Package: polars (via r-universe)

October 18, 2026

Title R Bindings for the 'polars' Rust Library

Version 2.0.0.9000

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

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

BugReports https://github.com/pola-rs/r-polars/issues

Depends R (>= 4.3)

Imports rlang (>= 1.2.0), S7 (>= 0.2.1)

Suggests arrow, bit64, blob, carrier (>= 0.2.0), cli, clock, curl, data.table, hms, jsonlite, knitr, mirai (>= 2.3.0), nanoarrow (>= 0.6.0), nycflights13, patrick (>= 0.3.0), pillar, pkgload, purrr (>= 1.1.0), reticulate (>= 1.43.0), rmarkdown, testthat (>= 3.3.2), tibble (>= 3.3.0), vctrs, withr

VignetteBuilder knitr

Config/Needs/dev devtools, lifecycle, readr, glue, RcppTOML, smvr

Config/Needs/lint fs, lintr

Config/Needs/website etiennebacher/altdoc, future.apply

Config/polars/lib-version 2.0.0-rc.3

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first lazyframe-frame, *-s3-base, polars_options, expr-expr, expr-*

Encoding UTF-8

SystemRequirements Cargo (Rust's package manager), rustc

Config/pak/sysreqs libclang-dev

Repository https://rpolars.r-universe.dev

Date/Publication 2026-10-18 12:36:27 UTC

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

RemoteRef HEAD

RemoteSha 157189d5cdd23d15edc0da201f9db127487e71f5

Contents

as.data.frame.polars_data_frame . . . 17
as.list.polars_data_frame . . . 19
as_nanoarrow_array_stream.polars_data_frame . . . 22
as_polars_df . . . 24
as_polars_expr . . . 27
as_polars_lf . . . 29
as_polars_series . . . 30
as_tibble.polars_data_frame . . . 36
check_polars . . . 39
cs . . . 42
cs___all . . . 43
cs___alpha . . . 44
cs___alphanumeric . . . 45
cs___array . . . 46
cs___binary . . . 47
cs___boolean . . . 48
cs___by_dtype . . . 49
cs___by_index . . . 50
cs___by_name . . . 51
cs___categorical . . . 52
cs___contains . . . 52
cs___date . . . 53
cs___datetime . . . 54
cs___decimal . . . 55
cs___digit . . . 56
cs___duration . . . 57
cs___empty . . . 58
cs___ends_with . . . 59
cs___enum . . . 60
cs___exclude . . . 61
cs___first . . . 62
cs___float . . . 63
cs___integer . . . 63
cs___last . . . 64
cs___list . . . 65
cs___matches . . . 66

cs___nested . . . 67
cs___numeric . . . 68
cs___signed_integer . . . 68
cs___starts_with . . . 69
cs___string . . . 70
cs___struct . . . 71
cs___temporal . . . 72
cs___time . . . 73
cs___unsigned_integer . . . 73
dataframe___bottom_k . . . 74
dataframe___cast . . . 75
dataframe___clear . . . 76
dataframe___clone . . . 77
dataframe___count . . . 78
dataframe___describe . . . 78
dataframe___drop . . . 79
dataframe___drop_nans . . . 80
dataframe___drop_nulls . . . 81
dataframe___equals . . . 82
dataframe___explode . . . 82
dataframe___fill_nan . . . 83
dataframe___fill_null . . . 84
dataframe___filter . . . 85
dataframe___gather_every . . . 85
dataframe___get_column . . . 86
dataframe___get_column_index . . . 87
dataframe___get_columns . . . 87
dataframe___glimpse . . . 88
dataframe___group_by . . . 89
dataframe___group_by_dynamic . . . 90
dataframe___hash_rows . . . 93
dataframe___head . . . 94
dataframe___is_duplicated . . . 95
dataframe___is_empty . . . 95
dataframe___is_unique . . . 96
dataframe___join . . . 96
dataframe___join_asof . . . 99
dataframe___join_where . . . 102
dataframe___lazy . . . 104
dataframe___map_columns . . . 104
dataframe___max . . . 105
dataframe___max_horizontal . . . 106
dataframe___mean . . . 106
dataframe___mean_horizontal . . . 107
dataframe___median . . . 107
dataframe___merge_sorted . . . 108
dataframe___min . . . 109
dataframe___min_horizontal . . . 109

dataframe___n_chunks . . . 110
dataframe___partition_by . . . 110
dataframe___pivot . . . 111
dataframe___quantile . . . 113
dataframe___rechunk . . . 114
dataframe___remove . . . 114
dataframe___rename . . . 115
dataframe___reverse . . . 116
dataframe___rolling . . . 116
dataframe___sample . . . 118
dataframe___select . . . 119
dataframe___select_seq . . . 120
dataframe___serialize . . . 121
dataframe___set_sorted . . . 122
dataframe___shift . . . 122
dataframe___slice . . . 123
dataframe___sort . . . 124
dataframe___std . . . 125
dataframe___sum . . . 126
dataframe___sum_horizontal . . . 126
dataframe___tail . . . 127
dataframe___to_dummies . . . 127
dataframe___to_series . . . 128
dataframe___to_struct . . . 129
dataframe___top_k . . . 130
dataframe___transpose . . . 131
dataframe___unique . . . 132
dataframe___unnest . . . 133
dataframe___unpivot . . . 134
dataframe___unstack . . . 135
dataframe___var . . . 136
dataframe___with_columns . . . 136
dataframe___with_columns_seq . . . 137
dataframe___with_row_index . . . 138
dataframe___write_csv . . . 139
dataframe___write_ipc . . . 141
dataframe___write_ipc_stream . . . 143
dataframe___write_json . . . 144
dataframe___write_ndjson . . . 145
dataframe___write_parquet . . . 146
datatype___to_dtype_expr . . . 148
datatype_expr___default_value . . . 148
datatype_expr___display . . . 150
datatype_expr___inner_dtype . . . 151
datatype_expr___matches . . . 151
datatype_expr___wrap_in_array . . . 152
datatype_expr___wrap_in_list . . . 153
expr___abs . . . 153

expr___add . . . 154
expr___alias . . . 155
expr___all . . . 155
expr___and . . . 156
expr___any . . . 157
expr___append . . . 158
expr___approx_n_unique . . . 158
expr___arccos . . . 159
expr___arccosh . . . 159
expr___arcsin . . . 160
expr___arcsinh . . . 160
expr___arctan . . . 161
expr___arctanh . . . 161
expr___arg_max . . . 162
expr___arg_min . . . 162
expr___arg_sort . . . 163
expr___arg_true . . . 163
expr___arg_unique . . . 164
expr___backward_fill . . . 164
expr___bin_intervals . . . 165
expr___bin_quantiles . . . 166
expr___bin_ranks . . . 167
expr___bitwise_and . . . 168
expr___bitwise_count_ones . . . 168
expr___bitwise_count_zeros . . . 169
expr___bitwise_leading_ones . . . 169
expr___bitwise_leading_zeros . . . 170
expr___bitwise_or . . . 170
expr___bitwise_trailing_ones . . . 171
expr___bitwise_trailing_zeros . . . 171
expr___bitwise_xor . . . 172
expr___bottom_k . . . 172
expr___bottom_k_by . . . 173
expr___cast . . . 174
expr___cbrt . . . 175
expr___ceil . . . 175
expr___clip . . . 176
expr___cos . . . 177
expr___cosh . . . 177
expr___cot . . . 178
expr___count . . . 178
expr___cum_count . . . 179
expr___cum_max . . . 179
expr___cum_min . . . 180
expr___cum_prod . . . 181
expr___cum_sum . . . 181
expr___cumulative_eval . . . 182
expr___cut . . . 183

expr___degrees . . . 184
expr___diff . . . 184
expr___dot . . . 185
expr___drop_nans . . . 185
expr___drop_nulls . . . 186
expr___entropy . . . 186
expr___eq . . . 187
expr___eq_missing . . . 188
expr___ewm_mean . . . 188
expr___ewm_mean_by . . . 190
expr___ewm_std . . . 191
expr___ewm_var . . . 193
expr___exclude . . . 194
expr___exp . . . 195
expr___explode . . . 195
expr___extend_constant . . . 196
expr___fill_nan . . . 197
expr___fill_null . . . 197
expr___filter . . . 198
expr___first . . . 199
expr___floor . . . 199
expr___floor_div . . . 200
expr___forward_fill . . . 200
expr___gather . . . 201
expr___gather_every . . . 202
expr___ge . . . 202
expr___get . . . 203
expr___gt . . . 204
expr___has_nulls . . . 204
expr___hash . . . 205
expr___head . . . 206
expr___hist . . . 206
expr___implode . . . 207
expr___index_of . . . 208
expr___interpolate . . . 208
expr___interpolate_by . . . 209
expr___is_between . . . 210
expr___is_close . . . 211
expr___is_duplicated . . . 212
expr___is_finite . . . 212
expr___is_first_distinct . . . 213
expr___is_in . . . 213
expr___is_infinite . . . 214
expr___is_last_distinct . . . 214
expr___is_nan . . . 215
expr___is_not_nan . . . 215
expr___is_not_null . . . 216
expr___is_null . . . 216

expr___is_unique . . . 217
expr___item . . . 217
expr___kurtosis . . . 218
expr___last . . . 219
expr___le . . . 220
expr___len . . . 220
expr___limit . . . 221
expr___log . . . 221
expr___log10 . . . 222
expr___log1p . . . 222
expr___lower_bound . . . 223
expr___lt . . . 223
expr___map_batches . . . 224
expr___max . . . 225
expr___max_by . . . 225
expr___mean . . . 226
expr___median . . . 226
expr___min . . . 227
expr___min_by . . . 227
expr___mod . . . 228
expr___mode . . . 228
expr___mul . . . 229
expr___n_unique . . . 230
expr___nan_max . . . 230
expr___nan_min . . . 231
expr___ne . . . 231
expr___ne_missing . . . 232
expr___not . . . 233
expr___null_count . . . 233
expr___or . . . 234
expr___over . . . 234
expr___pct_change . . . 236
expr___peak_max . . . 237
expr___peak_min . . . 237
expr___pow . . . 238
expr___product . . . 238
expr___qcut . . . 239
expr___quantile . . . 240
expr___radians . . . 241
expr___rank . . . 241
expr___reinterpret . . . 243
expr___repeat_by . . . 243
expr___replace . . . 244
expr___replace_strict . . . 245
expr___reshape . . . 247
expr___reverse . . . 248
expr___rle . . . 248
expr___rle_id . . . 249

expr___rolling . . . 249
expr___rolling_kurtosis . . . 251
expr___rolling_max . . . 252
expr___rolling_max_by . . . 253
expr___rolling_mean . . . 255
expr___rolling_mean_by . . . 256
expr___rolling_median . . . 258
expr___rolling_median_by . . . 259
expr___rolling_min . . . 261
expr___rolling_min_by . . . 263
expr___rolling_quantile . . . 265
expr___rolling_quantile_by . . . 266
expr___rolling_rank . . . 269
expr___rolling_rank_by . . . 270
expr___rolling_skew . . . 272
expr___rolling_std . . . 273
expr___rolling_std_by . . . 274
expr___rolling_sum . . . 277
expr___rolling_sum_by . . . 278
expr___rolling_var . . . 280
expr___rolling_var_by . . . 281
expr___round . . . 283
expr___round_sig_figs . . . 284
expr___sample . . . 285
expr___search_sorted . . . 286
expr___set_sorted . . . 287
expr___shift . . . 287
expr___shuffle . . . 288
expr___sign . . . 289
expr___sin . . . 289
expr___sinh . . . 290
expr___skew . . . 290
expr___slice . . . 291
expr___sort . . . 292
expr___sort_by . . . 293
expr___sqrt . . . 294
expr___std . . . 295
expr___sub . . . 295
expr___sum . . . 296
expr___tail . . . 296
expr___tan . . . 297
expr___tanh . . . 297
expr___to_physical . . . 298
expr___top_k . . . 298
expr___top_k_by . . . 299
expr___true_div . . . 300
expr___truncate . . . 301
expr___unique . . . 302

expr___unique_counts . . . 303
expr___upper_bound . . . 303
expr___value_counts . . . 304
expr___var . . . 305
expr___xor . . . 305
expr_arr_agg . . . 306
expr_arr_all . . . 307
expr_arr_any . . . 307
expr_arr_arg_max . . . 308
expr_arr_arg_min . . . 308
expr_arr_contains . . . 309
expr_arr_count_matches . . . 310
expr_arr_dot . . . 310
expr_arr_eval . . . 311
expr_arr_explode . . . 312
expr_arr_first . . . 312
expr_arr_get . . . 313
expr_arr_join . . . 314
expr_arr_last . . . 314
expr_arr_len . . . 315
expr_arr_max . . . 315
expr_arr_median . . . 316
expr_arr_min . . . 316
expr_arr_n_unique . . . 317
expr_arr_reverse . . . 317
expr_arr_shift . . . 318
expr_arr_sort . . . 318
expr_arr_std . . . 319
expr_arr_sum . . . 319
expr_arr_to_list . . . 320
expr_arr_to_struct . . . 320
expr_arr_unique . . . 321
expr_arr_var . . . 322
expr_bin_contains . . . 322
expr_bin_decode . . . 323
expr_bin_encode . . . 324
expr_bin_ends_with . . . 324
expr_bin_get . . . 325
expr_bin_reinterpret . . . 326
expr_bin_size . . . 327
expr_bin_starts_with . . . 327
expr_cat_physical . . . 328
expr_cat_to . . . 329
expr_dt_add_business_days . . . 330
expr_dt_base_utc_offset . . . 331
expr_dt_cast_time_unit . . . 332
expr_dt_century . . . 332
expr_dt_combine . . . 333

expr_dt_convert_time_zone . . . 334
expr_dt_date . . . 335
expr_dt_day . . . 335
expr_dt_days_in_month . . . 336
expr_dt_dst_offset . . . 336
expr_dt_epoch . . . 337
expr_dt_hour . . . 338
expr_dt_is_leap_year . . . 338
expr_dt_iso_year . . . 339
expr_dt_microsecond . . . 339
expr_dt_millennium . . . 340
expr_dt_millisecond . . . 341
expr_dt_minute . . . 341
expr_dt_month . . . 342
expr_dt_month_end . . . 343
expr_dt_month_start . . . 343
expr_dt_nanosecond . . . 344
expr_dt_offset_by . . . 344
expr_dt_ordinal_day . . . 346
expr_dt_quarter . . . 346
expr_dt_replace . . . 347
expr_dt_replace_time_zone . . . 348
expr_dt_round . . . 349
expr_dt_second . . . 351
expr_dt_strftime . . . 352
expr_dt_time . . . 352
expr_dt_timestamp . . . 353
expr_dt_to_string . . . 354
expr_dt_total_days . . . 355
expr_dt_total_hours . . . 356
expr_dt_total_microseconds . . . 356
expr_dt_total_milliseconds . . . 357
expr_dt_total_minutes . . . 358
expr_dt_total_nanoseconds . . . 359
expr_dt_total_seconds . . . 359
expr_dt_truncate . . . 360
expr_dt_week . . . 361
expr_dt_weekday . . . 362
expr_dt_year . . . 363
expr_list_agg . . . 363
expr_list_all . . . 364
expr_list_any . . . 365
expr_list_arg_max . . . 365
expr_list_arg_min . . . 366
expr_list_concat . . . 366
expr_list_contains . . . 367
expr_list_count_matches . . . 368
expr_list_diff . . . 368

expr_list_drop_nulls . . . 369
expr_list_eval . . . 369
expr_list_explode . . . 370
expr_list_first . . . 371
expr_list_gather . . . 372
expr_list_gather_every . . . 373
expr_list_get . . . 373
expr_list_head . . . 374
expr_list_join . . . 375
expr_list_last . . . 376
expr_list_len . . . 376
expr_list_max . . . 377
expr_list_mean . . . 377
expr_list_median . . . 378
expr_list_min . . . 378
expr_list_n_unique . . . 379
expr_list_reverse . . . 379
expr_list_sample . . . 380
expr_list_set_difference . . . 381
expr_list_set_intersection . . . 382
expr_list_set_symmetric_difference . . . 382
expr_list_set_union . . . 383
expr_list_shift . . . 384
expr_list_slice . . . 385
expr_list_sort . . . 385
expr_list_std . . . 386
expr_list_sum . . . 387
expr_list_tail . . . 387
expr_list_to_array . . . 388
expr_list_to_struct . . . 389
expr_list_unique . . . 390
expr_list_var . . . 390
expr_meta_eq . . . 391
expr_meta_has_multiple_outputs . . . 391
expr_meta_is_column . . . 392
expr_meta_is_column_selection . . . 392
expr_meta_is_literal . . . 393
expr_meta_is_regex_projection . . . 394
expr_meta_ne . . . 394
expr_meta_output_name . . . 395
expr_meta_pop . . . 396
expr_meta_root_names . . . 396
expr_meta_serialize . . . 397
expr_meta_tree_format . . . 398
expr_meta_undo_aliases . . . 399
expr_name_keep . . . 399
expr_name_prefix . . . 400
expr_name_prefix_fields . . . 400

expr_name_replace . . . 401
expr_name_suffix . . . 402
expr_name_suffix_fields . . . 402
expr_name_to_lowercase . . . 403
expr_name_to_uppercase . . . 403
expr_str_contains . . . 404
expr_str_contains_any . . . 405
expr_str_count_matches . . . 406
expr_str_decode . . . 407
expr_str_encode . . . 407
expr_str_ends_with . . . 408
expr_str_escape_regex . . . 409
expr_str_extract . . . 409
expr_str_extract_all . . . 410
expr_str_extract_groups . . . 411
expr_str_extract_many . . . 412
expr_str_find . . . 413
expr_str_find_many . . . 414
expr_str_head . . . 415
expr_str_join . . . 416
expr_str_json_decode . . . 417
expr_str_json_path_match . . . 417
expr_str_len_bytes . . . 418
expr_str_len_chars . . . 419
expr_str_normalize . . . 420
expr_str_pad_end . . . 420
expr_str_pad_start . . . 421
expr_str_replace . . . 421
expr_str_replace_all . . . 423
expr_str_replace_many . . . 424
expr_str_reverse . . . 425
expr_str_slice . . . 426
expr_str_split . . . 427
expr_str_split_exact . . . 428
expr_str_splitn . . . 429
expr_str_starts_with . . . 429
expr_str_strip_chars . . . 430
expr_str_strip_chars_end . . . 431
expr_str_strip_chars_start . . . 431
expr_str_strip_prefix . . . 432
expr_str_strip_suffix . . . 433
expr_str_strptime . . . 434
expr_str_tail . . . 436
expr_str_to_date . . . 437
expr_str_to_datetime . . . 438
expr_str_to_decimal . . . 439
expr_str_to_integer . . . 440
expr_str_to_lowercase . . . 441

expr_str_to_time . . . 441
expr_str_to_titlecase . . . 442
expr_str_to_uppercase . . . 443
expr_str_zfill . . . 443
expr_struct_drop . . . 444
expr_struct_field . . . 445
expr_struct_json_encode . . . 446
expr_struct_rename_fields . . . 446
expr_struct_unnest . . . 447
expr_struct_with_fields . . . 448
groupby___agg . . . 449
groupby___having . . . 450
groupby___head . . . 450
groupby___len . . . 451
groupby___max . . . 452
groupby___mean . . . 452
groupby___median . . . 453
groupby___min . . . 453
groupby___n_unique . . . 454
groupby___quantile . . . 455
groupby___sum . . . 456
groupby___tail . . . 456
infer_polars_dtype . . . 457
knit_print.polars_data_frame . . . 459
lazyframe___bottom_k . . . 460
lazyframe___cast . . . 461
lazyframe___clear . . . 462
lazyframe___clone . . . 463
lazyframe___collect . . . 464
lazyframe___collect_schema . . . 465
lazyframe___count . . . 466
lazyframe___describe . . . 466
lazyframe___drop . . . 467
lazyframe___drop_nans . . . 468
lazyframe___drop_nulls . . . 469
lazyframe___explain . . . 470
lazyframe___explode . . . 471
lazyframe___fill_nan . . . 472
lazyframe___fill_null . . . 473
lazyframe___filter . . . 474
lazyframe___first . . . 475
lazyframe___gather_every . . . 475
lazyframe___group_by . . . 476
lazyframe___group_by_dynamic . . . 477
lazyframe___head . . . 480
lazyframe___interpolate . . . 481
lazyframe___join . . . 481
lazyframe___join_asof . . . 484

lazyframe___join_where . . . 487
lazyframe___last . . . 489
lazyframe___max . . . 489
lazyframe___mean . . . 490
lazyframe___median . . . 490
lazyframe___merge_sorted . . . 491
lazyframe___min . . . 492
lazyframe___null_count . . . 492
lazyframe___pivot . . . 493
lazyframe___quantile . . . 495
lazyframe___remove . . . 496
lazyframe___rename . . . 497
lazyframe___reverse . . . 498
lazyframe___rolling . . . 498
lazyframe___select . . . 500
lazyframe___select_seq . . . 501
lazyframe___serialize . . . 502
lazyframe___set_sorted . . . 502
lazyframe___shift . . . 503
lazyframe___sink_batches . . . 504
lazyframe___sink_csv . . . 506
lazyframe___sink_ipc . . . 510
lazyframe___sink_ndjson . . . 513
lazyframe___slice . . . 515
lazyframe___sort . . . 515
lazyframe___sql . . . 516
lazyframe___std . . . 517
lazyframe___sum . . . 518
lazyframe___tail . . . 519
lazyframe___to_dot . . . 519
lazyframe___top_k . . . 520
lazyframe___unique . . . 521
lazyframe___unnest . . . 522
lazyframe___unpivot . . . 523
lazyframe___var . . . 524
lazyframe___with_columns . . . 524
lazyframe___with_columns_seq . . . 525
lazyframe___with_row_index . . . 526
lazygroupby___agg . . . 527
lazygroupby___having . . . 528
lazygroupby___head . . . 529
lazygroupby___len . . . 529
lazygroupby___max . . . 530
lazygroupby___mean . . . 530
lazygroupby___median . . . 531
lazygroupby___min . . . 532
lazygroupby___n_unique . . . 532
lazygroupby___quantile . . . 533

lazygroupby___sum . . . 534
lazygroupby___tail . . . 534
parquet_statistics . . . 535
pl . . . 538
pl___all . . . 539
pl___all_horizontal . . . 540
pl___any . . . 540
pl___any_horizontal . . . 541
pl___arg_sort_by . . . 542
pl___arg_where . . . 543
pl___coalesce . . . 544
pl___col . . . 544
pl___collect_all . . . 545
pl___concat . . . 546
pl___concat_arr . . . 548
pl___concat_list . . . 549
pl___concat_str . . . 550
pl___cum_sum . . . 551
pl___DataFrame . . . 552
pl___date . . . 553
pl___date_range . . . 554
pl___date_ranges . . . 556
pl___datetime . . . 558
pl___datetime_range . . . 560
pl___datetime_ranges . . . 562
pl___dtype_of . . . 564
pl___duration . . . 564
pl___element . . . 566
pl___explain_all . . . 566
pl___first . . . 567
pl___int_range . . . 568
pl___int_ranges . . . 569
pl___last . . . 569
pl___LazyFrame . . . 570
pl___len . . . 571
pl___linear_space . . . 572
pl___linear_spaces . . . 573
pl___list . . . 574
pl___lit . . . 575
pl___max . . . 576
pl___max_horizontal . . . 576
pl___mean_horizontal . . . 577
pl___min . . . 578
pl___min_horizontal . . . 579
pl___nth . . . 579
pl___PartitionBy . . . 580
pl___read_csv . . . 582
pl___read_ipc . . . 585

pl___read_ipc_stream . . . 587
pl___read_lines . . . 588
pl___read_ndjson . . . 589
pl___read_parquet . . . 591
pl___repeat_ . . . 593
pl___row_index . . . 594
pl___scan_csv . . . 595
pl___scan_ipc . . . 598
pl___scan_lines . . . 600
pl___scan_ndjson . . . 601
pl___scan_parquet . . . 603
pl___Series . . . 606
pl___show_versions . . . 607
pl___SQLContext . . . 607
pl___struct . . . 608
pl___sum . . . 609
pl___sum_horizontal . . . 610
pl___thread_pool_size . . . 610
pl___time_range . . . 611
pl___time_ranges . . . 612
pl___when . . . 614
pl_api_register_series_namespace . . . 616
polars_code_completion_activate . . . 617
polars_dtype . . . 618
polars_envvars . . . 621
polars_expr . . . 623
polars_info . . . 623
polars_options . . . 624
QueryOptFlags . . . 625
s3-arithmetic . . . 626
series___alias . . . 628
series___chunk_lengths . . . 628
series___is_empty . . . 629
series___n_chunks . . . 629
series___rechunk . . . 630
series___serialize . . . 631
series___shrink_dtype . . . 631
series___to_frame . . . 632
series___to_r_vector . . . 633
series_list_to_struct . . . 637
series_str_json_decode . . . 638
series_str_strptime . . . 639
series_str_to_datetime . . . 640
series_str_to_decimal . . . 642
series_struct_unnest . . . 643
sql_context___execute . . . 643
sql_context___register . . . 644
sql_context___register_many . . . 645

sql_context___tables 646
sql_context___unregister 646

Index **648**

`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(struct = "dataframe")`.

Usage

```
## S3 method for class 'polars_data_frame'
as.data.frame(
  x,
  ...,
  uint8 = c("integer", "raw"),
  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,
  ...,
  uint8 = c("integer", "raw"),
  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

`x`	A polars object
`...`	Passed to `as_polars_df()`.

`uint8` [**Experimental**] Determine how to convert Polars' UInt8 type values to R type. One of the followings:

- `"integer"` (default): Convert to the R's integer type.
- `"raw"`: Convert to the R's raw type. If the value is `null`, export as `00`.

`int64` [**Experimental**] 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` [**Experimental**] 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` [**Experimental**] Determine how to convert Polars' Time type values to R class. One of the followings:

- `"hms"` (default): Convert to the hms::hms class. If the hms package is not installed, a warning will be shown.
- `"ITime"`: Convert to the data.table::ITime class. The data.table package must be installed.

`decimal` [**Experimental**] 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`

[**Experimental**] 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 datetime 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` [**Experimental**] 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` [**Experimental**] 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

These S3 methods call `as_polars_df(x, ...)$get_columns()` with `rlang::set_names()`, or, `as_polars_df(x, ...)$to_struct()$to_r_vector() |> as.list()` depending on the `as_series` argument.

Usage

```
## S3 method for class 'polars_data_frame'
as.list(
  x,
  ...,
  as_series = TRUE,
  uint8 = c("integer", "raw"),
  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 = TRUE,
  uint8 = c("integer", "raw"),
  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

`x` A polars object

`...` Passed to `as_polars_df()`.

`as_series` Whether to convert each column to an R vector or a Series. If `TRUE` (default), return a list of Series, otherwise a list of vectors.

`uint8` **[Experimental]** Determine how to convert Polars' UInt8 type values to R type. One of the followings:

- `"integer"` (default): Convert to the R's integer type.
- `"raw"`: Convert to the R's raw type. If the value is `null`, export as `00`.

`int64` **[Experimental]** 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` **[Experimental]** 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` **[Experimental]** Determine how to convert Polars' Time type values to R class. One of the followings:

- `"hms"` (default): Convert to the hms::hms class. If the hms package is not installed, a warning will be shown.

- `"ITime"`: Convert to the data.table::ITime class. The data.table package must be installed.

`struct` [**Experimental**] 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.

`decimal` [**Experimental**] 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`

[**Experimental**] 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 datetime 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` [**Experimental**] 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` [**Experimental**] 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_nanoarrow_array_stream.polars_data_frame

Create a nanoarrow_array_stream from a Polars object

Description

Create a nanoarrow_array_stream from a Polars object

Usage

```
## S3 method for class 'polars_data_frame'
as_nanoarrow_array_stream(
  x,
  ...,
  schema = NULL,
  polars_compat_level = c("newest", "oldest")
)

## S3 method for class 'polars_lazy_frame'
as_nanoarrow_array_stream(
  x,
  ...,
  schema = NULL,
  polars_compat_level = c("newest", "oldest"),
  maintain_order = FALSE,
  chunk_size = NULL
)

## S3 method for class 'polars_series'
as_nanoarrow_array_stream(
  x,
  ...,
  schema = NULL,
  polars_compat_level = c("newest", "oldest")
)
```

Arguments

`x` A polars object

`...` Ignored.

`schema` **[Experimental]** An optional nanoarrow schema object. If specified, interpret the nanoarrow schema as a corresponding polars dtype and then convert the original object using `<Series>$cast()`. Note that the schema of the returned object cannot be fully controlled because Polars does not support all Arrow types.

For LazyFrame, this argument is not yet supported.

`polars_compat_level` **[Experimental]** Determines the compatibility level when exporting Polars' internal data structures. When specifying a new compatibility level, Polars exports its internal data structures that might not be interpretable by other Arrow implementations. The level can be specified as the name (e.g., `"newest"`) or as a scalar integer (Currently, `0` and `1` are supported).

- `"newest"` **[Experimental]** (default): Use the highest level, currently same as `1` (Low compatibility).
- `"oldest"`: Same as `0` (High compatibility).

`maintain_order` **[Experimental]** Maintain the order in which data is processed. Setting this to `FALSE` will be slightly faster.

`chunk_size` **[Experimental]** A positive integer or `NULL` (default). The number of rows each chunk collected from the lazy computation before being sent to the Arrow C stream. If `NULL`, Polars tries to compute an optimal chunk size automatically.

Value

A nanoarrow array stream

See Also

- `as_polars_series(<nanoarrow_array_stream>)`: Import an array stream as a Series via the Arrow C stream interface.

Examples

```
# Zero-copy round trip via nanoarrow
as_polars_series(letters[1:3], name = "letters") |>
  nanoarrow::as_nanoarrow_array_stream() |>
  as_polars_series()

# Specify the schema
as_polars_series(1:3, name = "numbers") |>
  nanoarrow::as_nanoarrow_array_stream(schema = nanoarrow::na_uint8()) |>
  as_polars_series()
```

```
# DataFrame support
pl$DataFrame(a = 1:3, b = letters[1:3]) |>
  nanoarrow::as_nanoarrow_array_stream() |>
  as_polars_df()

# Compatibility level
as_polars_series(letters[1:3]) |>
  nanoarrow::as_nanoarrow_array_stream(polars_compat_level = 1) |>
  nanoarrow::infer_nanoarrow_schema() |>
  format()

as_polars_series(letters[1:3]) |>
  nanoarrow::as_nanoarrow_array_stream(polars_compat_level = "oldest") |>
  nanoarrow::infer_nanoarrow_schema() |>
  format()
```

`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,
  ...,
  engine = c("auto", "in-memory", "streaming"),
```

```
  optimizations = pl$QueryOptFlags()
)

## 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

`x`	An R object.
`...`	Additional arguments passed to the methods.
`column_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.
`from_struct`	A logical. If `TRUE` (default) and the Series data type is a struct, the `<Series>$struct$unnest()` method is used to create a DataFrame from the struct Series. In this case, the `column_name` argument is ignored.
`engine`	The engine name to use for processing the query. One of the followings: • `"auto"` (default): Select the engine automatically. In Polars 1.16, the `"in-memory"` engine is selected for most cases. Starting with Polars 2.0, the streaming engine is selected instead. Use `"in-memory"` to keep the current execution engine explicitly. • `"in-memory"`: Use the in-memory engine. This is an escape hatch for queries whose incidental row order differs under streaming. • `"streaming"`: Use the streaming engine.
`optimizations`	[**Experimental**] A QueryOptFlags object to indicate optimization passes done during query optimization.

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 Series by `as_polars_series()`.
- All of the Series must be converted to the same length, except for the case of length 1, which will be recycled to match the length of the other Series if they have a length other than 1.

- 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 — *Create a Polars expression from an R object*

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, ..., keep_series = FALSE)

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

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

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

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

Arguments

`x`	An R object.
`...`	Additional arguments passed to the methods.
`keep_series`	A logical value indicating whether to treat the object as a Series or scalar value. If `TRUE`, the output is ensured to be a Series literal even if the length of the object is `1`.
`as_lit`	A logical value indicating whether to treat vector as literal values or not. This argument is always set to `TRUE` when calling this function from `pl$lit()`, and expects to return literal values. See examples for details.
`raw_as_binary`	A logical value indicating whether to convert raw vector to a Binary type scalar. If `TRUE` (default), the output is a Binary type scalar instead of UInt8 type literal.

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. Otherwise, the default method is called.

S3 method for raw:

If the `raw_as_binary` argument is `TRUE` (default), the raw vector is converted to a Binary type scalar. Otherwise, the default method is called.

S3 method for `NULL`:

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

Value

A polars expression

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(c("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)

# raw
as_polars_expr(as.raw(1))
as_polars_expr(as.raw(1), raw_as_binary = FALSE)
as_polars_expr(charToRaw("foo"))
as_polars_expr(charToRaw("foo"), raw_as_binary = FALSE)
```

```
# NULL
as_polars_expr(NULL)

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

# default method (for double)
as_polars_expr(double(0))
as_polars_expr(1)
as_polars_expr(NA_real_)
as_polars_expr(c(1, 2))

# 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")))

# default method (for Series)
as_polars_series(1) |>
  as_polars_expr()

# polars_expr
as_polars_expr(pl$col(c("a", "b")))
```

`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

`x`	An R object.
`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.
`...`	Additional arguments passed to the methods.
`strict`	A logical value to indicate whether throwing an error when the input list's elements have different data types. If `FALSE` (default), all elements are automatically cast to the super type, or, casting to the super type is failed, the value will be `null`. If `TRUE`, the first non-`NULL` 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 `tzone` 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-oclock or less than 0-oclock, 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(as.raw(c(0, 16, 255)))

# 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(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",
    "unique_quiet",
    "universal_quiet"
  ),
  uint8 = c("integer", "raw"),
  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",
```

```
    "unique_quiet",
    "universal_quiet"
  ),
  uint8 = c("integer", "raw"),
  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

`x` A polars object

`...` Passed to `as_polars_df()`.

`.name_repair` Treatment of problematic column names:

- `"minimal"`: No name repair or checks, beyond basic existence,
- `"unique"`: Make sure names are unique and not empty,
- `"check_unique"`: (default value), no name repair, but check they are `unique`,
- `"universal"`: Make the names `unique` and syntactic
- `"unique_quiet"`: Same as `"unique"`, but "quiet"
- `"universal_quiet"`: Same as `"universal"`, but "quiet"
- a function: apply custom name repair (e.g., `.name_repair = make.names` for names in the style of base R).
- A purrr-style anonymous function, see `rlang::as_function()`

This argument is passed on as `repair` to `vctrs::vec_as_names()`. See there for more details on these terms and the strategies used to enforce them.

`uint8` **[Experimental]** Determine how to convert Polars' UInt8 type values to R type. One of the followings:

- `"integer"` (default): Convert to the R's integer type.
- `"raw"`: Convert to the R's raw type. If the value is `null`, export as `00`.

`int64` **[Experimental]** 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` [**Experimental**] 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` [**Experimental**] Determine how to convert Polars' Time type values to R class. One of the followings:

- `"hms"` (default): Convert to the hms::hms class. If the hms package is not installed, a warning will be shown.
- `"ITime"`: Convert to the data.table::ITime class. The data.table package must be installed.

`decimal` [**Experimental**] 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`

[**Experimental**] 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 datetime 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` [**Experimental**] 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` [**Experimental**] 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

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` of `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:

- `*_df`: For polars data frames.
- `*_dtype`: For polars data types.
- `*_dtype_expr`: [**Experimental**] For polars data type expressions.
- `*_expr`: For polars expressions.
- `*_lf`: For polars lazy frames.
- `*_partitioning_scheme`: For polars partitioning schemes.
- `*_selector`: For polars selectors.
- `*_series`: For polars series.

Usage

```
is_polars_df(x)

is_polars_dtype(x)

is_polars_dtype_expr(x, ...)

is_polars_expr(x, ...)

is_polars_lf(x)

is_polars_selector(x, ...)
```

```
is_polars_series(x)

is_polars_partitioning_scheme(x)

is_list_of_polars_dtype(x, n = NULL)

is_list_of_polars_expr(x, n = NULL)

is_list_of_polars_lf(x, n = NULL)

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

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

check_polars_dtype_expr(
  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()
)

check_polars_partitioning_scheme(
  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()
)

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

Arguments

`x`	An object to check.
`...`	Arguments passed to `rlang::abort()`.
`n`	Expected length of a vector.
`allow_null`	If `TRUE`, `NULL` is allowed as a valid input.

`arg`	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.
`call`	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` 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

selector__as_expr()
```

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.

Selector operators combine selectors as sets. When one operand is a normal expression, the selector is materialized automatically and the operation is applied element-wise to the selected values. `<selector>$as_expr()` can be used to materialize a selector explicitly.

Examples

```
cs

df <- pl$DataFrame(
  colx = c("aa", "bb", "cc"),
  coly = c(TRUE, FALSE, TRUE),
  colz = c(1, 2, 3),
)

# Inverting the boolean selector will choose the non-boolean columns:
df$select(!cs$boolean())

# To invert the values in the selected boolean columns,
# we need to materialize the selector as a standard expression instead:
df$select(!cs$boolean()$as_expr())
```

---

cs__all — *Select all columns*

Description

Select all columns

Usage

```
cs__all()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by 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 — *Select all columns with alphabetic names (e.g. only letters)*

Description

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

Usage

```
cs__alpha(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}`) 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 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))
```

---

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

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 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__array — *Select all array columns*

Description

[Experimental]

Usage

```
cs__array(inner = NULL, ..., width = NULL)
```

Arguments

`inner`	An optional inner selector to select columns having specific inner data types. If `NULL`, all inner types are selected.
`...`	These dots are for future extensions and must be empty.
`width`	An optional integer specifying the width of the array columns to select. If `NULL`, all widths are selected.

Value

A Polars selector

See Also

- cs for the documentation on operators supported by selectors.
- `cs$by_dtype()`: Select all columns matching the given dtype(s).
- `cs$list()`: Select all list columns.
- `cs$nested()`: Select all nested columns.

Examples

```
df <- pl$DataFrame(
  foo = list(c("xx", "yy"), c("x", "y")),
  bar = list(123, 456),
  baz = c(2.0, 5.5),
  .schema_overrides = list(
    foo = pl$Array(pl$String, 2),
    bar = pl$Array(pl$Int64, 1)
  )
)

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

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

# If you want to select specific array columns,
# you can specify the inner data type and/or width:
df$select(cs$array(cs$string()))
df$select(cs$array(width = 1))
df$select(cs$array(cs$string() | cs$numeric(), width = 2))
```

---

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 selectors.

Examples

```
df <- pl$select(
  a = charToRaw("hello"),
  b = pl$lit("world"),
  c = charToRaw("!"),
  d = pl$lit(":"),
)

# 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 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` *Select all columns matching the given dtypes*

Description

Select all columns matching the given dtypes

Usage

```
cs__by_dtype(dtypes, ...)
```

Arguments

`dtypes`	A Polars data type or list of Polars data types to select.
`...`	These dots are for future extensions and must be empty.

Value

A Polars selector

See Also

cs for the documentation on operators supported by 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(c(pl$Date, pl$String)))

# Select all columns that are not of date or string dtype:
df$select(!cs$by_dtype(c(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, ..., require_all = TRUE)
```

Arguments

`indices`	0-based column indices to select. Negative indexing is supported.
`...`	These dots are for future extensions and must be empty.
`require_all`	Whether to match all indices (the default) or any of the indices.

