TRUE,
  hive_partitioning = NULL,
  glob = TRUE,
  schema = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  rechunk = FALSE,
  low_memory = FALSE,
  cache = TRUE,
  storage_options = NULL,
  retries = 2,
  include_file_paths = NULL,
  allow_missing_columns = FALSE
)
```

## Arguments

|               |  |
|---------------|--|
| <b>source</b> | Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the <code>storage_options</code> parameter. |
| <b>...</b>    | These dots are for future extensions and must be empty.  |
| <b>n_rows</b> | Stop reading from parquet file after reading <code>n_rows</code> .   |

|                                   |  |
|-----------------------------------|--|
| <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>parallel</code>             | This determines the direction and strategy of parallelism. "auto" will try to determine the optimal direction. 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. |
| <code>use_statistics</code>       | Use statistics in the parquet to determine if pages can be skipped from reading.   |
| <code>hive_partitioning</code>    | Infer statistics and schema from Hive partitioned sources and use them to prune reads.   |
| <code>glob</code>                 | Expand path given via globbing rules.  |
| <code>schema</code>               | Specify the datatypes of the columns. The datatypes must match the datatypes in the file(s). If there are extra columns that are not in the file(s), consider also enabling <code>allow_missing_columns</code> .   |
| <code>hive_schema</code>          | The column names and data types of the columns by which the data is partitioned. If NULL (default), the schema of the hive partitions is inferred.   |
| <code>try_parse_hive_dates</code> | Whether to try parsing hive values as date / datetime types.   |
| <code>rechunk</code>              | In case of reading multiple files via a glob pattern rechunk the final DataFrame into contiguous memory chunks.  |
| <code>low_memory</code>           | Reduce memory pressure at the expense of performance   |
| <code>cache</code>                | Cache the result after reading.  |
| <code>storage_options</code>      | Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here: <ul style="list-style-type: none"> <li>• <code>aws</code></li> <li>• <code>gcp</code></li> <li>• <code>azure</code></li> <li>• Hugging Face (<code>hf://</code>): Accepts an API key under the token parameter <code>c(token = YOUR_TOKEN)</code> or by setting the <code>HF_TOKEN</code> environment variable.</li> </ul> <p>If <code>storage_options</code> is not provided, Polars will try to infer the information from environment variables.</p>                                  |
| <code>retries</code>              | Number of retries if accessing a cloud instance fails.   |

**include\_file\_paths**

Character value indicating the column name that will include the path of the source file(s).

**allow\_missing\_columns**

When reading a list of parquet files, if a column existing in the first file cannot be found in subsequent files, the default behavior is to raise an error. However, if `allow_missing_columns` is set to `TRUE`, a full-NULL column is returned instead of erroring for the files that do not contain the column.

**Value**

A polars [LazyFrame](#)

**Examples**

```
# Write a Parquet file than we can then import as DataFrame
temp_file <- withr::local_tempfile(fileext = ".parquet")
as_polars_df(mtcars)$write_parquet(temp_file)
pl$scan_parquet(temp_file)$collect()

# Write a hive-style partitioned parquet dataset
temp_dir <- withr::local_tempdir()
as_polars_df(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\_\_Series

*Polars Series class (polars\_series)*

---

**Description**

Series are a 1-dimensional data structure, which are similar to [R vectors](#). Within a series all elements have the same Data Type.

**Usage**

```
pl__Series(name = NULL, values = NULL)
```

**Arguments**

**name** A single string or NULL. Name of the Series. Will be used as a column name when used in a [polars DataFrame](#). When not specified, name is set to an empty string.

**values** An R object. Passed as the x param of [as\\_polars\\_series\(\)](#).

## Details

The `pl$Series()` function mimics the constructor of the `Series` class of Python Polars. This function calls `as_polars_series()` internally to convert the input object to a Polars `Series`.

## Active bindings

- `dtype`: `$dtype` returns the data type of the `Series`.
- `name`: `$name` returns the name of the `Series`.
- `shape`: `$shape` returns a integer vector of length two with the number of length of the `Series` and width of the `Series` (always 1).

## See Also

- `as_polars_series()`

## Examples

```
# Constructing a Series by specifying name and values positionally:
s <- pl$Series("a", 1:3)
s

# Active bindings:
s$dtype
s$name
s$shape
```

---

`pl__show_versions`      *Print out the version of Polars and its optional dependencies*

---

## Description

**[Experimental]** Print out the version of Polars and its optional dependencies.

## Usage

```
pl__show_versions()
```

## Details

`cli` enhances the terminal output, especially error messages.

These packages may be used for exporting `Series` to R. See `<Series>$to_r_vector()` for details.

- `bit64`
- `blob`
- `clock`

- [data.table](#)
- [hms](#)
- [tibble](#)
- [vctrs](#)

### Value

NULL invisibly.

### Examples

```
pl$show_versions()
```

---

|            |   |
|------------|---|
| pl__struct | <i>Collect columns into a struct column</i> |
|------------|---|

---

### Description

Collect columns into a struct column

### Usage

```
pl__struct(..., .schema = NULL)
```

### Arguments

|         |  |
|---------|--|
| ...     | < <a href="#">dynamic-dots</a> > Name-value pairs of objects to be converted to polars <a href="#">expressions</a> by the <code>as_polars_expr()</code> function. Characters are parsed as column names, other non-expression inputs are parsed as <a href="#">literals</a> . Each name will be used as the expression name. |
| .schema | Optional schema that explicitly defines the struct field dtypes. If no columns or expressions are provided, .schema keys are used to define columns.   |

### Value

A polars [expression](#)

### Examples

```
# Collect all columns of a dataframe into a struct by passing pl$all().
df <- pl$DataFrame(
  int = 1:2,
  str = c("a", "b"),
  bool = c(TRUE, NA),
  list = list(1:2, 3L),
)
df$select(pl$struct(pl$all())$alias("my_struct"))
```



```

# Collect selected columns into a struct by either passing a list of
# columns, or by specifying each column as a positional argument.
df$select(pl$struct("int", FALSE)$alias("my_struct"))

# Name each struct field.
df$select(pl$struct(p = "int", q = "bool")$alias("my_struct"))$schema

# Pass a schema to specify the datatype of each field in the struct:
struct_schema <- list(int = pl$UInt32, list = pl$List(pl$Float32))
df$select(
  new_struct = pl$struct(pl$col("int", "list"), .schema = struct_schema)
)$unnest("new_struct")

```

---

|         |                       |
|---------|-----------------------|
| pl__sum | <i>Sum all values</i> |
|---------|-----------------------|

---

## Description

This function is syntactic sugar for `col(names)$sum()`.

## Usage

```
pl__sum(...)
```

## Arguments

... Name(s) of the columns to use in the aggregation.

## Value

A polars [expression](#)

## Examples

```

df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

# Get the sum of a column
df$select(pl$sum("a"))

# Get the sum of multiple columns
df$select(pl$sum("a", "b"))

```

---

pl\_\_sum\_horizontal     *Compute the sum horizontally across columns*

---

### Description

Compute the sum horizontally across columns

### Usage

```
pl__sum_horizontal(..., ignore_nulls = TRUE)
```

### Arguments

`...`     [<dynamic-dots>](#) Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

`ignore_nulls`     A logical. If TRUE, ignore null values (default). If FALSE, any null value in the input will lead to a null output.

