

Package: neopolars (via r-universe)

October 21, 2024

Title R Bindings for the 'polars' Rust Library

Version 0.0.0.9000

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

License MIT + file LICENSE

Depends R (>= 4.2)

Imports rlang (>= 1.1.0)

Suggests bit64, blob, cli, clock, data.table, hms, patrick, testthat
(>= 3.0.0), tibble, vctrs, withr

Config/Needs/dev devtools, lifecycle

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

SystemRequirements Cargo (Rust's package manager), rustc

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

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

RemoteRef next

RemoteSha 3dcf83e62136b61c45f18866a719189711cca518

Contents

as.data.frame.polars_data_frame	6
as.list.polars_data_frame	8
as_polars_df	10
as_polars_expr	13

as_polars_lf	16
as_polars_series	17
as_tibble.polars_data_frame	21
check_polars	24
cs	27
dataframe__cast	27
dataframe__clone	28
dataframe__drop	29
dataframe__equals	30
dataframe__filter	30
dataframe__get_columns	31
dataframe__group_by	32
dataframe__lazy	33
dataframe__select	33
dataframe__slice	34
dataframe__sort	35
dataframe__to_series	35
dataframe__to_struct	36
dataframe__with_columns	37
expr_arr_all	38
expr_arr_any	38
expr_arr_arg_max	39
expr_arr_arg_min	39
expr_arr_contains	40
expr_arr_count_matches	40
expr_arr_explode	41
expr_arr_first	41
expr_arr_get	42
expr_arr_join	43
expr_arr_last	43
expr_arr_max	44
expr_arr_median	44
expr_arr_min	45
expr_arr_n_unique	45
expr_arr_reverse	46
expr_arr_shift	46
expr_arr_sort	47
expr_arr_std	47
expr_arr_sum	48
expr_arr_to_list	48
expr_arr_unique	49
expr_arr_var	49
expr_bin_contains	50
expr_bin_decode	50
expr_bin_encode	51
expr_bin_ends_with	52
expr_bin_size	53
expr_bin_starts_with	53

expr_cat_get_categories	54
expr_cat_set_ordering	55
expr_dt_add_business_days	56
expr_dt_base_utc_offset	57
expr_dt_cast_time_unit	57
expr_dt_century	58
expr_dt_combine	59
expr_dt_convert_time_zone	60
expr_dt_date	60
expr_dt_day	61
expr_dt_dst_offset	62
expr_dt_epoch	62
expr_dt_hour	63
expr_dt_iso_year	63
expr_dt_is_leap_year	64
expr_dt_microsecond	64
expr_dt_millisecond	65
expr_dt_minute	66
expr_dt_month	66
expr_dt_month_end	67
expr_dt_month_start	67
expr_dt_nanosecond	68
expr_dt_offset_by	69
expr_dt_ordinal_day	70
expr_dt_quarter	71
expr_dt_replace_time_zone	71
expr_dt_round	73
expr_dt_second	74
expr_dt_strftime	75
expr_dt_time	76
expr_dt_timestamp	76
expr_dt_total_days	77
expr_dt_total_hours	78
expr_dt_total_microseconds	78
expr_dt_total_milliseconds	79
expr_dt_total_minutes	79
expr_dt_total_nanoseconds	80
expr_dt_total_seconds	81
expr_dt_to_string	81
expr_dt_truncate	82
expr_dt_week	83
expr_dt_weekday	84
expr_dt_with_time_unit	84
expr_dt_year	85
expr_list_all	86
expr_list_any	86
expr_list_arg_max	87
expr_list_arg_min	87

expr_list_concat	88
expr_list_contains	88
expr_list_count_matches	89
expr_list_diff	90
expr_list_drop_nulls	90
expr_list_eval	91
expr_list_explode	92
expr_list_first	92
expr_list_gather	93
expr_list_gather_every	94
expr_list_get	94
expr_list_head	95
expr_list_join	96
expr_list_last	96
expr_list_len	97
expr_list_max	97
expr_list_mean	98
expr_list_median	98
expr_list_min	99
expr_list_n_unique	99
expr_list_reverse	100
expr_list_sample	100
expr_list_set_difference	101
expr_list_set_intersection	102
expr_list_set_symmetric_difference	102
expr_list_set_union	103
expr_list_shift	104
expr_list_slice	105
expr_list_sort	105
expr_list_std	106
expr_list_sum	107
expr_list_tail	107
expr_list_to_array	108
expr_list_unique	109
expr_list_var	109
expr_str_contains	110
expr_str_contains_any	111
expr_str_count_matches	112
expr_str_decode	112
expr_str_encode	113
expr_str_ends_with	114
expr_str_extract	115
expr_str_extract_all	115
expr_str_extract_groups	116
expr_str_extract_many	117
expr_str_find	118
expr_str_head	119
expr_str_join	120