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 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)), require_all = FALSE))
df$select(cs$by_index(c(0, seq(101, 0, -25)), require_all = FALSE))

# 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(names, ..., require_all = TRUE, expand_patterns = FALSE)
```

Arguments

`names`	A character vector of column names to select.
`...`	These dots are for future extensions and must be empty.
`require_all`	Whether to match all names (the default) or any of the names.
`expand_patterns`	Whether to expand regex patterns (`^...$`) and wildcards (`*`) in names. Default is `FALSE` (treat names as literals).

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 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(c("foo", "bar")))

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

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

cs__categorical *Select all categorical columns*

Description

Select all categorical columns

Usage

```
cs__categorical()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by 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 *Select columns whose names contain the given literal substring(s)*

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 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 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 *Select all datetime columns*

Description

Select all datetime columns

Usage

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

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.

`time_zone` One of the followings. The value or each element of the vector will be passed to the `time_zone` argument of the `pl$Datetime()` function:

- A character vector of one or more timezone strings, as defined in `OlsonNames()`.
- `NULL` to select Datetime columns that do not have a timezone.
- `"*"` to select Datetime columns that have any timezone.
- A list of single timezone strings , `"*"`, and `NULL` to select Datetime columns that do not have a timezone or have the (specific) timezone. For example, the default value `list("*", NULL)` selects all Datetime columns.

Value

A Polars selector

See Also

cs for the documentation on operators supported by 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 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 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 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__empty` *Select no columns*

Description

This is useful for composition with other selectors.

Usage

```
cs__empty()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by selectors.

Examples

```
pl$DataFrame(a = 1, b = 2)$select(cs$empty())
```

`cs__ends_with` *Select columns that end with the given substring(s)*

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 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"))
```

`cs__enum` *Select all enum columns*

Description

[**Experimental**]

Usage

```
cs__enum()
```

Value

A Polars selector

See Also

- cs for the documentation on operators supported by selectors.
- `cs$by_dtype()`: Select all columns matching the given dtype(s).
- `cs$categorical()`: Select all categorical columns.

Examples

```
df <- pl$DataFrame(
  foo = c("xx", "yy"),
  bar = c("aa", "bb"),
  baz = c(2.0, 5.5),
  .schema_overrides = list(
    foo = pl$Enum(c("xx", "yy")),
    bar = pl$Enum(c("aa", "bb"))
  )
)

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

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

# If you want to select specific enum columns,
# you can use the `by_dtype()` selector:
df$select(cs$by_dtype(pl$Enum(c("aa", "bb"))))
```

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

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 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` *Select the first column in the current scope*

Description

Select the first column in the current scope

Usage

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

Arguments

`...`	These dots are for future extensions and must be empty.
`strict`	Require the column exists.

Value

A Polars selector

See Also

cs for the documentation on operators supported by 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 — *Select all float columns.*

Description

Select all float columns.

Usage

```
cs__float()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by 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 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 — *Select the last column in the current scope*

Description

Select the last column in the current scope

Usage

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

Arguments

`...`	These dots are for future extensions and must be empty.
`strict`	Require the column exists.

Value

A Polars selector

See Also

cs for the documentation on operators supported by 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__list` *Select all list columns*

Description

[**Experimental**]

Usage

```
cs__list(inner = NULL)
```

Arguments

`inner`	An optional inner selector to select columns having specific inner data types. If `NULL`, all inner types are selected.

Value

A Polars selector

See Also

- cs for the documentation on operators supported by selectors.
- `cs$by_dtype()`: Select all columns matching the given dtype(s).
- `cs$array()`: Select all array columns.
- `cs$nested()`: Select all nested columns.

Examples

```
df <- pl$DataFrame(
  foo = list(c("xx", "yy"), "x"),
  bar = list(c(123, 456), 789),
  baz = c(2.0, 5.5),
)

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

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

# If you want to select specific list columns,
# you can specify the inner data type with a selector:
df$select(cs$list(cs$string()))
```

---

cs__matches — *Select all columns that match the given regex pattern*

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 <https://docs.rs/regex/`

Value

A Polars selector

See Also

cs for the documentation on operators supported by 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$)"))
```

`cs__nested` *Select all nested columns*

Description

[**Experimental**] A nested column is a list, array or struct.

Usage

```
cs__nested()
```

Value

A Polars selector

See Also

- cs for the documentation on operators supported by selectors.
- `cs$by_dtype()`: Select all columns matching the given dtype(s).
- `cs$list()`: Select all list columns.
- `cs$array()`: Select all array columns.
- `cs$struct()`: Select all struct columns.

Examples

```
df <- pl$DataFrame(
  foo = data.frame(a = c("xx", "x"), b = c("yy", "y")),
  bar = c(123, 456),
  baz = c(2, 5.5),
  wow = list(c(1, 2), c(3)),
)

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

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

cs__numeric *Select all numeric columns.*

Description

Select all numeric columns.

Usage

```
cs__numeric()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by 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 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 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 — *Select all String (and, optionally, Categorical) string columns.*

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 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__struct` *Select all struct columns*

Description

[**Experimental**]

Usage

```
cs__struct()
```

Value

A Polars selector

See Also

- cs for the documentation on operators supported by selectors.
- `cs$by_dtype()`: Select all columns matching the given dtype(s).
- `cs$list()`: Select all list columns.
- `cs$array()`: Select all array columns.
- `cs$nested()`: Select all nested columns.

Examples

```
df <- pl$DataFrame(
  foo = data.frame(a = c("xx", "x"), b = c("yy", "y")),
  bar = data.frame(a = c(123, 456), b = c(789, 101)),
  baz = c(2.0, 5.5),
)

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

# Select all columns except for those that are struct:
```

```
df$select(!cs$struct())

# If you want to select specific struct columns,
# you can use the `by_dtype()` selector:
df$select(cs$by_dtype(pl$Struct(
  a = pl$String,
  b = pl$String
)))
```

`cs__temporal` *Select all temporal columns*

Description

Select all temporal columns

Usage

```
cs__temporal()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by 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 *Select all time columns*

Description

Select all time columns

Usage

```
cs__time()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by 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 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 unsigned):
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

`k`	Number of rows to return.
`...`	These dots are for future extensions and must be empty.
`by`	Column(s) used to determine the bottom rows. Accepts expression input. Strings are parsed as column names.
`reverse`	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 *Cast DataFrame column(s) to the specified dtype*

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

`...`	<`dynamic-dots`> 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__count` *Return the number of non-null elements for each column*

Description

Return the number of non-null elements for each column

Usage

```
dataframe__count()
```

Value

A polars DataFrame

Examples

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

`dataframe__describe` *Summary statistics for a DataFrame.*

Description

Summary statistics for a DataFrame.

Usage

```
dataframe__describe(
  percentiles = c(0.25, 0.5, 0.75),
  ...,
  interpolation = c(
    "nearest",
    "higher",
    "lower",
    "midpoint",
    "linear",
    "equiprobable"
  )
)
```

Arguments

`percentiles`	One or more percentiles to include in the summary statistics. All values must be in the range `[0; 1]`.
`...`	These dots are for future extensions and must be empty.
`interpolation`	Interpolation method for computing quantiles. Must be one of `"nearest"`, `"higher"`, `"lower"`, `"midpoint"`, or `"linear"`.

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  int = 1:3,
  float = c(0.5, NA, 2.5),
  string = c(letters[1:2], NA),
  date = c(as.Date("2024-01-20"), as.Date("2024-01-21"), NA),
  cat = factor(c(letters[1:2], NA)),
  bool = c(TRUE, FALSE, NA)
)
df

# Show default frame statistics:
df$describe()

# Customize which percentiles are displayed, applying linear interpolation:
df$describe(
  percentiles = c(0.1, 0.3, 0.5, 0.7, 0.9),
  interpolation = "linear"
)
```

`dataframe__drop` *Remove columns*

Description

Remove columns

Usage

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

Arguments

`...`	<`dynamic-dots`> Column names or selectors that should be removed.
`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 names or selectors for which are considered. If empty (default), use all columns (same as specifying with the selector cs$all()).

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 names or selectors for which are considered. If empty (default), use all columns (same as specifying with the selector `cs$all()`).

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.
`...`	These dots are for future extensions and must be empty.
`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(..., empty_as_null = FALSE, keep_nulls = TRUE)
```

Arguments

`...`	<`dynamic-dots`> Column names or selectors defining them. The underlying columns being exploded must be of the `List` or `Array` data type.
`empty_as_null`	Indicates to explode an empty list/array into a `null`. Defaults to `FALSE`.
`keep_nulls`	Indicates to explode a `null` list/array into a `null`.

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 = NULL,
  strategy = NULL,
  limit = NULL,
  ...,
  matches_supertype = TRUE
)
```

Arguments

`value`	Value used to fill null values.
`strategy`	Strategy used to fill null values. Must be one of: `"forward"`, `"backward"`, `"min"`, `"max"`, `"mean"`, `"zero"`, `"one"`, or `NULL` (default).
`limit`	Number of consecutive null values to fill when using the `"forward"` or `"backward"` strategy.
`...`	These dots are for future extensions and must be empty.
`matches_supertype`	Fill all matching supertypes of the fill `value` 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_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__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__glimpse *Show a dense preview of the DataFrame*

Description

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

Usage

```
dataframe__glimpse(..., max_items_per_column = 10, max_colname_length = 50)
```

Arguments

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

`max_items_per_column` Maximum number of items to show per column.

`max_colname_length` Maximum length of the displayed column names. Values that exceed this value are truncated with a trailing ellipsis.

Value

Returns a character value (invisibly)

Examples

```
df <- as_polars_df(iris)
df$glimpse()

df$glimpse(max_items_per_column = 3)

df$glimpse(max_items_per_column = 3, max_colname_length = 3)
```

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

An object of class `polars_group_by`

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 + 2 * every, start + 2 * every + period)
- …

where `start` is determined by `start_by`, `offset`, `every`, and the earliest datapoint. See the `start_by` argument description for details.

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

`index_column` Column used to group based on the time window. Often of type Date/Datetime. This column must be sorted in ascending order (or, if `group_by` 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 In64. Note that Int32 gets temporarily cast to Int64, so if performance matters, use an Int64 column.

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

`every` Interval of the window.

`period` Length of the window. If `NULL` (default), it will equal `every`.

`offset` Offset of the window, does not take effect if `start_by = "datapoint"`. Defaults to zero.

`include_boundaries` Add two columns `"_lower_boundary"` and `"_upper_boundary"` columns that show the boundaries of the window. This will impact performance because it's harder to parallelize.

`closed` Define which sides of the interval are closed (inclusive). Default is `"left"`.

`label` Define which label to use for the window:

- `"left"`: lower boundary of the window
- `"right"`: upper boundary of the window
- `"datapoint"`: 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.

`group_by` 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

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

An object of class `polars_group_by_dynamic`

See Also

- `<DataFrame>$rolling()`

Examples

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

# 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__hash_rows` *Hash and combine the rows in this DataFrame*

Description

The hash value is of type UInt64.

Usage

```
dataframe__hash_rows(seed = 0)
```

Arguments

`seed`	Random seed parameter. Defaults to 0.

Details

This implementation does not guarantee stable results across different Polars versions. Its stability is only guaranteed within a single version.

Value

A polars Series

Examples

```
df <- pl$DataFrame(
  foo = c(1, NA, 3, 4),
  ham = c("a", "b", NA, "d")
)
df$hash_rows(seed = 42)
```

---

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 polars 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"),
  nulls_equal = FALSE,
  coalesce = NULL,
  maintain_order = c("none", "left", "right", "left_right", "right_left"),
  allow_parallel = TRUE,
  force_parallel = FALSE
)
```

Arguments

`other` DataFrame to join with.

`on` Either a vector of column names or a list of expressions and/or strings. Use `left_on` and `right_on` if the column names to match on are different between the two DataFrames.

`how` One of the following methods:

- "inner": returns rows that have matching values in both tables
- "left": returns all rows from the left table, and the matched rows from the right table
- "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.

`nulls_equal` Join on null values. By default null values will never produce matches.

`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.

`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.

- `"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 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.

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

`other`	DataFrame to join with.
`...`	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.
`on`	Either a vector of column names or a list of expressions and/or strings. Use `left_on` and `right_on` if the column names to match on are different between the two LazyFrames.

`by_left`, `by_right`
: Same as `by` but only for the left or the right table. They must have the same length.

`by`
: Join on these columns before performing asof join. Either a vector of column names or a list of expressions and/or strings. Use `left_by` and `right_by` if the column names to match on are different between the two tables.

`strategy`
: Strategy for where to find match:

- `"backward"` (default): search for the last row in the right table whose `on` key is less than or equal to the left key.
- `"forward"`: search for the first row in the right table whose `on` key is greater than or equal to the left key.
- `"nearest"`: search for the last row in the right table whose value is nearest to the left key. String keys are not currently supported for a nearest search.

`suffix`
: Suffix to add to duplicated column names.

`tolerance`
: 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).

`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 `on` / `left_on` / `right_on` columns):

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

`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 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]

By default, 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,
  ...,
  how = c("inner", "left", "right"),
  suffix = "_right"
)
```

Arguments

`other`	DataFrame to join with.
`...`	<`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>"`.
`how`	Join strategy, one of `"inner"` (default), `"left"` or `"right"`.
`suffix`	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__map_columns` *Apply eager functions to columns of a DataFrame*

Description

[Experimental]

A higher-order function to apply functions to selected columns of a DataFrame, similar to purrr::map_at. The selected columns will be materialized as Series before the function is applied, and the return value of the function will be converted back to a Series by as_polars_series.

It is recommended to use `<dataframe>$with_columns()` unless they are using expressions that are only possible on Series and not on Expr. This is almost never the case, except for a very select few functions that cannot know the output datatype without looking at the data.

Usage

```
dataframe__map_columns(column_names, lambda)
```

Arguments

`column_names`	Column names or selectors specifying columns to apply the function to.
`lambda`	A function that will receive a Series as the first argument.

Value

A polars DataFrame

Examples

```
df1 <- pl$DataFrame(
  a = 1:4,
  b = c("10", "20", "30", "40"),
)

# Apply `<series>$shrink_dtype()` to the "a" column
df1$map_columns("a", \(s) s$shrink_dtype())

# Convert the "b" column to integer by the base R function `as.integer()`
df1$map_columns("b", \(s) s$to_r_vector() |> as.integer())

df2 <- pl$DataFrame(
  a = c('{"x":"a"}', NA, '{"x":"b"}', NA),
  b = c('{"a":1, "b": true}', NA, '{"a":2, "b": false}', NA),
)

# Apply `<series>$str$json_decode()` to both the "a" and "b" columns
dtype <- pl$Struct(value = pl$Int64)
df2$map_columns(c("a", "b"), \(s) s$str$json_decode(dtype))

# Use a selector to apply the function to all columns
df2$map_columns(cs$all(), \(s) s$str$json_decode(dtype))
```

`dataframe__max` *Aggregate the columns in the DataFrame to their maximum value*

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

Get the maximum value horizontally across columns.

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

Aggregate the columns in the DataFrame to their mean value

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

Take the mean of all values horizontally across columns.

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 `FALSE`, any null value in the input will lead to a null output.

Value

A polars Series

Examples

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

Arguments

`other`	Other DataFrame that must be merged.
`key`	Key that is sorted.
`...`	These dots are for future extensions and must be empty.
`maintain_order`	If `TRUE`, the output is guaranteed to have left-biased ordering for equal keys: rows from the left frame appear before rows from the right frame when their keys are equal.

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 — *Aggregate the columns in the DataFrame to their minimum value*

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 — *Get the minimum value horizontally across columns.*

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 names or selectors to group by. Must contain at least one column.
`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

Reshape data from long to wide format, known as "pivot wider".

Usage

```
dataframe__pivot(
  on,
  on_columns = NULL,
  ...,
  index = NULL,
  values = NULL,
  aggregate_function = NULL,
  maintain_order = TRUE,
  sort_columns = FALSE,
  separator = "_",
  column_naming = c("auto", "combine")
)
```

Arguments

- `on` — The column(s) whose values will be used as the new columns of the output.
- `on_columns` — What value combinations will be considered for the output table. If `NULL` (default), all unique values found in the `on` column(s) will be used.
- `...` — These dots are for future extensions and must be empty.
- `index` — The column(s) that remain from the input to the output. The output will have one row for each unique combination of the `index`'s values. If `NULL`, all remaining columns not specified in `on` and `values` will be used. At least one of `index` and `values` must be specified.
- `values` — The existing column(s) of values which will be moved under the new columns from `index`. If an aggregation is specified, these are the values on which the aggregation will be computed. If `NULL`, all remaining columns not specified in `on` and `index` will be used. At least one of `index` and `values` must be specified.
- `aggregate_function` — Choose from:
  - `NULL` (default): no aggregation takes place, will raise error if multiple values are in group. Same as `pl$element()$item(allow_empty = TRUE)`.
  - A predefined aggregate function string, one of `"min"`, `"max"`, `"first"`, `"last"`, `"sum"`, `"mean"`, `"median"`, `"len"`, `"item"`. Same as `pl$element()$<function>()`.
  - An expression to do the aggregation.
- `maintain_order` — Ensure the values of `index` are sorted by discovery order.
- `sort_columns` — Sort the transposed columns by name. Default is by order of discovery.
- `separator` — Used as separator/delimiter in generated column names in case of multiple values columns.
- `column_naming` — How resulting column names will be constructed. `"auto"` (default) combines with separator if there are multiple `values` columns, otherwise just uses the `on_columns` names. `"combine"` always combines the `values` columns' names with the `on_columns` names.

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")

# You can use selectors too - here we include all test scores
# in the pivoted table:
df$pivot("subject", values = cs$starts_with("test"))

# 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(),
)
```

`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")
)
```

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)
```

`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__remove` *Remove rows, dropping those that match the given predicate expression(s)*

Description

The original order of the remaining rows is preserved. Rows where the filter does not evaluate to `TRUE` are retained (this includes rows where the predicate evaluates as `null`).

Usage

```
dataframe__remove(...)
```

Arguments

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

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  ccy = c("USD", "EUR", "USD", "JPY"),
  year = c(2021, 2022, 2023, 2023),
  total = c(3245, NA, -6680, 25000),
)

# Remove rows matching a condition. Note that the row where `total` is null
# is kept:
df$remove(pl$col("total") >= 0)

# Note that this is *not* the same as simply inverting the condition in
# `$filter()` because `$filter()` doesn't keep predicates that evaluate to
# null:
df$filter(pl$col("total") < 0)

# We can use multiple conditions, combined with and/or operators:
df$remove((pl$col("total") >= 0) & (pl$col("ccy") == "USD"))

df$remove((pl$col("total") >= 0) | (pl$col("ccy") == "USD"))
```

---

dataframe__rename *Rename column names*

Description

Rename column names

Usage

```
dataframe__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 DataFrame

Examples

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

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

---

dataframe__reverse *Reverse the DataFrame*

Description

Reverse the DataFrame

Usage

```
dataframe__reverse()
```

Value

A polars DataFrame

Examples

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

df$reverse()
```

---

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 `<t_0, t_1, ..., t_n>`, then by default the windows created will be:

- `(t_0 - period, t_0]`
- `(t_1 - period, t_1]`

- …
- `(t_n - 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

```
dataframe__rolling(
  index_column,
  ...,
  period,
  offset = NULL,
  closed = c("right", "left", "both", "none"),
  group_by = NULL
)
```

Arguments

`index_column`	Column used to group based on the time window. Often of type Date/Datetime. This column must be sorted in ascending order (or, if `group_by` 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 In64. Note that Int32 gets temporarily cast to Int64, so if performance matters, use an Int64 column.
`...`	These dots are for future extensions and must be empty.
`period`	Length of the window - must be non-negative.
`offset`	Offset of the window. Default is `-period`.
`closed`	Define which sides of the interval are closed (inclusive). Default is `"left"`.
`group_by`	Also group by this column/these columns. Can be expressions or objects coercible to expressions.

Value

An object of class `polars_rolling_group_by`

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 *Sample from this DataFrame*

Description

Sample from this DataFrame

Usage

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

Arguments

`n`	Number of items to return. Accepts a scalar value or a Series; expressions are not supported. Values are parsed as literals. 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. Accepts a scalar value or a Series; expressions are not supported. Values are parsed as literals. Cannot be used with `n`.
`with_replacement`	Allow values to be sampled more than once.

`shuffle` If `TRUE`, explicitly shuffle the sampled data points. If `FALSE`, maintain their relative order. If `NULL` (default), use a high-performance algorithm without guaranteeing an order.

`seed` Seed for the random number generator. If `NULL` (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__select_seq

Select columns from this DataFrame

Description

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

Usage

```
dataframe__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 DataFrame

Examples

```
df <- pl$DataFrame(
  foo = 1:3,
  bar = 6:8,
  ham = letters[1:3]
)
df$select_seq("foo", bar2 = pl$col("bar") * 2)
```

dataframe__serialize *Serialize the DataFrame to a binary format*

Description

Serialize the DataFrame to a binary format. Currently, this format is uncompressed Arrow IPC stream format, so other Apache Arrow implementations may be able to read it.

Usage

```
dataframe__serialize()

pl__deserialize_df(data)
```

Arguments

data	A raw vector of serialized DataFrame.

Value

- `<dataframe>$serialize()` returns raw vector of serialized DataFrame.
- `pl$deserialize_df()` returns a deserialized DataFrame.

Examples

```
df <- pl$DataFrame(
  foo = 1:3,
  bar = 6:8,
)$cast(bar = pl$UInt8)

# Serialize the DataFrame to a binary format
serialized <- df$serialize()
serialized

# The bytes can later be deserialized back to a DataFrame
pl$deserialize_df(serialized)

# Other Apache Arrow implementations may be able to read it.
if (requireNamespace("arrow", quietly = TRUE)) {
  arrow::read_ipc_stream(serialized, as_data_frame = FALSE)
}
```

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

`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 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

`offset`	Start index, can be a negative value. This is 0-indexed, so `offset = 1` skips the first row.
`length`	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, ]
```

---

dataframe__sort *Sort a DataFrame by the given columns*

Description

Sort a DataFrame by the given columns

Usage

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

Arguments

`...`	<dynamic-dots> 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 `TRUE`, 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 — *Aggregate the columns of this DataFrame to their standard deviation values*

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 - 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 — *Aggregate the columns of this DataFrame to their sum values*

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 — *Sum all values horizontally across columns.*

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 polars DataFrame

Examples

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

df$tail(3)

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

dataframe__to_dummies — *Convert categorical variables into dummy/indicator variables*

Description

Convert categorical variables into dummy/indicator variables

Usage

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

Arguments

...	<dynamic-dots> Column names or selectors that should be converted to dummy variables. If empty (default), convert all columns (same as specifying with the selector cs$all()).
separator	Separator/delimiter used when generating column names.
drop_first	Remove the first category from the variables being encoded.
drop_nulls	A boolean indicating whether to generate columns for null values.

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  foo = c(1L, 2L),
  bar = c(3, NA),
  ham = c("a", "b")
)
df$to_dummies()

df$to_dummies(drop_first = TRUE)
df$to_dummies(drop_nulls = TRUE)

df$to_dummies("foo", "bar", separator = ":")
df$to_dummies(cs$integer(), separator=":")
df$to_dummies(cs$integer(), drop_first = TRUE, 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

index	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__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

`k`	Number of rows to return.
`...`	These dots are for future extensions and must be empty.
`by`	Column(s) used to determine the bottom rows. Accepts expression input. Strings are parsed as column names.
`reverse`	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 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__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

`...`	These dots are for future extensions and must be empty.
`include_header`	If set, the column names will be added as first column.
`header_name`	If `include_header` is set, this determines the name of the column that will be inserted.
`column_names`	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(
  ...,
  keep = c("any", "none", "first", "last"),
  maintain_order = FALSE
)
```

Arguments

`...` <`dynamic-dots`> Column names or selectors for which are considered. If empty (default), use all columns (same as specifying with the selector `cs$all()`).

`keep` Which of the duplicate rows to keep. Must be one of:

- `"any"`: does not give any guarantee of which row is kept. This allows more optimizations.
- `"none"`: don't keep duplicate rows.
- `"first"`: keep first unique row.
- `"last"`: keep last unique row.

`maintain_order` Keep the same order as the original data. This is more expensive to compute. Setting this to `TRUE` 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(c("bar", "ham"), maintain_order = TRUE)

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

---

dataframe__unnest — *Decompose struct columns into separate columns for each of their fields*

Description

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

Usage

```
dataframe__unnest(..., separator = NULL)
```

Arguments

`...`	<`dynamic-dots`> Name of the struct column(s) or selectors that should be unnested.
`separator`	`NULL` (default) or single string. Rename output column names as combination of the struct column name, name separator and field name.

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("a_and_c", separator = ":")
```

dataframe__unpivot *Unpivot a frame from wide to long format*

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

`on`	Column(s) or selector(s) to use as values variables. If `on` is empty (`cs$empty()`), no columns will be used. If set to `NULL` (default), all columns that are not in `index` will be used.
`...`	These dots are for future extensions and must be empty.
`index`	Column(s) or selector(s) to use as identifier variables.
`variable_name`	Name to give to the new column containing the names of the melted columns. Defaults to `"variable"`.
`value_name`	Name to give to the new column containing the values of the melted columns. Defaults to `"value"`.
	The row order of the result is not guaranteed; use `$sort()` when a stable order is required.

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"))
```

dataframe__unstack *Unstack a long table to a wide form without doing an aggregation*

Description

This can be much faster than a pivot, because it can skip the grouping phase.

Usage

```
dataframe__unstack(
  ...,
  step,
  how = c("vertical", "horizontal"),
  fill_values = NULL
)
```

Arguments

`...`	<dynamic-dots> Column name(s) and selector(s) to include in the operation. If empty, use all columns.
`step`	Number of rows in the unstacked frame.
`how`	Direction of the unstack. Must be one of `"vertical"` or `"horizontal"`.
`fill_values`	Fill values that don't fit the new size with this value. This can be a scalar value or a named list of the sort `list(<column_name> = <fill_value>)`. See examples.

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(x = LETTERS[1:8], y = 1:8)$with_columns(
  z = pl$int_ranges(pl$col("y"), pl$col("y") + 2, dtype = pl$UInt8)
)
df

df$unstack(step = 4, how = "vertical")
df$unstack(step = 2, how = "horizontal")
df$unstack(cs$numeric(), step = 5, fill_values = 0)
df$unstack("x", "y", step = 5, fill_values = list(y = 999, x = "foo"))
```

dataframe__var — *Aggregate the columns in the DataFrame to their variance value*

Description

Aggregate the columns in the DataFrame to their variance value

Usage

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

Arguments

`ddof`	"Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default `ddof` is 1.

Value

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(),
)
```

---

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(),
)
```

---

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,
  compression = c("uncompressed", "gzip", "zstd"),
  compression_level = NULL,
  check_extension = TRUE,
  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,
  decimal_comma = FALSE,
```

```
  null_value = "",
  quote_style = c("necessary", "always", "never", "non_numeric"),
  storage_options = NULL
)
```

Arguments

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

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

`include_bom` Logical, whether to include UTF-8 BOM in the CSV output.

`compression` [**Experimental**] What compression format to use. Must be one of `uncompressed` (default), `gzip`, or `zstd`.

`compression_level` [**Experimental**] The compression level to use, typically 0-9 or `NULL` to let the engine choose.

`check_extension` [**Experimental**] Whether to check if the filename matches the compression settings. Will raise an error if compression is set to `"uncompressed"` and the filename ends in one of `".gz"`, `".zst"`, `".zstd"`, or if `compression != "uncompressed"` and the file uses a mismatched extension. Only applies if file is a path.

`include_header` Logical, whether to include header in the CSV output.

`separator` Separate CSV fields with this symbol.

`line_terminator` String used to end each row.

`quote_char` Byte to use as quoting character.

`batch_size` Number of rows that will be processed per thread.

`datetime_format` A format string, with the specifiers defined by the chrono Rust crate. If no format specified, the default fractional-second precision is inferred from the maximum timeunit found in the frame's Datetime cols (if any).

`date_format` A format string, with the specifiers defined by the chrono Rust crate.

`time_format` A format string, with the specifiers defined by the chrono Rust crate.

`float_scientific` Whether to use scientific form always (`TRUE`), never (`FALSE`), or automatically (`NULL`) for Float32 and Float64 datatypes.

`float_precision` Number of decimal places to write, applied to both Float32 and Float64 datatypes.

`decimal_comma` If `TRUE`, use a comma `","` as the decimal separator instead of a point. Floats will be encapsulated in quotes if necessary.

`null_value` A string representing null values (defaulting to the empty string).

`quote_style` Determines the quoting strategy used. Must be one of:

- `"necessary"` (default): This puts quotes around fields only when necessary. They are necessary when fields contain a quote, delimiter or record terminator. Quotes are also necessary when writing an empty record (which is indistinguishable from a record with one empty field). This is the default.
- `"always"`: This puts quotes around every field. Always.
- `"never"`: This never puts quotes around fields, even if that results in invalid CSV data (e.g.: by not quoting strings containing the separator).
- `"non_numeric"`: This puts quotes around all fields that are non-numeric. Namely, when writing a field that does not parse as a valid float or integer, then quotes will be used even if they aren't strictly necessary.

`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.

Value

`NULL` invisibly.

Examples

```
tmpf <- tempfile()
as_polars_df(mtcars)$write_csv(tmpf)
pl$read_csv(tmpf, infer_schema_files = 10)

as_polars_df(mtcars)$write_csv(tmpf, separator = "|")
pl$read_csv(tmpf, separator = "|", infer_schema_files = 10)
```

`dataframe__write_ipc` *Write to Arrow IPC File Format*

Description

Write to Arrow IPC File Format

Usage

```
dataframe__write_ipc(
  path,
  ...,
  compression = c("uncompressed", "lz4", "zstd"),
  compat_level = c("newest", "oldest"),
  storage_options = NULL
)
```

Arguments

`path` A character. File path to which the file should be written.

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

`compression` Determines the compression algorithm. Must be one of:

- `"uncompressed"` or `NULL`: Write an uncompressed Arrow file.
- `"lz4"`: Fast compression/decompression.
- `"zstd"`: Good compression performance.

`compat_level` Determines the compatibility level when exporting Polars' internal data structures. When specifying a new compatibility level, Polars exports its internal data structures that might not be interpretable by other Arrow implementations. The level can be specified as the name (e.g., `"newest"`) or as a scalar integer (Currently, `0` and `1` are supported).

- `"newest"` [**Experimental**] (default): Use the highest level, currently same as `1` (Low compatibility).
- `"oldest"`: Same as `0` (High compatibility).

`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.

Value

`NULL` invisibly.

Examples

```
tmpf <- tempfile(fileext = ".arrow")
as_polars_df(mtcars)$write_ipc(tmpf, compression = "zstd")

pl$read_ipc(tmpf)
```

---

dataframe__write_ipc_stream

Write to Arrow IPC Stream Format

Description

Write to Arrow IPC Stream Format

Usage

```
dataframe__write_ipc_stream(
  path,
  ...,
  compression = c("uncompressed", "lz4", "zstd"),
  compat_level = c("newest", "oldest")
)
```

Arguments

`path`	A character. File path to which the file should be written.
`...`	These dots are for future extensions and must be empty.
`compression`	Determines the compression algorithm. Must be one of: • `"uncompressed"` or `NULL`: Write an uncompressed Arrow file. • `"lz4"`: Fast compression/decompression. • `"zstd"`: Good compression performance.
`compat_level`	Determines the compatibility level when exporting Polars' internal data structures. When specifying a new compatibility level, Polars exports its internal data structures that might not be interpretable by other Arrow implementations. The level can be specified as the name (e.g., `"newest"`) or as a scalar integer (Currently, `0` and `1` are supported). • `"newest"` [**Experimental**] (default): Use the highest level, currently same as `1` (Low compatibility). • `"oldest"`: Same as `0` (High compatibility).

Value

`NULL` invisibly.

Examples

```
tmpf <- tempfile(fileext = ".arrows")
as_polars_df(mtcars)$write_ipc_stream(tmpf, compression = "zstd")

nanoarrow::read_nanoarrow(tmpf)
```

---

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

`NULL` invisibly.

Examples

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

dat$select(pl$col(c("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,
  ...,
  compression = c("uncompressed", "gzip", "zstd"),
  compression_level = NULL,
  check_extension = TRUE
)
```

Arguments

`file`	File path to which the result will be written.
`...`	These dots are for future extensions and must be empty.
`compression`	[**Experimental**] What compression format to use. Must be one of `uncompressed` (default), `gzip`, or `zstd`.
`compression_level`	[**Experimental**] The compression level to use, typically 0-9 or `NULL` to let the engine choose.
`check_extension`	[**Experimental**] Whether to check if the filename matches the compression settings. Will raise an error if compression is set to `"uncompressed"` and the filename ends in one of `".gz"`, `".zst"`, `".zstd"`, or if `compression != "uncompressed"` and the file uses a mismatched extension. Only applies if file is a path.

Value

`NULL` invisibly.

Examples

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

dat$select(pl$col(c("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", "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,
  mkdir = FALSE
)
```

Arguments

`file` File path to which the result should be written. This should be a path to a directory if writing a partitioned dataset.

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

`compression` The compression method. Defaults to `"lz4"`. Must be one of:

- `"lz4"`: fast compression/decompression.
- `"uncompressed"`
- `"snappy"`: this guarantees that the parquet file will be compatible with older parquet readers.
- `"gzip"`
- `"brotli"`
- `"zstd"`: good compression performance.

`compression_level` `NULL` or integer. The level of compression to use. Only used if method is one of `"gzip"`, `"brotli"`, or `"zstd"`. Higher compression means smaller files on disk:

- `"gzip"`: min-level: 0, max-level: 9, default: 6.
- `"brotli"`: min-level: 0, max-level: 11, default: 1.
- `"zstd"`: min-level: 1, max-level: 22, default: 3.

`statistics` Whether statistics should be written to the Parquet headers. Possible values:

- `TRUE`: enable default set of statistics (default). Some statistics may be disabled.
- `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.

`partition_by` [**Experimental**] A character vector indicating column(s) to partition by. A partitioned dataset will be written if this is specified.

`partition_chunk_size_bytes`
[**Experimental**] 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).

`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.

`mkdir` Recursively create all the directories in the path.

Value

`NULL` invisibly.

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)
```

`datatype__to_dtype_expr`
Return a DataTypeExpr with a static DataType

Description

[**Experimental**] Return a DataTypeExpr with a static DataType.

Usage

```
datatype__to_dtype_expr()
```

Value

A polars datatype expression. This is not the same as a polars expression.

`datatype_expr__default_value`
Get a default value of a specific type

Description

[**Experimental**] Get a default value of a specific type:

- Integers and floats are their zero value as default, unless otherwise specified;
- Temporals are a physical zero as default;
- `pl$Decimal` is zero as default;
- `pl$String` and `pl$Binary` are an empty string;
- `pl$List` is an empty list, unless otherwise specified;
- `pl$Array` is the inner default value repeated over the shape;
- `pl$Struct` is the inner default value for all fields;
- `pl$Enum` is the first category if it exists;
- `pl$Null` and `pl$Categorical` are `null`.

Usage

```
datatype_expr__default_value(
  n = 1,
  ...,
  numeric_to_one = FALSE,
  num_list_values = 0
)
```

Arguments

`n`	Number of values in the output.
`...`	These dots are for future extensions and must be empty.
`numeric_to_one`	Use `1` instead of `0` as the default value for numeric types.
`num_list_values`	The amount of values a list contains.

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. Otherwise, the default method is called.

S3 method for raw:

If the `raw_as_binary` argument is `TRUE` (default), the raw vector is converted to a Binary type scalar. Otherwise, the default method is called.

S3 method for `NULL`:

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

Value

A polars expression

See Also

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

Examples

```
uint32 <- pl$UInt32$to_dtype_expr()
string <- pl$String$to_dtype_expr()
array <- pl$Array(pl$Float64, 2)$to_dtype_expr()

pl$select(
  uint32_default = uint32$default_value(),
  uint32_default_bis = uint32$default_value(numeric_to_one = TRUE),
  string_default = string$default_value(),
  array_default = array$default_value()
)

# Return more values with `n`
pl$select(uint32_default = uint32$default_value(n = 3))

# Customize the number of default values in a list with `num_list_values`
l <- pl$List(pl$Float64)$to_dtype_expr()
pl$select(
  list_default = l$default_value(),
  list_default_bis = l$default_value(num_list_values = 3),
)
```

---

datatype_expr__display

Get a formatted version of the output DataType

Description

[**Experimental**] Get a formatted version of the output DataType.

Usage

```
datatype_expr__display()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = 1:2, y = c("a", "b"), z = c(1, 2))
df$select(
  x = pl$dtype_of("x")$display(),
  y = pl$dtype_of("y")$display(),
  z = pl$dtype_of("z")$display(),
)$transpose(include_header = TRUE)
```

datatype_expr__inner_dtype

Get the inner DataType of a List or Array

Description

[**Experimental**] Get the inner DataType of a List or Array.

Usage

```
datatype_expr__inner_dtype()
```

Value

A polars datatype expression. This is not the same as a polars expression.

Examples

```
df <- pl$DataFrame(
  a = list(1L),
  b = list(list("a")),
  c = list(data.frame(x = 1, y = 2))
)

df$select(
  a_inner_dtype = pl$dtype_of("a")$inner_dtype()$display(),
  b_inner_dtype = pl$dtype_of("b")$inner_dtype()$display(),
  c_inner_dtype = pl$dtype_of("c")$inner_dtype()$display()
)
```

datatype_expr__matches

Get whether the output DataType matches a certain selector

Description

[**Experimental**] Get whether the output DataType matches a certain selector.

Usage

```
datatype_expr__matches(selector)
```

Arguments

`selector`	A selector presenting the data types to match.

Value

A polars datatype expression. This is not the same as a polars expression.

Examples

```
df <- pl$DataFrame(a = 1:3)
df$select(
  a_is_string = pl$dtype_of("a")$matches(cs$string()),
  a_is_integer = pl$dtype_of("a")$matches(cs$integer()),
)
```

---

datatype_expr__wrap_in_array

Get the DataType wrapped in an array

Description

[**Experimental**] Get the DataType wrapped in an array.

Usage

```
datatype_expr__wrap_in_array(..., width)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`width`	Width of the array.

Value

A polars datatype expression. This is not the same as a polars expression.

Examples

```
df <- pl$DataFrame(
  a = 1L,
  b = list("a"),
  c = data.frame(x = 1, y = 2)
)

df$select(
  a_wrapped = pl$dtype_of("a")$wrap_in_array(width = 1)$display(),
  b_wrapped = pl$dtype_of("b")$wrap_in_array(width = 1)$display(),
  c_wrapped = pl$dtype_of("c")$wrap_in_array(width = 1)$display()
)
```

datatype_expr__wrap_in_list

Get the DataType wrapped in a list

Description

[**Experimental**] Get the DataType wrapped in a list.

Usage

```
datatype_expr__wrap_in_list()
```

Value

A polars datatype expression. This is not the same as a polars expression.