### Value

A polars [expression](#)

### Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, NA),
  c = c("x", "y", "z")
)
df$with_columns(
  sum = pl$sum_horizontal("a", "b")
)
```

---

pl\_\_time\_range     *Generate a time range*

---

### Description

Generate a time range

**Usage**

```
pl__time_range(
  start = NULL,
  end = NULL,
  interval = "1h",
  ...,
  closed = c("both", "left", "none", "right")
)
```

**Arguments**

|                 |  |
|-----------------|--|
| <b>start</b>    | Lower bound of the time range. If omitted, defaults to 00:00:00.000.   |
| <b>end</b>      | Upper bound of the time range. If omitted, defaults to 23:59:59.999  |
| <b>interval</b> | Interval of the range periods, specified as a <a href="#">difftime</a> or using the Polars duration string language (see details). |
| <b>...</b>      | These dots are for future extensions and must be empty.  |
| <b>closed</b>   | Define which sides of the range are closed (inclusive). One of the following: "both" (default), "left", "right", "none".           |

**Value**

A polars [expression](#)

**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

```
pl$select(
  time = pl$time_range(
    start = hms::parse_hms("14:00:00"),
    interval = as.difftime("3:15:00")
  )
)
```

---

|                 |                                       |
|-----------------|---------------------------------------|
| pl__time_ranges | <i>Create a column of time ranges</i> |
|-----------------|---------------------------------------|

---

## Description

Create a column of time ranges

## Usage

```
pl__time_ranges(
  start = NULL,
  end = NULL,
  interval = "1h",
  ...,
  closed = c("both", "left", "none", "right")
)
```

## Arguments

|                 |  |
|-----------------|--|
| <b>start</b>    | Lower bound of the time range. If omitted, defaults to 00:00:00.000.   |
| <b>end</b>      | Upper bound of the time range. If omitted, defaults to 23:59:59.999  |
| <b>interval</b> | Interval of the range periods, specified as a <a href="#">difftime</a> or using the Polars duration string language (see details). |
| <b>...</b>      | These dots are for future extensions and must be empty.  |
| <b>closed</b>   | Define which sides of the range are closed (inclusive). One of the following: "both" (default), "left", "right", "none".           |

## Value

A polars [expression](#)

## 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

```
df <- pl$DataFrame(
  start = hms::parse_hms(c("09:00:00", "10:00:00")),
  end = hms::parse_hms(c("11:00:00", "11:00:00"))
)
df$with_columns(time_range = pl$time_ranges("start", "end"))
```

---

polars\_dtype

*Polars DataType class (polars\_dtype)*

---

## Description

Polars supports a variety of data types that fall broadly under the following categories:

- Numeric data types: signed integers, unsigned integers, floating point numbers, and decimals.
- Nested data types: lists, structs, and arrays.
- Temporal: dates, datetimes, times, and time deltas.
- Miscellaneous: strings, binary data, Booleans, categoricals, and enums.

All types support missing values represented by the special value `null`. This is not to be conflated with the special value `NaN` in floating number data types; see the section about floating point numbers for more information.

**Usage**

```

pl__Decimal(precision = NULL, scale = 0L)

pl__Datetime(time_unit = c("us", "ns", "ms"), time_zone = NULL)

pl__Duration(time_unit = c("us", "ns", "ms"))

pl__Categorical(ordering = c("physical", "lexical"))

pl__Enum(categories)

pl__Array(inner, shape)

pl__List(inner)

pl__Struct(...)

```

**Arguments**

|                         |   |
|-------------------------|---|
| <code>precision</code>  | Single integer or <code>NULL</code> (default), maximum number of digits in each number. If <code>NULL</code> , the precision is inferred. |
| <code>scale</code>      | Single integer or <code>NULL</code> . Number of digits to the right of the decimal point in each number. The default is 0.                |
| <code>time_unit</code>  | One of "us" (default, microseconds), "ns" (nanoseconds) or "ms"(milliseconds). Representing the unit of time.                             |
| <code>time_zone</code>  | A string or <code>NULL</code> (default). Representing the timezone.   |
| <code>ordering</code>   | One of "physical" (default) or "lexical". Ordering by order of appearance ("physical") or string value ("lexical").                       |
| <code>categories</code> | A character vector. Should not contain NA values and all values should be unique.   |
| <code>inner</code>      | A polars data type object.  |
| <code>shape</code>      | A integer-ish vector, representing the shape of the Array.  |
| <code>...</code>        | <dynamic-dots> Name-value pairs of polars data type. Each pair represents a field of the Struct.  |

**Details****Full data types table:**

| Type(s)                       | Details  |
|-------------------------------|--|
| Boolean                       | Boolean type that is bit packed efficiently.                         |
| Int8, Int16, Int32, Int64     | Varying-precision signed integer types.                              |
| UInt8, UInt16, UInt32, UInt64 | Varying-precision unsigned integer types.                            |
| Float32, Float64              | Varying-precision signed floating point numbers.                     |
| Decimal [Experimental]        | Decimal 128-bit type with optional precision and non-negative scale. |
| String                        | Variable length UTF-8 encoded string data, typically Human-readable. |

|                     |   |
|---------------------|---|
| Binary              | Stores arbitrary, varying length raw binary data.                               |
| Date                | Represents a calendar date.   |
| Time                | Represents a time of day.   |
| Datetime            | Represents a calendar date and time of day.                                     |
| Duration            | Represents a time duration.   |
| Array               | Arrays with a known, fixed shape per series; akin to numpy arrays.              |
| List                | Homogeneous 1D container with variable length.                                  |
| Categorical         | Efficient encoding of string data where the categories are inferred at runtime. |
| Enum [Experimental] | Efficient ordered encoding of a set of predetermined string categories.         |
| Struct              | Composite product type that can store multiple fields.                          |
| Null                | Represents null values.   |

## Examples

```

pl$Int8
pl$Int16
pl$Int32
pl$Int64
pl$UInt8
pl$UInt16
pl$UInt32
pl$UInt64
pl$Float32
pl$Float64
pl$Decimal(scale = 2)
pl$string
pl$Binary
pl$Date
pl$Time
pl$Datetime()
pl$Duration()
pl$Array(pl$Int32, c(2, 3))
pl$List(pl$Int32)
pl$Categorical()
pl$Enum(c("a", "b", "c"))
pl$Struct(a = pl$Int32, b = pl$string)
pl$Null

```

---

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\_expr

*Polars expression class* (polars\_expr)

---

## Description

An expression is a tree of operations that describe how to construct one or more [Series](#). As the outputs are [Series](#), it is straightforward to apply a sequence of expressions each of which transforms the output from the previous step. See examples for details.

## See Also

- [pl\\$lit\(\)](#): Create a literal expression.
- [pl\\$col\(\)](#): Create an expression representing column(s) in a [DataFrame](#).



## Examples

```

# An expression:
# 1. Select column `foo`,
# 2. Then sort the column (not in reversed order)
# 3. Then take the first two values of the sorted output
pl$col("foo")$sort()$head(2)

# Expressions will be evaluated inside a context, such as `$select()`
df <- pl$DataFrame(
  foo = c(1, 2, 1, 2, 3),
  bar = c(5, 4, 3, 2, 1),
)

df$select(
  pl$col("foo")$sort()$head(3), # Return 3 values
  pl$col("bar")$filter(pl$col("foo") == 1)$sum(), # Return a single value
)

```

---

`series_struct_unnest` *Convert this struct Series to a DataFrame with a separate column for each field*

---

## Description

Convert this struct Series to a DataFrame with a separate column for each field

## Usage