expr_str_json_decode	121
expr_str_json_path_match	121
expr_str_len_bytes	122
expr_str_len_chars	123
expr_str_pad_end	124
expr_str_pad_start	124
expr_str_replace	125
expr_str_replace_all	126
expr_str_replace_many	128
expr_str_reverse	129
expr_str_slice	129
expr_str_split	130
expr_str_splitn	130
expr_str_split_exact	131
expr_str_starts_with	132
expr_str_strip_chars	132
expr_str_strip_chars_end	133
expr_str_strip_chars_start	134
expr_str_strptime	134
expr_str_tail	137
expr_str_to_date	138
expr_str_to_datetime	139
expr_str_to_integer	140
expr_str_to_lowercase	141
expr_str_to_time	141
expr_str_to_uppercase	142
expr_str_zfill	142
lazyframe__collect	143
lazyframe__select	145
lazyframe__with_columns	146
pl	147
pl_api_register_series_namespace	148
pl_col	149
pl_DataFrame	150
pl_datetime_range	151
pl_datetime_ranges	153
pl_date_range	155
pl_date_ranges	157
pl_LazyFrame	159
pl_lit	160
pl_Series	161
pl_show_versions	162
pl_struct	163
polars_duration_string	164
polars_expr	164
series_struct_unnest	165
series__to_frame	166
series__to_r_vector	166

as.data.frame.polars_data_frame

Export the polars object as an R DataFrame

Description

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

Usage

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

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

Arguments

x	A polars object
...	Passed to <code>as_polars_df()</code> .
int64	Determine how to convert Polars' Int64, UInt32, or UInt64 type values to R type. One of the followings: <ul style="list-style-type: none"> "double" (default): Convert to the R's <code>double</code> type. Accuracy may be degraded.

	<ul style="list-style-type: none"> • "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	<p>Determine how to convert Polars' Date type values to R class. One of the followings:</p> <ul style="list-style-type: none"> • "Date" (default): Convert to the R's Date class. • "IDate": Convert to the data.table::IDate class.
time	<p>Determine how to convert Polars' Time type values to R class. One of the followings:</p> <ul style="list-style-type: none"> • "hms" (default): Convert to the hms::hms class. • "ITime": Convert to the data.table::ITime class. The data.table package must be installed.
decimal	<p>Determine how to convert Polars' Decimal type values to R type. One of the followings:</p> <ul style="list-style-type: none"> • "double" (default): Convert to the R's double type. • "character": Convert to the R's character type.
as_clock_class	<p>A logical value indicating whether to export datetimes and duration as the clock package's classes.</p> <ul style="list-style-type: none"> • 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	<p>Determine how to deal with ambiguous datetimes. Only applicable when <code>as_clock_class</code> is set to FALSE and datetime without timezone values are exported as POSIXct. Character vector or expression containing the followings:</p> <ul style="list-style-type: none"> • "raise" (default): Throw an error • "earliest": Use the earliest datetime • "latest": Use the latest datetime • "null": Return a NA value
non_existent	<p>Determine how to deal with non-existent datetimes. Only applicable when <code>as_clock_class</code> is set to FALSE and datetime without timezone values are exported as POSIXct. One of the followings:</p> <ul style="list-style-type: none"> • "raise" (default): Throw an error • "null": Return a NA value

Value

An [R data frame](#)

Examples

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

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

```
as.list.polars_data_frame
```