Examples

```
df <- pl$DataFrame(
  a = 1L,
  b = list("a"),
  c = data.frame(x = 1, y = 2)
)

df$select(
  a_wrapped = pl$dtype_of("a")$wrap_in_list()$display(),
  b_wrapped = pl$dtype_of("b")$wrap_in_list()$display(),
  c_wrapped = pl$dtype_of("c")$wrap_in_list()$display()
)
```

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` *Add two expressions*

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__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 `FALSE` 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(...)
```

Arguments

`...`	<`dynamic-dots`> One or more integer or boolean expressions to evaluate/combine.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  x = c(5, 6, 7, 4, 8),
  y = c(1.5, 2.5, 1.0, 4.0, -5.75),
  z = c(-9, 2, -1, 4, 8),
)
df$with_columns(
  (pl$col("x") >= pl$col("z"))$and(
    pl$col("y") >= pl$col("z"),
    pl$col("y") == pl$col("y"),
    pl$col("z") <= pl$col("x"),
    pl$col("y") != pl$col("x"),
  )$alias("all")
)
```

---

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 *Append expressions*

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 *Compute inverse hyperbolic cosine*

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 *Compute inverse sine*

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 *Compute inverse hyperbolic sine*

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 *Compute inverse tangent*

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 — *Get the index of the minimal value*

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 *Index of a sort*

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())
```

expr__arg_true *Return indices where expression is true*

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())
```

---

expr__arg_unique *Get the index of the first unique value*

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

[**Superseded**] This is an alias of `$fill_null(strategy = "backward")`.

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__bin_intervals *Bin values into discrete intervals delimited by breakpoints*

Description

[**Experimental**]

Usage

```
expr__bin_intervals(
  intervals = NULL,
  ...,
  n_bins = NULL,
  labels = NULL,
  include_intervals = FALSE,
  right_closed = FALSE
)
```

Arguments

`intervals`	Explicit numeric breakpoints for the bin boundaries.
`...`	These dots are for future extensions and must be empty.
`n_bins`	Number of equal-width bins. Supply exactly one of `intervals` and `n_bins`.
`labels`	Names of the bins. The number of labels must equal the number of bins. Use `NULL` to return `UInt32` bin indices.
`include_intervals`	Return a struct with the bin and its left and right boundaries.
`right_closed`	Use right-closed `(left, right]` bins instead of left-closed `[left, right)` bins.

Value

A polars expression

Examples

```
df <- pl$DataFrame(foo = -2:2)
df$select(bin = pl$col("foo")$bin_intervals(
  intervals = c(-1, 1), labels = c("low", "mid", "high"), right_closed = TRUE
))

# A single integer is a breakpoint; use n_bins for equal-width bins.
df$select(bin = pl$col("foo")$bin_intervals(intervals = 2L, labels = NULL))
df$select(bin = pl$col("foo")$bin_intervals(n_bins = 3, labels = NULL))
```

---

expr__bin_quantiles *Bin values into discrete intervals delimited by their quantiles*

Description

[**Experimental**]

Usage

```
expr__bin_quantiles(
  quantiles = NULL,
  ...,
  n_bins = NULL,
  labels = NULL,
  include_intervals = FALSE,
  right_closed = FALSE
)
```

Arguments

`quantiles`	Explicit non-decreasing quantile probabilities in `[0, 1]`.
`...`	These dots are for future extensions and must be empty.
`n_bins`	Number of bins based on evenly spaced quantile fractions. Supply exactly one of `quantiles` and `n_bins`.
`labels`	Names of the bins. The number of labels must equal the number of bins. Use `NULL` to return `UInt32` bin indices.
`include_intervals`	Return a struct with the bin and its left and right boundaries.
`right_closed`	Use right-closed `(left, right]` bins instead of left-closed `[left, right)` bins.

Value

A polars expression

Examples

```
df <- pl$DataFrame(foo = -2:2)
df$select(bin = pl$col("foo")$bin_quantiles(
  quantiles = c(0.25, 0.75), labels = c("low", "mid", "high")
))
df$select(bin = pl$col("foo")$bin_quantiles(n_bins = 3, labels = NULL))
```

---

expr__bin_ranks *Bin values by their position in sorted order*

Description

[**Experimental**]

Usage

```
expr__bin_ranks(
  ranks = NULL,
  ...,
  n_bins = NULL,
  labels = NULL,
  include_intervals = FALSE
)
```

Arguments

`ranks`	Explicit non-decreasing cumulative fractions in `[0, 1]`.
`...`	These dots are for future extensions and must be empty.
`n_bins`	Number of near-equal-sized bins. Supply exactly one of `ranks` and `n_bins`.
`labels`	Names of the bins. The number of labels must equal the number of bins. Use `NULL` to return `UInt32` bin indices.
`include_intervals`	Return a struct with the bin and its left and right boundaries.

Value

A polars expression

Examples

```
df <- pl$DataFrame(foo = -2:2)
df$select(bin = pl$col("foo")$bin_ranks(
  ranks = c(0.25, 0.75), labels = c("low", "mid", "high")
))
df$select(bin = pl$col("foo")$bin_ranks(n_bins = 3, labels = NULL))
```

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

`by`	Column(s) used to determine the smallest elements. Accepts expression input. Strings are parsed as column names.
`k`	Number of elements to return.
`...`	These dots are for future extensions and must be empty.
`reverse`	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 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 or DataTypeExpr to cast to. A DataTypeExpr can refer to the dtype of another expression.
`...`	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(cbrt = pl$col("a")$cbrt())
```

`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` *Compute hyperbolic cosine*

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 — *Compute cotangent*

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())
```

expr__count — *Get the number of non-null elements in the column*

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())
```

expr__cum_count — *Return the cumulative count of the non-null values in the column*

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 — *Return the cumulative product computed at every element.*

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__cumulative_eval

Return the cumulative count of the non-null values in the column

---

Description

[**Experimental**]

Usage

```
expr__cumulative_eval(expr, ..., min_samples = 1)
```

Arguments

`expr`	Expression to evaluate.
`...`	These dots are for future extensions and must be empty.
`min_samples`	Number of valid values (i.e. `length - null_count`) there should be in the window before the expression is evaluated.

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
  )
)
```

---

expr__cut — *Bin continuous values into discrete categories*

Description

[**Deprecated**]

Usage

```
expr__cut(
  breaks,
  ...,
  labels = NULL,
  left_closed = FALSE,
  include_breaks = FALSE
)
```

Arguments

`breaks`	Numeric vector of unique cut points.
`...`	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 cut points plus one.
`left_closed`	Set the intervals to be left-closed instead of right-closed.
`include_breaks`	Include a column with the right endpoint of the bin each observation falls in. This will change the data type of the output from a Categorical to a Struct.

Details

Deprecated in Polars 2.0.0 in favor of `$bin_intervals()`. Set `right_closed = TRUE` to keep the same interval closure. The replacement is experimental and requires `labels`; use `labels = NULL` for `UInt32` bin indices.

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")
```

---

expr__degrees — *Convert from radians to degrees*

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())
```

---

expr__diff — *Calculate the n-th discrete difference between elements*

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")
)
```

---

expr__dot — *Compute the dot/inner product between two Expressions*

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 — *Drop all floating point NaN values*

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 *Drop all floating point null values*

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())
```

---

expr__entropy *Compute entropy*

Description

Uses the formula `-sum(pk * log(pk)` where `pk` are discrete probabilities.

Usage

```
expr__entropy(base = exp(1), ..., normalize = TRUE)
```

Arguments

`base`	Numeric value used as base, defaults to `exp(1)`.
`...`	These dots are for future extensions and must be empty.
`normalize`	Normalize `pk` 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))
```

---

expr__eq *Check equality*

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

`other`	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"))
)
```

expr__eq_missing *Check equality without* `null` *propagation*

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 = NULL,
  span = NULL,
  half_life = NULL,
  alpha = NULL,
  adjust = TRUE,
  min_samples = 1,
  ignore_nulls = FALSE
)
```

Arguments

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

`com` Specify decay in terms of center of mass, γ, with

$$\alpha = \frac{1}{1+\gamma} \; \forall \; \gamma \geq 0$$

.

`span` Specify decay in terms of span, θ, with

$$\alpha = \frac{2}{\theta+1} \; \forall \; \theta \geq 1$$

`half_life` Specify decay in terms of half-life, λ, with

$$\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \; \forall \; \lambda > 0$$

`alpha` Specify smoothing factor alpha directly, $0 < \alpha \leq 1$.

`adjust` Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings:

- 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_samples` 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, null, x_2) are $(1-\alpha)^2$ and 1 if `adjust = TRUE`, and $(1-\alpha)^2$ and α 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, null, x_2) are $1-\alpha$ and 1 if `adjust = TRUE`, and $1-\alpha$ and α 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, ..., x_{n-1} at times t_0, t_1, ..., 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 τ 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 = NULL,
  span = NULL,
  half_life = NULL,
  alpha = NULL,
  adjust = TRUE,
  bias = FALSE,
  min_samples = 1,
  ignore_nulls = FALSE
)
```

Arguments

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

`com` Specify decay in terms of center of mass, γ, with

$$\alpha = \frac{1}{1+\gamma} \; \forall \; \gamma \geq 0$$

.

`span` Specify decay in terms of span, θ, with

$$\alpha = \frac{2}{\theta+1} \; \forall \; \theta \geq 1$$

`half_life` Specify decay in terms of half-life, λ, with

$$\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \; \forall \; \lambda > 0$$

`alpha` Specify smoothing factor alpha directly, $0 < \alpha \leq 1$.

`adjust` Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings:

- 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_samples` 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, null, x_2) are $(1-\alpha)^2$ and 1 if `adjust = TRUE`, and $(1-\alpha)^2$ and α 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, null, x_2) are $1-\alpha$ and 1 if `adjust = TRUE`, and $1-\alpha$ and α 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))
```

`expr__ewm_var` *Compute exponentially-weighted moving variance*

Description

Compute exponentially-weighted moving variance

Usage

```
expr__ewm_var(
  ...,
  com = NULL,
  span = NULL,
  half_life = NULL,
  alpha = NULL,
  adjust = TRUE,
  bias = FALSE,
  min_samples = 1,
  ignore_nulls = FALSE
)
```

Arguments

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

`com` Specify decay in terms of center of mass, γ, with

$$\alpha = \frac{1}{1+\gamma} \; \forall \, \gamma \geq 0$$

.

`span` Specify decay in terms of span, θ, with

$$\alpha = \frac{2}{\theta+1} \; \forall \, \theta \geq 1$$

`half_life` Specify decay in terms of half-life, λ, with

$$\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \; \forall \, \lambda > 0$$

`alpha` Specify smoothing factor alpha directly, $0 < \alpha \leq 1$.

`adjust` Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings:

- 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_samples` 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, null, x_2) are $(1-\alpha)^2$ and 1 if `adjust = TRUE`, and $(1-\alpha)^2$ and α 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, null, x_2) are $1-\alpha$ and 1 if `adjust = TRUE`, and $1-\alpha$ and α if `adjust = FALSE`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a")$ewm_var(com = 1, ignore_nulls = FALSE))
```

`expr__exclude` *Exclude columns from a multi-column expression.*

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 `^` 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 *Explode a list expression*

Description

This means that every item is expanded to a new row.

Usage

```
expr__explode(..., empty_as_null = FALSE, keep_nulls = TRUE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`empty_as_null`	Indicates to explode an empty list/array into a `null`. Defaults to `FALSE`.
`keep_nulls`	Indicates to explode a `null` list/array into a `null`.

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

Extend the Series with `n` *copies of a value*

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 = NULL, strategy = NULL, limit = NULL)
```

Arguments

value	Value used to fill null values. Can be NULL if strategy is specified. Accepts expression input, strings are parsed as column names.
strategy	Strategy used to fill null values. If value is NULL, 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 — *Get the first value*

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__floor — *Rounds down to the nearest integer value*

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 *Floor divide using two expressions*

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

[**Superseded**] This is an alias of `$fill_null(strategy = "forward")`.

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 *Take values by index*

Description

Take values by index

Usage

```
expr__gather(indices, ..., null_on_oob = FALSE)
```

Arguments

`indices`	An expression that leads to a UInt32 dtyped Series.
`...`	These dots are for future extensions and must be empty.
`null_on_oob`	If `TRUE`, return `null` if an index is out of bounds. Otherwise, raise an error.

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 *Take every* `n`*-th value in the Series and return as a new Series*

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, ..., null_on_oob = FALSE)
```

Arguments

`index`	An expression that leads to a UInt32 dtyped Series.
`...`	These dots are for future extensions and must be empty.
`null_on_oob`	If `TRUE`, return `null` if an index is out of bounds. Otherwise, raise an error.

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__has_nulls — *Check whether the expression contains one or more null values*

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__hash *Hash elements*

Description

Hash elements

Usage

```
expr__hash(seed = 0)
```

Arguments

seed	Integer, random seed parameter. Defaults to 0.

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(seed = 10))
```

expr__head — *Get the first n elements*

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 — *Bin values into buckets and count their occurrences*

Description

[**Experimental**]

Usage

```
expr__hist(
  bins = NULL,
  ...,
  bin_count = NULL,
  include_category = FALSE,
  include_breakpoint = FALSE
)
```

Arguments

- `bins` — Discretizations to make. If `NULL` (default), we determine the boundaries based on the data.
- `...` — These dots are for future extensions and must be empty.
- `bin_count` — If no bins provided, this will be used to determine the distance of the bins.
- `include_category` — Include a column that shows the intervals as categories.
- `include_breakpoint` — 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
  )
)
```

---

expr__implode — *Aggregate values into a list*

Description

Aggregate values into a list

Usage

```
expr__implode(..., maintain_order = TRUE)
```

Arguments

- `...` — These dots are for future extensions and must be empty.
- `maintain_order` — Whether to preserve the order of elements in the list. Setting this to `FALSE` can improve performance, especially within `$group_by()`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3, b = 4:6)
df$with_columns(pl$col("a")$implode())
```

---

expr__index_of — *Get the index of the first occurrence of a value, or* `NA` *if it's not found*

Description

Get the index of the first occurrence of a value, or `NA` if it's not found

Usage

```
expr__index_of(element)
```

Arguments

`element`	Value to find.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, NA, 17))
df$select(
  seventeen = pl$col("a")$index_of(17),
  null = pl$col("a")$index_of(NULL),
  fiftyfive = pl$col("a")$index_of(55),
)
```

---

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")
)
```

`expr__is_between` *Check if an expression is between the given lower and upper bounds*

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

`lower_bound`	Lower bound value. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals.
`upper_bound`	Upper bound value. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals.
`closed`	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_close` *Check if this expression is close, i.e. almost equal, to the other expression*

Description

Two values `a` and `b` are considered close if the following condition holds: $|a-b| \leq \max\{\text{rel_tol} \cdot \max\{|a|, |b|\}, \text{abs_tol}\}$

Usage

```
expr__is_close(other, ..., abs_tol = 0, rel_tol = 1e-09, nans_equal = FALSE)
```

Arguments

`other`	A literal or expression value to compare with.
`...`	These dots are for future extensions and must be empty.
`abs_tol`	Absolute tolerance. This is the maximum allowed absolute difference between two values. Must be non-negative.
`rel_tol`	Relative tolerance. This is the maximum allowed difference between two values, relative to the larger absolute value. Must be non-negative.
`nans_equal`	Whether `NaN` values should be considered equal.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1.5, 2.0, 2.5), b = c(1.55, 2.2, 3.0))
df$with_columns(
  is_close = pl$col("a")$is_close("b", abs_tol = 0.1)
)
```

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, ..., nulls_equal = FALSE)
```

Arguments

`other`	Accepts expression input. Strings are parsed as column names.
`...`	These dots are for future extensions and must be empty.
`nulls_equal`	A bool to indicate treating null as a distinct value. If `TRUE`, null values will not propagate.

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 *Check if elements are NULL*

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 *Return a boolean mask indicating unique values*

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__item *Get the single value*

Description

This raises an error if there is not exactly one value.

Usage

```
expr__item(..., allow_empty = FALSE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`allow_empty`	Allow having no values to return `null`.

Value

A polars expression

See Also

$get() to get a single value by index.

Examples

```
# This function may be useful to force ourselves to handle cases where we
# expect a single value but there are several. For example, we might expect
# `$mode()` to return a single value...
df <- pl$DataFrame(x = c("a", "a", "c"))
df$select(
  mode = pl$col("x")$mode()$item()
)

# ... but this is not always the case:
df <- pl$DataFrame(x = c("a", "b", "c"))
tryCatch(
  {
    df$select(
      mode = pl$col("x")$mode()$item()
    )
  },
  error = function(e) print(e)
)
```

---

expr__kurtosis *Compute the kurtosis (Fisher or Pearson)*

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

`...`	These dots are for future extensions and must be empty.
`fisher`	If `TRUE` (default), Fisher's definition is used (normal ==> 0.0). If `FALSE`, Pearson's definition is used (normal ==> 3.0).
`bias`	If `FALSE`, 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 — *Get the last value*

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 — *Check lower or equal inequality*

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 — *Return the number of elements in the column*

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 — *Get the first n rows*

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 — *Compute the logarithm*

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 *Compute the base-10 logarithm*

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 *Compute the natural logarithm plus one*

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 *Calculate the lower bound*

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 *Check strictly lower inequality*

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` *Apply a custom R function to a whole Series or sequence of Series.*

Description

[Experimental]

The output of this custom function is presumed to be either a Series, or an R vector that will be converted into a Series by `as_polars_series()`. By default, the function is applied to the complete Series, rather than element by element. Set `is_elementwise` to `TRUE` when the function can be safely applied element by element.

Usage

```
expr__map_batches(
  lambda,
  return_dtype = NULL,
  ...,
  is_elementwise = FALSE,
  returns_scalar = FALSE
)
```

Arguments

`lambda`	Function to apply.
`return_dtype`	Dtype of the output Series. Can be a DataType or a DataTypeExpr, such as one returned by `pl$dtype_of()`. It is recommended to set this whenever possible. If this is `NULL`, it tries to infer the datatype by calling the function with dummy data and looking at the output.
`...`	These dots are for future extensions and must be empty.
`is_elementwise`	Whether the function can be applied element by element. If `TRUE`, the function may be optimized by the query engine. Defaults to `FALSE`.
`returns_scalar`	Whether the function returns a single scalar value. Defaults to `FALSE`.

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) {
```

```
  x$to_r_vector() |>
    which.max()
})

# 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__max_by` *Get maximum value, ordered by another expression*

Description

[**Experimental**]

Usage

```
expr__max_by(by)
```

Arguments

`by`	Column used to determine the largest element. Accepts expression input. Strings are parsed as column names.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(-1, NaN, 1), b = c("x", "y", "z"))
df$with_columns(max_b_by_a = pl$col("b")$max_by("a"))
```

`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` *Get median value*

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 — *Get the minimum value*

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__min_by — *Get minimum value, ordered by another expression*

Description

[**Experimental**]

Usage

```
expr__min_by(by)
```

Arguments

by	Column used to determine the smallest element. Accepts expression input. Strings are parsed as column names.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(-1, NaN, 1), b = c("x", "y", "z"))
df$with_columns(min_b_by_a = pl$col("b")$min_by("a"))
```

expr__mod — *Modulo using two expressions*

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 — *Compute the most occurring value(s)*

Description

Compute the most occurring value(s)

Usage

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

Arguments

`...`	These dots are for future extensions and must be empty.
`maintain_order`	If `TRUE`, maintain order of data. This requires more work.

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__n_unique *Count unique values*

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__nan_max *Get the maximum value with NaN*

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 — *Get the minimum value with NaN*

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 — *Check inequality*

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 *Check inequality without* `null` *propagation*

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 — *Count null values*

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__or — *Apply logical OR on two expressions*

Description

Combine two boolean expressions with OR.

Usage

```
expr__or(...)
```

Arguments

...	<dynamic-dots> One or more integer or boolean expressions to evaluate/combine.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  x = c(5, 6, 7, 4, 8),
  y = c(1.5, 2.5, 1.0, 4.0, -5.75),
  z = c(-9, 2, -1, 4, 8),
)
df$with_columns(
  (pl$col("x") == pl$col("y"))$or(
    pl$col("y") == pl$col("z"),
    pl$col("y")$cast(pl$Int32) == pl$col("z"),
  )$alias("any")
)
```

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` *Computes percentage change between values*

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 — *Exponentiation using two expressions*

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 — *Compute the product of an expression.*

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` *Bin continuous values into discrete categories based on their quantiles*

Description

[**Deprecated**]

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.

Details

Deprecated in Polars 2.0.0 in favor of `$bin_quantiles()` or `$bin_ranks()`. Both replacements are experimental and require `labels`; use `labels = NULL` for `UInt32` bin indices. `$bin_quantiles()` keeps equal values together, while `$bin_ranks()` can split equal values across bins.

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 *Get quantile value(s)*

Description

Get quantile value(s)

Usage

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

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"))
df$select(pl$col("a")$quantile(0.3, interpolation = "equiprobable"))
```

---

expr__radians *Convert from degrees to radians*

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 *Assign ranks to data, dealing with ties appropriately*

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:

- `"average"` (default): The average of the ranks that would have been assigned to all the tied values is assigned to each value.
- `"min"`: The minimum of the ranks that would have been assigned to all the tied values is assigned to each value. (This is also referred to as "competition" ranking.)
- `"max"` : The maximum of the ranks that would have been assigned to all the tied values is assigned to each value.
- `"dense"`: Like 'min', but the rank of the next highest element is assigned the rank immediately after those assigned to the tied elements.
- `"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__reinterpret — *Reinterpret the underlying bits of a same-size numeric type.*

Description

Reinterpret the underlying bits of a same-size numeric type.

Usage

```
expr__reinterpret(..., signed = NULL, dtype = NULL)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`signed`	Whether to reinterpret as a signed or unsigned integer of the same width as the input. Must specify exactly one of `signed` and `dtype`.
`dtype`	Target data type for the reinterpretation. Must specify exactly one of `signed` and `dtype`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, 1, 2))$cast(pl$UInt64)

# Reinterpret column a as Int64
df$with_columns(
  reinterpreted = pl$col("a")$reinterpret(dtype = pl$Int64)
)
```

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

`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.

...	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 column or an Array column*

Description

Reshape this Expr to a flat column or an Array column

Usage

```
expr__reshape(dimensions)
```

Arguments

`dimensions`	A integer vector of length of the dimension size. If `-1` is used as the value for the first dimension, that dimension is inferred. Because the size of the Column may not be known in advance, it is only possible to use -1 for the first dimension.

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 Array with shape dimensions.

Value

A polars expression

Examples

```
df <- pl$DataFrame(foo = 1:8)

df$select(pl$col("foo")$reshape(8))
df$select(pl$col("foo")$reshape(c(2, 4)))

# Using `-1` for the first dimension to infer the other dimension
df$select(pl$col("foo")$reshape(c(-1, 4)))
df$select(pl$col("foo")$reshape(c(-1, 2)))

# We can have more than 2 dimensions
df$select(pl$col("foo")$reshape(c(2, 2, 2)))
df$select(pl$col("foo")$reshape(c(-1, 2, 2)))
```

expr__reverse *Reverse an expression*

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 *Compress the column data using run-length encoding*

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 — *Get a distinct integer ID for each run of identical values*

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 — *Create rolling groups based on a temporal or integer column*

Description

If you have a time series `<t_0, t_1, ..., t_n>`, then by default the windows created will be:

- `(t_0 - period, t_0]`
- `(t_1 - period, t_1]`
- …
- `(t_n - 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

`index_column`	Something coercible to an Expr. Strings are not parsed as columns. 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 temporarily cast to Int64, so if performance matters use an Int64 column.
`...`	These dots are for future extensions and must be empty.
`period`	Length of the window - must be non-negative.
`offset`	Offset of the window. Default is `-period`.
`closed`	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_kurtosis

Compute a rolling kurtosis

Description

[**Experimental**]

The window at a given row will include the row itself, and the `window_size - 1` elements before it.

Usage

```
expr__rolling_kurtosis(
  window_size,
  ...,
  fisher = TRUE,
  bias = TRUE,
  min_samples = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`...`	These dots are for future extensions and must be empty.
`fisher`	If `TRUE` (default), Fisher's definition is used (normal ==> 0.0). If `FALSE`, Pearson's definition is used (normal ==> 3.0).
`bias`	If `FALSE`, the calculations are corrected for statistical bias.
`min_samples`	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`.
`center`	If `TRUE`, set the labels at the center of the window.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, 4, 2, 9))

df$with_columns(
  rolling_kurtosis = pl$col("a")$rolling_kurtosis(window_size = 3)
)

# Center the values in the window
df$with_columns(
  rolling_kurtosis = pl$col("a")$rolling_kurtosis(window_size = 3, center = TRUE)
)
```

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_samples = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`...`	These dots are for future extensions and must be empty.
`min_samples`	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`.
`center`	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_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 `<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_max_by(
  by,
  window_size,
  ...,
  min_samples = 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.

`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_samples` The number of values in the window that should be non-null before computing a result. Defaults to 1.

`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 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_samples = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`...`	These dots are for future extensions and must be empty.
`min_samples`	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`.
`center`	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_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 `<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_mean_by(
  by,
  window_size,
  ...,
```

```
  min_samples = 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.

`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_samples` The number of values in the window that should be non-null before computing a result. Defaults to 1.

`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 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_samples = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`...`	These dots are for future extensions and must be empty.
`min_samples`	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`.
`center`	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_1 - window_size, t_1]`
- …
- `(t_n - window_size, t_n]`

Usage

```
expr__rolling_median_by(
  by,
  window_size,
  ...,
  min_samples = 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.
`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_samples`	The number of values in the window that should be non-null before computing a result. Defaults to 1.
`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 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_samples = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`...`	These dots are for future extensions and must be empty.
`min_samples`	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`.
`center`	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_samples = 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.
`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_samples` The number of values in the window that should be non-null before computing a result. Defaults to 1.

`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",
    "equiprobable"
  ),
  window_size,
  weights = NULL,
  ...,
  min_samples = NULL,
  center = FALSE
)
```

Arguments

`quantile`	Quantile between 0.0 and 1.0.
`interpolation`	Interpolation method. Must be one of `"nearest"`, `"higher"`, `"lower"`, `"midpoint"`, `"linear"`.
`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`...`	These dots are for future extensions and must be empty.
`min_samples`	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`.
`center`	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_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 `<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_quantile_by(
  by,
  window_size,
  ...,
  quantile,
  interpolation = c(
    "nearest",
    "higher",
    "lower",
    "midpoint",
    "linear",
    "equiprobable"
  ),
  min_samples = 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.

`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.

`quantile` Quantile between 0.0 and 1.0.

`interpolation` Interpolation method. Must be one of `"nearest"`, `"higher"`, `"lower"`, `"midpoint"`, `"linear"`.

`min_samples` The number of values in the window that should be non-null before computing a result. Defaults to 1.

`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 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_rank *Apply a rolling rank over values*

Description

[Experimental]

A window of length `window_size` will traverse the array. The values that fill this window will be ranked according to the `method` parameter. The resulting values will be the rank of the value that is at the end of the sliding window.

Usage

```
expr__rolling_rank(
  window_size,
  method = c("average", "min", "max", "dense", "random"),
  ...,
  seed = NULL,
  min_samples = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`method`	The method used to assign ranks to tied elements. Must be one of the following: • `"average"` (default): The average of the ranks that would have been assigned to all the tied values is assigned to each value. • `"min"`: The minimum of the ranks that would have been assigned to all the tied values is assigned to each value. (This is also referred to as "competition" ranking.) • `"max"`: The maximum of the ranks that would have been assigned to all the tied values is assigned to each value. • `"dense"`: Like `"min"`, but the rank of the next highest element is assigned the rank immediately after those assigned to the tied elements. • `"random"`: Choose a random rank for each value in a tie.
`...`	These dots are for future extensions and must be empty.
`seed`	Random seed used when `method = "random"`. If `NULL` (default), a random seed is generated for each rolling rank operation.
`min_samples`	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`.
`center`	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 = c(1, 4, 4, 1, 9))

df$select(pl$col("a")$rolling_rank(3, method = "average"))
```

---

expr__rolling_rank_by

Apply a rolling rank 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_rank_by(
  by,
  window_size,
  method = c("average", "min", "max", "dense", "random"),
  ...,
  seed = NULL,
  min_samples = 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.

`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".

`method` The method used to assign ranks to tied elements. Must be one of the following:

- `"average"` (default): The average of the ranks that would have been assigned to all the tied values is assigned to each value.
- `"min"`: The minimum of the ranks that would have been assigned to all the tied values is assigned to each value. (This is also referred to as "competition" ranking.)
- `"max"`: The maximum of the ranks that would have been assigned to all the tied values is assigned to each value.
- `"dense"`: Like `"min"`, but the rank of the next highest element is assigned the rank immediately after those assigned to the tied elements.
- `"random"`: Choose a random rank for each value in a tie.

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

`seed` Random seed used when `method = "random"`. If `NULL` (default), a random seed is generated for each rolling rank operation.

`min_samples` The number of values in the window that should be non-null before computing a result. Defaults to 1.

`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 rank with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_rank = pl$col("index")$rolling_rank_by(
    "date",
    window_size = "2h"
  )
)

# Compute the rolling rank with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_rank = pl$col("index")$rolling_rank_by(
    "date",
    window_size = "2h",
    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,
  min_samples = NULL,
  center = FALSE
)
```

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.
`min_samples`	The number of values in the window that should be non-null before computing a result. If set to `NULL` (default), it will be set equal to `window_size`.
`center`	Set the labels at the center of the window.

Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` - this method can cache the window size computation.

Value

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_samples = NULL,
  center = FALSE,
  ddof = 1
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`...`	These dots are for future extensions and must be empty.
`min_samples`	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`.
`center`	If `TRUE`, set the labels at the center of the window.
`ddof`	"Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default `ddof` is 1.

Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` - this method can cache the window size computation.

Value

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)
)
```

`expr__rolling_std_by` *Apply a rolling standard deviation 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_std_by(
  by,
  window_size,
  ...,
  min_samples = 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_samples`	The number of values in the window that should be non-null before computing a result. Defaults to 1.
`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 - ddof`, where `N` represents the number of elements. By default `ddof` is 1.

Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` - this method can cache the window size computation.

Value

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_samples = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`...`	These dots are for future extensions and must be empty.
`min_samples`	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`.
`center`	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_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 `<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_sum_by(
  by,
  window_size,
  ...,
  min_samples = 0,
  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.

`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_samples` The number of values in the window that should be non-null before computing a result. Defaults to 0.

`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 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_samples = NULL,
  center = FALSE,
  ddof = 1
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`...`	These dots are for future extensions and must be empty.
`min_samples`	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`.
`center`	If `TRUE`, set the labels at the center of the window.
`ddof`	"Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default `ddof` is 1.

Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` - this method can cache the window size computation.

Value

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 `<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_var_by(
  by,
  window_size,
  ...,
  min_samples = 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_samples` The number of values in the window that should be non-null before computing a result. Defaults to 1.

`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 - ddof`, where `N` represents the number of elements. By default `ddof` is 1.

Details

If you want to compute multiple aggregation statistics over the same dynamic window, consider using `$rolling()` - this method can cache the window size computation.

Value

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 = 0L,
  mode = c("half_to_even", "half_away_from_zero", "to_zero")
)
```

Arguments

`decimals`	Number of decimals to round by.
`mode`	The rounding strategy used. A "rounded value" is a value with at most `decimals` decimal places (e.g. integers when `decimals=0`, multiples of 0.1 when `decimals = 1`, 0.01 when `decimals = 2`, and so on).

Strategies that start with `half_` round all values to the *nearest* rounded value, only using the strategy to break ties when a value falls exactly between two rounded values (e.g. 0.5 when `decimals = 0`, 0.05 when `decimals = 1`). Other rounding strategies specify explicitly which rounded value is chosen and always apply (not just for tiebreaks).

- `"half_to_even"` (default): Round to the nearest value; break ties by choosing the nearest **even** value. For example, 0.5 rounds to 0, 1.5 rounds to 2, 2.5 rounds to 2. Also known as "banker's rounding"; this is the default because it tends to minimise cumulative rounding bias.
- `"half_away_from_zero"`: Round to the nearest value; break ties by rounding **away from zero**. For example, 0.5 rounds to 1, -0.5 rounds to -1, 2.5 rounds to 3. Also known as "commercial rounding".
- `"to_zero"`: Always round (truncate) **towards zero**, discarding the fractional part beyond `decimals`. For example, 0.9 rounds to 0, -0.9 rounds to 0, 1.29 rounds to 1.2 (with `decimals = 1`). Equivalent to `$truncate()`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(0.33, 0.52, 1.02, 1.17))
df$select(pl$col("a")$round(1))

df <- pl$DataFrame(
  f64 = c(-3.5, -2.5, -1.5, -0.5, 0.5, 1.5, 2.5, 3.5),
  d = c("-3.5", "-2.5", "-1.5", "-0.5", "0.5", "1.5", "2.5", "3.5")
)$cast(d = pl$Decimal(scale = 1))

df$with_columns(
  pl$all()$round(mode = "half_away_from_zero")$name$suffix("_away"),
  pl$all()$round(mode = "half_to_even")$name$suffix("_to_even"),
)
```

`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 = NULL,
  seed = NULL
)
```

Arguments

`n`	Number of items to return. Can be an Expr. Strings are parsed as column names. 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. Can be an Expr. Strings are parsed as column names. Cannot be used with `n`.
`with_replacement`	Allow values to be sampled more than once.
`shuffle`	If `TRUE`, explicitly shuffle the sampled data points. If `FALSE`, maintain their relative order. If `NULL` (default), use a high-performance algorithm without guaranteeing an order.
`seed`	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$DataFrame(a = 1:3)
df$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"),
  ...,
  descending = FALSE
)
```

Arguments

`element`	Expression or scalar value. Strings are parsed as literals.
`side`	Must be one of the following: • `"any"`: the index of the first suitable location found is given; • `"left"`: the index of the leftmost suitable location found is given; • `"right"`: the index the rightmost suitable location found is given.
`...`	These dots are for future extensions and must be empty.
`descending`	Boolean indicating whether the values are descending or not.

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, nulls_last = FALSE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`descending`	Whether the Series order is descending.
`nulls_last`	Whether the nulls are at the end.

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. Can be an Expr. Strings are parsed as column names. 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. Strings are parsed as literals.

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__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 — *Compute hyperbolic sine*

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 — *Compute the skewness*

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())
```

---

expr__slice — *Get a slice of this expression*

Description

Get a slice of this expression

Usage

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

Arguments

`offset`	Numeric or expression, zero-indexed. Indicates where to start the slice. A negative value is one-indexed and starts from the end.
`length`	Maximum number of elements contained in the slice. If `NULL` (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())
```

`expr__sort_by` *Sort this column by the ordering of another column, or multiple other columns.*

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

`...`	<dynamic-dots> Column(s) to sort by. Accepts expression input. Strings are parsed as column names.
`descending`	Sort in descending order. When sorting by multiple columns, can be specified per column by passing a sequence of booleans.
`nulls_last`	Place null values last; can specify a single boolean applying to all columns or a sequence of booleans for per-column control.
`multithreaded`	Sort using multiple threads.
`maintain_order`	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 — *Compute the standard deviation*

Description

Compute the standard deviation

Usage

```
expr__std(ddof = 1)
```

Arguments

`ddof`	"Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default `ddof` is 1.

Value

A polars expression

Examples

```
pl$DataFrame(a = c(1, 3, 5, 6))$
  select(pl$all()$std())
```

expr__sub — *Subtract two expressions*

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 — *Get sum value*

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 — *Get the last n elements*

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 *Compute tangent*

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 *Compute hyperbolic tangent*

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__to_physical *Cast to physical representation of the logical dtype*

Description

The following data types will be changed:

- Date -> Int32
- Datetime -> Int64
- Time -> Int64
- Duration -> Int64
- Categorical -> UInt32

Other data types will be left unchanged.

Note that the physical representations are an implementation detail and not guaranteed to be stable.

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__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

`by`	Column(s) used to determine the smallest elements. Accepts expression input. Strings are parsed as column names.
`k`	Number of elements to return.
`...`	These dots are for future extensions and must be empty.
`reverse`	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 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"))
```

---

expr__true_div *Divide two expressions*

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.: ±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__truncate` *Truncate numeric data toward zero to* `decimals` *number of decimal places*

Description

Truncate numeric data toward zero to `decimals` number of decimal places

Usage

```
expr__truncate(decimals = 0L)
```

Arguments

`decimals`	Number of decimal places to truncate to.

Details

Truncation discards the fractional part beyond the given number of decimals. For example, when truncating to 0 decimals, 0.25, -0.25, 0.99, and -0.99 will all round to 0. When truncating to 1 decimal 1.9999 rounds to 1.9 and -1.9999 rounds to -1.9. There is no tiebreak behaviour at midpoint values as there is with `$round()` so 0.5 and -0.5 will also round to 0 when `decimals = 0`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(n = c(-9.9999, 0.12345, 1.0251, 8.8765))
df$with_columns(
  t0 = pl$col("n")$truncate(0),
  t1 = pl$col("n")$truncate(1),
  t2 = pl$col("n")$truncate(2),
  t3 = pl$col("n")$truncate(3),
  t4 = pl$col("n")$truncate(4),
)
```

---

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

`...`	These dots are for future extensions and must be empty.
`sort`	Sort the output by count in descending order. If `FALSE` (default), the order of the output is random.
`parallel`	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.
`name`	Give the resulting count field a specific name. If `normalize` is `TRUE` it defaults to `"proportion"`, otherwise it defaults to `"count"`.
`normalize`	If `TRUE`, 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 - 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))
```

`expr_arr_agg` *Run any polars aggregation expression against the array's elements*

Description

This looks similar to `$arr$eval()`, but the key difference is that `$arr$agg()` automatically explodes the array if the expression inside returns a scalar (while `$arr$eval()` always returns an array).

Usage

```
expr_arr_agg(expr)
```

Arguments

`expr`	Expression to run. Note that you can select an element with `pl$element()`, `pl$first()`, and more. See Examples.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = list(c(1, NA), c(42, 13), c(NA, NA)))$
  cast(pl$Array(pl$Int64, 2))

# The column "null_count" has dtype u32 because `$null_count()` returns a
# scalar for each sub-array. Using `$arr$eval()` instead would return a
# column with dtype arr(u32).
df$with_columns(
  null_count = pl$col("a")$arr$agg(pl$element()$null_count())
)

# The column "no_nulls" has dtype arr(u32) because the expression doesn't
# guarantee to return a scalar.
df$with_columns(
  no_nulls = pl$col("a")$arr$agg(pl$element()$drop_nulls())
)
```

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

Description

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

Usage

```
expr_arr_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 `FALSE` values, the output is null.

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

Description

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

Usage

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

---

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

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

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, ..., nulls_equal = TRUE)
```

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.
`...`	These dots are for future extensions and must be empty.
`nulls_equal`	If `TRUE`, treat null as a distinct value. Null values will not propagate.

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$col("item")$cast(pl$Float64)),
  with_lit = pl$col("values")$arr$contains(pl$lit(1)$cast(pl$Float64))
)
```

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_dot

Compute the row-wise dot product with another Array expression

Description

Both inputs must contain arrays of the same width. Their inner data types are cast to a common supertype, which must be `Float32` or `Float64`. An input with one row is broadcast against the other input. If either input array is `null` for a row, the result for that row is `null`.