```
series_struct_unnest()
```

## Value

A polars [DataFrame](#)

## See Also

- [as\\_polars\\_df\(\)](#)

## Examples

```
s <- as_polars_series(data.frame(a = c(1, 3), b = c(2, 4)))
s$struct$unnest()
```

---

|               |                          |
|---------------|--------------------------|
| series__alias | <i>Rename the Series</i> |
|---------------|--------------------------|

---

### Description

`<Series>$rename()` is an alias for `<Series>$alias()`.

### Usage

```
series__alias(name)
series__rename(name)
```

### Arguments

name            The new name.

### Value

A [polars Series](#)

### Examples

```
series <- pl$Series("a", 1:3)

series$alias("b")
series$rename("b")
```

---

|                  |   |
|------------------|---|
| 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

An integer value

---

`series__to_frame`      *Cast this Series to a DataFrame*

---

### Description

Cast this Series to a DataFrame

### Usage

```
series__to_frame(name = NULL)
```

### Arguments

`name`            A character or NULL. If not NULL, name/rename the [Series](#) column in the new [DataFrame](#). If NULL, the column name is taken from the [Series](#) name.

### Value

A polars [DataFrame](#)

### See Also

- [as\\_polars\\_df\(\)](#)

### Examples

```
s <- pl$Series("a", c(123, 456))
df <- s$to_frame()
df

df <- s$to_frame("xyz")
df
```

---

`series__to_r_vector`      *Export the Series as an R vector*

---

### Description

Export the [Series](#) as an R [vector](#). But note that the Struct data type is exported as a [data.frame](#) by default for consistency, and a [data.frame](#) is not a vector. If you want to ensure the return value is a [vector](#), please set `ensure_vector = TRUE`, or use the [as.vector\(\)](#) function instead.

**Usage**

```
series__to_r_vector(
  ...,
  ensure_vector = FALSE,
  int64 = c("double", "character", "integer", "integer64"),
  date = c("Date", "IDate"),
  time = c("hms", "ITime"),
  struct = c("dataframe", "tibble"),
  decimal = c("double", "character"),
  as_clock_class = FALSE,
  ambiguous = c("raise", "earliest", "latest", "null"),
  non_existent = c("raise", "null")
)
```

**Arguments**

- ...** These dots are for future extensions and must be empty.
- ensure\_vector** A logical value indicating whether to ensure the return value is a [vector](#). When the Series has the Struct data type and this argument is `FALSE` (default), the return value is a [data.frame](#), not a [vector](#) (`is.vector(<data.frame>)` is `FALSE`). If `TRUE`, return a named [list](#) instead of a [data.frame](#).
- int64** Determine how to convert Polars' Int64, UInt32, or UInt64 type values to R type. One of the followings:
- `"double"` (default): Convert to the R's [double](#) type. Accuracy may be degraded.
  - `"character"`: Convert to the R's [character](#) type.
  - `"integer"`: Convert to the R's [integer](#) type. If the value is out of the range of R's integer type, export as `NA_integer_`.
  - `"integer64"`: Convert to the `bit64::integer64` class. The `bit64` package must be installed. If the value is out of the range of `bit64::integer64`, export as `bit64::NA_integer64_`.
- date** Determine how to convert Polars' Date type values to R class. One of the followings:
- `"Date"` (default): Convert to the R's [Date](#) class.
  - `"IDate"`: Convert to the `data.table::IDate` class.
- time** Determine how to convert Polars' Time type values to R class. One of the followings:
- `"hms"` (default): Convert to the `hms::hms` class.
  - `"ITime"`: Convert to the `data.table::ITime` class. The `data.table` package must be installed.
- struct** Determine how to convert Polars' Struct type values to R class. One of the followings:
- `"dataframe"` (default): Convert to the R's [data.frame](#) class.
  - `"tibble"`: Convert to the [tibble](#) class. If the `tibble` package is not installed, a warning will be shown.

|                             |   |
|-----------------------------|---|
| <code>decimal</code>        | Determine how to convert Polars' Decimal type values to R type. One of the followings: <ul style="list-style-type: none"> <li>• <code>"double"</code> (default): Convert to the R's <code>double</code> type.</li> <li>• <code>"character"</code>: Convert to the R's <code>character</code> type.</li> </ul>   |
| <code>as_clock_class</code> | A logical value indicating whether to export datetimes and duration as the <code>clock</code> package's classes. <ul style="list-style-type: none"> <li>• <code>FALSE</code> (default): Duration values are exported as <code>difftime</code> and date-time values are exported as <code>POSIXct</code>. Accuracy may be degraded.</li> <li>• <code>TRUE</code>: Duration values are exported as <code>clock_duration</code>, datetime without timezone values are exported as <code>clock_naive_time</code>, and datetime with timezone values are exported as <code>clock_zoned_time</code>. For this case, the <code>clock</code> package must be installed. Accuracy will be maintained.</li> </ul> |
| <code>ambiguous</code>      | Determine how to deal with ambiguous datetimes. Only applicable when <code>as_clock_class</code> is set to <code>FALSE</code> and datetime without timezone values are exported as <code>POSIXct</code> . Character vector or <code>expression</code> containing the followings: <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 NA value</li> </ul>   |
| <code>non_existent</code>   | Determine how to deal with non-existent datetimes. Only applicable when <code>as_clock_class</code> is set to <code>FALSE</code> and datetime without timezone values are exported as <code>POSIXct</code> . One of the followings: <ul style="list-style-type: none"> <li>• <code>"raise"</code> (default): Throw an error</li> <li>• <code>"null"</code>: Return a NA value</li> </ul>  |

## Details

The class/type of the exported object depends on the data type of the Series as follows:

- Boolean: `logical`.
- UInt8, UInt16, Int8, Int16, Int32: `integer`.
- Int64, UInt32, UInt64: `double`, `character`, `integer`, or `bit64::integer64`, depending on the `int64` argument.
- Float32, Float64: `double`.
- Decimal: `double`.
- String: `character`.
- Categorical: `factor`.
- Date: `Date` or `data.table::IDate`, depending on the `date` argument.
- Time: `hms::hms` or `data.table::ITime`, depending on the `time` argument.
- Datetime (without timezone): `POSIXct` or `clock_naive_time`, depending on the `as_clock_class` argument.

- Datetime (with timezone): `POSIXct` or `clock_zoned_time`, depending on the `as_clock_class` argument.
- Duration: `difftime` or `clock_duration`, depending on the `as_clock_class` argument.
- Binary: `blob::blob`.
- Null: `vctrs::unspecified`.
- List, Array: `vctrs::list_of`.
- Struct: `data.frame` or `tibble`, depending on the `struct` argument. If `ensure_vector = TRUE`, the top-level Struct is exported as a named `list` for to ensure the return value is a `vector`.

## Value

A `vector`

## Examples

```
# Struct values handling
series_struct <- as_polars_series(
  data.frame(
    a = 1:2,
    b = I(list(data.frame(c = "foo"), data.frame(c = "bar")))
  )
)
series_struct

## Export Struct as data.frame
series_struct$to_r_vector()

## Export Struct as data.frame,
## but the top-level Struct is exported as a named list
series_struct$to_r_vector(ensure_vector = TRUE)