Export the polars object as an R list

Description

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

Usage

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

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


Arguments

x	A polars object
...	Passed to <code>as_polars_df()</code> .
as_series	Whether to convert each column to an R vector or a Series . If TRUE, return a list of Series , otherwise a list of vectors (default).
int64	Determine how to convert Polars' Int64, UInt32, or UInt64 type values to R type. One of the followings: <ul style="list-style-type: none"> "double" (default): Convert to the R's double type. Accuracy may be degraded. "character": Convert to the R's character type. "integer": Convert to the R's integer type. If the value is out of the range of R's integer type, export as NA_integer_. "integer64": Convert to the bit64::integer64 class. The bit64 package must be installed. If the value is out of the range of bit64::integer64, export as bit64::NA_integer64_.
date	Determine how to convert Polars' Date type values to R class. One of the followings: <ul style="list-style-type: none"> "Date" (default): Convert to the R's Date class. "IDate": Convert to the data.table::IDate class.
time	Determine how to convert Polars' Time type values to R class. One of the followings: <ul style="list-style-type: none"> "hms" (default): Convert to the hms::hms class. "ITime": Convert to the data.table::ITime class. The data.table package must be installed.
struct	Determine how to convert Polars' Struct type values to R class. One of the followings: <ul style="list-style-type: none"> "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	Determine how to convert Polars' Decimal type values to R type. One of the followings: <ul style="list-style-type: none"> "double" (default): Convert to the R's double type. "character": Convert to the R's character type.
as_clock_class	A logical value indicating whether to export datetimes and duration as the clock package's classes. <ul style="list-style-type: none"> 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	Determine how to deal with ambiguous datetimes. Only applicable when <code>as_clock_class</code> is set to <code>FALSE</code> and datetime without timezone values are exported as <code>POSIXct</code> . Character vector or <code>expression</code> containing the followings: <ul style="list-style-type: none"> "raise" (default): Throw an error "earliest": Use the earliest datetime "latest": Use the latest datetime "null": Return a NA value
non_existent	Determine how to deal with non-existent datetimes. Only applicable when <code>as_clock_class</code> is set to <code>FALSE</code> and datetime without timezone values are exported as <code>POSIXct</code> . One of the followings: <ul style="list-style-type: none"> "raise" (default): Throw an error "null": Return a NA value

Details

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

Value

A `list`

See Also

- `<DataFrame>$get_columns()`

Examples

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

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

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

as_polars_df

Create a Polars DataFrame from an R object

Description

The `as_polars_df()` function creates a `polars DataFrame` from various R objects. `Polars DataFrame` is based on a sequence of `Polars Series`, so basically, the input object is converted to a `list` of `Polars Series` by `as_polars_series()`, then a `Polars DataFrame` is created from the list.

Usage

```

as_polars_df(x, ...)

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

## S3 method for class 'polars_series'
as_polars_df(x, ..., column_name = NULL, from_struct = TRUE)

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

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

## S3 method for class 'polars_lazy_frame'
as_polars_df(
  x,
  ...,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  comm_subplan_elim = TRUE,
  comm_subexpr_elim = TRUE,
  cluster_with_columns = TRUE,
  no_optimization = FALSE,
  streaming = FALSE
)

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

## S3 method for class 'data.frame'
as_polars_df(x, ...)

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

```

Arguments

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 <code><Series>\$struct\$unnest()</code>

	method is used to create a DataFrame from the struct Series . In this case, the <code>column_name</code> argument is ignored.
<code>type_coercion</code>	A logical, indicates type coercion optimization.
<code>predicate_pushdown</code>	A logical, indicates predicate pushdown optimization.
<code>projection_pushdown</code>	A logical, indicates projection pushdown optimization.
<code>simplify_expression</code>	A logical, indicates simplify expression optimization.
<code>slice_pushdown</code>	A logical, indicates slice pushdown optimization.
<code>comm_subplan_elim</code>	A logical, indicates try to cache branching subplans that occur on self-joins or unions.
<code>comm_subexpr_elim</code>	A logical, indicates try to cache common subexpressions.
<code>cluster_with_columns</code>	A logical, indicates to combine sequential independent calls to <code>with_columns</code> .
<code>no_optimization</code>	A logical. If TRUE, turn off (certain) optimizations.
<code>streaming</code>	A logical. If TRUE, process the query in batches to handle larger-than-memory data. If FALSE (default), the entire query is processed in a single batch. Note that streaming mode is considered unstable. It may be changed at any point without it being considered a breaking change.