Usage

```
expr_arr_dot(other)
```

Arguments

`other`	Array expression or query vector to compute the dot product with. A one-row input is broadcast against the other input.

Details

Elements are paired by position. Pairs where either element is `null` do not contribute to the sum. If a non-null row has no pairs where both elements are valid, the result is `0`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = list(c(1, 2), c(3, 4)),
  b = list(c(5, 6), c(7, 8))
)$cast(pl$Array(pl$Float64, 2))

df$with_columns(dot = pl$col("a")$arr$dot("b"))
```

---

expr_arr_eval *Run any polars expression on the sub-array's values*

Description

Run any polars expression on the sub-array's values

Usage

```
expr_arr_eval(expr, ..., as_list = FALSE)
```

Arguments

`expr`	Expression to run. Note that you can select an element with `pl$element()`, `pl$first()`, and more. See Examples.
`...`	These dots are for future extensions and must be empty.
`as_list`	Collect the resulting data into a list datatype (instead of array datatype). This allows for expressions which output a variable amount of data.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = list(c(1, 1), c(8, 5), c(3, 2))
)$cast(pl$Array(pl$Float64, 2))

df$with_columns(
  cum_sum = pl$col("a")$arr$eval(pl$element()$cum_sum())
```

```
)

# This would error without `as_list = TRUE` since `$unique()` doesn't return
# the same number of values in each row.
df$with_columns(
  cum_sum = pl$col("a")$arr$eval(pl$element()$unique(), as_list = TRUE)
)
```

---

expr_arr_explode *Explode array in separate rows*

Description

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

Usage

```
expr_arr_explode(..., empty_as_null = FALSE, keep_nulls = TRUE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`empty_as_null`	Indicates to explode an empty list/array into a `null`. Defaults to `FALSE`.
`keep_nulls`	Indicates to explode a `null` list/array into a `null`.

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). In R, out-of-bounds indices return `null` by default; pass `null_on_oob = FALSE` to raise an error.

Usage

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

Arguments

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

Value

A polars expression

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 *Join elements in every sub-array*

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`. The default is to propagate null values.

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 *Get the last value of the sub-arrays*

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_len *Return the number of elements in each sub-array*

Description

Return the number of elements in each sub-array

Usage

```
expr_arr_len()
```

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$with_columns(len = pl$col("a")$arr$len())
```

---

expr_arr_max *Compute the max value of the sub-arrays*

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 *Compute the median value of the sub-arrays*

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 *Compute the min value of the sub-arrays*

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 *Count the number of unique values in every sub-array*

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. Can be an Expr. Strings are parsed as column names. 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 — *Sort values in every sub-array*

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 *Compute the standard deviation of the sub-arrays*

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

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_to_struct — *Convert the Series of type Array to a Series of type Struct*

Description

Convert the Series of type Array to a Series of type Struct

Usage

```
expr_arr_to_struct(fields = NULL)
```

Arguments

`fields`	**[Experimental]** `NULL` (default) or a character vector of field names. If `NULL` is used, names are generated from the fixed array width.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  n = list(c(0, 1, 2), c(3, 4, 5)),
  .schema_overrides = list(n = pl$Array(pl$Int8, 3))
)

df$with_columns(struct = pl$col("n")$arr$to_struct())

# Convert array to struct with field name assignment by index from character:
df$select(pl$col("n")$arr$to_struct(c("a", "b", "c")))$unnest("n")
```

---

expr_arr_unique *Get the unique values in every sub-array*

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 - 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"))
)
```

---

expr_bin_decode *Decode values using the provided encoding*

Description

Decode values using the provided encoding

Usage

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

Arguments

`encoding`	A character, `"hex"` or `"base64"`. The encoding to use.
`...`	These dots are for future extensions and must be empty.
`strict`	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 *Encode a value using the provided encoding*

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_get` *Get the byte value at the given index*

Description

For example, index `0` would return the first byte of every binary value and index `-1` would return the last byte of every binary value.

Usage

```
expr_bin_get(index, ..., null_on_oob = FALSE)
```

Arguments

`index`	Index to return per binary value.
`...`	These dots are for future extensions and must be empty.
`null_on_oob`	If `TRUE`, return `null` if an index is out of bounds. Otherwise, raise an error.

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = c("\x01\x02\x03", "", "\x04\x05"))$cast(pl$Binary)

df$with_columns(
  get = pl$col("x")$bin$get(0, null_on_oob = TRUE)
)
```

---

expr_bin_reinterpret *Interpret bytes as another type*

Description

Supported types are numerical or temporal dtypes, or an `Array` of these dtypes.

Usage

```
expr_bin_reinterpret(..., dtype, endianness = c("little", "big"))
```

Arguments

`...`	These dots are for future extensions and must be empty.
`dtype`	A Polars DataType or DataTypeExpr indicating which type to interpret binary column into.
`endianness`	Which endianness to use when interpreting bytes. Must be one of `"little"` (default) or `"big"`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = blob::as_blob(c(5L, 35L)))

df$with_columns(
  bin2int = pl$col("x")$bin$reinterpret(
    dtype = pl$UInt8,
    endianness = "little"
  )
)
```

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_physical *Get the physical values of a Categorical or Enum data type*

Description

[**Experimental**] Get the physical representation of a Categorical or an Enum column, i.e. the integer codes used to store the categories.

Usage

```
expr_cat_physical()
```

Value

A polars expression

See Also

- `<expr>$cat$to()`

Examples

```
df <- pl$DataFrame(x = c("foo", "bar", "foo", "x", NA))$cast(
  pl$Enum(c("bar", "foo", "x"))
)

df$with_columns(physical = pl$col("x")$cat$physical())
```

`expr_cat_to` *Convert to a Categorical or Enum data type*

Description

[**Experimental**] Convert integer values to a Categorical or Enum data type. The input is converted to the physical type of the target data type, i.e. `UInt32` for Categorical and `UInt8`, `UInt16` or `UInt32` for Enum (depending on the number of categories).

Usage

```
expr_cat_to(dtype, ..., strict = TRUE)
```

Arguments

`dtype`	A Polars DataType or DataTypeExpr. Must be a Categorical or an Enum.
`...`	These dots are for future extensions and must be empty.
`strict`	If `TRUE` (default), raise an error if a value is not a valid category. If `FALSE`, such values become `null`.

Value

A polars expression

See Also

- `<expr>$cat$physical()`

Examples

```
dtype <- pl$Enum(c("bar", "foo", "x"))

df <- pl$DataFrame(x = c(1, 0, 1, 2, NA))$cast(pl$UInt8)
df$with_columns(cat = pl$col("x")$cat$to(dtype))

# Values that are not valid categories are converted to `null` when
# `strict = FALSE`
df2 <- pl$DataFrame(x = c(1, 0, 4))$cast(pl$UInt8)
df2$with_columns(cat = pl$col("x")$cat$to(dtype, strict = FALSE))
```

expr_dt_add_business_days

Offset by `n` *business days.*

Description

[**Experimental**] 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

`n`	An integer value or a polars expression representing the number of business days to offset by.
`...`	These dots are for future extensions and must be empty.
`week_mask`	Non-NA logical vector of length 7, representing the days of the week to count. The default is Monday to Friday (`c(TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE)`). If you wanted to count only Monday to Thursday, you would pass `c(TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE)`.
`holidays`	A Date class vector or a polars expression, representing the holidays to exclude from the count.
`roll`	What to do when the start date lands on a non-business day. Options are: • `"raise"`: raise an error; • `"forward"`: move to the next business day; • `"backward"`: move to the previous business day.

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 dtdst_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

Extract the century from underlying representation

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 *Combine Date and Time*

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

`time`	The number of epoch since or before (if negative) the Date. Can be an Expr or a PTime.
`time_unit`	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 — *Extract date from date(time)*

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 — *Extract day from underlying Date representation*

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_days_in_month

Extract the number of days in the month from the underlying Date representation.

Description

Extract the number of days in the month from the underlying Date representation.

Usage

```
expr_dt_days_in_month()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(date = as.Date(c("2020-01-01", "2020-02-01", "2020-03-01")))

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

---

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 dtbase_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")
)
```

expr_dt_hour *Extract hour from underlying Datetime representation*

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_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_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_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_millennium *Extract the millennium from underlying representation*

Description

Returns the millennium number in the calendar date.

Usage

```
expr_dt_millennium()
```

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(
  millennium = pl$col("date")$dt$millennium()
)
```

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 — *Extract month from underlying Date representation*

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 *Roll forward to the last day of the month*

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 *Replace time unit*

Description

Replace time unit

Usage

```
expr_dt_replace(
  ...,
  year = NULL,
  month = NULL,
  day = NULL,
  hour = NULL,
  minute = NULL,
  second = NULL,
  microsecond = NULL,
  ambiguous = c("raise", "earliest", "latest", "null")
)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`year`	Column or literal.
`month`	Column or literal, ranging from 1-12.
`day`	Column or literal, ranging from 1-31
`hour`	Column or literal, ranging from 0-23.
`minute`	Column or literal, ranging from 0-59.
`second`	Column or literal, ranging from 0-59.
`microsecond`	Column or literal, ranging from 0-999999.
`ambiguous`	Determine how to deal with ambiguous datetimes. 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 null value

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  date = as.Date(c("2024-04-01", "2025-03-16")),
  new_day = c(10, 15)
)
df$with_columns(replaced = pl$col("date")$dt$replace(day = "new_day"))
```

---

expr_dt_replace_time_zone

Replace time zone for an expression of type Datetime

Description

Different from dtconvert_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

time_zone	NULL or a character time zone from base::OlsonNames(). Pass NULL to unset time zone.
...	These dots are for future extensions and must be empty.
ambiguous	Determine how to deal with ambiguous datetimes. 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 null value

`non_existent` Determine how to deal with non-existent datetimes. One of the followings:

- `"raise"` (default): Throw an error
- `"null"`: Return a null value

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

`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$with_columns(round = pl$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` *Extract seconds from underlying Datetime representation*

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

`...`	These dots are for future extensions and must be empty.
`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 — *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. 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 = "1d1s"
  )
)
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_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$col("dtm")$dt$to_string("%A"),
  month_name = pl$col("dtm")$dt$to_string("%B"),
)
```

`expr_dt_total_days` *Extract the days from a Duration type*

Description

Extract the days from a Duration type

Usage

```
expr_dt_total_days(..., fractional = FALSE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`fractional`	A bool to indicate whether to include the fractional component of the day.

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(..., fractional = FALSE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`fractional`	A bool to indicate whether to include the fractional component of the day.

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(..., fractional = FALSE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`fractional`	A bool to indicate whether to include the fractional component of the day.

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(..., fractional = FALSE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`fractional`	A bool to indicate whether to include the fractional component of the day.

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$with_columns(
  diff_millisec = pl$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(..., fractional = FALSE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`fractional`	A bool to indicate whether to include the fractional component of the day.

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(..., fractional = FALSE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`fractional`	A bool to indicate whether to include the fractional component of the day.

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(..., fractional = FALSE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`fractional`	A bool to indicate whether to include the fractional component of the day.

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_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` *Extract weekday from underlying Date representation*

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_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_agg — *Run any polars aggregation expression against the lists' elements*

Description

This looks similar to `$list$eval()`, but the key difference is that `$list$agg()` automatically explodes the list if the expression inside returns a scalar (while `$list$eval()` always returns a list).

Usage

```
expr_list_agg(expr)
```

Arguments

`expr`	Expression to run. Note that you can select an element with `pl$element()`, `pl$first()`, and more. See Examples.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = list(c(1, NA), c(42, 13), c(NA, NA)))

# The column "null_count" has dtype u32 because `$null_count()` returns a
# scalar for each sub-list. Using `$list$eval()` instead would return a
# column with dtype list(u32).
df$with_columns(
  null_count = pl$col("a")$list$agg(pl$element()$null_count())
)

# The column "no_nulls" has dtype list(u32) because the expression doesn't
# guarantee to return a scalar.
df$with_columns(
  no_nulls = pl$col("a")$list$agg(pl$element()$drop_nulls())
)
```

---

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(..., 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 `FALSE` values, the output is null.

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 — *Evaluate whether any boolean value in a sub-list is true*

Description

Evaluate whether any boolean value in a sub-list is true

Usage

```
expr_list_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$DataFrame(
  a = list(c(TRUE, TRUE), c(FALSE, TRUE), c(FALSE, FALSE), NA, c())
)
df$with_columns(any = pl$col("a")$list$any())
```

expr_list_arg_max — *Retrieve the index of the maximum value in every sub-list*

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, ..., nulls_equal = TRUE)
```

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.
`...`	These dots are for future extensions and must be empty.
`nulls_equal`	If `TRUE`, treat null as a distinct value. Null values will not propagate.

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(1L)
)
```

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)
```

Arguments

`expr`	Expression to run. Note that you can select an element with `pl$element()`, `pl$first()`, and more. See Examples.

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(..., empty_as_null = FALSE, keep_nulls = TRUE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`empty_as_null`	Indicates to explode an empty list/array into a `null`. Defaults to `FALSE`.
`keep_nulls`	Indicates to explode a `null` list/array into a `null`.

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 *Get the first value of the sub-lists*

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 *Get several values by index in every sub-list*

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(indices, ..., null_on_oob = FALSE)
```

Arguments

`indices`	An Expr or something coercible to an Expr of datatype List, (see examples). 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.
`...`	These dots are for future extensions and must be empty.
`null_on_oob`	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(list(2L), null_on_oob = TRUE)
)

# To select different indices per row:
df$with_columns(
  gathered = pl$col("a")$list$gather(list(2L, c(0L, 3L), 1L), null_on_oob = TRUE)
)

# Indices must be an List(Int/Uint) type to work.
# So we may need to cast the column to List(UInt) first.
```

```
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

`n`	Gather every n-th row.
`offset`	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"))
)
```

---

expr_list_get

Get the value by index in every sub-list

Description

This allows to extract one value per list only. To extract several values by index, use `$list$gather()`. In R, out-of-bounds indices return `null` by default; pass `null_on_oob = FALSE` to raise an error.

Usage

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

Arguments

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

Value

A polars expression

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`. The default is to propagate null values.

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 — *Return the number of elements in each sub-list*

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 — *Compute the maximum value in every sub-list*

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 *Compute the minimum value in every sub-list*

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 *Count the number of unique values in every sub-lists*

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 = NULL,
  seed = NULL
)
```

Arguments

`n`	Number of items to return. Can be an Expr. Strings are parsed as column names. 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. Can be an Expr. Strings are parsed as column names. Cannot be used with `n`.
`with_replacement`	Allow values to be sampled more than once.
`shuffle`	If `TRUE`, explicitly shuffle the sampled data points. If `FALSE`, maintain their relative order. If `NULL` (default), use a high-performance algorithm without guaranteeing an order.
`seed`	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 *Slice every sub-list*

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 *Sort values in every sub-list*

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 *Compute the standard deviation in every sub-list*

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 - 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 *Sum all elements in every sub-list*

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())
```

expr_list_tail *Slice the last* `n` *values of every sub-list*

Description

Slice the last `n` values of every sub-list

Usage

```
expr_list_tail(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(
  tail_by_expr = pl$col("s")$list$tail("n"),
  tail_by_lit = pl$col("s")$list$tail(2)
)
```

---

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

Description

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

Usage

```
expr_list_to_array(width)
```

Arguments

`width`	Width of the resulting Array column.

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_to_struct *Convert the Series of type List to a Series of type Struct*

Description

Convert the Series of type List to a Series of type Struct

Usage

```
expr_list_to_struct(fields)
```

Arguments

`fields`	A character vector of field names.

Details

The field names must be supplied explicitly.

Value

A polars expression

See Also

- `<expr>$arr$to_struct()`
- `<series>$list$to_struct()`

Examples

```
df <- pl$DataFrame(n = list(c(0, 1), c(0, 1, 2)))

# Convert list to struct with explicit field names:

df$select(pl$col("n")$list$to_struct(c("one", "two", "three")))$unnest("n")

# Convert list to struct with field name assignment by
# index from a list of names:
df$select(pl$col("n")$list$to_struct(
  fields = c("one", "two", "three"))
)$unnest("n")
```

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 - 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$col(c("a", "b"))$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_literal` *Indicate if this expression is a literal value (optionally aliased)*

Description

Indicate if this expression is a literal value (optionally aliased)

Usage

```
expr_meta_is_literal(..., allow_aliasing = FALSE)
```

Arguments

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

`allow_aliasing` If `FALSE` (default), only a bare literal will match. Set to `TRUE` to also allow for aliased literals.

Value

A polars expression

Examples

```
e <- pl$lit(123)
e$meta$is_literal()

e <- pl$lit(123)$alias("foo")
e$meta$is_literal()
e$meta$is_literal(allow_aliasing = TRUE)
```

---

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 polars 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 — *Pop the latest expression and return the input(s) of the popped expression*

Description

Pop the latest expression and return the input(s) of the popped expression

Usage

```
expr_meta_pop(..., schema = NULL)
```

Arguments

...	These dots are for future extensions and must be empty.
schema	An optional schema. Must be NULL or a named list of DataType.

Value

A polars expression

Examples

```
e <- pl$col("foo") + pl$col("bar")
first <- e$meta$pop()[[1]]

first$meta$eq(pl$col("bar"))
first$meta$eq(pl$col("foo"))
```

expr_meta_root_names — *Get a list with the root column name*

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(..., as_dot = FALSE, schema = NULL)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`as_dot`	[**Experimental**] If `TRUE`, show the dot syntax that can be used in other packages, such as `DiagrammeR`.
`schema`	An optional schema. Must be `NULL` or a named list of DataType.

Value

A string, either with the tree itself (if `as_dot = FALSE`) or with the corresponding GraphViz code (if `as_dot = TRUE`).

Examples

```
my_expr <- (pl$col("foo") * pl$col("bar"))$sum()$over(pl$col("ham")) / 2
cat(my_expr$meta$tree_format())

## Not run:
# This output can be displayed with DiagrammeR for instance
graph <- my_expr$meta$tree_format(as_dot = TRUE)
DiagrammeR::grViz(graph)

## End(Not run)
```

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_name_keep

Keep the original root name of the expression.

Description

Keep the original root name of the expression.

Usage

```
expr_name_keep()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(alice = 1:3)
df$select(pl$col("alice")$alias("bob")$name$keep())
```

expr_name_prefix *Add a prefix to a column name*

Description

Add a prefix to a column name

Usage

```
expr_name_prefix(prefix)
```

Arguments

prefix	Prefix to be added to column name(s)

Value

A polars expression

See Also

$suffix() to add a suffix

Examples

```
dat <- as_polars_df(mtcars)

dat$select(
  pl$col("mpg"),
  pl$col("mpg")$name$prefix("name_"),
  pl$col(c("cyl", "drat"))$name$prefix("bar_")
)
```

expr_name_prefix_fields *Add a prefix to all fields name of a struct*

Description

Add a prefix to all fields name of a struct

Usage

```
expr_name_prefix_fields(prefix)
```

Arguments

prefix	Prefix to add to the field name.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1, b = 2)$select(
  pl$struct(pl$all())$alias("my_struct")
)

df$with_columns(
  pl$col("my_struct")$name$prefix_fields("col_")
)$unnest("my_struct")
```

---

expr_name_replace — *Replace matching regex/literal substring in the name with a new value*

Description

This will undo any previous renaming operations on the expression.

Due to implementation constraints, this method can only be called as the last expression in a chain. Only one name operation per expression will work.

Usage

```
expr_name_replace(pattern, value, ..., literal = FALSE)
```

Arguments

`pattern`	A valid regular expression pattern, compatible with the `regex crate <https://docs.rs/regex/`
`value`	String that will replace the matched substring.
`...`	These dots are for future extensions and must be empty.
`literal`	Treat `pattern` as a literal string, not a regex.

Value

A polars expression

Examples

```
df <- pl$DataFrame(n_foo = 1, n_bar = 2)
df$select(pl$all()$name$replace("^n_", "col_"))

df$select(pl$all()$name$replace("(a|e|i|o|u)", "@"))$schema
```

expr_name_suffix *Add a suffix to a column name*

Description

Add a suffix to a column name

Usage

```
expr_name_suffix(suffix)
```

Arguments

suffix	Suffix to be added to column name(s)

Value

A polars expression

See Also

$prefix() to add a prefix

Examples

```
dat <- as_polars_df(mtcars)

dat$select(
  pl$col("mpg"),
  pl$col("mpg")$name$suffix("_foo"),
  pl$col(c("cyl", "drat"))$name$suffix("_bar")
)
```

expr_name_suffix_fields *Add a suffix to all fields name of a struct*

Description

Add a suffix to all fields name of a struct

Usage

```
expr_name_suffix_fields(suffix)
```

Arguments

suffix	Suffix to add to the field name.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1, b = 2)$select(
  pl$struct(pl$all())$alias("my_struct")
)

df$with_columns(
  pl$col("my_struct")$name$suffix_fields("_post")
)$unnest("my_struct")
```

---

expr_name_to_lowercase

Make the root column name lowercase

Description

Due to implementation constraints, this method can only be called as the last expression in a chain.

Usage

```
expr_name_to_lowercase()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(Foo = 1:3, BAR = 4:6)
df$select(pl$all()$name$to_lowercase())
```

---

expr_name_to_uppercase

Make the root column name uppercase

Description

Due to implementation constraints, this method can only be called as the last expression in a chain.

Usage

```
expr_name_to_uppercase()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(Foo = 1:3, bar = 4:6)
df$select(pl$all()$name$to_uppercase())
```

---

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

`pattern`	A character or something can be coerced to a string Expr of a valid regex pattern, compatible with the regex crate.
`...`	These dots are for future extensions and must be empty.
`literal`	Logical. If `TRUE`, treat `pattern` as a literal string, not as a regular expression.
`strict`	Logical. If `TRUE` (default), raise an error if the underlying pattern is not a valid regex, otherwise mask out with a null value.

Details

To modify regular expression behaviour (such as case-sensitivity) with flags, use the inline `(?iLmsuxU)` syntax. See the regex crate's section on grouping and flags for additional information about the use of inline expression modifiers.

Value

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 `(?i)` syntax example
pl$DataFrame(s = c("AAA", "aAa", "aaa"))$with_columns(
  default_match = pl$col("s")$str$contains("AA"),
  insensitive_match = pl$col("s")$str$contains("(?i)AA")
)

df <- pl$DataFrame(txt = c("Crab", "cat and dog", "rab$bit", NA))
df$with_columns(
  regex = pl$col("txt")$str$contains("cat|bit"),
  literal = pl$col("txt")$str$contains("rab$", literal = TRUE)
)
```

---

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` String patterns to search. Accepts expression input. Bare character vectors are interpreted as column names. For literal patterns, use a list or an expression such as `pl$lit(...)$implode()`.

`...` 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(list(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

`pattern`	A character or something can be coerced to a string Expr of a valid regex pattern, compatible with the regex crate.
`...`	These dots are for future extensions and must be empty.
`literal`	Logical. If `TRUE`, treat `pattern` 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 — *Encode a value using the provided encoding*

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 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_escape_regex

Returns string values with all regular expression meta characters escaped

Description

Returns string values with all regular expression meta characters escaped

Usage

```
expr_str_escape_regex()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(text = c("abc", "def", NA, r"(abc(\w+))"))
df$with_columns(escaped = pl$col("text")$str$escape_regex())
```

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 = 1L)
```

Arguments

`pattern`	A valid regular expression pattern containing at least one capture group, compatible with the regex crate.
`group_index`	Index of the targeted capture group. Group 0 means the whole pattern, the first group begins at index 1. Defaults to the first capture group.

Details

To modify regular expression behaviour (such as multi-line matching) with flags, use the inline `(?iLmsuxU)` syntax. See the example.

See the regex crate's section on grouping and flags for additional information about the use of inline expression modifiers.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  url = c(
    "http://vote.com/ballon_dor?error=404&ref=unknown",
    "http://vote.com/ballon_dor?ref=polars&candidate=messi",
    "http://vote.com/ballon_dor?candidate=ronaldo&ref=polars"
  )
)
df$select(
  extracted = pl$col("url")$str$extract(r"(candidate=(\w+))", 1),
  referer = pl$col("url")$str$extract(r"(ref=(\w+))", 1),
  error = pl$col("url")$str$extract(r"(error=(\w+))", 1)
)

# Using the multi-line mode flag `(?m)`
df <- pl$DataFrame(
  lines = c("I Like\nThose\nOdds", "This is\nThe Way")
)
df$with_columns(
  with_m_flag = pl$col("lines")$str$extract(r"((?m)^(T\w+))", 1),
  without_flag = pl$col("lines")$str$extract(r"(^(T\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

[**Experimental**] This method supports matching on string literals only, and does not support regular expression matching.

Usage

```
expr_str_extract_many(
  patterns,
  ...,
  ascii_case_insensitive = FALSE,
  overlapping = FALSE,
  leftmost = FALSE
)
```

Arguments

`patterns`	String patterns to search. Accepts expression input. Strings are parsed as column names, and other non-expression inputs are parsed as literals. To use the same character vector for all rows, use `list(c(...))` instead of `c(...)` (see Examples).
`...`	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.
`overlapping`	Whether matches can overlap.
`leftmost`	Whether to guarantee in case there are overlapping matches that the leftmost match is used. In case there are multiple candidates for the leftmost match, the pattern which comes first in `patterns` is used.

Value

A polars expression

Examples

```
df <- pl$DataFrame(values = "discontent")
patterns <- list(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 — *Return the index position of the first substring matching a pattern*

Description

Return the index position of the first substring matching a pattern

Usage

```
expr_str_find(pattern, ..., literal = FALSE, strict = TRUE)
```

Arguments

`pattern`	A character or something can be coerced to a string Expr of a valid regex pattern, compatible with the regex crate.
`...`	These dots are for future extensions and must be empty.
`literal`	Logical. If `TRUE`, treat `pattern` as a literal string, not as a regular expression.
`strict`	Logical. If `TRUE` (default), raise an error if the underlying pattern is not a valid regex, otherwise mask out with a null value.

Details

To modify regular expression behaviour (such as case-sensitivity) with flags, use the inline `(?iLmsuxU)` syntax. See the regex crate's section on grouping and flags for additional information about the use of inline expression modifiers.

Value

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_find_many *Use the Aho-Corasick algorithm to find many matches*

Description

[**Experimental**] The function will return the bytes offset of the start of each match. The return type will be List(UInt32). This method supports matching on string literals only, and does not support regular expression matching.

Usage

```
expr_str_find_many(
  patterns,
  ...,
  ascii_case_insensitive = FALSE,
  overlapping = FALSE,
  leftmost = FALSE
)
```

Arguments

`patterns`	String patterns to search. Accepts expression input. Strings are parsed as column names, and other non-expression inputs are parsed as literals. To use the same character vector for all rows, use `list(c(...))` instead of `c(...)` (see Examples).
`...`	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.
`overlapping`	Whether matches can overlap.
`leftmost`	Whether to guarantee in case there are overlapping matches that the leftmost match is used. In case there are multiple candidates for the leftmost match, the pattern which comes first in `patterns` is used.

Value

A polars expression

Examples

```
df <- pl$DataFrame(values = "discontent")
patterns <- list(c("winter", "disco", "onte", "discontent"))

df$with_columns(
  matches = pl$col("values")$str$find_many(patterns, overlapping = FALSE),
  matches_overlapping = pl$col("values")$str$find_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")$str$find_many("patterns"))
```

---

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")
)
```

---

expr_str_join — *Vertically concatenate the string values in the column to a single string value.*

Description

Vertically concatenate the string values in the column to a single string value.

Usage

```
expr_str_join(delimiter = "", ..., ignore_nulls = TRUE)
```

Arguments

`delimiter`	The delimiter to insert between consecutive string values.
`...`	These dots are for future extensions and must be empty.
`ignore_nulls`	Ignore null values (default). If `FALSE`, null values will be propagated: if the column contains any null values, the output is null.

Value

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. Throw errors if encounter invalid json strings.

Usage

```
expr_str_json_decode(dtype, ...)
```

Arguments

dtype	The dtype to cast the extracted value to.
...	These dots are for future extensions and must be empty.

Value

A polars expression

See Also

- <series>strjson_decode()

Examples

```
df <- pl$DataFrame(
  json_val = c('{"a":1, "b": true}', NA, '{"a":2, "b": false}')
)

dtype <- pl$Struct(a = pl$UInt8, b = pl$Boolean)
df$select(
  pl$col("json_val")$str$json_decode(dtype)
)$unnest("json_val")
```

expr_str_json_path_match *Extract the first match of JSON string with the provided JSONPath expression*

Description

Extract the first match of JSON string with the provided JSONPath expression

Usage

```
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_normalize *Returns the Unicode normal form of the string values*

Description

This uses the forms described in Unicode Standard Annex 15: https://www.unicode.org/reports/tr15/.

Usage

```
expr_str_normalize(form = c("NFC", "NFKC", "NFD", "NFKD"))
```

Arguments

form	Unicode form to use. Must be one of: "NFC", "NFKC", "NFD", "NFKD".

Value

A polars expression

Examples

```
df <- pl$DataFrame(text = c("01²", "    "))

new <- df$with_columns(
  nfc = pl$col("text")$str$normalize("NFC"),
  nfkc = pl$col("text")$str$normalize("NFKC"),
)
new

new$select(pl$all()$str$len_bytes())
```

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	Pad the string until it reaches this length. Strings with length equal to or greater than this value are returned as-is. Can be integer or expression.
fill_char	Fill with this ASCII character.

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`	Pad the string until it reaches this length. Strings with length equal to or greater than this value are returned as-is. Can be integer or expression.
`fill_char`	Fill with this ASCII character.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c("cow", "monkey", NA, "hippopotamus"))
df$select(pl$col("a")$str$pad_start(8, "*"))
```

---

expr_str_replace *Replace first matching regex/literal substring with a new string value*

Description

Replace first matching regex/literal substring with a new string value

Usage

```
expr_str_replace(pattern, value, ..., literal = FALSE, n = 1L)
```

Arguments

`pattern`	A character or something can be coerced to a string Expr of a valid regex pattern, compatible with the regex crate.
`value`	A character or an Expr of string that will replace the matched substring.
`...`	These dots are for future extensions and must be empty.
`literal`	Logical. If `TRUE`, treat `pattern` as a literal string, not as a regular expression.
`n`	A number of matches to replace. Note that regex replacement with `n > 1` not yet supported, so raise an error if `n > 1` and `pattern` includes regex pattern and `literal = FALSE`.

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

`pattern`	A character or something can be coerced to a string Expr of a valid regex pattern, compatible with the regex crate.
`value`	A character or an Expr of string that will replace the matched substring.
`...`	These dots are for future extensions and must be empty.
`literal`	Logical. If `TRUE`, treat `pattern` as a literal string, not as a regular expression.

Details

To modify regular expression behaviour (such as case-sensitivity) with flags, use the inline `(?iLmsuxU)` syntax. See the regex crate's section on grouping and flags for additional information about the use of inline expression modifiers.

Value

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,
  leftmost = FALSE
)
```

Arguments

`patterns`	String patterns to search. Accepts expression input. Bare character vectors are interpreted as column names. For literal patterns, use a list or an expression such as `pl$lit(...)$implode()`.

`replace_with` A list of strings, such as `list(c(...))`, or an Expr of dtype `List(String)` used as replacements. A single replacement is applied to all matches. Otherwise, the number of replacements must match the `patterns` argument.

`...` 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.

`leftmost` Whether to guarantee in case there are overlapping matches that the leftmost match is used. In case there are multiple candidates for the leftmost match, the pattern which comes first in `patterns` is used.

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(list(c("you", "me")), list(c("")))
)

# if there are more than one replacement, the patterns and replacements are
# matched
df$with_columns(
  fake_pronouns = pl$col("lyrics")$str$replace_many(list(c("you", "me")), list(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

`offset`	Start index. Negative indexing is supported.
`length`	Length of the slice. If `NULL` (default), the slice is taken to the end of the string.

Value

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` *Split the string by a substring*

Description

Split the string by a substring

Usage

```
expr_str_split(by, ..., inclusive = FALSE, literal = TRUE, strict = TRUE)
```

Arguments

`by`	Substring (or regex if `literal = FALSE`) 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.
`literal`	If `TRUE` (default), treat `by` as a literal string, not as a regular expression.
`strict`	If `TRUE` (default), raise an error if the underlying pattern is not a valid regex, otherwise mask out with a null value.

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")))

df <- pl$DataFrame(s = c("foo1bar", "foo99bar", "foo1bar2baz"))
df
df$with_columns(
  pl$col("s")$str$split(by = "\\d+", literal = FALSE)$alias("split_regex"),
  pl$col("s")$str$split(by = "\\d+", literal = FALSE, inclusive = TRUE)$alias(
    "split_regex_inclusive"
  ),
)

df <- pl$DataFrame(
  s = c("foo1bar", "foo bar", "foo-bar baz"),
  by = c("\\d", "\\s", "-"),
```

```
)
df$with_columns(
  pl$col("s")$str$split(by = pl$col("by"), literal = FALSE)$alias("split_regex"),
  pl$col("s")$str$split(by = pl$col("by"), literal = FALSE, inclusive = TRUE)$alias(
    "split_regex_inclusive"
  ),
)
```

`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_splitn — *Split the string by a substring, restricted to returning at most* `n` *items*

Description

If the number of possible splits is less than `n-1`, the remaining field elements will be null. If the number of possible splits is `n-1` or greater, the last (nth) substring will contain the remainder of the string.

Usage

```
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_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

`dtype`	The data type to convert into. Can be either `pl$Date`, `pl$Datetime`, or `pl$Time`.
`format`	Format to use for conversion. Refer to the chrono crate documentation 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.
`...`	These dots are for future extensions and must be empty.
`strict`	If `TRUE` (default), raise an error if a single string cannot be parsed. If `FALSE`, produce a polars `null`.
`exact`	If `TRUE` (default), require an exact format match. If `FALSE`, allow the format to match anywhere in the target string. Conversion to the Time type is always exact. Note that using `exact = FALSE` introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.
`cache`	Use a cache of unique, converted dates to apply the datetime conversion.
`ambiguous`	Determine how to deal with ambiguous datetimes. 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 null value

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()`
- `<series>$str$to_datetime()`

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"))

# 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` *Convert a String column into a Date column*

Description

Convert a String column into a Date column

Usage

```
expr_str_to_date(format = NULL, ..., strict = TRUE, exact = TRUE, cache = TRUE)
```

Arguments

`format`	Format to use for conversion. Refer to the chrono crate documentation 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.
`...`	These dots are for future extensions and must be empty.
`strict`	If `TRUE` (default), raise an error if a single string cannot be parsed. If `FALSE`, produce a polars `null`.
`exact`	If `TRUE` (default), require an exact format match. If `FALSE`, allow the format to match anywhere in the target string. Conversion to the Time type is always exact. Note that using `exact = FALSE` introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.
`cache`	Use a cache of unique, converted dates to apply the datetime conversion.

Value

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

`format`	Format to use for conversion. Refer to the chrono crate documentation 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.
`...`	These dots are for future extensions and must be empty.
`time_unit`	Unit of time for the resulting Datetime column. If `NULL` (default), the time unit is inferred from the format string if given, e.g.: `"%F %T%.3f"` => `pl$Datetime("ms")`. If no fractional second component is found, the default is `"us"` (microsecond).
`time_zone`	for the resulting Datetime column.
`strict`	If `TRUE` (default), raise an error if a single string cannot be parsed. If `FALSE`, produce a polars `null`.
`exact`	If `TRUE` (default), require an exact format match. If `FALSE`, allow the format to match anywhere in the target string. Note that using `exact = FALSE` introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.
`cache`	Use a cache of unique, converted dates to apply the datetime conversion.
`ambiguous`	Determine how to deal with ambiguous datetimes. 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 null value

Value

A polars expression

See Also

- `<expr>$str$strptime()`
- `<series>$str$to_datetime()`

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_zone = "UTC"))
```

---

expr_str_to_decimal *Convert a String column into a Decimal column*

Description

[**Experimental**]

Usage

```
expr_str_to_decimal(..., scale)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`scale`	Number of digits after the comma to use for the decimals.

Value

A polars expression

See Also

- `<series>$str$to_decimal()`

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(scale = 2))
```

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, dtype = pl$Int64, strict = TRUE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`base`	A positive integer or expression which is the base of the string we are parsing. Characters are parsed as column names. Default: `10L`.
`dtype`	A polars integer dtype (e.g. `pl$UInt8`, `pl$Int32`, etc.). The default is `pl$Int64`.
`strict`	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$to_integer(
    base = 2,
    dtype = pl$Int32,
    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

`format`	Format to use for conversion. Refer to the chrono crate documentation 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.
`...`	These dots are for future extensions and must be empty.
`strict`	If `TRUE` (default), raise an error if a single string cannot be parsed. If `FALSE`, produce a polars `null`.
`cache`	Use a cache of unique, converted dates to apply the datetime conversion.

Value

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_titlecase

Convert a string to titlecase

Description

Transform to titlecase variant.

Usage

```
expr_str_to_titlecase()
```

Details

This method is only available with the "nightly" feature. See `polars_info()` for more details.

Value

A polars expression

Examples

```
pl$select(
  pl$lit(c("hello there", "HI, THERE", NA))$str$to_titlecase()
)$to_series()
```

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_struct_drop *Drop one or more fields from the struct*

Description

Drop one or more fields from the struct

Usage

```
expr_struct_drop(names, ..., strict = TRUE)
```

Arguments

`names`	Character vector of the names of the fields to drop.
`...`	These dots are for future extensions and must be empty.
`strict`	If `TRUE` (default), raise an error if any of the specified fields doesn't exist in the struct.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  aaa = c(1, 2),
  bbb = c("ab", "cd"),
  ccc = c(TRUE, NA)
)$select(struct_col = pl$struct("aaa", "bbb", "ccc"))
df

df$select(pl$col("struct_col")$struct$drop("aaa"))

# Dropping an unknown field is an error unless `strict = FALSE`
df$select(pl$col("struct_col")$struct$drop("zzz", strict = FALSE))
```

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("*"))
```

groupby__agg *Compute aggregations for each group of a group by operation*

Description

Compute aggregations for each group of a group by operation

Usage

```
groupby__agg(...)
```

Arguments

...	<dynamic-dots> Aggregations to compute for each group of the group by operation. Accepts expression input. Strings are parsed as column names.

Value

A polars DataFrame

Examples

```
# Compute the aggregation of the columns for each group.
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"), pl$col("c"))

# Compute the sum of a column for each group.
df$group_by("a")$agg(pl$col("b")$sum())

# Compute multiple aggregates at once by passing a list of expressions.
df$group_by("a")$agg(pl$sum("b"), pl$col("c")$mean())

# Use keyword arguments to easily name your expression inputs.
df$group_by("a")$agg(
  b_sum = pl$sum("b"),
  c_mean_squared = (pl$col("c") ** 2)$mean()
)
```

groupby__having — *Filter groups with a list of predicates after aggregation*

Description

Using this method is equivalent to adding the predicates to the aggregation and filtering afterwards.

This method can be chained and all conditions will be combined using `&`.

Usage

```
groupby__having(...)
```

Arguments

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

Value

An object of class `polars_group_by`

Examples

```
df <- pl$DataFrame(x = c("a", "b", "a", "b", "c"))

# Only keep groups that contain more than one element:
df$group_by("x")$having(
  pl$len() > 1
)$agg()
```

groupby__head — *Get the first* `n` *rows of each group*

Description

Get the first `n` rows of each group

Usage

```
groupby__head(n = 5)
```

Arguments

`n`	Number of rows to return.

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  letters = c("c", "c", "a", "c", "a", "b"),
  nrs = 1:6
)
df

df$group_by("letters")$head(2)$sort("letters")
```

---

groupby__len *Return the number of rows in each group*

Description

Return the number of rows in each group

Usage

```
groupby__len(name = NULL)
```

Arguments

name	Assign a name to the resulting column. If `NULL`, defaults to `"len"`.