## Export Struct as tibble
series_struct$to_r_vector(struct = "tibble")

## Export Struct as tibble,
## but the top-level Struct is exported as a named list
series_struct$to_r_vector(struct = "tibble", ensure_vector = TRUE)

# Integer values handling
series_uint64 <- as_polars_series(
  c(NA, "0", "4294967295", "18446744073709551615")
)$cast(pl$UInt64)
series_uint64

## Export UInt64 as double
series_uint64$to_r_vector(int64 = "double")

## Export UInt64 as character
series_uint64$to_r_vector(int64 = "character")
```

```
## Export UInt64 as integer (overflow occurs)
series_uint64$to_r_vector(int64 = "integer")

## Export UInt64 as bit64::integer64 (overflow occurs)
if (requireNamespace("bit64", quietly = TRUE)) {
  series_uint64$to_r_vector(int64 = "integer64")
}

# Duration values handling
series_duration <- as_polars_series(
  c(NA, -1000000000, -10, -1, 1000000000)
)$cast(pl$Duration("ns"))
series_duration

## Export Duration as difftime
series_duration$to_r_vector(as_clock_class = FALSE)

## Export Duration as clock_duration
if (requireNamespace("clock", quietly = TRUE)) {
  series_duration$to_r_vector(as_clock_class = TRUE)
}

# Datetime values handling
series_datetime <- as_polars_series(
  as.POSIXct(
    c(NA, "1920-01-01 00:00:00", "1970-01-01 00:00:00", "2020-01-01 00:00:00"),
    tz = "UTC"
  )
)$cast(pl$Datetime("ns", "UTC"))
series_datetime