Details

The default method of `as_polars_df()` throws an error, so we need to define methods for the classes we want to support.

S3 method for `list`:

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

S3 method for `data.frame`:

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

S3 method for `polars_series`:

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

S3 method for `polars_lazy_frame`:

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

Value

A polars [DataFrame](#)

See Also

- [as.list\(<polars_data_frame>\)](#): Export the DataFrame as an R list.
- [as.data.frame\(<polars_data_frame>\)](#): Export the DataFrame as an R data frame.

Examples

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

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

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

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

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

as_polars_expr

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, ...)
```

```

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

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

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

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

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

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

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

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

```

Arguments

x	An R object.
...	Additional arguments passed to the methods.
structify	A logical. If TRUE, convert multi-column expressions to a single struct expression by calling <code>pl\$struct()</code> . Otherwise (default), done nothing.
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 <code>pl\$lit()</code> , and expects to return literal values. See examples for details.

Details

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

Default S3 method:

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

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

S3 method for [character](#):

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

Value

A polars [expression](#)

Literal scalar mapping

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

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

These types' NA is converted to a null literal with casting to the corresponding Polars type.

The [raw](#) type vector is converted to a Binary scalar.

- raw: Binary

NULL is converted to a Null type null literal.

- NULL: Null

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

See Also

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

Examples

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

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

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

```

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

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

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

# NULL
as_polars_expr(NULL)

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

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

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

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

```

as_polars_lf

Create a Polars LazyFrame from an R object

Description

The `as_polars_lf()` function creates a [LazyFrame](#) from various R objects. It is basically a shortcut for `as_polars_df(x, ...)` with the `$lazy()` method.

Usage

```
as_polars_lf(x, ...)
```



```
## Default S3 method:
as_polars_lf(x, ...)

## S3 method for class 'polars_lazy_frame'
as_polars_lf(x, ...)
```

Arguments

`x` An R object.

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

Details

Default S3 method:

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

Value

A polars [LazyFrame](#)

as_polars_series	<i>Create a Polars Series from an R object</i>
------------------	--

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 '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 'difftime'
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 '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. 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](#):

Since polars Data type's unit is day and physical type is integer, sub-day precision of the R's [Date](#) object will be ignored.

S3 method for hms:

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

S3 method for polars_data_frame:

This method is a shortcut for `<DataFrame>$to_struct()`.

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.

Examples

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

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

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

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

# raw
as_polars_series(charToRaw("foo"))

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

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

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

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

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

# 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)))
}

```

as_tibble.polars_data_frame

Export the polars object as a tibble data frame

Description

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

Usage

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

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

Arguments

<code>x</code>	A polars object
<code>...</code>	Passed to <code>as_polars_df()</code> .
<code>.name_repair</code>	Treatment of problematic column names: <ul style="list-style-type: none"> • "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 • a function: apply custom name repair (e.g., <code>.name_repair = make.names</code> for names in the style of base R).