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  a = c("Apple", "Apple", "Orange"),
  b = c(1, NA, 2)
)
df$group_by("a")$len()

df$group_by("a")$len("n")
```

groupby__max — *Reduce the groups to the maximal value*

Description

Reduce the groups to the maximal value

Usage

```
groupby__max()
```

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
df

df$group_by("grp")$max()
```

groupby__mean — *Return the mean per group*

Description

Return the mean per group

Usage

```
groupby__mean()
```

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
df

df$group_by("grp")$mean()
```

---

groupby__median *Return the median per group*

Description

Return the median per group

Usage

```
groupby__median()
```

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
df

df$group_by("grp")$median()
```

---

groupby__min *Reduce the groups to the minimal value*

Description

Reduce the groups to the minimal value

Usage

```
groupby__min()
```

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
df

df$group_by("grp")$min()
```

---

groupby__n_unique *Count the unique values per group*

Description

Count the unique values per group

Usage

```
groupby__n_unique()
```

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
df

df$group_by("grp")$n_unique()
```

groupby__quantile *Compute the quantile per group*

Description

Compute the quantile per group

Usage

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

Arguments

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

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
df

df$group_by("grp")$quantile(0.5)
```

groupby__sum *Return the sum per group*

Description

Return the sum per group

Usage

```
groupby__sum()
```

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
df

df$group_by("grp")$sum()
```

groupby__tail *Get the last* `n` *rows of each group*

Description

Get the last `n` rows of each group

Usage

```
groupby__tail(n = 5)
```

Arguments

`n`	Number of rows to return.

Value

A polars DataFrame

Examples

```
df <- pl$DataFrame(
  letters = c("c", "c", "a", "c", "a", "b"),
  nrs = 1:6
)
df

df$group_by("letters")$tail(2)$sort("letters")
```

---

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 order 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

`x` An R object.

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

`strict` A logical value to indicate whether throwing an error when the input list's elements have different data types. If `FALSE` (default), all elements are automatically cast to the super type, or, casting to the super type is failed, the value will be `null`. If `TRUE`, the first non-`NULL` element's data type is used as the data type of the inner Series.

`infer_dtype_length` 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 `Inf`, 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))
```

---

knit_print.polars_data_frame

knit print polars DataFrame

Description

Mimics Python Polars' NotebookFormatter for HTML outputs.

Usage

```
## S3 method for class 'polars_data_frame'
knit_print(x, ...)
```

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

Arguments

`x`	A polars object
`...`	Additional arguments passed to the S3 method. Currently ignored, except two optional arguments `options` and `inline`; see the references below.

Details

Outputs HTML tables if the output format is HTML and the document's `df_print` option is not `"default"` or `"tibble"`.

Or, the output format can be enforced with R's `options` function as follows:

- `options(polars.df_knitr_print = "default")` for the default print method.
- `options(polars.df_knitr_print = "html")` for the HTML table.

Value

`x` invisibly or knit_asis object.

Examples

```
# Using the default print method
withr::with_options(
  list(polars.df_knitr_print = "default"),
  knitr::knit_print(as_polars_df(mtcars))
)

# Returning HTML table
withr::with_options(
  list(polars.df_knitr_print = "html"),
  knitr::knit_print(as_polars_df(mtcars))
)
```

---

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

`k`	Number of rows to return.
`...`	These dots are for future extensions and must be empty.
`by`	Column(s) used to determine the bottom rows. Accepts expression input. Strings are parsed as column names.
`reverse`	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()
```

`lazyframe__cast` *Cast LazyFrame column(s) to the specified dtype(s)*

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

`...`	<dynamic-dots> 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 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 *Clone a LazyFrame*

Description

This makes a very cheap deep copy/clone of an existing LazyFrame. Rarely useful as `LazyFrame`s 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(
  ...,
  engine = c("auto", "in-memory", "streaming"),
  optimizations = pl$QueryOptFlags()
)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`engine`	The engine name to use for processing the query. One of the followings: • `"auto"` (default): Select the engine automatically. In Polars 1.16, the `"in-memory"` engine is selected for most cases. Starting with Polars 2.0, the streaming engine is selected instead. Use `"in-memory"` to keep the current execution engine explicitly. • `"in-memory"`: Use the in-memory engine. This is an escape hatch for queries whose incidental row order differs under streaming. • `"streaming"`: Use the streaming engine.
`optimizations`	[**Experimental**] A QueryOptFlags object to indicate optimization passes done during query optimization.

Value

A polars DataFrame

See Also

- `$sink_parquet()` streams query to a parquet file.
- `$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(
  engine = "streaming"
)
```

---

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__describe — *Creates a summary of statistics for a LazyFrame, returning a DataFrame.*

Description

This method does not maintain the laziness of the frame, and will collect the final result. This could potentially be an expensive operation.

We do not guarantee the output of `describe()` to be stable. It will show statistics that we deem informative, and may be updated in the future. Using `describe()` programmatically (versus interactive exploration) is not recommended for this reason.

Usage

```
lazyframe__describe(
  percentiles = c(0.25, 0.5, 0.75),
  ...,
  interpolation = c(
    "nearest",
    "higher",
    "lower",
    "midpoint",
    "linear",
    "equiprobable"
  )
)
```

Arguments

`percentiles`	One or more percentiles to include in the summary statistics. All values must be in the range `[0; 1]`.
`...`	These dots are for future extensions and must be empty.
`interpolation`	Interpolation method for computing quantiles. Must be one of `"nearest"`, `"higher"`, `"lower"`, `"midpoint"`, or `"linear"`.

Details

The median is included by default as the 50% percentile.

Value

A polars DataFrame

Examples

```
lf <- pl$LazyFrame(
  int = 1:3,
  float = c(0.5, NA, 2.5),
  string = c(letters[1:2], NA),
  date = c(as.Date("2024-01-20"), as.Date("2024-01-21"), NA),
  cat = factor(c(letters[1:2], NA)),
  bool = c(TRUE, FALSE, NA)
)
lf$collect()

# Show default frame statistics:
lf$describe()

# Customize which percentiles are displayed, applying linear interpolation:
lf$describe(
  percentiles = c(0.1, 0.3, 0.5, 0.7, 0.9),
  interpolation = "linear"
)
```

`lazyframe__drop` *Remove columns*

Description

Remove columns

Usage

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

Arguments

`...`	<`dynamic-dots`> Column names or selectors that should be removed.
`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 names or selectors for which are considered. If empty (default), use all columns (same as specifying with the selector `cs$all()`).

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 NaN.
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 names or selectors for which are considered. If empty (default), use all columns (same as specifying with the selector cs$all()).

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"),
  engine = c("auto", "in-memory", "streaming"),
  optimized = TRUE,
  optimizations = pl$QueryOptFlags()
)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`format`	The format to use for displaying the logical plan. Must be either `"plain"` (default) or `"tree"`.
`engine`	The engine name to use for processing the query. One of the followings: • `"auto"` (default): Select the engine automatically. In Polars 1.16, the `"in-memory"` engine is selected for most cases. Starting with Polars 2.0, the streaming engine is selected instead. Use `"in-memory"` to keep the current execution engine explicitly.

- `"in-memory"`: Use the in-memory engine. This is an escape hatch for queries whose incidental row order differs under streaming.
- `"streaming"`: Use the streaming engine.

`optimized` Return an optimized query plan. If `TRUE` (default), the subsequent optimization flags control which optimizations run.

`optimizations` [**Experimental**] A QueryOptFlags object to indicate optimization passes done during query optimization.

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 writeLines() for better printing)
lazy_query$explain(optimized = FALSE) |> writeLines()

# 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() |> writeLines()

# You can disable specific optimizations.
lazy_query$explain(
  optimizations = pl$QueryOptFlags(predicate_pushdown = FALSE)
) |>
  writeLines()

# Also possible to see this as tree format
lazy_query$explain(format = "tree") |> writeLines()
```

---

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(..., empty_as_null = FALSE, keep_nulls = TRUE)
```

Arguments

`...`	<`dynamic-dots`> Column names or selectors defining them. The underlying columns being exploded must be of the `List` or `Array` data type.
`empty_as_null`	Indicates to explode an empty list/array into a `null`. Defaults to `FALSE`.
`keep_nulls`	Indicates to explode a `null` list/array into a `null`.

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 = NULL,
  strategy = NULL,
  limit = NULL,
  ...,
  matches_supertype = TRUE
)
```

Arguments

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

lazyframe__filter *Filter the rows in the LazyFrame based on a predicate expression*

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 + 2 * every, start + 2 * every + period)
- …

where `start` is determined by `start_by`, `offset`, `every`, and the earliest datapoint. See the `start_by` argument description for details.

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

`index_column`	Column used to group based on the time window. Often of type Date/Datetime. This column must be sorted in ascending order (or, if `group_by` 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 In64. Note that Int32 gets temporarily cast to Int64, so if performance matters, use an Int64 column.
`...`	These dots are for future extensions and must be empty.
`every`	Interval of the window.
`period`	Length of the window. If `NULL` (default), it will equal `every`.
`offset`	Offset of the window, does not take effect if `start_by = "datapoint"`. Defaults to zero.

`include_boundaries`
: Add two columns `"_lower_boundary"` and `"_upper_boundary"` columns that show the boundaries of the window. This will impact performance because it's harder to parallelize.

`closed`
: Define which sides of the interval are closed (inclusive). Default is `"left"`.

`label`
: Define which label to use for the window:
  - `"left"`: lower boundary of the window
  - `"right"`: upper boundary of the window
  - `"datapoint"`: 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.

`group_by`
: 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

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

An object of class `polars_lazy_group_by`

See Also

- <LazyFrame>$rolling()

Examples

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

Join LazyFrames

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"),
  nulls_equal = FALSE,
```

```
  coalesce = NULL,
  maintain_order = c("none", "left", "right", "left_right", "right_left"),
  allow_parallel = TRUE,
  force_parallel = FALSE
)
```

Arguments

`other` LazyFrame to join with.

`on` Either a vector of column names or a list of expressions and/or strings. Use `left_on` and `right_on` if the column names to match on are different between the two LazyFrames.

`how` One of the following methods:

- "inner": returns rows that have matching values in both tables
- "left": returns all rows from the left table, and the matched rows from the right table
- "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.

`nulls_equal` Join on null values. By default null values will never produce matches.

`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.

`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.

- `"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.

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

`other` LazyFrame to join with.

`...` 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.

`on` Either a vector of column names or a list of expressions and/or strings. Use `left_on` and `right_on` if the column names to match on are different between the two LazyFrames.

`by_left`, `by_right` Same as `by` but only for the left or the right table. They must have the same length.

`by` Join on these columns before performing asof join. Either a vector of column names or a list of expressions and/or strings. Use `left_by` and `right_by` if the column names to match on are different between the two tables.

`strategy` Strategy for where to find match:

- `"backward"` (default): search for the last row in the right table whose `on` key is less than or equal to the left key.
- `"forward"`: search for the first row in the right table whose `on` key is greater than or equal to the left key.
- `"nearest"`: search for the last row in the right table whose value is nearest to the left key. String keys are not currently supported for a nearest search.

`suffix` Suffix to add to duplicated column names.

`tolerance` 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).

`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 `on` / `left_on` / `right_on` columns):

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

`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]

By default, 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,
  ...,
  how = c("inner", "left", "right"),
  suffix = "_right"
)
```

Arguments

other	LazyFrame to join with.
...	<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>".
how	Join strategy, one of "inner" (default), "left" or "right".
suffix	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()

# Keep all rows of the left frame, even those without a match
east$join_where(
  west,
  pl$col("dur") < pl$col("time"),
  pl$col("rev") < pl$col("cost"),
```

```
  how = "left"
)$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 — *Aggregate the columns in the LazyFrame to their mean value*

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 — *Aggregate the columns in the LazyFrame to their median value*

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, ..., maintain_order = FALSE)
```

Arguments

`other`	Other LazyFrame that must be merged.
`key`	Key that is sorted.
`...`	These dots are for future extensions and must be empty.
`maintain_order`	If `TRUE`, the output is guaranteed to have left-biased ordering for equal keys: rows from the left frame appear before rows from the right frame when their keys are equal.

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__pivot *Pivot a frame from long to wide format*

Description

Reshape data from long to wide format, known as "pivot wider".

Unlike for DataFrame, the values contained in the columns must be specified beforehand using `on_columns`.

Usage

```
lazyframe__pivot(
  on,
  on_columns,
  ...,
  index = NULL,
  values = NULL,
  aggregate_function = NULL,
  maintain_order = FALSE,
  separator = "_",
  column_naming = c("auto", "combine")
)
```

Arguments

`on` The column(s) whose values will be used as the new columns of the output.

`on_columns` What value combinations will be considered for the output table. Something can be converted to a DataFrame by `as_polars_series(on_columns) |> as_polars_df()`. If `on` contains multiple columns, the DataFrame passed to `on_columns` must have exactly the same columns in the same order as `on`. See examples for details.

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

`index` The column(s) that remain from the input to the output. The output will have one row for each unique combination of the `index`'s values. If `NULL`, all remaining columns not specified in `on` and `values` will be used. At least one of `index` and `values` must be specified.

`values` The existing column(s) of values which will be moved under the new columns from `index`. If an aggregation is specified, these are the values on which the aggregation will be computed. If `NULL`, all remaining columns not specified in `on` and `index` will be used. At least one of `index` and `values` must be specified.

`aggregate_function`
Choose from:

- `NULL` (default): no aggregation takes place, will raise error if multiple values are in group. Same as `pl$element()$item(allow_empty = TRUE)`.

- A predefined aggregate function string, one of `"min"`, `"max"`, `"first"`, `"last"`, `"sum"`, `"mean"`, `"median"`, `"len"`, `"item"`. Same as `pl$element()$<function>()`.
- An expression to do the aggregation.

`maintain_order`
: Ensure the values of `index` are sorted by discovery order.

`separator`
: Used as separator/delimiter in generated column names in case of multiple values columns.

`column_naming`
: How resulting column names will be constructed. `"auto"` (default) combines with separator if there are multiple `values` columns, otherwise just uses the `on_columns` names. `"combine"` always combines the `values` columns' names with the `on_columns` names.

Value

A polars LazyFrame

Examples

```
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$lazy()$pivot(
  "subject",
  on_columns = c("maths", "physics"),
  index = "name",
  values = "test_1",
)$collect()

# You can use selectors too - here we include all test scores in the pivoted table:
df$lazy()$pivot(
  "subject",
  on_columns = c("maths", "physics"),
  values = cs$starts_with("test"),
)$collect()

# If you end up with multiple values per cell, you can specify how to aggregate
# them with `aggregate_function`:
lf <- pl$LazyFrame(
  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),
)
```

```
lf$pivot(
  "col", on_columns = c("a", "b"), index = "ix", aggregate_function = "sum"
)$collect()

# You can also pass a custom aggregation function using `pl$element()` expressions:
lf <- pl$LazyFrame(
  col1 = c("a", "a", "a", "b", "b", "b"),
  col2 = c("x", "x", "x", "x", "y", "y"),
  col3 = c(6, 7, 3, 2, 5, 7),
)
lf$pivot(
  "col2",
  on_columns = c("x", "y"),
  index = "col1",
  values = "col3",
  aggregate_function = pl$element()$tanh()$mean(),
)$collect()

# Note that `on_columns` must contain all combinations of the values in `on`.
# For example, you can use the `expand.grid()` function to create all combinations
# of multiple columns as follows:
as_polars_lf(datasets::penguins)$pivot(
  on = c("species", "sex"),
  on_columns = expand.grid(
    species = c("Adelie", "Gentoo", "Chinstrap"),
    sex = c("male", "female")
  ),
  index = "island",
  values = "body_mass",
  aggregate_function = "mean",
)$collect()
```

`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",
```

```
    "equiprobable"
  )
)
```

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__remove` *Remove rows, dropping those that match the given predicate expression(s)*

Description

The original order of the remaining rows is preserved. Rows where the filter does not evaluate to `TRUE` are retained (this includes rows where the predicate evaluates as `null`).

Usage

```
lazyframe__remove(...)
```

Arguments

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

Value

A polars LazyFrame

Examples

```
lf <- pl$LazyFrame(
  ccy = c("USD", "EUR", "USD", "JPY"),
  year = c(2021, 2022, 2023, 2023),
  total = c(3245, NA, -6680, 25000),
)

# Remove rows matching a condition. Note that the row where `total` is null
# is kept:
```

```
lf$remove(pl$col("total") >= 0)$collect()

# Note that this is *not* the same as simply inverting the condition in
# `$filter()` because `$filter()` doesn't keep predicates that evaluate to
# null:
lf$filter(pl$col("total") < 0)$collect()

# We can use multiple conditions, combined with and/or operators:
lf$remove((pl$col("total") >= 0) & (pl$col("ccy") == "USD"))$collect()

lf$remove((pl$col("total") >= 0) | (pl$col("ccy") == "USD"))$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 `<t_0, t_1, ..., t_n>`, then by default the windows created will be:

- `(t_0 - period, t_0]`
- `(t_1 - period, t_1]`
- …
- `(t_n - 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

```
lazyframe__rolling(
  index_column,
  ...,
  period,
  offset = NULL,
  closed = c("right", "left", "both", "none"),
  group_by = NULL
)
```

Arguments

`index_column`	Column used to group based on the time window. Often of type Date/Datetime. This column must be sorted in ascending order (or, if `group_by` 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 In64. Note that Int32 gets temporarily cast to Int64, so if performance matters, use an Int64 column.
`...`	These dots are for future extensions and must be empty.
`period`	Length of the window - must be non-negative.
`offset`	Offset of the window. Default is `-period`.
`closed`	Define which sides of the interval are closed (inclusive). Default is `"left"`.
`group_by`	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

An object of class `polars_lazy_group_by`

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()
```

`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", bar2 = pl$col("bar") * 2)$collect()
```

lazyframe__serialize *Serialize the logical plan of this LazyFrame*

Description

Serialize the logical plan to a binary representation (raw vector).

Usage

```
lazyframe__serialize()

pl__deserialize_lf(data)
```

Arguments

`data`	A raw vector of serialized LazyFrame.

Value

- `<lazyframe>$serialize()` returns a raw vector.
- `pl$deserialize_lf()` returns a deserialized LazyFrame.

Examples

```
lf <- pl$LazyFrame(a = 1:3)$sum()

# Serialize the logical plan to a binary representation.
serialized <- lf$serialize()
serialized

# The bytes can later be deserialized back into a LazyFrame.
pl$deserialize_lf(serialized)$collect()
```

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_batches

Evaluate the query and call a user-defined function for every ready batch

Description

[**Experimental**] This allows streaming results that are larger than RAM in certain cases.

Note that this method is much slower than native sinks. Only use it if you cannot implement your logic otherwise.

`<lazyframe>$sink_batches()` is a shortcut for `<lazyframe>$lazy_sink_batches()$collect()`.

Usage

```
lazyframe__sink_batches(
  lambda,
  ...,
  chunk_size = NULL,
  maintain_order = TRUE,
  engine = c("auto", "in-memory", "streaming"),
  optimizations = pl$QueryOptFlags()
)

lazyframe__lazy_sink_batches(
  lambda,
  ...,
  chunk_size = NULL,
  maintain_order = TRUE
)
```

Arguments

`lambda` A function that will receive a DataFrame as the first argument and called for side effects (e.g., writing to a file). If the function returns `TRUE` and using the streaming engine, this signals that no more results are needed, allowing for early stopping.

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

`chunk_size` A positive integer or `NULL` (default). The number of rows that are buffered before the callback is called.

`maintain_order` Maintain the order in which data is processed. Setting this to `FALSE` will be slightly faster.

`engine` The engine name to use for processing the query. One of the followings:

- `"auto"` (default): Select the engine automatically. In Polars 1.16, the `"in-memory"` engine is selected for most cases. Starting with Polars 2.0, the streaming engine is selected instead. Use `"in-memory"` to keep the current execution engine explicitly.

- "in-memory": Use the in-memory engine. This is an escape hatch for queries whose incidental row order differs under streaming.
- "streaming": Use the streaming engine.

optimizations [**Experimental**] A QueryOptFlags object to indicate optimization passes done during query optimization.

Value

- <lazyframe>$sink_batches() returns NULL invisibly.
- <lazyframe>$lazy_sink_batches() returns a new LazyFrame.

Examples

```
lf <- as_polars_lf(mtcars)

# Each batch is a Polars DataFrame
lf$sink_batches(\(df) print(df), chunk_size = 10)

# We can stop reading the batches by returning `TRUE`:
lf$sort("cyl")$sink_batches(
  \(df) {
    print(df)

    # We want to stop if this condition is respected:
    max(df[["cyl"]])$to_r_vector() > 4
  },
  chunk_size = 10
)

# One usecase for this function is to export larger-than-RAM data to file
# formats for which polars doesn't provide a writer out of the box.
# The example below writes a LazyFrame by batches to a CSV file for the
# sake of the example, but one could replace `write.csv()` by
# `haven::write_dta()`, `saveRDS()`, or other functions.
#
# Note that if `chunk_size` is missing, then Polars tries to compute it
# automatically. However, depending on the characteristics of the data (for
# instance very long string elements), this can lead to out-of-memory errors.
# It is therefore recommended to set `chunk_size` manually.

withr::with_tempdir({
  file_idx <- 1

  lf$sink_batches(
    \(df) {
      dest <- paste0("file_", file_idx, ".csv")
      cat(sprintf("Writing %s rows to %s\n", nrow(df), dest))
      write.csv(as.data.frame(df), dest)
      file_idx <<- file_idx + 1
    }
  )
```

```
  cat("\nFiles in the directory:\n")
  cat(list.files("."))
  cat("\n\n")

  pl$read_csv(".", infer_schema_files = NULL)
})

# The number of rows in each chunk can be adjusted with `chunk_size`.
withr::with_tempdir({
  file_idx <- 1

  lf$sink_batches(
    \(df) {
      dest <- paste0("file_", file_idx, ".csv")
      cat(sprintf("Writing %s rows to %s\n", nrow(df), dest))
      write.csv(as.data.frame(df), dest)
      file_idx <<- file_idx + 1
    }
  )

  cat("\nFiles in the directory:\n")
  cat(list.files("."))
})

# To avoid manually creating paths and incrementing `file_idx` in the
# anonymous function, we can use function factories:
withr::with_tempdir({
  writer_factory <- function(dir) {
    i <- 0
    function(df) {
      i <<- i + 1
      dest <- file.path(dir, sprintf("%03d.csv", i))
      cat(sprintf("Writing %s rows to %s\n", nrow(df), dest))
      as.data.frame(df) |>
        write.csv(dest, row.names = FALSE)
    }
  }

  writer <- writer_factory(".")

  lf$sink_batches(writer, chunk_size = 10)

  cat("\nFiles in the directory:\n")
  cat(list.files("."))
})
```

`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.

- `<lazyframe>$lazy_sink_*()` don't write directly to the output file(s) until `$collect()` is called. This is useful if you want to save a query to review or run later.
- `<lazyframe>$sink_*()` write directly to the output file(s) (they are shortcuts for `<lazyframe>$lazy_sink_*()$collect()`).

Usage

```
lazyframe__sink_csv(
  path,
  ...,
  include_bom = FALSE,
  compression = c("uncompressed", "gzip", "zstd"),
  compression_level = NULL,
  check_extension = TRUE,
  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,
  decimal_comma = FALSE,
  null_value = "",
  quote_style = c("necessary", "always", "never", "non_numeric"),
  maintain_order = TRUE,
  storage_options = NULL,
  sync_on_close = c("none", "data", "all"),
  mkdir = FALSE,
  engine = c("auto", "in-memory", "streaming"),
  optimizations = pl$QueryOptFlags()
)

lazyframe__lazy_sink_csv(
  path,
  ...,
  include_bom = FALSE,
  compression = c("uncompressed", "gzip", "zstd"),
  compression_level = NULL,
  check_extension = TRUE,
  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,
  decimal_comma = FALSE,
  null_value = "",
  quote_style = c("necessary", "always", "never", "non_numeric"),
  maintain_order = TRUE,
  storage_options = NULL,
  sync_on_close = c("none", "data", "all"),
  mkdir = FALSE
)
```

Arguments

`path` A character. File path to which the file should be written.

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

`include_bom` Logical, whether to include UTF-8 BOM in the CSV output.

`compression` [**Experimental**] What compression format to use. Must be one of `uncompressed` (default), `gzip`, or `zstd`.

`compression_level` [**Experimental**] The compression level to use, typically 0-9 or `NULL` to let the engine choose.

`check_extension` [**Experimental**] Whether to check if the filename matches the compression settings. Will raise an error if compression is set to `"uncompressed"` and the filename ends in one of `".gz"`, `".zst"`, `".zstd"`, or if `compression != "uncompressed"` and the file uses a mismatched extension. Only applies if file is a path.

`include_header` Logical, whether to include header in the CSV output.

`separator` Separate CSV fields with this symbol.

`line_terminator` String used to end each row.

`quote_char` Byte to use as quoting character.

`batch_size` Number of rows that will be processed per thread.

`datetime_format` A format string, with the specifiers defined by the chrono Rust crate. If no format specified, the default fractional-second precision is inferred from the maximum timeunit found in the frame's Datetime cols (if any).

`date_format` A format string, with the specifiers defined by the chrono Rust crate.

`time_format` A format string, with the specifiers defined by the chrono Rust crate.

`float_scientific`
: Whether to use scientific form always (`TRUE`), never (`FALSE`), or automatically (`NULL`) for Float32 and Float64 datatypes.

`float_precision`
: Number of decimal places to write, applied to both Float32 and Float64 datatypes.

`decimal_comma`
: If `TRUE`, use a comma `","` as the decimal separator instead of a point. Floats will be encapsulated in quotes if necessary.

`null_value`
: A string representing null values (defaulting to the empty string).

`quote_style`
: Determines the quoting strategy used. Must be one of:

- `"necessary"` (default): This puts quotes around fields only when necessary. They are necessary when fields contain a quote, delimiter or record terminator. Quotes are also necessary when writing an empty record (which is indistinguishable from a record with one empty field). This is the default.
- `"always"`: This puts quotes around every field. Always.
- `"never"`: This never puts quotes around fields, even if that results in invalid CSV data (e.g.: by not quoting strings containing the separator).
- `"non_numeric"`: This puts quotes around all fields that are non-numeric. Namely, when writing a field that does not parse as a valid float or integer, then quotes will be used even if they aren't strictly necessary.

`maintain_order`
: Maintain the order in which data is processed. Setting this to `FALSE` will be slightly faster.

`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.

`sync_on_close`
: Sync to disk when before closing a file. Must be one of:

- `"none"`: does not sync;
- `"data"`: syncs the file contents;
- `"all"`: syncs the file contents and metadata.

`mkdir`
: Recursively create all the directories in the path.

`engine`
: The engine name to use for processing the query. One of the followings:

- `"auto"` (default): Select the engine automatically. In Polars 1.16, the `"in-memory"` engine is selected for most cases. Starting with Polars 2.0, the streaming engine is selected instead. Use `"in-memory"` to keep the current execution engine explicitly.
- `"in-memory"`: Use the in-memory engine. This is an escape hatch for queries whose incidental row order differs under streaming.
- `"streaming"`: Use the streaming engine.

`optimizations` [**Experimental**] A QueryOptFlags object to indicate optimization passes done during query optimization.

Value

- `<lazyframe>$sink_*()` returns `NULL` invisibly.
- `<lazyframe>$lazy_sink_*()` returns a new LazyFrame.

Examples

```
# Sink table 'mtcars' from mem to CSV
tmpf <- tempfile(fileext = ".csv")
as_polars_lf(mtcars)$sink_csv(tmpf)

# Create a query that can be run in streaming end-to-end
tmpf2 <- tempfile(fileext = ".csv")
lf <- pl$scan_csv(tmpf, infer_schema_files = 10)$select(pl$col("cyl") * 2)$lazy_sink_csv(tmpf2)
lf$explain() |>
  cat()

# Execute the query and write to disk
lf$collect()

# Load CSV directly into a DataFrame / memory
pl$read_csv(tmpf2, infer_schema_files = 10)
```

`lazyframe__sink_ipc` *Evaluate the query in streaming mode and write to Arrow IPC File Format*

Description

[**Experimental**]

This allows streaming results that are larger than RAM to be written to disk.

- `<lazyframe>$lazy_sink_*()` don't write directly to the output file(s) until `$collect()` is called. This is useful if you want to save a query to review or run later.
- `<lazyframe>$sink_*()` write directly to the output file(s) (they are shortcuts for `<lazyframe>$lazy_sink_*()$collect()`).

Usage

```
lazyframe__sink_ipc(
  path,
  ...,
  compression = c("uncompressed", "lz4", "zstd"),
  compat_level = c("newest", "oldest"),
  maintain_order = TRUE,
  storage_options = NULL,
  sync_on_close = c("none", "data", "all"),
  mkdir = FALSE,
  engine = c("auto", "in-memory", "streaming"),
  optimizations = pl$QueryOptFlags()
)

lazyframe__lazy_sink_ipc(
  path,
  ...,
  compression = c("uncompressed", "lz4", "zstd"),
  compat_level = c("newest", "oldest"),
  maintain_order = TRUE,
  storage_options = NULL,
  sync_on_close = c("none", "data", "all"),
  mkdir = FALSE
)
```

Arguments

`path` A character. File path to which the file should be written.

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

`compression` Determines the compression algorithm. Must be one of:

- `"uncompressed"` or `NULL`: Write an uncompressed Arrow file.
- `"lz4"`: Fast compression/decompression.
- `"zstd"`: Good compression performance.

`compat_level` Determines the compatibility level when exporting Polars' internal data structures. When specifying a new compatibility level, Polars exports its internal data structures that might not be interpretable by other Arrow implementations. The level can be specified as the name (e.g., `"newest"`) or as a scalar integer (Currently, `0` and `1` are supported).

- `"newest"` [**Experimental**] (default): Use the highest level, currently same as `1` (Low compatibility).
- `"oldest"`: Same as `0` (High compatibility).

`maintain_order` Maintain the order in which data is processed. Setting this to `FALSE` will be slightly faster.

`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.

`sync_on_close`
: Sync to disk when before closing a file. Must be one of:

  - `"none"`: does not sync;
  - `"data"`: syncs the file contents;
  - `"all"`: syncs the file contents and metadata.

`mkdir`
: Recursively create all the directories in the path.

`engine`
: The engine name to use for processing the query. One of the followings:

  - `"auto"` (default): Select the engine automatically. In Polars 1.16, the `"in-memory"` engine is selected for most cases. Starting with Polars 2.0, the streaming engine is selected instead. Use `"in-memory"` to keep the current execution engine explicitly.
  - `"in-memory"`: Use the in-memory engine. This is an escape hatch for queries whose incidental row order differs under streaming.
  - `"streaming"`: Use the streaming engine.

`optimizations`
: [**Experimental**] A QueryOptFlags object to indicate optimization passes done during query optimization.

Value

- `<lazyframe>$sink_*()` returns `NULL` invisibly.
- `<lazyframe>$lazy_sink_*()` returns a new LazyFrame.

Examples

```
tmpf <- tempfile(fileext = ".arrow")
as_polars_lf(mtcars)$sink_ipc(tmpf, compression = "zstd")

pl$read_ipc(tmpf)
```

lazyframe__sink_ndjson

Evaluate the query in streaming mode and write to a NDJSON file

Description

[**Experimental**]

This allows streaming results that are larger than RAM to be written to disk.

Usage

```
lazyframe__sink_ndjson(
  path,
  ...,
  compression = c("uncompressed", "gzip", "zstd"),
  compression_level = NULL,
  check_extension = TRUE,
  maintain_order = TRUE,
  storage_options = NULL,
  sync_on_close = c("none", "data", "all"),
  mkdir = FALSE,
  engine = c("auto", "in-memory", "streaming"),
  optimizations = pl$QueryOptFlags()
)

lazyframe__lazy_sink_ndjson(
  path,
  ...,
  compression = c("uncompressed", "gzip", "zstd"),
  compression_level = NULL,
  check_extension = TRUE,
  maintain_order = TRUE,
  storage_options = NULL,
  sync_on_close = c("none", "data", "all"),
  mkdir = FALSE
)
```

Arguments

`path`	A character. File path to which the file should be written.
`...`	These dots are for future extensions and must be empty.
`compression`	[**Experimental**] What compression format to use. Must be one of `uncompressed` (default), `gzip`, or `zstd`.
`compression_level`	[**Experimental**] The compression level to use, typically 0-9 or `NULL` to let the engine choose.

`check_extension`
: [**Experimental**] Whether to check if the filename matches the compression settings. Will raise an error if compression is set to `"uncompressed"` and the filename ends in one of `".gz"`, `".zst"`, `".zstd"`, or if `compression != "uncompressed"` and the file uses a mismatched extension. Only applies if file is a path.

`maintain_order`
: Maintain the order in which data is processed. Setting this to `FALSE` will be slightly faster.

`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.

`sync_on_close`
: Sync to disk when before closing a file. Must be one of:

- `"none"`: does not sync;
- `"data"`: syncs the file contents;
- `"all"`: syncs the file contents and metadata.

`mkdir`
: Recursively create all the directories in the path.

`engine`
: The engine name to use for processing the query. One of the followings:

- `"auto"` (default): Select the engine automatically. In Polars 1.16, the `"in-memory"` engine is selected for most cases. Starting with Polars 2.0, the streaming engine is selected instead. Use `"in-memory"` to keep the current execution engine explicitly.
- `"in-memory"`: Use the in-memory engine. This is an escape hatch for queries whose incidental row order differs under streaming.
- `"streaming"`: Use the streaming engine.

`optimizations`
: [**Experimental**] A QueryOptFlags object to indicate optimization passes done during query optimization.

Value

- `<lazyframe>$sink_*()` returns `NULL` invisibly.
- `<lazyframe>$lazy_sink_*()` returns a new LazyFrame.

Examples

```
dat <- as_polars_lf(head(mtcars))
```

```
destination <- tempfile()

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

`lazyframe__slice` *Get a slice of the LazyFrame.*

Description

Get a slice of the LazyFrame.

Usage

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

Arguments

`offset`	Start index. Negative indexing is supported.
`length`	Length of the slice. If `NULL` (default), all rows starting at the offset will be selected.

Value

A polars 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

...	<dynamic-dots> 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 TRUE, 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__sql *Execute a SQL query against the LazyFrame*

Description

[**Experimental**] Execute a SQL query against the LazyFrame.

Usage

```
lazyframe__sql(query, ..., table_name = "self")
```

Arguments

`query`	SQL query to execute.
`...`	These dots are for future extensions and must be empty.
`table_name`	Optionally provide an explicit name for the table that represents the calling frame (defaults to `"self"`).

Details

The calling frame is automatically registered as a table in the SQL context under the name `"self"`.

More control over registration and execution behaviour is available by using the SQLContext object.

Value

A polars LazyFrame

Examples

```
lf1 <- pl$LazyFrame(a = 1:3, b = 6:8, c = c("z", "y", "x"))

# Query the LazyFrame using SQL:
lf1$sql("SELECT c, b FROM self WHERE a > 1")$collect()

# Apply SQL transforms (aliasing "self" to "frame") then filter natively
# (you can freely mix SQL and native operations):
lf1$sql(
  query = "
       SELECT
          a,
          (a % 2 == 0) AS a_is_even,
          (b::float4 / 2) AS 'b/2',
          CONCAT_WS(':', c, c, c) AS c_c_c
       FROM frame
       ORDER BY a
 ",
  table_name = "frame",
)$filter(!pl$col("c_c_c")$str$starts_with("x"))$collect()
```

---

lazyframe__std — *Aggregate the columns of this LazyFrame to their standard deviation values*

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 - 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()
```

---

lazyframe__sum — *Aggregate the columns of this LazyFrame to their sum values*

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()
```

lazyframe__tail *Get the last* `n` *rows.*

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__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, optimizations = pl$QueryOptFlags())
```

Arguments

`optimized`	Optimize the query plan.
`optimizations`	**[Experimental]** A QueryOptFlags object to indicate optimization passes done during query optimization.

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__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

`k`	Number of rows to return.
`...`	These dots are for future extensions and must be empty.
`by`	Column(s) used to determine the bottom rows. Accepts expression input. Strings are parsed as column names.
`reverse`	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__unique *Drop duplicate rows*

Description

Drop duplicate rows

Usage

```
lazyframe__unique(
  ...,
  keep = c("any", "none", "first", "last"),
  maintain_order = FALSE
)
```

Arguments

`...` <dynamic-dots> Column names or selectors for which are considered. If empty (default), use all columns (same as specifying with the selector `cs$all()`).

`keep` Which of the duplicate rows to keep. Must be one of:

- `"any"`: does not give any guarantee of which row is kept. This allows more optimizations.
- `"none"`: don't keep duplicate rows.
- `"first"`: keep first unique row.
- `"last"`: keep last unique row.

`maintain_order` Keep the same order as the original data. This is more expensive to compute. Setting this to `TRUE` 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$unique(c("bar", "ham"), maintain_order = TRUE)$collect()

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

---

lazyframe__unnest — *Decompose struct columns into separate columns for each of their fields*

Description

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

Usage

```
lazyframe__unnest(..., separator = NULL)
```

Arguments

`...`	<`dynamic-dots`> Name of the struct column(s) or selectors that should be unnested.
`separator`	`NULL` (default) or single string. Rename output column names as combination of the struct column name, name separator and field name.

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$unnest("a_and_c", separator = ":")$collect()
```

lazyframe__unpivot *Unpivot a frame from wide to long format*

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

`on`	Column(s) or selector(s) to use as values variables. If `on` is empty (`cs$empty()`), no columns will be used. If set to `NULL` (default), all columns that are not in `index` will be used.
`...`	These dots are for future extensions and must be empty.
`index`	Column(s) or selector(s) to use as identifier variables.
`variable_name`	Name to give to the new column containing the names of the melted columns. Defaults to `"variable"`.
`value_name`	Name to give to the new column containing the values of the melted columns. Defaults to `"value"`.
	The row order of the result is not guaranteed; use `$sort()` when a stable order is required.

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()
```

lazyframe__var — *Aggregate the columns in the LazyFrame to their variance value*

Description

Aggregate the columns in the LazyFrame to their variance value

Usage

```
lazyframe__var(ddof = 1)
```

Arguments

`ddof`	"Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default `ddof` is 1.

Value

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()
```

---

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()
```

---

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()
```

---

lazygroupby__agg *Compute aggregations for each group of a group by operation*

Description

Compute aggregations for each group of a group by operation

Usage

```
lazygroupby__agg(...)
```

Arguments

`...`	<`dynamic-dots`> Aggregations to compute for each group of the group by operation. Accepts expression input. Strings are parsed as column names.