## Export zoned datetime as POSIXct
series_datetime$to_r_vector(as_clock_class = FALSE)

## Export zoned datetime as clock_zoned_time
if (requireNamespace("clock", quietly = TRUE)) {
  series_datetime$to_r_vector(as_clock_class = TRUE)
}
```

# Index

- \* datasets
  - cs, [37](#)
  - pl, [447](#)
- <DataFrame>\$get\_columns(), [18](#)
- <DataFrame>\$group\_by\_dynamic(), [100](#)
- <DataFrame>\$partition\_by(), [75](#)
- <DataFrame>\$rolling(), [77](#)
- <DataFrame>\$to\_struct(), [29](#)
- <Expr>\$floor\_div(), [314](#), [382](#)
- <Expr>\$mod(), [290](#)
- <Expr>\$str\$contains(), [211](#)
- <Expr>\$str\$replace(), [227](#)
- <Expr>\$str\$replace\_all(), [226](#)
- <Expr>\$str\$strptime(), [239](#), [241](#), [244](#)
- <Expr>\$str\$to\_date(), [237](#)
- <Expr>\$str\$to\_datetime(), [237](#)
- <Expr>\$str\$to\_time(), [237](#)
- <Expr>\$true\_div(), [290](#)
- <LazyFrame>\$collect(), [21](#), [474](#)
- <LazyFrame>\$group\_by\_dynamic(), [425](#)
- <LazyFrame>\$head(), [435](#)
- <LazyFrame>\$rolling(), [406](#)
- <Series>\$alias(), [514](#)
- <Series>\$rename(), [514](#)
- <Series>\$struct\$unnest(), [20](#), [21](#)
- <Series>\$to\_frame(), [21](#)
- <Series>\$to\_r\_vector(), [18](#), [29](#), [503](#)
- <expr>\$dt\$offset\_by(), [469](#)
- <expr>\$dt\$offset\_by(1d), [469](#)
- \$cast(), [329](#), [458](#), [474](#)
- \$collect(), [421](#)
- \$collect\_in\_background(), [393](#), [421](#)
- \$drop\_nans(), [275](#)
- \$drop\_nulls(), [275](#)
- \$dt\$base\_utc\_offset(), [148](#)
- \$dt\$convert\_time\_zone(), [158](#)
- \$dt\$dst\_offset(), [143](#)
- \$dt\$to\_string(), [161](#)
- \$eq(), [277](#)
- \$eq\_missing(), [276](#)
- \$explode(), [288](#)
- \$head(), [307](#)
- \$list\$gather(), [182](#)
- \$list\$get(), [180](#)
- \$ne\_missing(), [317](#)
- \$profile(), [393](#)
- \$replace(), [331](#)
- \$replace\_strict(), [330](#)
- \$rolling(), [336](#), [337](#), [339](#), [341](#), [342](#), [344](#), [345](#), [347](#), [349](#), [350](#), [353-355](#), [357](#), [358](#), [360](#), [362](#), [363](#), [425](#)
- \$set\_difference(), [191](#)
- \$set\_symmetric\_difference(), [189](#)
- \$sink\_ipc(), [393](#), [421](#)
- \$sink\_parquet(), [393](#), [421](#)
- \$sort(), [262](#), [379](#)
- \$str\$contains(), [219](#)
- \$str\$ends\_with(), [210](#), [219](#)
- \$str\$find(), [210](#)
- \$str\$start\_with(), [210](#), [219](#)
- \$strip\_chars\_end(), [235](#)
- \$strip\_chars\_start(), [235](#)
- \$to\_frame(), [21](#)
- \$value\_counts(), [382](#), [383](#)
  
- abort(), [36](#)
- abs(n), [79](#), [107](#)
- Arithmetic operators, [246](#), [290](#), [314](#), [315](#), [324](#), [376](#), [382](#)
- as.data.frame(<polars\_data\_frame>), [21](#)
- as.data.frame(<polars\_object>), [34](#)
- as.data.frame.polars\_data\_frame, [14](#)
- as.data.frame.polars\_lazy\_frame
  - (as.data.frame.polars\_data\_frame), [14](#)
- as.list(<polars\_data\_frame>), [21](#), [73](#)
- as.list.polars\_data\_frame, [16](#)



- as.list.polars\_lazy\_frame  
(*as.list.polars\_data\_frame*),  
16
- as.vector(), 515
- as\_polars\_df, 19
- as\_polars\_df(), 15, 17, 19, 25, 29, 32,  
458, 513, 515
- as\_polars\_df(x, ...), 25
- as\_polars\_expr, 22
- as\_polars\_expr(), 22, 101, 115, 116, 338,  
342, 345, 348, 352, 356, 359, 363,  
426, 427, 441, 442, 452, 460, 469,  
470, 476, 504
- as\_polars\_lf, 25
- as\_polars\_lf(), 25
- as\_polars\_series, 26
- as\_polars\_series(), 19–21, 23, 24, 26,  
28, 110, 387, 458, 474, 476, 502,  
503
- as\_tibble.polars\_data\_frame, 31
- as\_tibble.polars\_lazy\_frame  
(*as\_tibble.polars\_data\_frame*),  
31
  
- base::OlsonNames(), 146, 158
- bit64, 15, 17, 33, 503, 516
- bit64::integer64, 15, 17, 33, 516, 517
- bit64::NA\_integer64\_, 15, 17, 33, 516
- blob, 503
- blob::blob, 518
  
- character, 15, 17, 18, 23, 33, 516, 517
- check\_list\_of\_polars\_dtype  
(*check\_polars*), 34
- check\_polars, 34, 387
- check\_polars\_df (*check\_polars*), 34
- check\_polars\_dtype (*check\_polars*), 34
- check\_polars\_expr (*check\_polars*), 34
- check\_polars\_lf (*check\_polars*), 34
- check\_polars\_selector (*check\_polars*),  
34
- check\_polars\_series (*check\_polars*),  
34
- chrono strftime documentation, 162,  
169
- cli, 503
- clock, 15, 18, 33, 503, 517
- clock\_duration, 15, 18, 29, 33, 517, 518
- clock\_naive\_time, 15, 18, 33, 517
- clock\_zoned\_time, 15, 18, 33, 517, 518
- cs, 37, 38–61
- cs\_\_all, 38
- cs\_\_alpha, 38
- cs\_\_alphanumeric, 39
- cs\_\_binary, 40
- cs\_\_boolean, 41
- cs\_\_by\_dtype, 42
- cs\_\_by\_index, 43
- cs\_\_by\_name, 44
- cs\_\_categorical, 45
- cs\_\_contains, 45
- cs\_\_date, 46
- cs\_\_datetime, 47
- cs\_\_decimal, 48
- cs\_\_digit, 49
- cs\_\_duration, 50
- cs\_\_ends\_with, 51
- cs\_\_exclude, 52
- cs\_\_first, 53
- cs\_\_float, 53
- cs\_\_integer, 54
- cs\_\_last, 55
- cs\_\_matches, 55
- cs\_\_numeric, 56
- cs\_\_signed\_integer, 57
- cs\_\_starts\_with, 58
- cs\_\_string, 59
- cs\_\_temporal, 60
- cs\_\_time, 60
- cs\_\_unsigned\_integer, 61
  
- data type, 28, 162, 169, 453
- data.frame, 17, 21, 28, 29, 515, 516, 518
- data.table, 15, 17, 33, 504, 516
- data.table::IDate, 15, 17, 33, 516, 517
- data.table::ITime, 15, 17, 33, 516, 517
- DataFrame, 20, 21, 25, 31, 62–64, 66, 67,  
69–72, 79, 83, 85, 88–92, 94, 95,  
97, 98, 101–109, 111–113, 115,  
117, 118, 320, 393, 458, 474, 484,  
486, 487, 489, 491, 512, 513, 515
- DataFrame (*pl\_\_DataFrame*), 458
- DataFrame\$filter(), 287
- dataframe\_\_bottom\_k, 62
- dataframe\_\_cast, 63
- dataframe\_\_clear, 64
- dataframe\_\_clone, 64
- dataframe\_\_drop, 65

- dataframe\_\_drop\_nans, 66
- dataframe\_\_drop\_nulls, 67
- dataframe\_\_equals, 68
- dataframe\_\_explode, 69
- dataframe\_\_fill\_nan, 69
- dataframe\_\_fill\_null, 70
- dataframe\_\_filter, 71
- dataframe\_\_gather\_every, 72
- dataframe\_\_get\_column, 72
- dataframe\_\_get\_column\_index, 74
- dataframe\_\_get\_columns, 73
- dataframe\_\_group\_by, 74
- dataframe\_\_group\_by\_dynamic, 75
- dataframe\_\_head, 79
- dataframe\_\_is\_duplicated, 79