	<ul style="list-style-type: none"> • A purrr-style anonymous function, see <code>rlang::as_function()</code> <p>This argument is passed on as repair to <code>vctrs::vec_as_names()</code>. See there for more details on these terms and the strategies used to enforce them.</p>
int64	<p>Determine how to convert Polars' Int64, UInt32, or UInt64 type values to R type. One of the followings:</p> <ul style="list-style-type: none"> • "double" (default): Convert to the R's <code>double</code> type. Accuracy may be degraded. • "character": Convert to the R's <code>character</code> type. • "integer": Convert to the R's <code>integer</code> type. If the value is out of the range of R's integer type, export as <code>NA_integer_</code>. • "integer64": Convert to the <code>bit64::integer64</code> class. The <code>bit64</code> package must be installed. If the value is out of the range of <code>bit64::integer64</code>, export as <code>bit64::NA_integer64_</code>.
date	<p>Determine how to convert Polars' Date type values to R class. One of the followings:</p> <ul style="list-style-type: none"> • "Date" (default): Convert to the R's <code>Date</code> class. • "IDate": Convert to the <code>data.table::IDate</code> class.
time	<p>Determine how to convert Polars' Time type values to R class. One of the followings:</p> <ul style="list-style-type: none"> • "hms" (default): Convert to the <code>hms::hms</code> class. • "ITime": Convert to the <code>data.table::ITime</code> class. The <code>data.table</code> package must be installed.
decimal	<p>Determine how to convert Polars' Decimal type values to R type. One of the followings:</p> <ul style="list-style-type: none"> • "double" (default): Convert to the R's <code>double</code> type. • "character": Convert to the R's <code>character</code> type.
as_clock_class	<p>A logical value indicating whether to export datetimes and duration as the <code>clock</code> package's classes.</p> <ul style="list-style-type: none"> • FALSE (default): Duration values are exported as <code>difftime</code> and datetime values are exported as <code>POSIXct</code>. Accuracy may be degraded. • TRUE: Duration values are exported as <code>clock_duration</code>, datetime without timezone values are exported as <code>clock_naive_time</code>, and datetime with timezone values are exported as <code>clock_zoned_time</code>. For this case, the <code>clock</code> package must be installed. Accuracy will be maintained.
ambiguous	<p>Determine how to deal with ambiguous datetimes. Only applicable when <code>as_clock_class</code> is set to FALSE and datetime without timezone values are exported as <code>POSIXct</code>. Character vector or <code>expression</code> containing the followings:</p> <ul style="list-style-type: none"> • "raise" (default): Throw an error • "earliest": Use the earliest datetime • "latest": Use the latest datetime • "null": Return a NA value
non_existent	<p>Determine how to deal with non-existent datetimes. Only applicable when <code>as_clock_class</code> is set to FALSE and datetime without timezone values are exported as <code>POSIXct</code>. One of the followings:</p>

- "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 or FALSE depending on the class of the object. `check_*` functions throw an informative error if the object is not the correct class. Suffixes are corresponding to the polars object classes:

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

Usage

```
is_polars_dtype(x)

is_polars_df(x)

is_polars_expr(x, ...)

is_polars_lf(x)

is_polars_selector(x, ...)

is_polars_series(x)

is_list_of_polars_dtype(x, n = NULL)

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

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

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

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

check_polars_selector(
```

```

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

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

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

```

Arguments

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

Details

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

Value

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

Examples

```
is_polars_df(as_polars_df(mtcars))
```

```
is_polars_df(mtcars)

# Use `check_polars_*` functions in a function
# to ensure the input is a polars object
sample_func <- function(x) {
  check_polars_df(x)
  TRUE
}

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

cs

Polars column selector function namespace

Description

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

Usage

```
cs
```

Format

An object of class `polars_object` of length 2.

Examples

```
cs

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

dataframe__cast

Cast DataFrame column(s) to the specified dtype

Description

Cast DataFrame column(s) to the specified dtype

Usage

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

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__clone

Clone a DataFrame

Description

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

Usage

```
dataframe__clone()
```

Value

A polars [DataFrame](#)

Examples

```
df1 <- as_polars_df(iris)  
  
# Assigning does not copy the DataFrame (environment object), calling  
# $clone() creates a new environment.  
df2 <- df1  
df3 <- df1$clone()  
rlang::env_label(df1)  
rlang::env_label(df2)  
rlang::env_label(df3)  
  
# Cloning can be useful to add attributes to data used in a function without  
# adding those attributes to the original object.
```

```

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

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

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

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

```

dataframe__drop

Drop columns of a DataFrame

Description

Drop columns of a DataFrame

Usage

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

Arguments

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

Value

A polars [DataFrame](#)

Examples

```

as_polars_df(mtcars)$drop(c("mpg", "hp"))

# equivalent
as_polars_df(mtcars)$drop("mpg", "hp")

```

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.

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__filter *Filter rows of a DataFrame*

Description

Filter rows of a DataFrame

Usage

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

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__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__group_by *Group a DataFrame*

Description

Group a DataFrame

Usage

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

Details

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

Value

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

See Also

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

Examples

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

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

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

# Group by multiple columns by passing a list of column names.
df$group_by(c("a", "b"))$agg(pl$max("c"))

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

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



```
pl$col("c")$mean()
)
```

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

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

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__slice	<i>Get a slice of the DataFrame.</i>
------------------	--------------------------------------

Description

Get a slice of the DataFrame.