Value

A polars LazyFrame

Examples

```
# Compute the aggregation of the columns for each group.
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"), pl$col("c"))$collect()

# Compute the sum of a column for each group.
lf$group_by("a")$agg(pl$col("b")$sum())$collect()

# Compute multiple aggregates at once by passing a list of expressions.
lf$group_by("a")$agg(pl$sum("b"), pl$col("c")$mean())$collect()

# Use keyword arguments to easily name your expression inputs.
lf$group_by("a")$agg(
  b_sum = pl$sum("b"),
  c_mean_squared = (pl$col("c") ** 2)$mean()
)$collect()
```

---

lazygroupby__having *Filter groups with a list of predicates after aggregation*

Description

Using this method is equivalent to adding the predicates to the aggregation and filtering afterwards.

This method can be chained and all conditions will be combined using `&`.

Usage

```
lazygroupby__having(...)
```

Arguments

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

Value

A lazy groupby

Examples

```
lf <- pl$LazyFrame(x = c("a", "b", "a", "b", "c"))

# Only keep groups that contain more than one element:
lf$group_by("x")$having(
  pl$len() > 1
)$agg()$collect()
```

lazygroupby__head *Get the first* `n` *rows of each group*

Description

Get the first `n` rows of each group

Usage

```
lazygroupby__head(n = 5)
```

Arguments

`n`	Number of rows to return.

Value

A polars LazyFrame

Examples

```
lf <- pl$LazyFrame(
  letters = c("c", "c", "a", "c", "a", "b"),
  nrs = 1:6
)
lf$collect()

lf$group_by("letters")$head(2)$sort("letters")$collect()
```

lazygroupby__len *Return the number of rows in each group*

Description

Return the number of rows in each group

Usage

```
lazygroupby__len(name = NULL)
```

Arguments

`name`	Assign a name to the resulting column. If `NULL`, defaults to `"len"`.

Value

A polars LazyFrame

Examples

```
lf <- pl$LazyFrame(
  a = c("Apple", "Apple", "Orange"),
  b = c(1, NA, 2)
)
lf$group_by("a")$len()$collect()

lf$group_by("a")$len("n")$collect()
```

---

lazygroupby__max *Reduce the groups to the maximal value*

Description

Reduce the groups to the maximal value

Usage

```
lazygroupby__max()
```

Value

A polars LazyFrame

Examples

```
lf <- pl$LazyFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
lf$collect()

lf$group_by("grp")$max()$collect()
```

---

lazygroupby__mean *Return the mean per group*

Description

Return the mean per group

Usage

```
lazygroupby__mean()
```

Value

A polars LazyFrame

Examples

```
lf <- pl$LazyFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
lf$collect()

lf$group_by("grp")$mean()$collect()
```

---

lazygroupby__median *Return the median per group*

Description

Return the median per group

Usage

```
lazygroupby__median()
```

Value

A polars LazyFrame

Examples

```
lf <- pl$LazyFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
lf$collect()

lf$group_by("grp")$median()$collect()
```

lazygroupby__min *Reduce the groups to the minimal value*

Description

Reduce the groups to the minimal value

Usage

```
lazygroupby__min()
```

Value

A polars LazyFrame

Examples

```
lf <- pl$LazyFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
lf$collect()

lf$group_by("grp")$min()$collect()
```

lazygroupby__n_unique *Count the unique values per group*

Description

Count the unique values per group

Usage

```
lazygroupby__n_unique()
```

Value

A polars LazyFrame

Examples

```
lf <- pl$LazyFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
lf$collect()

lf$group_by("grp")$n_unique()$collect()
```

`lazygroupby__quantile`

Compute the quantile per group

Description

Compute the quantile per group

Usage

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

Arguments

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

Value

A polars LazyFrame

Examples

```
lf <- pl$LazyFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
```

```
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
lf$collect()

lf$group_by("grp")$quantile(0.5)$collect()
```

---

lazygroupby__sum *Return the sum per group*

Description

Return the sum per group

Usage

```
lazygroupby__sum()
```

Value

A polars LazyFrame

Examples

```
lf <- pl$LazyFrame(
  grp = c("c", "c", "a", "c", "a", "b"),
  x = c(0.5, 0.5, 4, 10, 13, 14),
  y = 1:6,
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
)
lf$collect()

lf$group_by("grp")$sum()$collect()
```

---

lazygroupby__tail *Get the last* `n` *rows of each group*

Description

Get the last `n` rows of each group

Usage

```
lazygroupby__tail(n = 5)
```

Arguments

`n`	Number of rows to return.

Value

A polars LazyFrame

Examples

```
lf <- pl$LazyFrame(
  letters = c("c", "c", "a", "c", "a", "b"),
  nrs = 1:6
)
lf$collect()

lf$group_by("letters")$tail(2)$sort("letters")$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.

- `<lazyframe>$lazy_sink_*()` don't write directly to the output file(s) until `$collect()` is called. This is useful if you want to save a query to review or run later.
- `<lazyframe>$sink_*()` write directly to the output file(s) (they are shortcuts for `<lazyframe>$lazy_sink_*()$collect()`).

Usage

```
parquet_statistics(
  ...,
  min = TRUE,
  max = TRUE,
  distinct_count = TRUE,
  null_count = TRUE
)

lazyframe__sink_parquet(
  path,
  ...,
  compression = c("lz4", "uncompressed", "snappy", "gzip", "brotli", "zstd"),
  compression_level = NULL,
  statistics = TRUE,
  row_group_size = NULL,
  data_page_size = NULL,
  maintain_order = TRUE,
  storage_options = NULL,
  sync_on_close = c("none", "data", "all"),
```

```
  mkdir = FALSE,
  engine = c("auto", "in-memory", "streaming"),
  optimizations = pl$QueryOptFlags()
)

lazyframe__lazy_sink_parquet(
  path,
  ...,
  compression = c("lz4", "uncompressed", "snappy", "gzip", "brotli", "zstd"),
  compression_level = NULL,
  statistics = TRUE,
  row_group_size = NULL,
  data_page_size = NULL,
  maintain_order = TRUE,
  storage_options = NULL,
  sync_on_close = c("none", "data", "all"),
  mkdir = FALSE
)
```

Arguments

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

`min` Include stats on the minimum values in the column.

`max` Include stats on the maximum values in the column.

`distinct_count` Include stats on the number of distinct values in the column.

`null_count` Include stats on the number of null values in the column.

`path` A character. File path to which the file should be written.

`compression` The compression method. Defaults to `"lz4"`. Must be one of:

- `"lz4"`: fast compression/decompression.
- `"uncompressed"`
- `"snappy"`: this guarantees that the parquet file will be compatible with older parquet readers.
- `"gzip"`
- `"brotli"`
- `"zstd"`: good compression performance.

`compression_level` `NULL` or integer. The level of compression to use. Only used if method is one of `"gzip"`, `"brotli"`, or `"zstd"`. Higher compression means smaller files on disk:

- `"gzip"`: min-level: 0, max-level: 9, default: 6.
- `"brotli"`: min-level: 0, max-level: 11, default: 1.
- `"zstd"`: min-level: 1, max-level: 22, default: 3.

`statistics` Whether statistics should be written to the Parquet headers. Possible values:

- `TRUE`: enable default set of statistics (default). Some statistics may be disabled.
- `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.

`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.

`sync_on_close`
: Sync to disk when before closing a file. Must be one of:

- `"none"`: does not sync;
- `"data"`: syncs the file contents;
- `"all"`: syncs the file contents and metadata.

`mkdir`
: Recursively create all the directories in the path.

`engine`
: The engine name to use for processing the query. One of the followings:

- `"auto"` (default): Select the engine automatically. In Polars 1.16, the `"in-memory"` engine is selected for most cases. Starting with Polars 2.0, the streaming engine is selected instead. Use `"in-memory"` to keep the current execution engine explicitly.
- `"in-memory"`: Use the in-memory engine. This is an escape hatch for queries whose incidental row order differs under streaming.
- `"streaming"`: Use the streaming engine.

`optimizations`
: **[Experimental]** A QueryOptFlags object to indicate optimization passes done during query optimization.

Value

- `<lazyframe>$sink_*()` returns `NULL` invisibly.
- `<lazyframe>$lazy_sink_*()` returns a new LazyFrame.

Examples

```
# Sink table 'mtcars' from mem to parquet
tmpf <- tempfile()
as_polars_lf(mtcars)$sink_parquet(tmpf)

# Create a query that can be run in streaming end-to-end
tmpf2 <- tempfile()
lf <- pl$scan_parquet(tmpf)$select(pl$col("cyl") * 2)$lazy_sink_parquet(tmpf2)
lf$explain() |>
  cat()

# Execute the query and write to disk
lf$collect()

# Load parquet directly into a DataFrame / memory
pl$read_parquet(tmpf2)
```

---

pl — *Polars top-level function namespace*

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
```

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__all` *Either return an expression representing all columns, or evaluate a bitwise AND operation*

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

Arguments

`names`	The name(s) or data type of the column(s) to use in the aggregation. A character vector or a Polars data type/list of data types can be supplied.
`...`	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 `FALSE` 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 *Apply the AND logical horizontally across columns*

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

Arguments

`names`	The name(s) or data type of the column(s) to use in the aggregation. A character vector or a Polars data type/list of data types can be supplied.
`...`	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, 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_sort_by *Return the row indices that would sort the column(s)*

Description

Return the row indices that would sort the column(s)

Usage

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

Arguments

`...`	<dynamic-dots> 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 `TRUE`, streaming is not possible and performance might be worse since this requires a stable search.

Value

A polars expression

Examples

```
# Pass a single column name to compute the arg sort by that column.
df <- pl$DataFrame(
  a = c(0, 1, 1, 0),
  b = c(3, 2, 3, 2),
  c = c(1, 2, 3, 4)
)
df$select(pl$arg_sort_by("a"))

# Compute the arg sort by multiple columns by either passing a list of
# columns, or by specifying each column as a positional argument.
df$select(pl$arg_sort_by("a", "b", descending = TRUE))

# Use gather to apply the arg sort to other columns.
df$select(pl$col("c")$gather(pl$arg_sort_by("a")))
```

`pl__arg_where` *Return indices where* `condition` *evaluates to* `TRUE`

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 — *Folds the columns from left to right, keeping the first non-null value*

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(c("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(names)
```

Arguments

`names` The name(s) or data type of the column(s) to represent. It accepts one of the following:

- A character vector representing column names
  - Regular expressions starting with `^` and ending with `$` are allowed.
  - Single wildcard `"*"` has a special meaning: check the examples.
- A Polars DataType or list of Polars data types

Value

A polars expression

Examples

```
# a single column by a character
pl$col("foo")

# multiple columns by characters
pl$col(c("foo", "bar"))

# multiple columns by polars data types
pl$col(c(pl$Float64, pl$String))

# Single `"*"` is converted to a wildcard expression
pl$col("*")

# 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__collect_all *Collect multiple LazyFrames at the same time*

Description

This can run all the computation graphs in parallel or combined. Common Subplan Elimination is applied on the combined plan, meaning that diverging queries will run only once.

Usage

```
pl__collect_all(
  lazy_frames,
  ...,
  engine = c("auto", "in-memory", "streaming"),
  optimizations = pl$QueryOptFlags()
)
```

Arguments

`lazy_frames`	A list of LazyFrames to collect.
`...`	These dots are for future extensions and must be empty.
`engine`	The engine name to use for processing the query. One of the followings: • `"auto"` (default): Select the engine automatically. In Polars 1.16, the `"in-memory"` engine is selected for most cases. Starting with Polars 2.0, the streaming engine is selected instead. Use `"in-memory"` to keep the current execution engine explicitly. • `"in-memory"`: Use the in-memory engine. This is an escape hatch for queries whose incidental row order differs under streaming. • `"streaming"`: Use the streaming engine.
`optimizations`	**[Experimental]** A QueryOptFlags object to indicate optimization passes done during query optimization.

Value

A list containing all the collected DataFrames, in the same order as the input LazyFrames.

Examples

```
lf <- as_polars_lf(mtcars)$with_columns(sqrt_mpg = pl$col("mpg")$sqrt())

cyl_4 <- lf$filter(pl$col("cyl") == 4)
cyl_6 <- lf$filter(pl$col("cyl") == 6)

# We could do `cyl_4$collect()` and `cyl_6$collect()`, but this would be
# wasteful because `sqrt_mpg` would be computed twice.
# `pl$collect_all()` executes only once the parts of the query that are
# identical across LazyFrames.
pl$collect_all(list(cyl_4, cyl_6))
```

---

pl__concat — *Combine multiple DataFrames, LazyFrames, or Series into a single object*

Description

Combine multiple DataFrames, LazyFrames, or Series into a single object

Usage

```
pl__concat(
  ...,
  how = "vertical",
  rechunk = FALSE,
  parallel = TRUE,
  strict = deprecated()
)
```

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. All input frames must have the same height; raises an error otherwise; • `"horizontal_extend"`: stacks Series from DataFrames horizontally and fills with `null` if the lengths don't match; • `"align"`, `"align_full"`, `"align_left"`, `"align_right"`: Combines frames horizontally, auto-determining the common key columns and aligning rows using the same logic as `align_frames` (note that `"align"` is an alias for `"align_full"`). The "align" strategy determines the type of join used to align the frames, equivalent to the "how" parameter on `align_frames`. Note that the common join columns are automatically coalesced, but other column collisions will raise an error (if you need more control over this you should use a suitable `join` method directly). Series only support the `"vertical"` strategy.
`rechunk`	Make sure that the result data is in contiguous memory.
`parallel`	Only relevant for LazyFrames. This determines if the concatenated lazy computations may be executed in parallel.
`strict`	[**Deprecated**] Deprecated. `how = "horizontal"` already requires equal heights. Use `how = "horizontal_extend"` to pad shorter frames with null.

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_extend")

# 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")

df_a1 <- pl$DataFrame(id = 1:2, x = 3:4)
df_a2 <- pl$DataFrame(id = 2:3, y = 5:6)
df_a3 <- pl$DataFrame(id = c(1L, 3L), z = 7:8)
pl$concat(df_a1, df_a2, df_a3, how = "align")
pl$concat(df_a1, df_a2, df_a3, how = "align_left")
pl$concat(df_a1, df_a2, df_a3, how = "align_right")
```

`pl__concat_arr` *Horizontally concatenate columns into a single array column*

Description

[**Experimental**]

Usage

```
pl__concat_arr(...)
```

Arguments

`...`	<dynamic-dots> Columns to concatenate into a single array column. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals.

Value

A polars expression

Examples

```
# Concatenate two existing array columns.
df <- pl$DataFrame(a = list(1:2, 3:4, 5:6), b = list(4, 1, NA))$cast(
  a = pl$Array(pl$Int64, 2),
  b = pl$Array(pl$Int64, 1)
)

df$with_columns(concat_arr = pl$concat_arr("a", "b"))

# Concatenate two existing non-array columns.
df <- pl$DataFrame(a = c(NA, 5, 6), b = c(6, 5, NA))
df$with_columns(concat_arr = pl$concat_arr("a", "b"))

# Concatenate mixed array and non-array columns.
df <- pl$DataFrame(a = list(NA, 5L, 6L), b = c(6L, 5L, NA))$cast(
  a = pl$Array(pl$Int32, 1)
)
df$with_columns(concat_arr = pl$concat_arr("a", "b"))


# Unit-length columns are broadcasted:
df$with_columns(concat_arr = pl$concat_arr("a", pl$sum("b")))
```

`pl__concat_list` *Horizontally concatenate columns into a single list column*

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 = " ",
  )
)
```

---

pl__cum_sum *Cumulatively sum all values*

Description

This function is syntactic sugar for `col(names)$cum_sum()`.

Usage

```
pl__cum_sum(names, ...)
```

Arguments

`names`	The name(s) or data type of the column(s) to use in the aggregation. A character vector or a Polars data type/list of data types can be supplied.
`...`	These dots are for future extensions and must be empty.

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(c("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 or length 1. 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`: `$width` returns a integer with the number of columns of the DataFrame.
- `flags`: `$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__date — *Create a Polars literal expression of type Date*

Description

Create a Polars literal expression of type Date

Usage

```
pl__date(year, month, day)
```

Arguments

year	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of year.
month	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of month. Range: 1-12.
day	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of day. Range: 1-31.

Value

A polars expression

Examples

```
df <- pl$DataFrame(month = 1:3, day = 4:6)
df$with_columns(pl$date(2024, pl$col("month"), pl$col("day")))

# We can also use `pl$date()` for filtering:
df <- pl$DataFrame(
  start = rep(as.Date("2024-01-01"), 3),
  end = as.Date(c("2024-05-01", "2024-07-01", "2024-09-01"))
)
df$filter(pl$col("end") > pl$date(2024, 6, 1))
```

---

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

`start`	Lower bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`end`	Upper bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`interval`	Interval of the range periods, specified as a difftime object or using the Polars duration string language. See the `Polars duration string language` 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_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 *Create a column of date ranges*

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

`start`	Lower bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`end`	Upper bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`interval`	Interval of the range periods, specified as a difftime object or using the Polars duration string language. See the `Polars duration string language` 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__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

`year`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of year.
`month`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of month. Range: 1-12.
`day`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of day. Range: 1-31.
`hour`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of hour. Range: 0-23.
`minute`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of minute. Range: 0-59.
`second`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of second. Range: 0-59.
`microsecond`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of microsecond. Range: 0-999999.

`...`	These dots are for future extensions and must be empty.
`time_unit`	One of `"us"` (default, microseconds), `"ns"` (nanoseconds) or `"ms"`(milliseconds). Representing the unit of time.
`time_zone`	A string or `NULL` (default). Representing the timezone.
`ambiguous`	Determine how to deal with ambiguous datetimes. 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 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

`start`	Lower bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`end`	Upper bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`interval`	Interval of the range periods, specified as a difftime object or using the Polars duration string language. See the `Polars duration string language` 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"`.
`time_unit`	Time unit of the resulting the Datetime data type. One of `"ns"`, `"us"`, `"ms"` or `NULL`
`time_zone`	Time zone of the resulting Datetime 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

`start`	Lower bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`end`	Upper bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`interval`	Interval of the range periods, specified as a difftime object or using the Polars duration string language. See the `Polars duration string language` 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"`.
`time_unit`	Time unit of the resulting the Datetime data type. One of `"ns"`, `"us"`, `"ms"` or `NULL`
`time_zone`	Time zone of the resulting Datetime 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__dtype_of — *Get a lazily evaluated DataType of a column or expression*

Description

[**Experimental**] Get a lazily evaluated DataType of a column or expression

Usage

```
pl__dtype_of(col_or_expr)
```

Arguments

`col_or_expr`	Either a string for the column_name or an expression.

Value

A polars datatype expression. This is not the same as a polars expression.

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

`...`	These dots are for future extensions and must be empty.
`weeks`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of weeks, or `NULL` (default).
`days`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of days, or `NULL` (default).
`hours`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of hours, or `NULL` (default).
`minutes`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of minutes, or `NULL` (default).
`seconds`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of seconds, or `NULL` (default).
`milliseconds`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of milliseconds, or `NULL` (default).
`microseconds`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of microseconds, or `NULL` (default).
`nanoseconds`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of nanoseconds, or `NULL` (default).
`time_unit`	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 `nanoseconds` 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__explain_all — *Explain multiple LazyFrames as if passed to* `pl$collect_all()`

Description

Common Subplan Elimination is applied on the combined plan, meaning that diverging queries will run only once.

Usage

```
pl__explain_all(lazy_frames, ..., optimizations = pl$QueryOptFlags())
```

Arguments

`lazy_frames`	A list of LazyFrames to collect.
`...`	These dots are for future extensions and must be empty.
`optimizations`	**[Experimental]** A QueryOptFlags object to indicate optimization passes done during query optimization.

Value

A character value containing the query plan.

Examples

```
lf <- as_polars_lf(mtcars)$with_columns(sqrt_mpg = pl$col("mpg")$sqrt())

cyl_4 <- lf$filter(pl$col("cyl") == 4)
cyl_6 <- lf$filter(pl$col("cyl") == 6)

# Note that `sqrt_mpg` is present in the plans for `cyl_4` and `cyl_6` but
# only once in the plan produced by `pl$explain_all()`. This is because
# `pl$explain_all()` eliminates parts of the queries that are shared across
# multiple plans.
pl$explain_all(list(cyl_4, cyl_6)) |>
  writeLines()
```

---

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 — *Generate a range of integers*

Description

Generate a range of integers

Usage

```
pl__int_range(start = 0, end = NULL, step = 1, ..., dtype = pl$Int64)
```

Arguments

`start`	Start of the range (inclusive). Defaults to 0.
`end`	End of the range (exclusive). If `NULL` (default), the value of `start` is used and `start` is set to 0.
`step`	Step size of the range.
`...`	These dots are for future extensions and must be empty.
`dtype`	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 — *Generate a range of integers for each row of the input columns*

Description

Generate a range of integers for each row of the input columns

Usage

```
pl__int_ranges(start = 0, end = NULL, step = 1, ..., dtype = pl$Int64)
```

Arguments

`start`	Start of the range (inclusive). Defaults to 0.
`end`	End of the range (exclusive). If `NULL` (default), the value of `start` is used and `start` is set to 0.
`step`	Step size of the range.
`...`	These dots are for future extensions and must be empty.
`dtype`	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 — *Get the last column of the context*

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 or length 1. 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__linear_space *Create sequence of evenly-spaced points*

Description

[**Experimental**]

Usage

```
pl__linear_space(
  start,
  end,
  num_samples,
  ...,
  closed = c("both", "left", "none", "right")
)
```

Arguments

`start`	Lower bound of the range.
`end`	Upper bound of the range.
`num_samples`	Number of samples in the output sequence.
`...`	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"`.

Details

`linear_space` works with numeric and temporal dtypes. When the `start` and `end` parameters are `Date` dtypes, the output sequence consists of equally-spaced `Datetime` elements with millisecond precision.

Value

A polars expression

Examples

```
pl$select(
  pl$linear_space(start = 0, end = 1, num_samples = 3)
)
pl$select(
  pl$linear_space(start = 0, end = 1, num_samples = 3, closed = "left")
)
pl$select(
  pl$linear_space(start = 0, end = 1, num_samples = 3, closed = "right")
)
pl$select(
```

```
  pl$linear_space(start = 0, end = 1, num_samples = 3, closed = "none")
)

# Date endpoints generate a sequence of Datetime values:
pl$select(
  pl$linear_space(
    start = as.Date("2025-01-01"),
    end = as.Date("2025-02-01"),
    num_samples = 3,
    closed = "right"
  )
)

# You can generate a sequence using the length of the dataframe:
df <- pl$DataFrame(a = c(1, 2, 3, 4, 5))
df$with_columns(ls = pl$linear_space(0, 1, pl$len()))
```

`pl__linear_spaces` *Create sequence of evenly-spaced points for each row between* `start` *and* `end`

Description

[**Experimental**] The number of values in each sequence is determined by `num_samples`.

Usage

```
pl__linear_spaces(
  start,
  end,
  num_samples,
  ...,
  closed = c("both", "left", "none", "right"),
  as_array = FALSE
)
```

Arguments

`start`	Lower bound of the range.
`end`	Upper bound of the range.
`num_samples`	Number of samples in the output sequence.
`...`	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"`.
`as_array`	Return result as a fixed-length `Array`. `num_samples` must be a constant.

Details

`linear_space` works with numeric and temporal dtypes. When the `start` and `end` parameters are `Date` dtypes, the output sequence consists of equally-spaced `Datetime` elements with millisecond precision.

Value

A polars expression

Examples

```
df <- pl$DataFrame(start = c(1, -1), end = c(3, 2), num_samples = c(4, 5))
df$with_columns(ls = pl$linear_spaces("start", "end", "num_samples"))

df$with_columns(ls = pl$linear_spaces("start", "end", 3, as_array = TRUE))
```

---

pl__list — *Collect columns into a list column*

Description

Unlike `pl$concat_list()`, list-typed inputs are not extended: the value of each input becomes a single element of the output list. This means that `List(T)` inputs produce a `List(List(T))` output.

Usage

```
pl__list(...)
```

Arguments

`...`	<`dynamic-dots`> Columns to collect into a list column. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals.

Value

A polars expression

Examples

```
# Wrap non-list columns into a list (same as pl$concat_list() in this case).
df <- pl$DataFrame(a = c(1, 2, 3), b = c(4, 5, 6))
df$with_columns(a_b = pl$list("a", "b"))

# Collect list columns into a list of lists. pl$concat_list() would instead
# concatenate them into a single list.
df <- pl$DataFrame(a = list(1:2, 3L, 4:5), b = list(6L, 7:8, 9L))
df$with_columns(
```

```
  list = pl$list("a", "b"),
  concat_list = pl$concat_list("a", "b")
)
```

`pl__lit` *Return an expression representing a literal value*

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 `x` param of `as_polars_expr()`.
`dtype`	A polars data type or `NULL` (default). If not `NULL`, casted to the specified data type.

Value

A polars expression

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(names, ...)
```

Arguments

`names`	The name(s) or data type of the column(s) to use in the aggregation. A character vector or a Polars data type/list of data types can be supplied.
`...`	These dots are for future extensions and must be empty.

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(c("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(names, ...)
```

Arguments

`names`	The name(s) or data type of the column(s) to use in the aggregation. A character vector or a Polars data type/list of data types can be supplied.
`...`	These dots are for future extensions and must be empty.

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(c("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 *Get the nth column(s) of the context*

Description

Get the nth column(s) of the context

Usage

```
pl__nth(indices, strict = TRUE)
```

Arguments

indices	One or more indices representing the columns to retrieve.
strict	[**Experimental**] Passed to cs$by_index()'s require_all argument.

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__PartitionBy *Partitioning scheme to write files*

Description

[Experimental]

Partitioning schemes are used to write multiple files with `sink_*` and `write_*` methods.

- `pl$PartitionBy()`: Configuration for writing to multiple output files. Supports partitioning by key expressions, file size limits, or both.

Usage

```
pl__PartitionBy(
  base_path,
  ...,
  key = NULL,
  include_key = NULL,
  max_rows_per_file = NULL,
  approximate_bytes_per_file = NULL
)
```

Arguments

`base_path`	The base path for the output files. Use the `mkdir` option of the `sink_*` methods to ensure directories in the path are created.
`...`	These dots are for future extensions and must be empty.
`key`	Something can be coerced to a list of expressions, or `NULL` (default). Used to partition by.
`include_key`	A bool indicating whether to include the key columns in the output files. Can only be used if `key` is specified, otherwise should be `NULL`.

`max_rows_per_file`
: An integer-ish value indicating the maximum size in rows of each of the generated files.

`approximate_bytes_per_file`
: An integer-ish value indicating approximate number of bytes to write to each file, or `NULL`. This is measured as the estimated size of the DataFrame in memory. Defaults to approximately 4GB when `key` is specified without `max_rows_per_file`; otherwise unlimited.

Examples

```
# Partitioning by columns
temp_dir_1 <- withr::local_tempdir()
as_polars_lf(mtcars)$sink_parquet(
  pl$PartitionBy(
    temp_dir_1,
    key = c("cyl", "am"),
    include_key = FALSE
  ),
  mkdir = TRUE
)
list.files(temp_dir_1, recursive = TRUE)

# Partitioning by max row size
temp_dir_2 <- withr::local_tempdir()
as_polars_lf(mtcars)$sink_csv(
  pl$PartitionBy(
    temp_dir_2,
    max_rows_per_file = 10
  ),
  mkdir = TRUE
)

files <- list.files(temp_dir_2, full.names = TRUE)
files
lapply(files, \(x) nrow(read.csv(x)))

# Partitioning by both key and size
temp_dir_3 <- withr::local_tempdir()
as_polars_lf(mtcars)$sink_parquet(
  pl$PartitionBy(
    temp_dir_3,
    key = "cyl",
    max_rows_per_file = 5,
    approximate_bytes_per_file = 1000000
  ),
  mkdir = TRUE
)
list.files(temp_dir_3, recursive = TRUE)
```

pl__read_csv *New DataFrame from CSV*

Description

New DataFrame from CSV

Usage

```
pl__read_csv(
  source,
  ...,
  has_header = TRUE,
  separator = ",",
  comment_prefix = NULL,
  quote_char = '"',
  skip_rows = 0,
  schema = NULL,
  schema_overrides = NULL,
  null_values = NULL,
  empty_string_is_null = TRUE,
  ignore_errors = FALSE,
  infer_schema = TRUE,
  infer_schema_length = 100,
  infer_schema_files = 10,
  n_rows = NULL,
  encoding = c("utf8", "utf8-lossy"),
  low_memory = FALSE,
  skip_rows_after_header = 0,
  row_index_name = NULL,
  row_index_offset = 0,
  try_parse_dates = FALSE,
  eol_char = "\n",
  raise_if_empty = NULL,
  truncate_ragged_lines = NULL,
  decimal_comma = FALSE,
  glob = TRUE,
  storage_options = NULL,
  include_file_paths = NULL,
  missing_columns = c("raise", "insert"),
  extra_columns = c("raise", "ignore")
)
```

Arguments

`source`	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.

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

`has_header` Indicate if the first row of dataset is a header or not.If `FALSE`, column names will be autogenerated in the following format: `"column_x"` with `x` being an enumeration over every column in the dataset starting at 0.

`separator` Single byte character to use as separator in the file.

`comment_prefix`
A string, which can be up to 5 symbols in length, used to indicate the start of a comment line. For instance, it can be set to `#` or `//`.

`quote_char` Single byte character used for quoting. Set to `NULL` to turn off special handling and escaping of quotes.

`skip_rows` Start reading after a particular number of rows. The header will be parsed at this offset.

`schema` Provide the schema. This means that polars doesn't do schema inference. This argument expects the complete schema, whereas `schema_overrides` can be used to partially overwrite a schema. This must be a list. With `has_header = TRUE`, fields are matched by name and schema names must match the CSV header. With `has_header = FALSE`, fields are matched by position and the schema length must match the input width. Empty string names are valid; names must not contain `NA` values.

`schema_overrides`
Overwrite dtypes during inference. This must be a list. If the list is named, dtypes partially overwrite inferred columns by name. An empty string is a valid column name. If the list is unnamed, dtypes overwrite columns by position and must include one dtype for every CSV column. Names must not contain `NA` values.

`null_values` 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. `c(col1 = "a")`).

`empty_string_is_null`
By default, a missing string value is considered to be `NA`. Setting this parameter to `FALSE` will consider missing string values as an empty character instead.

`ignore_errors` Keep reading the file even if some lines yield errors. You can also use `infer_schema = FALSE` to read all columns as UTF8 to check which values might cause an issue.

`infer_schema` If `TRUE` (default), the schema is inferred from the data using the first `infer_schema_length` rows. When `FALSE`, the schema is not inferred and will be `pl$String` if not specified in `schema` or `schema_overrides`.

`infer_schema_length`
The maximum number of rows to scan for schema inference. This applies individually to each file included according to `infer_schema_files`. If `NULL`, the full data may be scanned (this is slow). Set `infer_schema = FALSE` to read all columns as `pl$String`.

`infer_schema_files`
[Experimental] How many files to use when inferring the schema. Defaults to 10.

`n_rows` Stop reading from the source after reading `n_rows`.

`encoding` Either `"utf8"` or `"utf8-lossy"`. Lossy means that invalid UTF8 values are replaced with "?" characters.

`low_memory` Reduce memory pressure at the expense of performance.

`skip_rows_after_header`
Skip this number of rows when the header is parsed.

`row_index_name`
If not `NULL`, this will insert a row index column with the given name.

`row_index_offset`
Offset to start the row index column (only used if the name is set by `row_index_name`).

`try_parse_dates`
Try to automatically parse dates. Most ISO8601-like formats can be inferred, as well as a handful of others. If this does not succeed, the column remains of data type `pl$String`.

`eol_char` Single byte end of line character (default: `"\n"`). When encountering a file with Windows line endings (`"\r\n"`), one can go with the default `"\n"`. The extra `"\r"` will be removed when processed.

`raise_if_empty`
If `FALSE`, parsing an empty file returns an empty DataFrame or LazyFrame. Defaults to `FALSE` when `has_header = FALSE` and `schema` is supplied, and `TRUE` otherwise.

`truncate_ragged_lines`
Truncate lines that are longer than the schema. If `NULL` (the default), resolves to `TRUE` when `extra_columns = "ignore"` and `FALSE` otherwise.

`decimal_comma` Parse floats using a comma as the decimal separator instead of a period.

`glob` Expand path given via globbing rules.

`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.

`include_file_paths`
Include the path of the source file(s) as a column with this name.

`missing_columns`
Configuration for behavior when columns defined in the schema are missing from the data:

- `"raise"` (default): Raises an error.

- `"insert"`: Inserts the missing columns using NULLs as the row values.

`extra_columns` **[Experimental]** How to handle columns present in the CSV data but absent from the schema. The default, `"raise"`, raises an error. `"ignore"` drops extra columns. When `extra_columns = "ignore"`, `truncate_ragged_lines` must be omitted or `TRUE`; omitted truncation is enabled automatically.

Value

A polars DataFrame

Examples

```
my_file <- tempfile()
write.csv(iris, my_file)
pl$read_csv(my_file, infer_schema_files = 10)
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,
  row_index_name = NULL,
  row_index_offset = 0L,
  storage_options = NULL,
  hive_partitioning = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  include_file_paths = NULL
)
```

Arguments

`source`	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.
`...`	These dots are for future extensions and must be empty.
`n_rows`	Stop reading from the source after reading `n_rows`.

`row_index_name`
: If not `NULL`, this will insert a row index column with the given name.

`row_index_offset`
: Offset to start the row index column (only used if the name is set by `row_index_name`).

`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.

`hive_partitioning`
: Infer statistics and schema from Hive partitioned sources and use them to prune reads. If `NULL` (default), it is automatically enabled when a single directory is passed, and otherwise disabled.

`hive_schema`
: [**Experimental**] A list containing the column names and data types of the columns by which the data is partitioned, e.g. `list(a = pl$String, b = pl$Float32)`. If `NULL` (default), the schema of the Hive partitions is inferred.

`try_parse_hive_dates`
: Whether to try parsing hive values as date / datetime types.

`include_file_paths`
: Include the path of the source file(s) as a column with this name.

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
)
```

Arguments

`source`	A character of the path to an Arrow IPC stream file.
`...`	These dots are for future extensions and must be empty.
`columns`	A character vector of column names to read.
`n_rows`	Stop reading from the source after reading `n_rows`.
`row_index_name`	If not `NULL`, this will insert a row index column with the given name.
`row_index_offset`	Offset to start the row index column (only used if the name is set by `row_index_name`).

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_lines` *Read lines into a string column from a file*

Description

[**Experimental**]

Usage

```
pl__read_lines(
  source,
  ...,
  name = "lines",
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  glob = TRUE,
  storage_options = NULL,
  include_file_paths = NULL
)
```

Arguments

`source`	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.
`...`	These dots are for future extensions and must be empty.
`name`	Name to use for the output column.
`n_rows`	Stop reading from the source after reading `n_rows`.
`row_index_name`	If not `NULL`, this will insert a row index column with the given name.
`row_index_offset`	Offset to start the row index column (only used if the name is set by `row_index_name`).
`glob`	Expand path given via globbing rules.

`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.

`include_file_paths`
: Include the path of the source file(s) as a column with this name.

Value

A polars DataFrame

Examples

```
dest <- withr::local_tempfile()
writeLines("Hello\nworld", dest)
pl$read_lines(dest)
```

`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,
  row_index_name = NULL,
  row_index_offset = 0L,
```

```
  ignore_errors = FALSE,
  storage_options = NULL,
  include_file_paths = NULL
)
```

Arguments

`source` Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.

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

`schema` Provide a list of datatypes for the fields in the NDJSON input. Named elements are matched by name.

`schema_overrides` Overwrite dtypes during schema inference. This must be a list of datatypes. Named elements are matched by name.

`infer_schema_length` The maximum number of rows to scan for schema inference. This applies individually to each file included according to `infer_schema_files`. If `NULL`, the full data may be scanned (this is slow). Set `infer_schema = FALSE` to read all columns as `pl$String`.

`batch_size` Number of rows to read in each batch.

`n_rows` Stop reading from the source after reading `n_rows`.

`low_memory` Reduce memory pressure at the expense of performance.

`row_index_name` If not `NULL`, this will insert a row index column with the given name.

`row_index_offset` Offset to start the row index column (only used if the name is set by `row_index_name`).

`ignore_errors` Keep reading the file even if some lines yield errors. You can also use `infer_schema = FALSE` to read all columns as UTF8 to check which values might cause an issue.

`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.

`include_file_paths` Include the path of the source file(s) as a column with this name.

Value

A polars DataFrame

Examples

```
ndjson_filename <- tempfile()
jsonlite::stream_out(iris, file(ndjson_filename), verbose = FALSE)
pl$read_ndjson(ndjson_filename)
```

---

pl__read_parquet *Read into a DataFrame from Parquet file*

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,
  low_memory = FALSE,
  cache = TRUE,
  storage_options = NULL,
  include_file_paths = NULL,
  missing_columns = c("raise", "insert")
)
```

Arguments

`source`	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.
`...`	These dots are for future extensions and must be empty.
`n_rows`	Stop reading from the source after reading `n_rows`.

`row_index_name`
: If not `NULL`, this will insert a row index column with the given name.

`row_index_offset`
: Offset to start the row index column (only used if the name is set by `row_index_name`).

`parallel`
: This determines the direction and strategy of parallelism.
  - `"auto"` (default): Will try to determine the optimal direction.
  - `"prefiltered"`: [**Experimental**] 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. Falls back to `"auto"` if no predicate is given.
  - `"columns"`, `"row_groups"`: Use the specified direction.
  - `"none"`: No parallelism.

`use_statistics`
: Use statistics in the parquet to determine if pages can be skipped from reading.

`hive_partitioning`
: Infer statistics and schema from Hive partitioned sources and use them to prune reads. If `NULL` (default), it is automatically enabled when a single directory is passed, and otherwise disabled.

`glob`
: Expand path given via globbing rules.

`schema`
: [**Experimental**] Named list of 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 setting `missing_columns = "insert"`.

`hive_schema`
: [**Experimental**] A list containing the column names and data types of the columns by which the data is partitioned, e.g. `list(a = pl$String, b = pl$Float32)`. If `NULL` (default), the schema of the Hive partitions is inferred.

`try_parse_hive_dates`
: Whether to try parsing hive values as date / datetime types.

`low_memory`
: Reduce memory pressure at the expense of performance

`cache`
: 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.

`include_file_paths`
: Include the path of the source file(s) as a column with this name.

`missing_columns`
: Configuration for behavior when columns defined in the schema are missing from the data:

- `"raise"` (default): Raises an error.
- `"insert"`: Inserts the missing columns using NULLs as the row values.

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)
```

---

pl__repeat_ *Construct a column of length n“ filled with the given value*

Description

Construct a column of length n“ filled with the given value

Usage

```
pl__repeat_(value, n, ..., dtype = NULL)
```

Arguments

`value`	Value to repeat.
`n`	Length of the resulting column
`...`	These dots are for future extensions and must be empty.
`dtype`	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__row_index` *Generate a row index*

Description

[**Experimental**] The length of the returned sequence will match the context length. If you would like to generate sequences with custom offsets / length / step size / datatypes, it is recommended to use `pl$int_range()` instead.