- dataframe\_\_is\_empty, 80
- dataframe\_\_is\_unique, 81
- dataframe\_\_join, 81
- dataframe\_\_join\_asof, 84
- dataframe\_\_join\_where, 87
- dataframe\_\_lazy, 88
- dataframe\_\_max, 89
- dataframe\_\_max\_horizontal, 89
- dataframe\_\_mean, 90
- dataframe\_\_mean\_horizontal, 90
- dataframe\_\_median, 91
- dataframe\_\_merge\_sorted, 91
- dataframe\_\_min, 92
- dataframe\_\_min\_horizontal, 92
- dataframe\_\_n\_chunks, 93
- dataframe\_\_partition\_by, 94
- dataframe\_\_pivot, 94
- dataframe\_\_quantile, 97
- dataframe\_\_rechunk, 97
- dataframe\_\_rename, 98
- dataframe\_\_rolling, 99
- dataframe\_\_sample, 100
- dataframe\_\_select, 101
- dataframe\_\_set\_sorted, 102
- dataframe\_\_shift, 103
- dataframe\_\_slice, 103
- dataframe\_\_sort, 104
- dataframe\_\_std, 105
- dataframe\_\_sum, 106
- dataframe\_\_sum\_horizontal, 106
- dataframe\_\_tail, 107
- dataframe\_\_to\_dummies, 108
- dataframe\_\_to\_series, 109
- dataframe\_\_to\_struct, 110
- dataframe\_\_top\_k, 107
- dataframe\_\_transpose, 110
- dataframe\_\_unique, 111
- dataframe\_\_unnest, 113
- dataframe\_\_unpivot, 113
- dataframe\_\_var, 114
- dataframe\_\_with\_columns, 115
- dataframe\_\_with\_columns\_seq, 116
- dataframe\_\_with\_row\_index, 117
- dataframe\_\_write\_csv, 118
- dataframe\_\_write\_json, 120
- dataframe\_\_write\_ndjson, 121
- dataframe\_\_write\_parquet, 121
- DataFrames, 454
- DataType, 29, 386
- DataType (*polars\_dtype*), 509
- Date, 15, 17, 29, 33, 143, 516, 517
- Date/Time/Datetime, 162, 169
- Datetime, 240, 462, 464, 466, 467
- difftime, 15, 18, 29, 33, 462, 464, 466, 467, 507, 508, 517, 518
- double, 15, 17, 18, 33, 516, 517
- dtype, 29
- Duration, 162, 169, 469
- environment, 448
- environment class, 37, 447
- Expr, 23, 128, 182, 210–212, 218, 225, 227, 386
- Expr (*polars\_expr*), 512
- Expr\$struct\$field(\*), 208
- expr\_\_abs, 245
- expr\_\_add, 246
- expr\_\_agg\_groups, 247
- expr\_\_alias, 247
- expr\_\_all, 248
- expr\_\_and, 249
- expr\_\_any, 249
- expr\_\_append, 250
- expr\_\_approx\_n\_unique, 251
- expr\_\_arccos, 251
- expr\_\_arccosh, 252
- expr\_\_arcsin, 252
- expr\_\_arcsinh, 253
- expr\_\_arctan, 253
- expr\_\_arctanh, 254
- expr\_\_arg\_max, 254
- expr\_\_arg\_min, 255

`expr__arg_sort`, 255  
`expr__arg_true`, 256  
`expr__arg_unique`, 256  
`expr__backward_fill`, 257  
`expr__bitwise_and`, 257  
`expr__bitwise_count_ones`, 258  
`expr__bitwise_count_zeros`, 258  
`expr__bitwise_leading_ones`, 259  
`expr__bitwise_leading_zeros`, 259  
`expr__bitwise_or`, 260  
`expr__bitwise_trailing_ones`, 260  
`expr__bitwise_trailing_zeros`, 261  
`expr__bitwise_xor`, 261  
`expr__bottom_k`, 262  
`expr__bottom_k_by`, 262  
`expr__cast`, 263  
`expr__cbrt`, 264  
`expr__ceil`, 265  
`expr__clip`, 265  
`expr__cos`, 266  
`expr__cosh`, 267  
`expr__cot`, 267  
`expr__count`, 268  
`expr__cum_count`, 269  
`expr__cum_max`, 270  
`expr__cum_min`, 270  
`expr__cum_prod`, 271  
`expr__cum_sum`, 272  
`expr__cumulative_eval`, 268  
`expr__cut`, 272  
`expr__degrees`, 273  
`expr__diff`, 274  
`expr__dot`, 274  
`expr__drop_nans`, 275  
`expr__drop_nulls`, 275  
`expr__entropy`, 276  
`expr__eq`, 276, 277  
`expr__eq_missing`, 277, 277  
`expr__ewm_mean`, 278  
`expr__ewm_mean_by`, 279  
`expr__ewm_std`, 280  
`expr__ewm_var`, 282  
`expr__exclude`, 283  
`expr__exp`, 284  
`expr__explode`, 285  
`expr__extend_constant`, 285  
`expr__fill_nan`, 286  
`expr__fill_null`, 286  
`expr__filter`, 287  
`expr__first`, 288  
`expr__flatten`, 288  
`expr__floor`, 289  
`expr__floor_div`, 289  
`expr__floordiv` (`expr__floor_div`), 289  
`expr__forward_fill`, 290  
`expr__gather`, 291  
`expr__gather_every`, 291  
`expr__ge`, 292  
`expr__get`, 292  
`expr__gt`, 293  
`expr__has_nulls`, 294  
`expr__hash`, 294  
`expr__head`, 295  
`expr__hist`, 295  
`expr__implode`, 296  
`expr__interpolate`, 297  
`expr__interpolate_by`, 297  
`expr__is_between`, 298  
`expr__is_duplicated`, 299  
`expr__is_finite`, 299  
`expr__is_first_distinct`, 300  
`expr__is_in`, 300  
`expr__is_infinite`, 301  
`expr__is_last_distinct`, 302  
`expr__is_nan`, 302  
`expr__is_not_nan`, 303  
`expr__is_not_null`, 303  
`expr__is_null`, 304  
`expr__is_unique`, 305  
`expr__kurtosis`, 305  
`expr__last`, 306  
`expr__le`, 306  
`expr__len`, 307  
`expr__limit`, 307  
`expr__log`, 308  
`expr__log10`, 308  
`expr__log1p`, 309  
`expr__lower_bound`, 309  
`expr__lt`, 310  
`expr__map_batches`, 310  
`expr__max`, 312  
`expr__mean`, 312  
`expr__median`, 313  
`expr__min`, 313  
`expr__mod`, 314  
`expr__mode`, 314

`expr__mul`, 315  
`expr__n_unique`, 319  
`expr__nan_max`, 316  
`expr__nan_min`, 316  
`expr__ne`, 317, 318  
`expr__ne_missing`, 317, 317  
`expr__not`, 318  
`expr__null_count`, 319  
`expr__or`, 320  
`expr__over`, 320  
`expr__pct_change`, 322  
`expr__peak_max`, 323  
`expr__peak_min`, 323  
`expr__pow`, 324  
`expr__product`, 324  
`expr__qcut`, 325  
`expr__quantile`, 326  
`expr__radians`, 327  
`expr__rank`, 327  
`expr__rechunk`, 328  
`expr__reinterpret`, 329  
`expr__repeat_by`, 329  
`expr__replace`, 330  
`expr__replace_strict`, 331  
`expr__reshape`, 333  
`expr__reverse`, 334  
`expr__rle`, 334  
`expr__rle_id`, 335  
`expr__rolling`, 335  
`expr__rolling_max`, 337  
`expr__rolling_max_by`, 338  
`expr__rolling_mean`, 340  
`expr__rolling_mean_by`, 341  
`expr__rolling_median`, 343  
`expr__rolling_median_by`, 344  
`expr__rolling_min`, 346  
`expr__rolling_min_by`, 348  
`expr__rolling_quantile`, 350  
`expr__rolling_quantile_by`, 351  
`expr__rolling_skew`, 353  
`expr__rolling_std`, 354  
`expr__rolling_std_by`, 355  
`expr__rolling_sum`, 358  
`expr__rolling_sum_by`, 359  
`expr__rolling_var`, 361  
`expr__rolling_var_by`, 362  
`expr__round`, 364  
`expr__round_sig_figs`, 365  
`expr__sample`, 365  
`expr__search_sorted`, 366  
`expr__set_sorted`, 367  
`expr__shift`, 368  
`expr__shrink_dtype`, 368  
`expr__shuffle`, 369  
`expr__sign`, 370  
`expr__sin`, 370  
`expr__sinh`, 371  
`expr__skew`, 371  
`expr__slice`, 372  