Usage

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

Arguments

offset	Start index, can be a negative value. This is 0-indexed, so offset = 1 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	<i>Sort a DataFrame</i>
-----------------	-------------------------

Description

Sort a DataFrame

Usage

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

Value

A polars [DataFrame](#)

Examples

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

dataframe__to_series	<i>Select column as Series at index location</i>
----------------------	--

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)  
  
# doesn't error if the column isn't there  
df$to_series(index = 8)
```

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__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

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

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

as_polars_df(iris)$with_columns(
  SW_add_2 = (pl$col("Sepal.Width") + 2),
  # unnamed expr will keep name "Sepal.Length"
  pl$col("Sepal.Length")$abs()
)
```

`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()
```

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()
```

Value

A polars [expression](#)

Examples

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

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

Description

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

Usage

```
expr_arr_arg_max()
```

Value

A polars [expression](#)

Examples

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

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

Description

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

Usage

```
expr_arr_arg_min()
```

Value

A polars [expression](#)

Examples

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

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

Description

Check if sub-arrays contain the given item

Usage

```
expr_arr_contains(item)
```

Arguments

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

Value

A polars [expression](#)

Examples

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

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

Description

Count how often a value occurs in every sub-array

Usage

```
expr_arr_count_matches(element)
```

Arguments

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

Value

A polars [expression](#)

Examples

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

expr_arr_explode	<i>Explode array in separate rows</i>
------------------	---------------------------------------

Description

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

Usage

```
expr_arr_explode()
```

Value

A polars [expression](#)

Examples

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

expr_arr_first	<i>Get the first value of the sub-arrays</i>
----------------	--

Description

Get the first value of the sub-arrays

Usage

```
expr_arr_first()
```

Value

A polars [expression](#)

Examples

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

 expr_arr_get

Get the value by index in every sub-array

Description

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

Usage

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

Arguments

index	An Expr or something coercible to an Expr, that must return a single index.
...	Dots which should be empty.
null_on_oob	If TRUE, return null if an index is out of bounds. Otherwise, raise an error.

Value

Expr

Examples

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

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

Description

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

Usage

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

Arguments

separator	String to separate the items with. Can be an Expr. Strings are not parsed as columns.
...	Dots which should be empty.

Value

A polars [expression](#)

Examples

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

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

Description

Get the last value of the sub-arrays

Usage

```
expr_arr_last()
```

Value

A polars [expression](#)

Examples

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

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

Description

Compute the max value of the sub-arrays

Usage

```
expr_arr_max()
```

Value

A polars [expression](#)

Examples

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

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

Description

Compute the median value of the sub-arrays

Usage

```
expr_arr_median()
```

Value

A polars [expression](#)

Examples

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

 expr_arr_min

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)
```

Value

A polars [expression](#)

Examples

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

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

Description

Sort values in every sub-array

Usage

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

Arguments

... Dots which should be empty.

Examples

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

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

Description

Compute the standard deviation of the sub-arrays

Usage

```
expr_arr_std(ddof = 1)
```

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	<i>Compute the sum of the sub-arrays</i>
--------------	--

Description

Compute the sum of the sub-arrays

Usage

```
expr_arr_sum()
```

Value

A polars [expression](#)

Examples

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

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

Description

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

Usage

```
expr_arr_to_list()
```

Value

A polars [expression](#)

Examples

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

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

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

Description

Get the unique values in every sub-array

Usage

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

Arguments

... Dots which should be empty.

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	<i>Compute the variance of the sub-arrays</i>
--------------	---

Description

Compute the variance of the sub-arrays

Usage

```
expr_arr_var(ddof = 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.
...	Dots which should 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_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

sub Prefix substring.

Value

A polars [expression](#)

Examples

```
colors <- pl$DataFrame(  
  name = c("black", "yellow", "blue"),  
  code = as_polars_series(c("x00x00x00", "xffxffx00", "x00x00xff"))$cast(pl$Binary),  
  prefix = as_polars_series(c("x00", "xffx00", "xffxff"))$cast(pl$Binary)  
)  
  
colors$select(  
  "name",  
  starts_with_lit = pl$col("code")$bin$starts_with("xff"),  
  starts_with_expr = pl$col("code")$bin$starts_with(pl$col("prefix"))  
)
```

expr_cat_get_categories

Get the categories stored in this data type

Description

Get the categories stored in this data type

Usage

```
expr_cat_get_categories()
```

Value

A polars [expression](#)

Examples

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

expr_cat_set_ordering Set Ordering

Description

Determine how this categorical series should be sorted.