Usage

```
pl__row_index(name = "index")
```

Arguments

`name`	Name of the returned column.

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = c("A", "A", "B", "B", "B"))
df$with_columns(pl$row_index(), pl$row_index("another_index"))

df$group_by("x")$agg(pl$row_index())$sort("x")

df$select(pl$row_index())
```

---

pl__scan_csv — *Lazily read from a CSV file or multiple files via glob patterns*

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,
  empty_string_is_null = TRUE,
  ignore_errors = FALSE,
  infer_schema = TRUE,
  infer_schema_length = 100,
  infer_schema_files = 10,
  n_rows = NULL,
  encoding = c("utf8", "utf8-lossy"),
  low_memory = FALSE,
  skip_rows_after_header = 0,
  row_index_name = NULL,
  row_index_offset = 0,
  try_parse_dates = FALSE,
  eol_char = "\n",
  raise_if_empty = NULL,
  truncate_ragged_lines = NULL,
  decimal_comma = FALSE,
  glob = TRUE,
  storage_options = NULL,
```

```
  include_file_paths = NULL,
  missing_columns = c("raise", "insert"),
  extra_columns = c("raise", "ignore")
)
```

Arguments

`source` Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.

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

`has_header` Indicate if the first row of dataset is a header or not.If `FALSE`, column names will be autogenerated in the following format: `"column_x"` with `x` being an enumeration over every column in the dataset starting at 0.

`separator` Single byte character to use as separator in the file.

`comment_prefix` A string, which can be up to 5 symbols in length, used to indicate the start of a comment line. For instance, it can be set to `#` or `//`.

`quote_char` Single byte character used for quoting. Set to `NULL` to turn off special handling and escaping of quotes.

`skip_rows` Start reading after a particular number of rows. The header will be parsed at this offset.

`schema` Provide the schema. This means that polars doesn't do schema inference. This argument expects the complete schema, whereas `schema_overrides` can be used to partially overwrite a schema. This must be a list. With `has_header = TRUE`, fields are matched by name and schema names must match the CSV header. With `has_header = FALSE`, fields are matched by position and the schema length must match the input width. Empty string names are valid; names must not contain `NA` values.

`schema_overrides` Overwrite dtypes during inference. This must be a list. If the list is named, dtypes partially overwrite inferred columns by name. An empty string is a valid column name. If the list is unnamed, dtypes overwrite columns by position and must include one dtype for every CSV column. Names must not contain `NA` values.

`null_values` 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. `c(col1 = "a")`).

`empty_string_is_null` By default, a missing string value is considered to be `NA`. Setting this parameter to `FALSE` will consider missing string values as an empty character instead.

`ignore_errors` Keep reading the file even if some lines yield errors. You can also use `infer_schema = FALSE` to read all columns as UTF8 to check which values might cause an issue.

`infer_schema` If `TRUE` (default), the schema is inferred from the data using the first `infer_schema_length` rows. When `FALSE`, the schema is not inferred and will be `pl$String` if not specified in `schema` or `schema_overrides`.

`infer_schema_length`
: The maximum number of rows to scan for schema inference. This applies individually to each file included according to `infer_schema_files`. If `NULL`, the full data may be scanned (this is slow). Set `infer_schema = FALSE` to read all columns as `pl$String`.

`infer_schema_files`
: **[Experimental]** How many files to use when inferring the schema. Defaults to 10.

`n_rows`
: Stop reading from the source after reading `n_rows`.

`encoding`
: Either `"utf8"` or `"utf8-lossy"`. Lossy means that invalid UTF8 values are replaced with "?" characters.

`low_memory`
: Reduce memory pressure at the expense of performance.

`skip_rows_after_header`
: Skip this number of rows when the header is parsed.

`row_index_name`
: If not `NULL`, this will insert a row index column with the given name.

`row_index_offset`
: Offset to start the row index column (only used if the name is set by `row_index_name`).

`try_parse_dates`
: Try to automatically parse dates. Most ISO8601-like formats can be inferred, as well as a handful of others. If this does not succeed, the column remains of data type `pl$String`.

`eol_char`
: Single byte end of line character (default: `"\n"`). When encountering a file with Windows line endings (`"\r\n"`), one can go with the default `"\n"`. The extra `"\r"` will be removed when processed.

`raise_if_empty`
: If `FALSE`, parsing an empty file returns an empty DataFrame or LazyFrame. Defaults to `FALSE` when `has_header = FALSE` and `schema` is supplied, and `TRUE` otherwise.

`truncate_ragged_lines`
: Truncate lines that are longer than the schema. If `NULL` (the default), resolves to `TRUE` when `extra_columns = "ignore"` and `FALSE` otherwise.

`decimal_comma`
: Parse floats using a comma as the decimal separator instead of a period.

`glob`
: Expand path given via globbing rules.

`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.

`include_file_paths`
: Include the path of the source file(s) as a column with this name.

`missing_columns`
: Configuration for behavior when columns defined in the schema are missing from the data:
  - `"raise"` (default): Raises an error.
  - `"insert"`: Inserts the missing columns using NULLs as the row values.

`extra_columns`
: [**Experimental**] How to handle columns present in the CSV data but absent from the schema. The default, `"raise"`, raises an error. `"ignore"` drops extra columns. When `extra_columns = "ignore"`, `truncate_ragged_lines` must be omitted or `TRUE`; omitted truncation is enabled automatically.

Value

A polars LazyFrame

Examples

```
my_file <- tempfile()
write.csv(iris, my_file)
lazy_frame <- pl$scan_csv(my_file, infer_schema_files = 10)
lazy_frame$collect()
unlink(my_file)
```

---

pl__scan_ipc

Lazily read from an Arrow IPC (Feather v2) file or multiple files via glob patterns

---

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,
  row_index_name = NULL,
  row_index_offset = 0L,
  storage_options = NULL,
  hive_partitioning = NULL,
  hive_schema = NULL,
```

```
  try_parse_hive_dates = TRUE,
  include_file_paths = NULL
)
```

Arguments

`source` Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.

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

`n_rows` Stop reading from the source after reading `n_rows`.

`row_index_name`
If not `NULL`, this will insert a row index column with the given name.

`row_index_offset`
Offset to start the row index column (only used if the name is set by `row_index_name`).

`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.

`hive_partitioning`
Infer statistics and schema from Hive partitioned sources and use them to prune reads. If `NULL` (default), it is automatically enabled when a single directory is passed, and otherwise disabled.

`hive_schema` [**Experimental**] A list containing the column names and data types of the columns by which the data is partitioned, e.g. `list(a = pl$String, b = pl$Float32)`. If `NULL` (default), the schema of the Hive partitions is inferred.

`try_parse_hive_dates`
Whether to try parsing hive values as date / datetime types.

`include_file_paths`
Include the path of the source file(s) as a column with this name.

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_lines` *Construct a LazyFrame which scans lines into a string column from a file*

Description

[**Experimental**]

Usage

```
pl__scan_lines(
  source,
  ...,
  name = "lines",
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  glob = TRUE,
  storage_options = NULL,
  include_file_paths = NULL
)
```

Arguments

`source`	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.
`...`	These dots are for future extensions and must be empty.

`name` Name to use for the output column.

`n_rows` Stop reading from the source after reading `n_rows`.

`row_index_name`
If not `NULL`, this will insert a row index column with the given name.

`row_index_offset`
Offset to start the row index column (only used if the name is set by `row_index_name`).

`glob` Expand path given via globbing rules.

`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.

`include_file_paths`
Include the path of the source file(s) as a column with this name.

Value

A polars LazyFrame

Examples

```
dest <- withr::local_tempfile()
writeLines("Hello\nworld", dest)
pl$scan_lines(dest)$collect()
```

`pl__scan_ndjson` *Lazily read from a local or cloud-hosted NDJSON file(s)*

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,
  row_index_name = NULL,
  row_index_offset = 0L,
  ignore_errors = FALSE,
  storage_options = NULL,
  include_file_paths = NULL
)
```

Arguments

`source` Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.

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

`schema` Provide a list of datatypes for the fields in the NDJSON input. Named elements are matched by name.

`schema_overrides` Overwrite dtypes during schema inference. This must be a list of datatypes. Named elements are matched by name.

`infer_schema_length` The maximum number of rows to scan for schema inference. This applies individually to each file included according to `infer_schema_files`. If `NULL`, the full data may be scanned (this is slow). Set `infer_schema = FALSE` to read all columns as `pl$String`.

`batch_size` Number of rows to read in each batch.

`n_rows` Stop reading from the source after reading `n_rows`.

`low_memory` Reduce memory pressure at the expense of performance.

`row_index_name` If not `NULL`, this will insert a row index column with the given name.

`row_index_offset` Offset to start the row index column (only used if the name is set by `row_index_name`).

`ignore_errors` Keep reading the file even if some lines yield errors. You can also use `infer_schema = FALSE` to read all columns as UTF8 to check which values might cause an issue.

`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.

`include_file_paths`
: Include the path of the source file(s) as a column with this name.

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` *Lazily read from a local or cloud-hosted parquet file (or files)*

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,
  low_memory = FALSE,
```

```
  cache = TRUE,
  storage_options = NULL,
  include_file_paths = NULL,
  missing_columns = c("raise", "insert")
)
```

Arguments

`source` Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.

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

`n_rows` Stop reading from the source after reading `n_rows`.

`row_index_name`
If not `NULL`, this will insert a row index column with the given name.

`row_index_offset`
Offset to start the row index column (only used if the name is set by `row_index_name`).

`parallel` This determines the direction and strategy of parallelism.

- `"auto"` (default): Will try to determine the optimal direction.
- `"prefiltered"`: [**Experimental**] 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. Falls back to `"auto"` if no predicate is given.
- `"columns"`, `"row_groups"`: Use the specified direction.
- `"none"`: No parallelism.

`use_statistics`
Use statistics in the parquet to determine if pages can be skipped from reading.

`hive_partitioning`
Infer statistics and schema from Hive partitioned sources and use them to prune reads. If `NULL` (default), it is automatically enabled when a single directory is passed, and otherwise disabled.

`glob` Expand path given via globbing rules.

`schema` [**Experimental**] Named list of 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 setting `missing_columns = "insert"`.

`hive_schema` [**Experimental**] A list containing the column names and data types of the columns by which the data is partitioned, e.g. `list(a = pl$String, b = pl$Float32)`. If `NULL` (default), the schema of the Hive partitions is inferred.

`try_parse_hive_dates`
Whether to try parsing hive values as date / datetime types.

`low_memory` Reduce memory pressure at the expense of performance

`cache` 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.

`include_file_paths`
Include the path of the source file(s) as a column with this name.

`missing_columns`
Configuration for behavior when columns defined in the schema are missing from the data:

- `"raise"` (default): Raises an error.
- `"insert"`: Inserts the missing columns using NULLs as the row values.

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__SQLContext *Initialize a new* `SQLContext`

Description

[**Experimental**]

Usage

```
pl__SQLContext(...)
```

Arguments

...	<dynamic-dots> Elements that are known in the current `SQLContext`. It accepts any R object that can be converted to a LazyFrame via `as_polars_lf()`. All elements must be named.

Value

An object of class `"polars_sql_context"`

Examples

```
pl$SQLContext(mtcars = mtcars)

pl$SQLContext(mtcars = mtcars, a = data.frame(x = 1))
```

---

pl__struct *Collect columns into a struct column*

Description

Collect columns into a struct column

Usage

```
pl__struct(..., .schema = NULL)
```

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.
.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(c("int", "list")), .schema = struct_schema)
)$unnest("new_struct")
```

`pl__sum` *Sum all values*

Description

This function is syntactic sugar for `col(names)$sum()`.

Usage

```
pl__sum(names, ...)
```

Arguments

`names`	The name(s) or data type of the column(s) to use in the aggregation. A character vector or a Polars data type/list of data types can be supplied.
`...`	These dots are for future extensions and must be empty.

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(c("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__thread_pool_size *Return the number of threads in the Polars thread pool*

Description

Return the number of threads in the Polars thread pool

Usage

```
pl__thread_pool_size()
```

Details

The threadpool size can be overridden by setting the `POLARS_MAX_THREADS` environment variable before process start. It cannot be modified once the package is loaded. It is strongly recommended not to override this value as it will be set automatically by the engine.

Value

The integer of threads used by polars engine.

See Also

- `polars_info()` shows the thread pool size and other information.

Examples

```
pl$thread_pool_size()
```

---

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

`start`	Lower bound of the time range. If omitted, defaults to `00:00:00.000`.
`end`	Upper bound of the time range. If omitted, defaults to `23:59:59.999`
`interval`	Interval of the range periods, specified as a difftime or using the Polars duration string language (see details).
`...`	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".

Examples

```
pl$select(
  time = pl$time_range(
    start = hms::parse_hms("14:00:00"),
    interval = as.difftime("3:15:00")
  )
)
```

---

pl__time_ranges *Create a column of time ranges*

Description

Create a column of time ranges

Usage

```
pl__time_ranges(
  start = NULL,
  end = NULL,
  interval = "1h",
  ...,
  closed = c("both", "left", "none", "right")
)
```

Arguments

`start`	Lower bound of the time range. If omitted, defaults to `00:00:00.000`.
`end`	Upper bound of the time range. If omitted, defaults to `23:59:59.999`
`interval`	Interval of the range periods, specified as a difftime or using the Polars duration string language (see details).
`...`	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".

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"))
```

---

pl__when *Start a when-then-otherwise expression*

Description

Always initiated with a `pl$when()$then()` and optionally followed by chaining one or more `$when()$then()` statements.

An optional `$otherwise()` can be appended at the end. If not declared, a default of `$otherwise(NA)` is used.

Similar to pl$coalesce, the value from the first condition that evaluates to `TRUE` will be picked. If all conditions are `FALSE`, the `otherwise` value is picked.

Usage

```
pl__when(...)
```

Arguments

`...`	<`dynamic-dots`> Condition(s) that must be met in order to apply the subsequent statement. Accepts one or more boolean expressions, which are implicitly combined with `&`.

Details

Polars computes all expressions passed to when-then-otherwise in parallel and filters afterwards. This means each expression must be valid on its own, regardless of the conditions in the when-then-otherwise chain.

String inputs e.g. `when("string")`, `then("string")` or `otherwise("string")` are parsed as column names. pl$lit() can be used to create string values.

Value

A polars expression

Examples

```
# Below we add a column with the value 1, where column "foo" > 2 and the
# value 1 + column "bar" where it isn't.
df <- pl$DataFrame(foo = c(1, 3, 4), bar = c(3, 4, 0))
df$with_columns(
  val = pl$when(pl$col("foo") > 2)$then(1)$otherwise(1 + pl$col("bar"))
)

# Note that when-then always executes all expressions.
# The results are folded left to right, picking the then value from the first
# when condition that is true.
# If no when condition is true the otherwise value is picked.
df$with_columns(
  when = pl$col("foo") > 2,
  then = 1,
  otherwise = 1 + pl$col("bar")
)$with_columns(
  val = pl$when("when")$then("then")$otherwise("otherwise")
)

# Strings are parsed as column names
df$with_columns(
  val = pl$when(pl$col("foo") > 2)$then("foo")$otherwise("bar")
)

# Use pl$lit() to create literal values
df$with_columns(
  val = pl$when(pl$col("foo") > 2)$then(pl$lit("foo"))$otherwise(pl$lit("bar"))
)

# Multiple when-then statements can be chained.
df$with_columns(
  val = pl$when(pl$col("foo") > 2)$
    then(1)$
    when(pl$col("bar") > 2)$
    then(4)$
    otherwise(-1)
)

# The otherwise statement is optional and defaults to $otherwise(NA) if not given.
# This idiom is commonly used to null out values.
df$with_columns(pl$when(pl$col("foo") == 3)$then("bar"))

# Multiple predicates passed to when are combined with &
df$with_columns(
  val = pl$when(pl$col("foo") > 2, pl$col("bar") < 3)$
    then(pl$lit("Yes"))$
    otherwise(pl$lit("No"))
)

# Structs can be used as a way to return multiple values.
# Here we swap the "foo" and "bar" values when "foo" is greater than 2.
```

```
df$with_columns(
  pl$when(pl$col("foo") > 2)$
    then(pl$struct(foo = "bar", bar = "foo"))$
    otherwise(pl$struct("foo", "bar"))$
    struct$
    unnest()
)

# The output name of a when-then expression comes from the first then branch.
# Here we try to set all columns to 0 if any column contains a value less than 2.
tryCatch(
  df$with_columns(
    pl$when(pl$any_horizontal(pl$all() < 2))$then(0)$otherwise(pl$all())
  ),
  error = function(e) e
)

# name$keep() could be used to give preference to the column expression.
df$with_columns(
  pl$when(pl$any_horizontal(pl$all() < 2))$then(0)$otherwise(pl$all())$name$keep()
)

# The logic could also be changed to move the column expression inside then.
df$with_columns(
  pl$when(pl$any_horizontal(pl$all() < 2)$not())$then(pl$all())$otherwise(0)
)
```

---

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

`name`	Name under which the functionality will be accessed.
`ns_fn`	A function returns a new environment 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")
```

---

polars_code_completion_activate

Polars code completion

Description

[**Experimental**] This only works in RStudio.

Usage

```
polars_code_completion_activate(..., verbose = TRUE)

polars_code_completion_deactivate(..., verbose = TRUE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`verbose`	Inform whether code completion is active or not.

Details

This intercepts Rstudio internal functions `.rs.getCompletionsFunction` & `.rs.getCompletionsDollar` in the loaded session environment `tools:rstudio`.

Pressing tab will literally evaluate the left-hand-side in order to compute possible suggestions. Therefore, it may be slow on large data or large queries.

Examples

```
if (interactive()) {
  # Activate completion
  polars_code_completion_activate()

  # Add a "$" and press tab to see functions for LazyFrame
  pl$LazyFrame(x = 1)

  # Add a "$" after pl$col("x") and press tab to see functions for expressions
  pl$LazyFrame(x = 1)$select(pl$col("x"))

  # Add a "$" in count_matches() and press tab to see possible arguments
  pl$LazyFrame(x = 1)$select(pl$col("x")$str$count_matches())

  # Deactivate (restarting the R session also deactivates code completion)
  polars_code_completion_deactivate()
}
```

---

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, arrays, and maps.
- 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 = 38L, scale = 0L)

pl__Datetime(time_unit = c("us", "ns", "ms"), time_zone = NULL)

pl__Duration(time_unit = c("us", "ns", "ms"))

pl__Categorical()

pl__Enum(categories)

pl__Array(inner, shape)

pl__List(inner)

pl__Map(key, value)

pl__Struct(...)
```

Arguments

`precision`	Single integer should be in the range `1` to `38`. The maximum number of digits in each number.
`scale`	Single integer. Number of digits to the right of the decimal point in each number. The default is `0`.
`time_unit`	One of `"us"` (default, microseconds), `"ns"` (nanoseconds) or `"ms"`(milliseconds). Representing the unit of time.
`time_zone`	A string or `NULL` (default). Representing the timezone.
`categories`	A character vector. Should not contain `NA` values and all values should be unique.
`inner`	A polars data type object.
`shape`	A integer-ish vector, representing the shape of the Array.
`key`	A polars data type for the Map keys. Keys cannot have the `Null` type.
`value`	A polars data type for the Map values.
`...`	<`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.
`Float16`, `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.
`Map`	Key/value entries; their types are exposed through `$key` and `$value`.
`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.

Maps:

Use `pl$Map(key, value)` to define a Map data type. To create a Map Series from R values, first create a compatible `List(Struct)` Series and explicitly cast it to the Map type. Ordinary R lists are not implicitly inferred as Map 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$Int32)
pl$Map(pl$String, pl$Int64)
pl$Categorical()
pl$Enum(c("a", "b", "c"))
pl$Struct(a = pl$Int32, b = pl$String)
pl$Null
pl$Unknown
```

`polars_envvars` *Get polars environment variables*

Description

[**Experimental**] Get polars environment variables

Usage

```
polars_envvars()
```

Details

The following envvars are available (in alphabetical order, with the default value in parenthesis):

- `POLARS_FMT_MAX_COLS` (`5`): Set the number of columns that are visible when displaying tables. If negative, all columns are displayed.
- `POLARS_FMT_MAX_ROWS` (`8`): Set the number of rows that are visible when displaying tables. If negative, all rows are displayed. This applies to both DataFrame and Series.
- `POLARS_FMT_STR_LEN` (`32`): Maximum number of characters to display;
- `POLARS_FMT_TABLE_CELL_ALIGNMENT` (`"LEFT"`): set the table cell alignment. Can be `"LEFT"`, `"CENTER"`, `"RIGHT"`;
- `POLARS_FMT_TABLE_CELL_LIST_LEN` (`3`): Maximum number of elements of list variables to display;
- `POLARS_FMT_TABLE_CELL_NUMERIC_ALIGNMENT` (`"LEFT"`): Set the table cell alignment for numeric columns. Can be `"LEFT"`, `"CENTER"`, `"RIGHT"`;
- `POLARS_FMT_TABLE_DATAFRAME_SHAPE_BELOW` (`"0"`): print the DataFrame shape information below the data when displaying tables. Can be `"0"` or `"1"`.
- `POLARS_FMT_TABLE_FORMATTING` (`"UTF8_FULL_CONDENSED"`): Set table formatting style. Possible values:
  - `"ASCII_FULL"`: ASCII, with all borders and lines, including row dividers.
  - `"ASCII_FULL_CONDENSED"`: Same as ASCII_FULL, but with dense row spacing.
  - `"ASCII_NO_BORDERS"`: ASCII, no borders.
  - `"ASCII_BORDERS_ONLY"`: ASCII, borders only.
  - `"ASCII_BORDERS_ONLY_CONDENSED"`: ASCII, borders only, dense row spacing.
  - `"ASCII_HORIZONTAL_ONLY"`: ASCII, horizontal lines only.
  - `"ASCII_MARKDOWN"`: ASCII, Markdown compatible.
  - `"UTF8_FULL"`: UTF8, with all borders and lines, including row dividers.
  - `"UTF8_FULL_CONDENSED"`: Same as UTF8_FULL, but with dense row spacing.
  - `"UTF8_NO_BORDERS"`: UTF8, no borders.
  - `"UTF8_BORDERS_ONLY"`: UTF8, borders only.
  - `"UTF8_HORIZONTAL_ONLY"`: UTF8, horizontal lines only.

  - "NOTHING": No borders or other lines.
- POLARS_FMT_TABLE_HIDE_COLUMN_DATA_TYPES ("0"): Hide table column data types (i64, f64, str etc.). Can be "0" or "1".
- POLARS_FMT_TABLE_HIDE_COLUMN_NAMES ("0"): Hide table column names. Can be "0" or "1".
- POLARS_FMT_TABLE_HIDE_COLUMN_SEPARATOR ("0"): Hide the "---" separator between the column names and column types. Can be "0" or "1".
- POLARS_FMT_TABLE_HIDE_DATAFRAME_SHAPE_INFORMATION ("0"): Hide the DataFrame shape information when displaying tables. Can be "0" or "1".
- POLARS_FMT_TABLE_INLINE_COLUMN_DATA_TYPE ("0"): Moves the data type inline with the column name (to the right, in parentheses). Can be "0" or "1".
- POLARS_FMT_TABLE_ROUNDED_CORNERS ("0"): Apply rounded corners to UTF8-styled tables (only applies to UTF8 formats).
- POLARS_MAX_THREADS (<variable>): Maximum number of threads used to initialize the thread pool. The thread pool is locked once polars is loaded, so this envvar must be set before loading the package.
- POLARS_TABLE_WIDTH (<variable>): Set the maximum width of a table in characters.
- POLARS_VERBOSE ("0"): Enable additional verbose/debug logging.
- POLARS_WARN_UNSTABLE ("0"): Issue a warning when unstable functionality is used. Enabling this setting may help avoid functionality that is still evolving, potentially reducing maintenance burden from API changes and bugs. Can be "0" or "1".

The following configuration options are present in the Python API but currently cannot be changed in R: decimal separator, thousands separator, float precision, float formatting, trimming decimal zeros.

Value

polars_envvars() returns a named list where the names are the names of environment variables and values are their values.

Examples

```
polars_envvars()

# By default, long string values are truncated.
df <- pl$DataFrame(x = "This is a very very very long sentence.")
df

# We can temporarily allow for wider columns with `withr::with_envvar()`:
withr::with_envvar(
  list(POLARS_FMT_STR_LEN = "50"),
  print(df)
)

# Or we could set it permanently with `Sys.setenv(POLARS_FMT_STR_LEN = "50")`.
```

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 `<DataFrame>$select()`
df <- pl$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
)
```

polars_info — *Report information of the package*

Description

[**Experimental**] This function reports the following information:

- Package versions (the Polars R package version and the dependent Rust Polars crate version)
- Compatibility information of Polars.
- Number of threads used by Polars
- Rust feature flags (See `vignette("install", "polars")` for details)

Usage

```
polars_info()
```

Value

A list with information of the package

Examples

```
polars_info()

polars_info()$versions

polars_info()$features$nightly
```

---

polars_options *Get and reset polars options*

Description

[**Experimental**] `polars_options()` returns a list of options for polars. Options can be set with `options()`. Note that **options must be prefixed with "polars."**, e.g to modify the option `to_r_vector.int64` you need to pass `options(polars.to_r_vector.int64 =)`. See below for a description of all options.

`polars_options_reset()` brings all polars options back to their default value.

Usage

```
polars_options()

polars_options_reset()
```

Details

The following options are available (in alphabetical order, with the default value in parenthesis):

- `compat_level` will affect `as_nanoarrow_array_stream(<series>)`'s `polars_compat_level` argument and `<lazyframe>$sink_ipc()`'s `compat_level` argument. See the documentation of those functions for details.
- for all `to_r_vector.*` options, see arguments of to_r_vector().
- `df_knitr_print` (TODO: possible values??)

Value

`polars_options()` returns a named list where the names are option names and values are option values.

`polars_options_reset()` doesn't return anything.

Examples

```
library(hms)
polars_options()
withr::with_options(
  list(polars.to_r_vector.int64 = "character"),
  polars_options()
)
```

`QueryOptFlags` *The set of the optimizations considered during query optimization*

Description

[**Experimental**]

Usage

```
pl__QueryOptFlags(
  ...,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  comm_subplan_elim = TRUE,
  comm_subexpr_elim = TRUE,
  cluster_with_columns = TRUE,
  check_order_observe = TRUE,
  fast_projection = TRUE,
  join_order = TRUE,
  row_estimate = TRUE
)
```

Arguments

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

`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 trying to cache branching subplans that occur on self-joins or unions.

`comm_subexpr_elim`
: A logical, indicates trying to cache common subexpressions.

`cluster_with_columns`
: A logical, indicates to combine sequential independent calls to with_columns.

`check_order_observe`
: A logical, indicates not to maintain order if the order would not be observed.

`fast_projection`
: A logical, indicates to replace simple projections with a faster inlined projection that skips the expression engine.

`join_order` A logical, indicates whether joins may be reordered.

`row_estimate` A logical, indicates whether row estimates are used in cost planning.

Value

A `QueryOptFlags` object.

Examples

```
opt_flags <- pl$QueryOptFlags()
opt_flags
```

`s3-arithmetic` *Arithmetic operators for Polars objects*

Description

Arithmetic operators for Polars objects

Usage

```
## S3 method for class 'polars_expr'
e1 + e2

## S3 method for class 'polars_expr'
e1 - e2

## S3 method for class 'polars_expr'
e1 * e2

## S3 method for class 'polars_expr'
e1 / e2

## S3 method for class 'polars_expr'
```

```
e1 %% e2

## S3 method for class 'polars_expr'
e1 %/% e2

## S3 method for class 'polars_expr'
e1 ^ e2

## S3 method for class 'polars_expr'
e1 < e2
```

Arguments

e1, e2	Polars objects of numeric type or objects that can be coerced to a polars object of numeric type. Only + can work with two string inputs.

Value

A Polars object the same type as the input.

See Also

- <Expr>$add()
- <Expr>$sub()
- <Expr>$mul()
- <Expr>$true_div()
- <Expr>$pow()
- <Expr>$mod()
- <Expr>$floor_div()

Examples

```
pl$lit(5) + 10
5 + pl$lit(10)
pl$lit(5) + pl$lit(10)
+pl$lit(1)

# This will not raise an error as it is not actually evaluated.
expr <- pl$lit(5) + "10"
expr

# Will raise an error as it is evaluated.
tryCatch(
  pl$select(expr),
  error = function(e) e
)

# `+` accepts two string inputs
pl$select(pl$lit("a") + "b")
```

```
as_polars_series(5) + 10
+as_polars_series(5)
-as_polars_series(5)
```

---

series__alias *Rename the Series*

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__chunk_lengths *Get the length of each individual chunk*

Description

Get the length of each individual chunk

Usage

```
series__chunk_lengths()
```

Value

A numeric vector

Examples

```
s <- pl$Series("a", c(1, 2, 3))
s$chunk_lengths()

s2 <- pl$Series("a", c(4, 5, 6))

# Concatenate Series with rechunk = TRUE
pl$concat(s, s2, rechunk = TRUE)$chunk_lengths()

# Concatenate Series with rechunk = FALSE
pl$concat(s, s2, rechunk = FALSE)$chunk_lengths()
```

---

series__is_empty *Check if the Series is empty*

Description

Check if the Series is empty

Usage

```
series__is_empty()
```

Value

`TRUE` or `FALSE`

Examples

```
s <- pl$Series("a", integer())
s$is_empty()
```

---

series__n_chunks *Get the number of chunks that this Series contains*

Description

Get the number of chunks that this Series contains

Usage

```
series__n_chunks()
```

Value

An integer value

Examples

```
s <- pl$Series("a", c(1, 2, 3))
s$n_chunks()

s2 <- pl$Series("a", c(4, 5, 6))

# Concatenate Series with rechunk = TRUE
pl$concat(s, s2, rechunk = TRUE)$n_chunks()

# Concatenate Series with rechunk = FALSE
pl$concat(s, s2, rechunk = FALSE)$n_chunks()
```

---

series__rechunk *Create a single chunk of memory for this Series*

Description

Create a single chunk of memory for this Series

Usage

```
series__rechunk(..., in_place = FALSE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`in_place`	Bool to indicate if the operation should be done in place.

Value

A polars Series

Examples

```
s <- pl$Series("a", c(1, 2, 3))
s$n_chunks()

s2 <- pl$Series("a", c(4, 5, 6))
s <- pl$concat(s, s2, rechunk = FALSE)
s$n_chunks()

s$rechunk()$n_chunks()
```

series__serialize *Serialize and deserialize a Series*

Description

Serialize and deserialize a Series

Usage

```
series__serialize()

pl__deserialize_series(data)
```

Arguments

data	A raw vector of serialized Series.

Details

Similar to `polars.Series.__getstate__()` and `polars.Series.__setstate__()` in Python Polars.

Value

- `<Series>$serialize()` returns a raw vector of serialized Series.
- `pl$deserialize_series()` returns a deserialized Series.

Examples

```
serialized <- as_polars_series(1:3)$serialize()
serialized

pl$deserialize_series(serialized)
```

series__shrink_dtype *Shrink numeric values to the minimal required datatype*

Description

Shrink numeric values to the minimal required datatype

Usage

```
series__shrink_dtype()
```

Value

A polars Series

Examples

```
s <- as_polars_series(1:6)
s

s$shrink_dtype()
```

---

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.

Usage

```
series__to_r_vector(
  ...,
  uint8 = c("integer", "raw"),
  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.

`uint8` [**Experimental**] Determine how to convert Polars' UInt8 type values to R type. One of the followings:

- `"integer"` (default): Convert to the R's integer type.
- `"raw"`: Convert to the R's raw type. If the value is `null`, export as `00`.

`int64` [**Experimental**] 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` [**Experimental**] 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` [**Experimental**] Determine how to convert Polars' Time type values to R class. One of the followings:

- `"hms"` (default): Convert to the hms::hms class. If the hms package is not installed, a warning will be shown.
- `"ITime"`: Convert to the data.table::ITime class. The data.table package must be installed.

`struct` [**Experimental**] 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.

`decimal` [**Experimental**] 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`
[**Experimental**] 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 datetime 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` [**Experimental**] 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` [**Experimental**] 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

The class/type of the exported object depends on the DataType of the Series as follows:

- Boolean: logical.
- UInt8: integer or raw, depending on the `uint8` argument.

- 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.
- Map: vctrs::list_of of data.frame objects with typed `key` and `value` columns.
- Struct: data.frame or tibble, depending on the `struct` argument.

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 normal R data frame
series_struct$to_r_vector()

## Export Struct as tibble data frame
series_struct$to_r_vector(struct = "tibble")

# UInt8 values handling
series_uint8 <- as_polars_series(c(NA, 0, 255))$cast(pl$UInt8)
series_uint8

## Export UInt8 as integer
```

```
series_uint8$to_r_vector(uint8 = "integer")

## Export UInt8 as raw (`null` is exported as `00`)
series_uint8$to_r_vector(uint8 = "raw")

# Other Integer values handlings
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)
}
```

---

series_list_to_struct

Convert the series of type List to a series of type Struct

Description

Convert the series of type List to a series of type Struct

Usage

```
series_list_to_struct(fields)
```

Arguments

fields	A character vector of field names.

Value

A polars Series

See Also

- <expr>$list$to_struct()

Examples

```
# Convert list to struct with explicit field names:
s1 <- as_polars_series(list(0:2, 0:1))
s2 <- s1$list$to_struct(c("one", "two", "three"))
s2
s2$struct$fields

# Convert list to struct with field name assignment by
# index from a list of names:
s1$list$to_struct(fields = c("one", "two", "three"))$struct$unnest()
```

series_str_json_decode

Parse string values as JSON

Description

Throws an error if invalid JSON strings are encountered.

Usage

```
series_str_json_decode(dtype = NULL, ..., infer_schema_length = 100L)
```

Arguments

`dtype`	The dtype to cast the extracted value to, or `NULL` (default). If `NULL`, the dtype will be inferred from the JSON value.
`...`	These dots are for future extensions and must be empty.
`infer_schema_length`	The maximum number of rows to scan for schema inference. If set to `NULL`, the full data may be scanned *(this is slow)*. Only used if the `dtype` argument is `NULL`.

Value

A polars Series

See Also

- `<expr>$str$json_decode()`

Examples

```
s1 <- as_polars_series(c('{"a":1, "b": true}', NA, '{"a":2, "b": false}'))

s2 <- s1$str$json_decode()
s2
s2$dtype

s3 <- s1$str$json_decode(pl$Struct(a = pl$UInt8, b = pl$Boolean))
s3
s3$dtype
```

series_str_strptime *Convert a String Series into a Date/Datetime/Time Series*

Description

Similar to the `strptime()` function.

Usage

```
series_str_strptime(
  dtype,
  format = NULL,
  ...,
  strict = TRUE,
  exact = TRUE,
  cache = TRUE,
  ambiguous = c("raise", "earliest", "latest", "null")
)
```

Arguments

`dtype`	The data type to convert into. Can be either `pl$Date`, `pl$Datetime`, or `pl$Time`.
`format`	Format to use for conversion. Refer to the chrono crate documentation 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.
`...`	These dots are for future extensions and must be empty.
`strict`	If `TRUE` (default), raise an error if a single string cannot be parsed. If `FALSE`, produce a polars `null`.
`exact`	If `TRUE` (default), require an exact format match. If `FALSE`, allow the format to match anywhere in the target string. Note that using `exact = FALSE` introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.
`cache`	Use a cache of unique, converted dates to apply the datetime conversion.
`ambiguous`	Determine how to deal with ambiguous datetimes. Character vector or Series containing the followings: • `"raise"` (default): Throw an error • `"earliest"`: Use the earliest datetime • `"latest"`: Use the latest datetime • `"null"`: Return a null value

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 Series

See Also

- `<expr>$str$strptime()`
- `<series>$str$to_datetime()`

Examples

```
s1 <- as_polars_series(c("2020-01-01 01:00Z", "2020-01-01 02:00Z"))

s1$str$strptime(pl$Datetime(), "%Y-%m-%d %H:%M%#z")

# Auto infer format
s1$str$strptime(pl$Datetime())

# Datetime with timezone is interpreted as UTC timezone
s2 <- as_polars_series(c("2020-01-01T01:00:00+09:00"))
s2$str$strptime(pl$Datetime())

# Dealing with different formats.
s3 <- as_polars_series(
  c(
    "2021-04-22",
    "2022-01-04 00:00:00",
    "01/31/22",
    "Sun Jul  8 00:34:60 2001"
  )
)

pl$select(pl$coalesce(
  s3$str$strptime(pl$Date, "%F", strict = FALSE),
  s3$str$strptime(pl$Date, "%F %T", strict = FALSE),
  s3$str$strptime(pl$Date, "%D", strict = FALSE),
  s3$str$strptime(pl$Date, "%c", strict = FALSE),
))$to_series()
```

---

series_str_to_datetime

Convert a String Series into a Datetime Series

Description

Convert a String Series into a Datetime Series

Usage

```
series_str_to_datetime(
  format = NULL,
  ...,
  time_unit = NULL,
  time_zone = NULL,
  strict = TRUE,
  exact = TRUE,
  cache = TRUE,
  ambiguous = c("raise", "earliest", "latest", "null")
)
```

Arguments

`format`	Format to use for conversion. Refer to the chrono crate documentation 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.
`...`	These dots are for future extensions and must be empty.
`time_unit`	Unit of time for the resulting Datetime column. If `NULL` (default), the time unit is inferred from the format string if given, e.g.: `"%F %T%.3f"` => `pl$Datetime("ms")`. If no fractional second component is found, the default is `"us"` (microsecond).
`time_zone`	for the resulting Datetime column.
`strict`	If `TRUE` (default), raise an error if a single string cannot be parsed. If `FALSE`, produce a polars `null`.
`exact`	If `TRUE` (default), require an exact format match. If `FALSE`, allow the format to match anywhere in the target string. Note that using `exact = FALSE` introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.
`cache`	Use a cache of unique, converted dates to apply the datetime conversion.
`ambiguous`	Determine how to deal with ambiguous datetimes. Character vector or Series containing the followings: • `"raise"` (default): Throw an error • `"earliest"`: Use the earliest datetime • `"latest"`: Use the latest datetime • `"null"`: Return a null value

Value

A polars Series

See Also

- <expr>strto_datetime()

Examples

```
s <- as_polars_series(c("2020-01-01 01:00Z", "2020-01-01 02:00Z"))
s$str$to_datetime("%Y-%m-%d %H:%M%#z")
s$str$to_datetime(time_unit = "ms")
```

---

series_str_to_decimal

Convert a String Series into a Decimal Series

Description

[**Experimental**]

Usage

```
series_str_to_decimal(..., scale = NULL, inference_length = 100L)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`scale`	Number of digits after the comma to use for the decimals, or `NULL` (default). If `NULL`, the method will infer the scale from the data.
`inference_length`	Number of elements to parse to determine the `precision` and `scale` of the decimal data type.