`expr__sort`, 373  
`expr__sort_by`, 374  
`expr__sqrt`, 375  
`expr__std`, 376  
`expr__sub`, 376  
`expr__sum`, 377  
`expr__tail`, 377  
`expr__tan`, 378  
`expr__tanh`, 378  
`expr__to_physical`, 380  
`expr__top_k`, 379  
`expr__top_k_by`, 379  
`expr__true_div`, 381  
`expr__truediv (expr__true_div)`, 381  
`expr__unique`, 382  
`expr__unique_counts`, 383  
`expr__upper_bound`, 383  
`expr__value_counts`, 384  
`expr__var`, 385  
`expr__xor`, 385  
`expr_arr_all`, 124  
`expr_arr_any`, 124  
`expr_arr_arg_max`, 125  
`expr_arr_arg_min`, 125  
`expr_arr_contains`, 126  
`expr_arr_count_matches`, 126  
`expr_arr_explode`, 127  
`expr_arr_first`, 127  
`expr_arr_get`, 128  
`expr_arr_join`, 129  
`expr_arr_last`, 129  
`expr_arr_max`, 130  
`expr_arr_median`, 130  
`expr_arr_min`, 131  
`expr_arr_n_unique`, 131  
`expr_arr_reverse`, 132  
`expr_arr_shift`, 132

- expr\_arr\_sort, 133
- expr\_arr\_std, 133
- expr\_arr\_sum, 134
- expr\_arr\_to\_list, 135
- expr\_arr\_unique, 135
- expr\_arr\_var, 136
- expr\_bin\_contains, 136
- expr\_bin\_decode, 137
- expr\_bin\_encode, 138
- expr\_bin\_ends\_with, 139
- expr\_bin\_size, 139
- expr\_bin\_starts\_with, 140
- expr\_cat\_get\_categories, 141
- expr\_cat\_set\_ordering, 141
- expr\_dt\_add\_business\_days, 142
- expr\_dt\_base\_utc\_offset, 143
- expr\_dt\_cast\_time\_unit, 144
- expr\_dt\_century, 145
- expr\_dt\_combine, 145
- expr\_dt\_convert\_time\_zone, 146
- expr\_dt\_date, 147
- expr\_dt\_day, 147
- expr\_dt\_dst\_offset, 148
- expr\_dt\_epoch, 148
- expr\_dt\_hour, 149
- expr\_dt\_is\_leap\_year, 150
- expr\_dt\_iso\_year, 150
- expr\_dt\_microsecond, 151
- expr\_dt\_millisecond, 151
- expr\_dt\_minute, 152
- expr\_dt\_month, 153
- expr\_dt\_month\_end, 153
- expr\_dt\_month\_start, 154
- expr\_dt\_nanosecond, 154
- expr\_dt\_offset\_by, 155
- expr\_dt\_ordinal\_day, 156
- expr\_dt\_quarter, 157
- expr\_dt\_replace\_time\_zone, 158
- expr\_dt\_round, 159
- expr\_dt\_second, 161
- expr\_dt\_strftime, 161
- expr\_dt\_time, 162
- expr\_dt\_timestamp, 163
- expr\_dt\_to\_string, 168
- expr\_dt\_total\_days, 164
- expr\_dt\_total\_hours, 164
- expr\_dt\_total\_microseconds, 165
- expr\_dt\_total\_milliseconds, 166
- expr\_dt\_total\_minutes, 166
- expr\_dt\_total\_nanoseconds, 167
- expr\_dt\_total\_seconds, 168
- expr\_dt\_truncate, 170
- expr\_dt\_week, 171
- expr\_dt\_weekday, 172
- expr\_dt\_with\_time\_unit, 172
- expr\_dt\_year, 173
- expr\_list\_all, 173
- expr\_list\_any, 174
- expr\_list\_arg\_max, 174
- expr\_list\_arg\_min, 175
- expr\_list\_concat, 175
- expr\_list\_contains, 176
- expr\_list\_count\_matches, 177
- expr\_list\_diff, 177
- expr\_list\_drop\_nulls, 178
- expr\_list\_eval, 178
- expr\_list\_explode, 179
- expr\_list\_first, 180
- expr\_list\_gather, 180
- expr\_list\_gather\_every, 181
- expr\_list\_get, 182
- expr\_list\_head, 183
- expr\_list\_join, 183
- expr\_list\_last, 184
- expr\_list\_len, 185
- expr\_list\_max, 185
- expr\_list\_mean, 186
- expr\_list\_median, 186
- expr\_list\_min, 187
- expr\_list\_n\_unique, 187
- expr\_list\_reverse, 188
- expr\_list\_sample, 188
- expr\_list\_set\_difference, 189
- expr\_list\_set\_intersection, 190
- expr\_list\_set\_symmetric\_difference, 191
- expr\_list\_set\_union, 192
- expr\_list\_shift, 192
- expr\_list\_slice, 193
- expr\_list\_sort, 194
- expr\_list\_std, 195
- expr\_list\_sum, 195
- expr\_list\_tail, 196
- expr\_list\_to\_array, 196
- expr\_list\_unique, 197
- expr\_list\_var, 198

- expr\_meta\_eq, 198
- expr\_meta\_has\_multiple\_outputs, 199
- expr\_meta\_is\_column, 199
- expr\_meta\_is\_column\_selection, 200
- expr\_meta\_is\_regex\_projection, 201
- expr\_meta\_ne, 201
- expr\_meta\_output\_name, 202
- expr\_meta\_pop, 203
- expr\_meta\_root\_names, 203
- expr\_meta\_serialize, 204
- expr\_meta\_tree\_format, 205
- expr\_meta\_undo\_aliases, 205
- expr\_str\_contains, 210
- expr\_str\_contains\_any, 211
- expr\_str\_count\_matches, 212
- expr\_str\_decode, 212
- expr\_str\_encode, 213
- expr\_str\_ends\_with, 214
- expr\_str\_extract, 215
- expr\_str\_extract\_all, 215
- expr\_str\_extract\_groups, 216
- expr\_str\_extract\_many, 217
- expr\_str\_find, 218
- expr\_str\_head, 219
- expr\_str\_join, 220
- expr\_str\_json\_decode, 221
- expr\_str\_json\_path\_match, 221
- expr\_str\_len\_bytes, 222
- expr\_str\_len\_chars, 223
- expr\_str\_pad\_end, 224
- expr\_str\_pad\_start, 224
- expr\_str\_replace, 225
- expr\_str\_replace\_all, 226
- expr\_str\_replace\_many, 228
- expr\_str\_reverse, 229
- expr\_str\_slice, 229
- expr\_str\_split, 230
- expr\_str\_split\_exact, 231
- expr\_str\_splitn, 230
- expr\_str\_starts\_with, 232
- expr\_str\_strip\_chars, 232
- expr\_str\_strip\_chars\_end, 233
- expr\_str\_strip\_chars\_start, 234
- expr\_str\_strip\_prefix, 234
- expr\_str\_strip\_suffix, 235
- expr\_str\_strptime, 236
- expr\_str\_tail, 238
- expr\_str\_to\_date, 239
- expr\_str\_to\_datetime, 240
- expr\_str\_to\_decimal, 241
- expr\_str\_to\_integer, 242
- expr\_str\_to\_lowercase, 243
- expr\_str\_to\_time, 243
- expr\_str\_to\_uppercase, 244
- expr\_str\_zfill, 244
- expr\_struct\_field, 206
- expr\_struct\_json\_encode, 207
- expr\_struct\_rename\_fields, 207
- expr\_struct\_unnest, 208
- expr\_struct\_with\_fields, 209
- expression, 15, 18, 22, 23, 33, 124–127, 129–132, 134–154, 156–158, 160–199, 201–225, 227–239, 241–277, 279, 280, 282–330, 332–337, 339, 341, 342, 344, 346, 347, 349, 350, 353–355, 357, 358, 360, 362–385, 449–453, 456, 457, 460–462, 464, 466, 468, 470–473, 475–481, 492, 504–508, 517
- expression (*polars\_expr*), 512
- expressions, 22, 101, 115, 116, 426, 427, 441, 442, 504
- factor, 517
- GroupBy, 75
- GroupByDynamic, 77
- hms, 29, 504
- hms::hms, 15, 17, 33, 516, 517
- infer\_polars\_dtype, 386
- infer\_polars\_dtype(), 20, 28, 29, 36, 386
- integer, 15, 17, 33, 516, 517
- is\_convertible\_to\_polars\_expr (*infer\_polars\_dtype*), 386
- is\_convertible\_to\_polars\_expr(), 386
- is\_convertible\_to\_polars\_series (*infer\_polars\_dtype*), 386