Usage

```
expr_cat_set_ordering(ordering)
```

Arguments

ordering string either 'physical' or 'lexical'

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

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(  
  cats = factor(c("z", "z", "k", "a", "b")),  
  vals = c(3, 1, 2, 2, 3)  
)  
  
# sort by the string value of categories  
df$with_columns(  
  pl$col("cats")$cat$set_ordering("lexical")  
)$sort("cats", "vals")  
  
# sort by the underlying value of categories  
df$with_columns(  
  pl$col("cats")$cat$set_ordering("physical")  
)$sort("cats", "vals")
```

```
expr_dt_add_business_days
    Offset by n business days.
```

Description

Offset by n business days.

Usage

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

Arguments

n	Number of business days to offset by. Can be a single integer or an expression.
week_mask	Which days of the week to count. The default is Monday to Friday. If you wanted to count only Monday to Thursday, you would pass c(TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE).
holidays	Holidays to exclude from the count.
roll	What to do when the start date lands on a non-business day. Options are: <ul style="list-style-type: none"> "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))

# 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 = c("ns", "us", "ms"))
```

Arguments

`time_unit` Time unit, one of 'ns', 'us', or 'ms'.

Value

A polars [expression](#)

Examples

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

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

Description

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

Usage

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

Arguments

time	The number of epoch since or before (if negative) the Date. Can be an Expr or a PTime.
time_unit	Time unit, one of 'ns', 'us', or 'ms'.

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

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$DataFrame(date = pl$date_range(as.Date("2001-1-1"), as.Date("2001-1-3")))

df$with_columns(
  epoch_ns = pl$col("date")$dt$epoch(),
  epoch_s = pl$col("date")$dt$epoch(time_unit = "s")
)
```

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

Description

Returns the hour number from 0 to 23.

Usage

```
expr_dt_hour()
```

Value

A polars [expression](#)

Examples

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

expr_dt_iso_year	<i>Extract ISO year from underlying Date representation</i>
------------------	---

Description

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

Usage

```
expr_dt_iso_year()
```

Value

A polars [expression](#)

Examples

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

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

Description

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

Usage

```
expr_dt_is_leap_year()
```

Value

A polars [expression](#)

Examples

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

expr_dt_microsecond *Extract microseconds from underlying Datetime representation*

Description

Extract microseconds from underlying Datetime representation

Usage

```
expr_dt_microsecond()
```


Value

A polars [expression](#)

Examples

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

expr_dt_millisecond *Extract milliseconds from underlying Datetime representation*

Description

Extract milliseconds from underlying Datetime representation

Usage

```
expr_dt_millisecond()
```

Value

A polars [expression](#)

Examples

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

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

Description

For datetimes, the time of day is preserved.

Usage

```
expr_dt_month_end()
```

Value

A polars [expression](#)

Examples

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

expr_dt_month_start	<i>Roll backward to the first day of the month</i>
---------------------	--

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	<i>Offset a date by a relative time offset</i>
-------------------	--

Description

This differs from `pl$col("foo") + timedelta` in that it can take months and leap years into account. Note that only a single minus sign is allowed in the by string, as the first character.

Usage

```
expr_dt_offset_by(by)
```

Arguments

`by` optional string encoding duration see details.

Details

The by are created with the following string language:

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

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

These strings can be combined:

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

Value

A polars [expression](#)

Examples

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

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

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

expr_dt_ordinal_day *Extract ordinal day from underlying Date representation*

Description

Returns the day of year starting from 1. The return value ranges from 1 to 366 (the last day of year differs across years).

Usage

```
expr_dt_ordinal_day()
```

Value

A polars [expression](#)

Examples

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

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

expr_dt_quarter *Extract quarter from underlying Date representation*

Description

Returns the quarter ranging from 1 to 4.

Usage

```
expr_dt_quarter()
```

Value

A polars [expression](#)

Examples

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

expr_dt_replace_time_zone *Replace time zone for an expression of type Datetime*

Description

Different from [\\$dt\\$convert_time_zone\(\)](#), this will also modify the underlying timestamp and will ignore the original time zone.

Usage