Value

A polars Series

See Also

- <expr>strto_decimal()

Examples

```
s <- as_polars_series(c(
  "40.12",
  "3420.13",
  "120134.19",
  "3212.98",
  "12.90",
  "143.09",
  "143.9"
```

```
))

s$str$to_decimal()
s$str$to_decimal(scale = 4)
```

`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()
```

`sql_context__execute` *Parse the given SQL query and execute it against the registered frame data*

Description

[**Experimental**]

Usage

```
sql_context__execute(query)
```

Arguments

`query`	A valid string SQL query.

Value

A polars LazyFrame

Examples

```
# Declare frame data and register with a SQLContext:
df <- pl$DataFrame(
  title = c(
    "The Godfather",
    "The Dark Knight",
    "Schindler's List",
    "Pulp Fiction",
    "The Shawshank Redemption"
  ),
  release_year = c(1972, 2008, 1993, 1994, 1994),
  budget = c(6 * 1e6, 185 * 1e6, 22 * 1e6, 8 * 1e6, 25 * 1e6),
  gross = c(134821952, 533316061, 96067179, 107930000, 28341469),
  imdb_score = c(9.2, 9, 8.9, 8.9, 9.3)
)

ctx <- pl$SQLContext(films = df)
ctx$execute(
  "
     SELECT title, release_year, imdb_score
     FROM films
     WHERE release_year > 1990
     ORDER BY imdb_score DESC
     "
)$collect()

# Execute a GROUP BY query:
ctx$execute(
  "
  SELECT
       MAX(release_year / 10) * 10 AS decade,
       SUM(gross) AS total_gross,
       COUNT(title) AS n_films,
  FROM films
  GROUP BY (release_year / 10) -- decade
  ORDER BY total_gross DESC
  "
)$collect()
```

---

sql_context__register

Register a single frame as a table, using the given name

Description

[**Experimental**]

Usage

```
sql_context__register(name, frame = NULL)
```

Arguments

`name`	Name of the table.
`frame`	Object to associate with this table name.

Value

An object of class `"polars_sql_context"`

Examples

```
df <- pl$DataFrame(x = 1)
ctx <- pl$SQLContext()
ctx$register("frame_data", df)$execute("SELECT * FROM frame_data")$collect()
```

---

sql_context__register_many

Register multiple eager/lazy frames as tables, using the associated names

---

Description

[**Experimental**]

Usage

```
sql_context__register_many(...)
```

Arguments

`...`	<dynamic-dots> Elements that are known in the current `SQLContext`. It accepts any R object that can be converted to a LazyFrame via `as_polars_lf()`. All elements must be named.

Value

An object of class `"polars_sql_context"`

Examples

```
df <- pl$DataFrame(x = 1)
df2 <- pl$DataFrame(x = 2)
df3 <- pl$DataFrame(x = 3)

ctx <- pl$SQLContext()
ctx$register_many(tab1 = df, tab2 = df2, tab3 = df3)
```

sql_context__tables *Return a list of the registered table names*

Description

[**Experimental**]

Usage

```
sql_context__tables()
```

Details

This method will return the same values as the "SHOW TABLES" SQL statement, but as a vector instead of a frame.

Value

A character vector

Examples

```
# Executing as SQL:
frame_data <- pl$DataFrame(x = 1)
ctx <- pl$SQLContext(hello_world=frame_data, foo = data.frame(x = 2))
ctx$execute("SHOW TABLES")$collect()

# Calling the method:
ctx$tables()
```

sql_context__unregister *Unregister one or more frames by name*

Description

[**Experimental**]

Usage

```
sql_context__unregister(names)
```

Arguments

names	Names of the tables to unregister.

Value

An object of class "polars_sql_context"

Examples

```
df <- pl$DataFrame(ints = 9:5)
lf1 <- pl$LazyFrame(text = letters[1:3])
lf2 <- pl$LazyFrame(misc = "testing1234")

# Register with a SQLContext object:
ctx <- pl$SQLContext(test1 = df, test2 = lf1, test3 = lf2)
ctx$tables()

# Unregister one or more of the tables:
ctx$unregister(c("test1", "test3"))$tables()
ctx$unregister("test2")$tables()
```

Index

*.polars_expr *(s3-arithmetic)*, 626
+.polars_expr *(s3-arithmetic)*, 626
-.polars_expr *(s3-arithmetic)*, 626
/.polars_expr *(s3-arithmetic)*, 626
<.polars_expr *(s3-arithmetic)*, 626
<DataFrame>$get_columns(), *22*
<DataFrame>$group_by_dynamic(), *117*
<DataFrame>$partition_by(), *89*
<DataFrame>$rolling(), *92*
<DataFrame>$to_struct(), *33*
<Expr>$add(), *627*
<Expr>$floor_div(), *228*, *301*, *627*
<Expr>$mod(), *200*, *627*
<Expr>$mul(), *627*
<Expr>$pow(), *627*
<Expr>strcontains(), *405*
<Expr>strreplace(), *423*
<Expr>strreplace_all(), *422*
<Expr>strstrptime(), *437*, *442*
<Expr>$sub(), *627*
<Expr>$true_div(), *200*, *627*
<LazyFrame>$collect(), *26*, *570*
<LazyFrame>$group_by_dynamic(), *499*
<LazyFrame>$head(), *519*
<LazyFrame>$rolling(), *479*
<Series>$alias(), *628*
<Series>$rename(), *628*
<Series>$struct$unnest(), *25, 26*
<Series>$to_frame(), *26*
<Series>$to_r_vector(), *21*, *34*, *607*
<dataframe>$with_columns(), *104*
<expr>arrto_struct(), *389*
<expr>catphysical(), *329*
<expr>$cat$to(), *328*
<expr>dtoffset_by(), *564*
<expr>dtoffset_by(1d), *564*
<expr>$list$to_struct(), *637*
<expr>strjson_decode(), *638*
<expr>strstrptime(), *439*, *640*
<expr>strto_date(), *435*
<expr>strto_datetime(), *435*, *642*
<expr>strto_decimal(), *642*
<expr>strto_time(), *435*
<lazyframe>$sink_ipc(), *624*
<series>$list$to_struct(), *389*
<series>strjson_decode(), *417*
<series>strto_datetime(), *435*, *439*, *640*
<series>strto_decimal(), *439*
arreval(), *306*
$bin_intervals(), *183*
$bin_quantiles(), *239*
$bin_ranks(), *239*
$cast(), *552*, *570*
$collect(), *507*, *510*, *535*
$drop_nans(), *186*
$drop_nulls(), *185*
dtbase_utc_offset(), *336*
dtconvert_time_zone(), *348*
dtdst_offset(), *331*
dtto_string(), *352*
$eq(), *188*
$eq_missing(), *187*
$get(), *218*
$head(), *221*
$list$eval(), *363*
$list$gather(), *373*
$list$get(), *372*
$ne_missing(), *231*
$prefix(), *402*
$replace(), *245*
$replace_strict(), *244*
$rolling(), *250*, *252*, *254*, *256*, *257*, *259*, *260*, *262*, *264*, *266*, *268*, *270*, *271*, *273*, *274*, *276*, *277*, *279*, *281*, *282*, *499*
$round(), *301*
$set_difference(), *382*

$set_symmetric_difference(), *381*
$sink_ipc(), *464*
$sink_parquet(), *464*
$sort(), *172, 173, 298, 299*
strcontains(), *414*
strends_with(), *404, 414*
strfind(), *404*
strstart_with(), *404, 414*
$strip_chars_end(), *433*
$strip_chars_start(), *432*
$suffix(), *400*
$to_frame(), *26*
$truncate(), *284*
$value_counts(), *302, 303*
%/%.polars_expr *(s3-arithmetic)*, 626
%%.polars_expr *(s3-arithmetic)*, 626
^.polars_expr *(s3-arithmetic)*, 626

abort(), *42*
abs(n), *94, 127*
Arithmetic operators, *154, 200, 228, 229, 238, 295, 301*
Array, *326*
array, *67*
as.data.frame(<polars_data_frame>), *26*
as.data.frame(<polars_object>), *38*
as.data.frame.polars_data_frame, 17
as.data.frame.polars_lazy_frame *(as.data.frame.polars_data_frame)*, 17
as.list(<polars_data_frame>), *26, 88*
as.list.polars_data_frame, 19
as.list.polars_lazy_frame *(as.list.polars_data_frame)*, 19
as_nanoarrow_array_stream(<series>), *624*
as_nanoarrow_array_stream.polars_data_frame, 22
as_nanoarrow_array_stream.polars_lazy_frame *(as_nanoarrow_array_stream.polars_data_frame)*, 22
as_nanoarrow_array_stream.polars_series *(as_nanoarrow_array_stream.polars_data_frame)*, 22
as_polars_df, 24
as_polars_df(), *17, 20, 24, 30, 33, 34, 37, 552, 632, 643*
as_polars_df(x, ...), *29*
as_polars_expr, 27
as_polars_expr(), *27, 119, 120, 136, 138, 253, 257, 260, 263, 267, 270, 275, 278, 282, 500, 501, 524, 526, 544, 554, 558, 565, 575, 608*
as_polars_lf, 29
as_polars_lf(), *29*
as_polars_series, 30, *104*
as_polars_series(), *24–26, 28, 30, 33, 129, 149, 224, 458, 459, 552, 570, 575, 606*
as_polars_series(<nanoarrow_array_stream>), *23*
as_tibble.polars_data_frame, 36
as_tibble.polars_lazy_frame *(as_tibble.polars_data_frame)*, 36

base::OlsonNames(), *334, 348*
Binary, *27, 28, 149*
bit64, *18, 20, 37, 607, 633*
bit64::integer64, *18, 20, 37, 633, 635*
bit64::NA_integer64_, *18, 20, 37, 633*
blob, *607*
blob::blob, *635*

Categorical, *328, 329*
character, *18, 20, 21, 28, 37, 38, 149, 633–635*
check_list_of_polars_dtype *(check_polars)*, 39
check_list_of_polars_lf *(check_polars)*, 39
check_polars, 39, *459*
check_polars_df *(check_polars)*, 39
check_polars_dtype *(check_polars)*, 39
check_polars_dtype_expr *(check_polars)*, 39
check_polars_expr *(check_polars)*, 39
check_polars_lf *(check_polars)*, 39
check_polars_partitioning_scheme *(check_polars)*, 39
check_polars_selector *(check_polars)*, 39
check_polars_series *(check_polars)*, 39
cli, *607*
clock, *18, 21, 38, 607, 634*

clock_duration, *18*, *21*, *33*, *38*, *634*, *635*
clock_naive_time, *18*, *21*, *38*, *634*, *635*
clock_zoned_time, *18*, *21*, *38*, *634*, *635*
cs, 42, *44*, *46–74*
cs$all(), *80*, *81*, *128*, *132*, *468*, *469*, *521*
cs$array(), *65*, *67*, *71*
cs$by_dtype(), *47*, *60*, *65*, *67*, *71*
cs$by_index(), *579*
cs$categorical(), *60*
cs$empty(), *134*, *523*
cs$list(), *47*, *67*, *71*
cs$nested(), *47*, *65*, *71*
cs$struct(), *67*
cs__all, 43
cs__alpha, 44
cs__alphanumeric, 45
cs__array, 46
cs__binary, 47
cs__boolean, 48
cs__by_dtype, 49
cs__by_index, 50
cs__by_name, 51
cs__categorical, 52
cs__contains, 52
cs__date, 53
cs__datetime, 54
cs__decimal, 55
cs__digit, 56
cs__duration, 57
cs__empty, 58
cs__ends_with, 59
cs__enum, 60
cs__exclude, 61
cs__first, 62
cs__float, 63
cs__integer, 63
cs__last, 64
cs__list, 65
cs__matches, 66
cs__nested, 67
cs__numeric, 68
cs__signed_integer, 68
cs__starts_with, 69
cs__string, 70
cs__struct, 71
cs__temporal, 72
cs__time, 73
cs__unsigned_integer, 73
data type, *33*, *352*, *354*, *539*, *541*, *545*, *551*, *576*, *578*, *609*
data types, *46*, *65*
data.frame, *21*, *26*, *33*, *634*, *635*
data.table, *18*, *21*, *38*, *607*, *634*
data.table::IDate, *18*, *20*, *38*, *633*, *635*
data.table::ITime, *18*, *21*, *38*, *634*, *635*
DataFrame, *25*, *26*, *30*, *36*, *75–81*, *83–86*, *94*, *98*, *100*, *103–106*, *108*, *109*, *111*, *112*, *114–116*, *119–128*, *130–133*, *135–139*, *234*, *449–456*, *464*, *467*, *493*, *504*, *552*, *570*, *585–587*, *589*, *591*, *593*, *621*, *623*, *632*, *643*
DataFrame *(pl__DataFrame)*, 552
DataFrame$filter(), *198*
dataframe__bottom_k, 74
dataframe__cast, 75
dataframe__clear, 76
dataframe__clone, 77
dataframe__count, 78
dataframe__describe, 78
dataframe__drop, 79
dataframe__drop_nans, 80
dataframe__drop_nulls, 81
dataframe__equals, 82
dataframe__explode, 82
dataframe__fill_nan, 83
dataframe__fill_null, 84
dataframe__filter, 85
dataframe__gather_every, 85
dataframe__get_column, 86
dataframe__get_column_index, 87
dataframe__get_columns, 87
dataframe__glimpse, 88
dataframe__group_by, 89
dataframe__group_by_dynamic, 90
dataframe__hash_rows, 93
dataframe__head, 94
dataframe__is_duplicated, 95
dataframe__is_empty, 95
dataframe__is_unique, 96
dataframe__join, 96
dataframe__join_asof, 99
dataframe__join_where, 102
dataframe__lazy, 104
dataframe__map_columns, 104
dataframe__max, 105

dataframe__max_horizontal, 106
dataframe__mean, 106
dataframe__mean_horizontal, 107
dataframe__median, 107
dataframe__merge_sorted, 108
dataframe__min, 109
dataframe__min_horizontal, 109
dataframe__n_chunks, 110
dataframe__partition_by, 110
dataframe__pivot, 111
dataframe__quantile, 113
dataframe__rechunk, 114
dataframe__remove, 114
dataframe__rename, 115
dataframe__reverse, 116
dataframe__rolling, 116
dataframe__sample, 118
dataframe__select, 119
dataframe__select_seq, 120
dataframe__serialize, 121
dataframe__set_sorted, 122
dataframe__shift, 122
dataframe__slice, 123
dataframe__sort, 124
dataframe__std, 125
dataframe__sum, 126
dataframe__sum_horizontal, 126
dataframe__tail, 127
dataframe__to_dummies, 127
dataframe__to_series, 128
dataframe__to_struct, 129
dataframe__top_k, 130
dataframe__transpose, 131
dataframe__unique, 132
dataframe__unnest, 133
dataframe__unpivot, 134
dataframe__unstack, 135
dataframe__var, 136
dataframe__with_columns, 136
dataframe__with_columns_seq, 137
dataframe__with_row_index, 138
dataframe__write_csv, 139
dataframe__write_ipc, 141
dataframe__write_ipc_stream, 143
dataframe__write_json, 144
dataframe__write_ndjson, 145
dataframe__write_parquet, 146
DataFrames, *547*
DataType, *34*, *224*, *396*, *398*, *457*, *634*
DataType *(polars_dtype)*, 618
datatype__to_dtype_expr, 148
datatype_expr__default_value, 148
datatype_expr__display, 150
datatype_expr__inner_dtype, 151
datatype_expr__matches, 151
datatype_expr__wrap_in_array, 152
datatype_expr__wrap_in_list, 153
datatypes, *590*, *592*, *602*, *604*
Date, *18*, *20*, *33*, *38*, *330*, *633*, *635*
Date/Time/Datetime, *352*, *354*
Datetime, *438*, *555*, *556*, *560*, *562*, *641*
decimal data type, *642*
difftime, *18*, *21*, *33*, *38*, *555*, *556*, *560*, *562*, *611*, *613*, *634*, *635*
double, *18*, *20*, *21*, *37*, *38*, *633–635*
dtype, *33*, *440*, *638*
Duration, *352*, *354*, *564*

Enum, *328*, *329*
environment, *616*
environment class, *42*, *538*
Expr, *28*, *104*, *149*, *404*, *406*, *413*, *422*, *423*, *457*
Expr *(polars_expr)*, 623
Expr$struct$field(*), *447*
expr__abs, 153
expr__add, 154
expr__alias, 155
expr__all, 155
expr__and, 156
expr__any, 157
expr__append, 158
expr__approx_n_unique, 158
expr__arccos, 159
expr__arccosh, 159
expr__arcsin, 160
expr__arcsinh, 160
expr__arctan, 161
expr__arctanh, 161
expr__arg_max, 162
expr__arg_min, 162
expr__arg_sort, 163
expr__arg_true, 163
expr__arg_unique, 164
expr__backward_fill, 164
expr__bin_intervals, 165
expr__bin_quantiles, 166

expr__bin_ranks, 167
expr__bitwise_and, 168
expr__bitwise_count_ones, 168
expr__bitwise_count_zeros, 169
expr__bitwise_leading_ones, 169
expr__bitwise_leading_zeros, 170
expr__bitwise_or, 170
expr__bitwise_trailing_ones, 171
expr__bitwise_trailing_zeros, 171
expr__bitwise_xor, 172
expr__bottom_k, 172
expr__bottom_k_by, 173
expr__cast, 174
expr__cbrt, 175
expr__ceil, 175
expr__clip, 176
expr__cos, 177
expr__cosh, 177
expr__cot, 178
expr__count, 178
expr__cum_count, 179
expr__cum_max, 179
expr__cum_min, 180
expr__cum_prod, 181
expr__cum_sum, 181
expr__cumulative_eval, 182
expr__cut, 183
expr__degrees, 184
expr__diff, 184
expr__dot, 185
expr__drop_nans, 185
expr__drop_nulls, 186
expr__entropy, 186
expr__eq, 187, *188*
expr__eq_missing, *187*, 188
expr__ewm_mean, 188
expr__ewm_mean_by, 190
expr__ewm_std, 191
expr__ewm_var, 193
expr__exclude, 194
expr__exp, 195
expr__explode, 195
expr__extend_constant, 196
expr__fill_nan, 197
expr__fill_null, 197
expr__filter, 198
expr__first, 199
expr__floor, 199
expr__floor_div, 200
expr__floordiv *(expr__floor_div)*, 200
expr__forward_fill, 200
expr__gather, 201
expr__gather_every, 202
expr__ge, 202
expr__get, 203
expr__gt, 204
expr__has_nulls, 204
expr__hash, 205
expr__head, 206
expr__hist, 206
expr__implode, 207
expr__index_of, 208
expr__interpolate, 208
expr__interpolate_by, 209
expr__is_between, 210
expr__is_close, 211
expr__is_duplicated, 212
expr__is_finite, 212
expr__is_first_distinct, 213
expr__is_in, 213
expr__is_infinite, 214
expr__is_last_distinct, 214
expr__is_nan, 215
expr__is_not_nan, 215
expr__is_not_null, 216
expr__is_null, 216
expr__is_unique, 217
expr__item, 217
expr__kurtosis, 218
expr__last, 219
expr__le, 220
expr__len, 220
expr__limit, 221
expr__log, 221
expr__log10, 222
expr__log1p, 222
expr__lower_bound, 223
expr__lt, 223
expr__map_batches, 224
expr__max, 225
expr__max_by, 225
expr__mean, 226
expr__median, 226
expr__min, 227
expr__min_by, 227
expr__mod, 228

expr__mode, 228
expr__mul, 229
expr__n_unique, 230
expr__nan_max, 230
expr__nan_min, 231
expr__ne, 231, *232*
expr__ne_missing, *231*, 232
expr__not, 233
expr__null_count, 233
expr__or, 234
expr__over, 234
expr__pct_change, 236
expr__peak_max, 237
expr__peak_min, 237
expr__pow, 238
expr__product, 238
expr__qcut, 239
expr__quantile, 240
expr__radians, 241
expr__rank, 241
expr__reinterpret, 243
expr__repeat_by, 243
expr__replace, 244
expr__replace_strict, 245
expr__reshape, 247
expr__reverse, 248
expr__rle, 248
expr__rle_id, 249
expr__rolling, 249
expr__rolling_kurtosis, 251
expr__rolling_max, 252
expr__rolling_max_by, 253
expr__rolling_mean, 255
expr__rolling_mean_by, 256
expr__rolling_median, 258
expr__rolling_median_by, 259
expr__rolling_min, 261
expr__rolling_min_by, 263
expr__rolling_quantile, 265
expr__rolling_quantile_by, 266
expr__rolling_rank, 269
expr__rolling_rank_by, 270
expr__rolling_skew, 272
expr__rolling_std, 273
expr__rolling_std_by, 274
expr__rolling_sum, 277
expr__rolling_sum_by, 278
expr__rolling_var, 280
expr__rolling_var_by, 281
expr__round, 283
expr__round_sig_figs, 284
expr__sample, 285
expr__search_sorted, 286
expr__set_sorted, 287
expr__shift, 287
expr__shuffle, 288
expr__sign, 289
expr__sin, 289
expr__sinh, 290
expr__skew, 290
expr__slice, 291
expr__sort, 292
expr__sort_by, 293
expr__sqrt, 294
expr__std, 295
expr__sub, 295
expr__sum, 296
expr__tail, 296
expr__tan, 297
expr__tanh, 297
expr__to_physical, 298
expr__top_k, 298
expr__top_k_by, 299
expr__true_div, 300
expr__truediv *(expr__true_div)*, 300
expr__truncate, 301
expr__unique, 302
expr__unique_counts, 303
expr__upper_bound, 303
expr__value_counts, 304
expr__var, 305
expr__xor, 305
expr_arr_agg, 306
expr_arr_all, 307
expr_arr_any, 307
expr_arr_arg_max, 308
expr_arr_arg_min, 308
expr_arr_contains, 309
expr_arr_count_matches, 310
expr_arr_dot, 310
expr_arr_eval, 311
expr_arr_explode, 312
expr_arr_first, 312
expr_arr_get, 313
expr_arr_join, 314
expr_arr_last, 314

expr_arr_len, 315
expr_arr_max, 315
expr_arr_median, 316
expr_arr_min, 316
expr_arr_n_unique, 317
expr_arr_reverse, 317
expr_arr_shift, 318
expr_arr_sort, 318
expr_arr_std, 319
expr_arr_sum, 319
expr_arr_to_list, 320
expr_arr_to_struct, 320
expr_arr_unique, 321
expr_arr_var, 322
expr_bin_contains, 322
expr_bin_decode, 323
expr_bin_encode, 324
expr_bin_ends_with, 324
expr_bin_get, 325
expr_bin_reinterpret, 326
expr_bin_size, 327
expr_bin_starts_with, 327
expr_cat_physical, 328
expr_cat_to, 329
expr_dt_add_business_days, 330
expr_dt_base_utc_offset, 331
expr_dt_cast_time_unit, 332
expr_dt_century, 332
expr_dt_combine, 333
expr_dt_convert_time_zone, 334
expr_dt_date, 335
expr_dt_day, 335
expr_dt_days_in_month, 336
expr_dt_dst_offset, 336
expr_dt_epoch, 337
expr_dt_hour, 338
expr_dt_is_leap_year, 338
expr_dt_iso_year, 339
expr_dt_microsecond, 339
expr_dt_millennium, 340
expr_dt_millisecond, 341
expr_dt_minute, 341
expr_dt_month, 342
expr_dt_month_end, 343
expr_dt_month_start, 343
expr_dt_nanosecond, 344
expr_dt_offset_by, 344
expr_dt_ordinal_day, 346
expr_dt_quarter, 346
expr_dt_replace, 347
expr_dt_replace_time_zone, 348
expr_dt_round, 349
expr_dt_second, 351
expr_dt_strftime, 352
expr_dt_time, 352
expr_dt_timestamp, 353
expr_dt_to_string, 354
expr_dt_total_days, 355
expr_dt_total_hours, 356
expr_dt_total_microseconds, 356
expr_dt_total_milliseconds, 357
expr_dt_total_minutes, 358
expr_dt_total_nanoseconds, 359
expr_dt_total_seconds, 359
expr_dt_truncate, 360
expr_dt_week, 361
expr_dt_weekday, 362
expr_dt_year, 363
expr_list_agg, 363
expr_list_all, 364
expr_list_any, 365
expr_list_arg_max, 365
expr_list_arg_min, 366
expr_list_concat, 366
expr_list_contains, 367
expr_list_count_matches, 368
expr_list_diff, 368
expr_list_drop_nulls, 369
expr_list_eval, 369
expr_list_explode, 370
expr_list_first, 371
expr_list_gather, 372
expr_list_gather_every, 373
expr_list_get, 373
expr_list_head, 374
expr_list_join, 375
expr_list_last, 376
expr_list_len, 376
expr_list_max, 377
expr_list_mean, 377
expr_list_median, 378
expr_list_min, 378
expr_list_n_unique, 379
expr_list_reverse, 379
expr_list_sample, 380
expr_list_set_difference, 381

expr_list_set_intersection, 382
expr_list_set_symmetric_difference, 382
expr_list_set_union, 383
expr_list_shift, 384
expr_list_slice, 385
expr_list_sort, 385
expr_list_std, 386
expr_list_sum, 387
expr_list_tail, 387
expr_list_to_array, 388
expr_list_to_struct, 389
expr_list_unique, 390
expr_list_var, 390
expr_meta_eq, 391
expr_meta_has_multiple_outputs, 391
expr_meta_is_column, 392
expr_meta_is_column_selection, 392
expr_meta_is_literal, 393
expr_meta_is_regex_projection, 394
expr_meta_ne, 394
expr_meta_output_name, 395
expr_meta_pop, 396
expr_meta_root_names, 396
expr_meta_serialize, 397
expr_meta_tree_format, 398
expr_meta_undo_aliases, 399
expr_name_keep, 399
expr_name_prefix, 400
expr_name_prefix_fields, 400
expr_name_replace, 401
expr_name_suffix, 402
expr_name_suffix_fields, 402
expr_name_to_lowercase, 403
expr_name_to_uppercase, 403
expr_str_contains, 404
expr_str_contains_any, 405
expr_str_count_matches, 406
expr_str_decode, 407
expr_str_encode, 407
expr_str_ends_with, 408
expr_str_escape_regex, 409
expr_str_extract, 409
expr_str_extract_all, 410
expr_str_extract_groups, 411
expr_str_extract_many, 412
expr_str_find, 413
expr_str_find_many, 414
expr_str_head, 415
expr_str_join, 416
expr_str_json_decode, 417
expr_str_json_path_match, 417
expr_str_len_bytes, 418
expr_str_len_chars, 419
expr_str_normalize, 420
expr_str_pad_end, 420
expr_str_pad_start, 421
expr_str_replace, 421
expr_str_replace_all, 423
expr_str_replace_many, 424
expr_str_reverse, 425
expr_str_slice, 426
expr_str_split, 427
expr_str_split_exact, 428
expr_str_splitn, 429
expr_str_starts_with, 429
expr_str_strip_chars, 430
expr_str_strip_chars_end, 431
expr_str_strip_chars_start, 431
expr_str_strip_prefix, 432
expr_str_strip_suffix, 433
expr_str_strptime, 434
expr_str_tail, 436
expr_str_to_date, 437
expr_str_to_datetime, 438
expr_str_to_decimal, 439
expr_str_to_integer, 440
expr_str_to_lowercase, 441
expr_str_to_time, 441
expr_str_to_titlecase, 442
expr_str_to_uppercase, 443
expr_str_zfill, 443
expr_struct_drop, 444
expr_struct_field, 445
expr_struct_json_encode, 446
expr_struct_rename_fields, 446
expr_struct_unnest, 447
expr_struct_with_fields, 448
expression, *18*, *21*, *27*, *28*, *38*, *112*, *149*, *150*, *153–188*, *190–192*, *194–235*, *237*, *238*, *240–244*, *246–252*, *254*, *256*, *257*, *259*, *261*, *262*, *264*, *266*, *268*, *270*, *271*, *273*, *274*, *276*, *277*, *279*, *281*, *282*, *284–299*, *301–397*, *399–413*, *415–423*, *425–448*, *494*, *539–545*, *548–551*, *554*, *555*, *557*,

559, 560, 562, 565–569, 571, 572, 574–580, 594, 608–611, 613, 614, 634
expression (polars_expr), 623
expressions, 27, 119, 120, 136, 138, 500, 501, 524, 526, 580, 608

factor, 635

groupby__agg, 449
groupby__having, 450
groupby__head, 450
groupby__len, 451
groupby__max, 452
groupby__mean, 452
groupby__median, 453
groupby__min, 453
groupby__n_unique, 454
groupby__quantile, 455
groupby__sum, 456
groupby__tail, 456

hms, 18, 20, 33, 38, 607, 634
hms::hms, 18, 20, 38, 634, 635

infer_polars_dtype, 457
infer_polars_dtype(), 25, 33, 34, 42, 457
integer, 18, 20, 37, 633–635
is_convertible_to_polars_expr (infer_polars_dtype), 457
is_convertible_to_polars_expr(), 457
is_convertible_to_polars_series (infer_polars_dtype), 457
is_convertible_to_polars_series(), 457
is_list_of_polars_dtype (check_polars), 39
is_list_of_polars_expr (check_polars), 39
is_list_of_polars_lf (check_polars), 39
is_polars (check_polars), 39
is_polars_df (check_polars), 39
is_polars_dtype (check_polars), 39
is_polars_dtype_expr (check_polars), 39
is_polars_expr (check_polars), 39
is_polars_lf (check_polars), 39
is_polars_partitioning_scheme (check_polars), 39
is_polars_selector (check_polars), 39
is_polars_series (check_polars), 39

knit_asis, 460
knit_print.polars_data_frame, 459
knit_print.polars_series (knit_print.polars_data_frame), 459

LazyFrame, 29, 30, 104, 134, 461–463, 466, 468, 469, 472–475, 480, 481, 483, 485, 488–492, 494, 496–498, 500–503, 505, 510, 512, 514–527, 529–535, 538, 570, 598, 599, 601, 603, 605, 644
LazyFrame (pl__LazyFrame), 570
LazyFrame$filter(), 198
lazyframe__bottom_k, 460
lazyframe__cast, 461
lazyframe__clear, 462
lazyframe__clone, 463
lazyframe__collect, 464
lazyframe__collect_schema, 465
lazyframe__count, 466
lazyframe__describe, 466
lazyframe__drop, 467
lazyframe__drop_nans, 468
lazyframe__drop_nulls, 469
lazyframe__explain, 470
lazyframe__explode, 471
lazyframe__fill_nan, 472
lazyframe__fill_null, 473
lazyframe__filter, 474
lazyframe__first, 475
lazyframe__gather_every, 475
lazyframe__group_by, 476
lazyframe__group_by_dynamic, 477
lazyframe__head, 480
lazyframe__interpolate, 481
lazyframe__join, 481
lazyframe__join_asof, 484
lazyframe__join_where, 487
lazyframe__last, 489
lazyframe__lazy_sink_batches (lazyframe__sink_batches), 504

lazyframe__lazy_sink_csv (*lazyframe__sink_csv*), 506
lazyframe__lazy_sink_ipc (*lazyframe__sink_ipc*), 510
lazyframe__lazy_sink_ndjson (*lazyframe__sink_ndjson*), 513
lazyframe__lazy_sink_parquet (*parquet_statistics*), 535
lazyframe__limit (*lazyframe__head*), 480
lazyframe__max, 489
lazyframe__mean, 490
lazyframe__median, 490
lazyframe__merge_sorted, 491
lazyframe__min, 492
lazyframe__null_count, 492
lazyframe__pivot, 493
lazyframe__quantile, 495
lazyframe__remove, 496
lazyframe__rename, 497
lazyframe__reverse, 498
lazyframe__rolling, 498
lazyframe__select, 500
lazyframe__select_seq, 501
lazyframe__serialize, 502
lazyframe__set_sorted, 502
lazyframe__shift, 503
lazyframe__sink_batches, 504
lazyframe__sink_csv, 506
lazyframe__sink_ipc, 510
lazyframe__sink_ndjson, 513
lazyframe__sink_parquet (*parquet_statistics*), 535
lazyframe__slice, 515
lazyframe__sort, 515
lazyframe__sql, 516
lazyframe__std, 517
lazyframe__sum, 518
lazyframe__tail, 519
lazyframe__to_dot, 519
lazyframe__top_k, 520
lazyframe__unique, 521
lazyframe__unnest, 522
lazyframe__unpivot, 523
lazyframe__var, 524
lazyframe__with_columns, 524
lazyframe__with_columns_seq, 525
lazyframe__with_row_index, 526
LazyFrames, *547*
lazygroupby__agg, 527
lazygroupby__having, 528
lazygroupby__head, 529
lazygroupby__len, 529
lazygroupby__max, 530
lazygroupby__mean, 530
lazygroupby__median, 531
lazygroupby__min, 532
lazygroupby__n_unique, 532
lazygroupby__quantile, 533
lazygroupby__sum, 534
lazygroupby__tail, 534
list, *21*, *25*, *32*, *33*, *67*, *458*
literals, *119*, *120*, *136*, *138*, *500*, *501*, *524*, *526*, *608*
logical, *634*

NA_integer_, *18*, *20*, *37*, *633*
nanoarrow array stream, *23*
nanoarrow schema object, *23*
nanoarrow::infer_nanoarrow_schema(), *457*

OlsonNames(), *54*
options(), *624*

parquet_statistics, 535
parquet_statistics(), *147*, *537*
PartitionBy (*pl__PartitionBy*), 580
pl, 538
pl$all(), *155*
pl$arg_sort_by(), *163*
pl$coalesce, *614*
pl$col(), *28*, *149*, *623*
pl$concat_list(), *574*
pl$Date, *434*, *639*
pl$date_range(), *557*
pl$date_ranges(), *555*
pl$Datetime, *434*, *639*
pl$Datetime(), *54*
pl$Datetime(ms), *435*, *438*, *640*, *641*
pl$datetime_range(), *563*
pl$datetime_ranges(), *561*
pl$dtype_of(), *224*
pl$int_range(), *594*
pl$lit(), *27*, *614*, *623*
pl$PartitionBy(), *580*
pl$Time, *434*, *639*

pl__all, 539
pl__all_horizontal, 540
pl__any, 540
pl__any_horizontal, 541
pl__arg_sort_by, 542
pl__arg_where, 543
pl__Array *(polars_dtype)*, 618
pl__Categorical *(polars_dtype)*, 618
pl__coalesce, 544
pl__col, 544
pl__collect_all, 545
pl__concat, 546
pl__concat_arr, 548
pl__concat_list, 549
pl__concat_str, 550
pl__cum_sum, 551
pl__DataFrame, 552
pl__date, 553
pl__date_range, 554
pl__date_ranges, 556
pl__Datetime *(polars_dtype)*, 618
pl__datetime, 558
pl__datetime_range, 560
pl__datetime_ranges, 562
pl__Decimal *(polars_dtype)*, 618
pl__deserialize_df *(dataframe__serialize)*, 121
pl__deserialize_lf *(lazyframe__serialize)*, 502
pl__deserialize_series *(series__serialize)*, 631
pl__dtype_of, 564
pl__Duration *(polars_dtype)*, 618
pl__duration, 564
pl__element, 566
pl__Enum *(polars_dtype)*, 618
pl__explain_all, 566
pl__first, 567
pl__int_range, 568
pl__int_ranges, 569
pl__last, 569
pl__LazyFrame, 570
pl__len, 571
pl__linear_space, 572
pl__linear_spaces, 573
pl__List *(polars_dtype)*, 618
pl__list, 574
pl__lit, 575
pl__Map *(polars_dtype)*, 618
pl__max, 576
pl__max_horizontal, 576
pl__mean_horizontal, 577
pl__min, 578
pl__min_horizontal, 579
pl__nth, 579
pl__PartitionBy, 580
pl__QueryOptFlags *(QueryOptFlags)*, 625
pl__read_csv, 582
pl__read_ipc, 585
pl__read_ipc_stream, 587
pl__read_lines, 588
pl__read_ndjson, 589
pl__read_parquet, 591
pl__repeat_, 593
pl__row_index, 594
pl__scan_csv, 595
pl__scan_ipc, 598
pl__scan_lines, 600
pl__scan_ndjson, 601
pl__scan_parquet, 603
pl__Series, 606
pl__show_versions, 607
pl__SQLContext, 607
pl__Struct *(polars_dtype)*, 618
pl__struct, 608
pl__sum, 609
pl__sum_horizontal, 610
pl__thread_pool_size, 610
pl__time_range, 611
pl__time_ranges, 612
pl__when, 614
pl_api_register_series_namespace, 616
polars data frames, *39*
polars data types, *39*
Polars DataFrame, *24*
polars DataFrame, *24*, *32*, *606*
Polars DataType, *545*
polars DataType, *458*
polars expression, *148*, *151–153*, *330*, *554*, *558*, *564*, *565*
polars expressions, *39*
polars lazy frames, *39*
polars partitioning schemes, *39*
polars selectors, *39*
Polars Series, *552*

polars Series, 30, 33, 86, 94–96, 106, 107, 109, 126, 628, 630, 631, 637, 638, 640–642
polars series, 39
polars_code_completion_activate, 617
polars_code_completion_deactivate (polars_code_completion_activate), 617
polars_data_frame, 33
polars_data_frame (pl__DataFrame), 552
polars_dtype, 618
polars_envvars, 621
polars_expr, 623
polars_info, 623
polars_info(), 442, 611
polars_lazy_frame, 26, 33
polars_lazy_frame (pl__LazyFrame), 570
polars_options, 624
polars_options_reset (polars_options), 624
polars_partitioning_scheme (pl__PartitionBy), 580
polars_selector (cs), 42
polars_series, 26
polars_series (pl__Series), 606
polars_sql_context (pl__SQLContext), 607
POSIXct, 18, 19, 21, 33, 38, 634, 635
POSIXlt, 33
purrr::map_at, 104

QueryOptFlags, 25, 464, 471, 505, 510, 512, 514, 519, 537, 546, 567, 625

R data frame, 19
R Data Frames, 552
R vector, 20
R vectors, 552, 606
raw, 18, 20, 27, 28, 37, 149, 633, 634
rlang, 42
rlang::abort(), 41
rlang::as_function(), 37
rlang::set_names(), 19

s3-arithmetic, 626
Selector (cs), 42
selector, 43, 44, 46–74, 151
selector__as_expr (cs), 42
selectors, 79–81, 83, 111, 128, 132, 133, 468, 469, 472, 521, 522
Series, 20, 23–28, 33, 42, 87, 104, 105, 129, 149, 457, 547, 552, 570, 607, 621, 623, 631–634, 639, 641
Series (pl__Series), 606
series__alias, 628
series__chunk_lengths, 628
series__is_empty, 629
series__n_chunks, 629
series__rechunk, 630
series__rename (series__alias), 628
series__serialize, 631
series__shrink_dtype, 631
series__to_frame, 632
series__to_r_vector, 633
series_list_to_struct, 637
series_str_json_decode, 638
series_str_strptime, 639
series_str_to_datetime, 640
series_str_to_decimal, 642
series_struct_unnest, 643
sql_context__execute, 643
sql_context__register, 644
sql_context__register_many, 645
sql_context__tables, 646
sql_context__unregister, 646
SQLContext, 517
SQLContext (pl__SQLContext), 607
strptime(), 434, 639
struct, 67
struct dtype, 25
struct type, 24

the vctrs package, 458
tibble, 21, 38, 607, 634, 635
to_r_vector(), 624

UInt64, 93
UInt8, 27

vctrs, 607
vctrs::list_of, 635
vctrs::list_of(), 33
vctrs::unspecified, 635
vctrs::vec_as_names(), 37
vctrs_rcrd, 33
vector, 633, 635

vectors, *20*