- is\_convertible\_to\_polars\_series(), 386
- is\_list\_of\_polars\_dtype (*check\_polars*), 34
- is\_polars (*check\_polars*), 34
- is\_polars\_df (*check\_polars*), 34
- is\_polars\_dtype (*check\_polars*), 34

- `is_polars_expr` (*check\_polars*), 34
- `is_polars_lf` (*check\_polars*), 34
- `is_polars_selector` (*check\_polars*), 34
- `is_polars_series` (*check\_polars*), 34
- `LazyFrame`, 25, 89, 114, 389–391, 394–397, 399–403, 408, 411, 413, 415–419, 422, 423, 426–428, 432–435, 438–441, 443, 444, 474, 496, 497, 499, 502
- `LazyFrame` (*pl\_\_LazyFrame*), 474
- `LazyFrame$filter()`, 287
- `lazyframe__bottom_k`, 388
- `lazyframe__cast`, 389
- `lazyframe__clear`, 390
- `lazyframe__clone`, 391
- `lazyframe__collect`, 392
- `lazyframe__collect_schema`, 394
- `lazyframe__count`, 394
- `lazyframe__drop`, 395
- `lazyframe__drop_nans`, 396
- `lazyframe__drop_nulls`, 397
- `lazyframe__explain`, 397
- `lazyframe__explode`, 399
- `lazyframe__fill_nan`, 400
- `lazyframe__fill_null`, 400
- `lazyframe__filter`, 401
- `lazyframe__first`, 402
- `lazyframe__gather_every`, 403
- `lazyframe__group_by`, 403
- `lazyframe__group_by_dynamic`, 404
- `lazyframe__head`, 408
- `lazyframe__interpolate`, 408
- `lazyframe__join`, 409
- `lazyframe__join_asof`, 411
- `lazyframe__join_where`, 415
- `lazyframe__last`, 416
- `lazyframe__limit` (*lazyframe\_\_head*), 408
- `lazyframe__max`, 416
- `lazyframe__mean`, 417
- `lazyframe__median`, 417
- `lazyframe__merge_sorted`, 418
- `lazyframe__min`, 418
- `lazyframe__null_count`, 419
- `lazyframe__profile`, 419
- `lazyframe__quantile`, 422
- `lazyframe__rename`, 422
- `lazyframe__reverse`, 423
- `lazyframe__rolling`, 424
- `lazyframe__select`, 425
- `lazyframe__select_seq`, 426
- `lazyframe__serialize`, 427
- `lazyframe__set_sorted`, 428
- `lazyframe__shift`, 428
- `lazyframe__sink_csv`, 429
- `lazyframe__sink_parquet` (*parquet\_statistics*), 444
- `lazyframe__slice`, 432
- `lazyframe__sort`, 432
- `lazyframe__std`, 433
- `lazyframe__sum`, 434
- `lazyframe__tail`, 434
- `lazyframe__to_dot`, 436
- `lazyframe__top_k`, 435
- `lazyframe__unique`, 438
- `lazyframe__unnest`, 439
- `lazyframe__unpivot`, 439
- `lazyframe__var`, 440
- `lazyframe__with_columns`, 441
- `lazyframe__with_columns_seq`, 442
- `lazyframe__with_row_index`, 443
- `LazyFrames`, 454, 455
- `LazyGroupBy`, 406, 425
- `list`, 18, 20, 28, 387, 516, 518
- `literals`, 101, 115, 116, 426, 427, 441, 442, 504
- `logical`, 517
- `map_elements()`, 310
- `NA_integer_`, 15, 17, 33, 516
- `nanoarrow::infer_nanoarrow_schema()`, 386
- `OlsonNames()`, 47
- `parquet_statistics`, 444
- `parquet_statistics()`, 122, 446
- `pl`, 447
- `pl$all()`, 248
- `pl$arg_sort_by()`, 255
- `pl$col()`, 23, 512
- `pl$date_range()`, 468
- `pl$date_ranges()`, 467
- `pl$Datetime()`, 47
- `pl$Datetime(ms)`, 237, 240
- `pl$datetime_range()`, 465

- pl\$datetime\_ranges(), 463
- pl\$lit(), 22, 23, 512
- pl\$struct(), 23
- pl\_\_all, 449
- pl\_\_all\_horizontal, 449
- pl\_\_any, 450
- pl\_\_any\_horizontal, 451
- pl\_\_arg\_where, 452
- pl\_\_Array (*polars\_dtype*), 509
- pl\_\_Categorical (*polars\_dtype*), 509
- pl\_\_coalesce, 452
- pl\_\_col, 453
- pl\_\_concat, 454
- pl\_\_concat\_list, 455
- pl\_\_concat\_str, 456
- pl\_\_cum\_sum, 457
- pl\_\_DataFrame, 458
- pl\_\_date\_range, 465
- pl\_\_date\_ranges, 467
- pl\_\_Datetime (*polars\_dtype*), 509
- pl\_\_datetime, 459
- pl\_\_datetime\_range, 461
- pl\_\_datetime\_ranges, 463
- pl\_\_Decimal (*polars\_dtype*), 509
- pl\_\_Duration (*polars\_dtype*), 509
- pl\_\_duration, 469
- pl\_\_element, 470
- pl\_\_Enum (*polars\_dtype*), 509
- pl\_\_first, 471
- pl\_\_int\_range, 472
- pl\_\_int\_ranges, 473
- pl\_\_last, 473
- pl\_\_LazyFrame, 474
- pl\_\_len, 475
- pl\_\_List (*polars\_dtype*), 509
- pl\_\_lit, 476
- pl\_\_max, 477
- pl\_\_max\_horizontal, 478
- pl\_\_mean\_horizontal, 478
- pl\_\_min, 479
- pl\_\_min\_horizontal, 480
- pl\_\_nth, 481
- pl\_\_read\_csv, 481
- pl\_\_read\_ipc, 484
- pl\_\_read\_ipc\_stream, 486
- pl\_\_read\_ndjson, 487
- pl\_\_read\_parquet, 489
- pl\_\_repeat\_, 492
- pl\_\_scan\_csv, 493
- pl\_\_scan\_ipc, 496
- pl\_\_scan\_ndjson, 498
- pl\_\_scan\_parquet, 500
- pl\_\_Series, 502
- pl\_\_show\_versions, 503
- pl\_\_Struct (*polars\_dtype*), 509
- pl\_\_struct, 504
- pl\_\_sum, 505
- pl\_\_sum\_horizontal, 506
- pl\_\_time\_range, 506
- pl\_\_time\_ranges, 508
- pl\_api\_register\_series\_namespace, 448
- polars data frames, 34
- Polars DataFrame, 19
- polars DataFrame, 19, 28, 502
- polars DataType, 387
- Polars DataType(s), 453
- polars expression, 142, 460, 469, 470
- polars expressions, 34
- polars lazy frames, 34
- polars selectors, 34
- Polars Series, 458
- polars Series, 26, 29, 72, 80, 81, 89, 90, 92, 106, 514
- polars series, 34
- polars\_data\_frame, 29
- polars\_data\_frame (*pl\_DataFrame*), 458
- polars\_dtype, 509
- polars\_duration\_string, 511
- polars\_expr, 512
- polars\_lazy\_frame, 21, 29
- polars\_lazy\_frame (*pl\_LazyFrame*), 474
- polars\_series, 21
- polars\_series (*pl\_Series*), 502
- POSIXct, 15, 16, 18, 29, 33, 517, 518
- POSIXlt, 29
- R data frame, 16
- R Data Frames, 458
- R vector, 17
- R vectors, 458, 502
- raw, 23, 476
- rlang, 36
- rlang::abort(), 36
- rlang::as\_function(), 32
- RollingGroupBy, 100



Series, [17](#), [19–21](#), [23](#), [28](#), [29](#), [36](#), [73](#), [110](#),  
[386](#), [454](#), [455](#), [458](#), [474](#), [503](#), [512](#),  
[515](#)

Series (*pl\_\_Series*), [502](#)

series\_\_alias, [514](#)

series\_\_n\_chunks, [514](#)

series\_\_rename (*series\_\_alias*), [514](#)

series\_\_to\_frame, [515](#)

series\_\_to\_r\_vector, [515](#)

series\_struct\_unnest, [513](#)

strptime(), [236](#)

struct dtype, [20](#)

struct type, [19](#)

the vctrs package, [387](#)

tibble, [17](#), [34](#), [504](#), [516](#), [518](#)

vctrs, [504](#)

vctrs::list\_of, [518](#)

vctrs::list\_of(), [28](#)

vctrs::unspecified, [518](#)

vctrs::vec\_as\_names(), [32](#)

vctrs\_rcrd, [28](#)

vector, [515](#), [516](#), [518](#)

vectors, [17](#)