```
expr_dt_replace_time_zone(
  time_zone,
  ...,
  ambiguous = c("raise", "earliest", "latest", "null"),
  non_existent = c("raise", "null")
)
```

Arguments

time_zone	NULL or a character time zone from <code>base::OlsonNames()</code> . 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: <ul style="list-style-type: none"> • "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: <ul style="list-style-type: none"> • "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	<i>Round datetime</i>
---------------	-----------------------

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

`fractional` If TRUE, include the fractional component of the second.

Value

A polars [expression](#)

Examples

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

df$with_columns(
  second = pl$col("datetime")$dt$second(),
  second_fractional = pl$col("datetime")$dt$second(fractional = TRUE)
)
```

expr_dt_strftime	<i>Convert date/time/datetime to string</i>
------------------	---

Description

Similar to `$cast(pl$String)`, but this method allows you to customize the formatting of the resulting string. This is an alias for `dtto_string()`.

Usage

```
expr_dt_strftime(format)
```

Arguments

format	Format to use. See chrono docs for specifying the format: https://docs.rs/chrono/latest/chrono/format/strftime/index.html .
--------	---

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	<i>Extract time</i>
--------------	---------------------

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	<i>Get timestamp in the given time unit</i>
-------------------	---

Description

Get timestamp in the given time unit

Usage

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

Arguments

time_unit Time unit, one of 'ns', 'us', or 'ms'.

Value

A polars [expression](#)

Examples

```
df <- pl$select(  
  date = pl$datetime_range(  
    start = as.Date("2001-1-1"),  
    end = as.Date("2001-1-3"),  
    interval = "1d1s"  
  )  
)  
df$select(  
  pl$col("date"),  
  pl$col("date")$dt$timestamp()$alias("timestamp_ns"),  
  pl$col("date")$dt$timestamp(time_unit = "ms")$alias("timestamp_ms")  
)
```

expr_dt_total_days *Extract the days from a Duration type*

Description

Extract the days from a Duration type

Usage

```
expr_dt_total_days()
```

Value

A polars [expression](#)

Examples

```
df <- pl$select(  
  date = pl$datetime_range(  
    start = as.Date("2020-3-1"),  
    end = as.Date("2020-5-1"),  
    interval = "1mo1s"  
  )  
)  
df$with_columns(  
  diff_days = pl$col("date")$diff()$dt$total_days()  
)
```

expr_dt_total_hours *Extract the hours from a Duration type*

Description

Extract the hours from a Duration type

Usage

```
expr_dt_total_hours()
```

Value

A polars [expression](#)

Examples

```
df <- pl$select(  
  date = pl$date_range(  
    start = as.Date("2020-1-1"),  
    end = as.Date("2020-1-4"),  
    interval = "1d"  
  )  
)  
df$with_columns(  
  diff_hours = pl$col("date")$diff()$dt$total_hours()  
)
```

expr_dt_total_microseconds
 Extract the microseconds from a Duration type

Description

Extract the microseconds from a Duration type

Usage

```
expr_dt_total_microseconds()
```

Value

A polars [expression](#)

Examples

```
df <- pl$select(date = pl$datetime_range(  
  start = as.POSIXct("2020-1-1", tz = "GMT"),  
  end = as.POSIXct("2020-1-1 00:00:01", tz = "GMT"),  
  interval = "1ms"  
)  
)  
df$with_columns(  
  diff_microsec = pl$col("date")$diff()$dt$total_microseconds()  
)
```

expr_dt_total_milliseconds

Extract the milliseconds from a Duration type

Description

Extract the milliseconds from a Duration type

Usage

```
expr_dt_total_milliseconds()
```

Value

A polars [expression](#)

Examples

```
df <- pl$select(date = pl$datetime_range(  
  start = as.POSIXct("2020-1-1", tz = "GMT"),  
  end = as.POSIXct("2020-1-1 00:00:01", tz = "GMT"),  
  interval = "1ms"  
)  
)  
df$with_columns(  
  diff_millisec = pl$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()
```

Value

A polars [expression](#)

Examples

```
df <- pl$select(  
  date = pl$date_range(  
    start = as.Date("2020-1-1"),  
    end = as.Date("2020-1-4"),  
    interval = "1d"  
  )  
)  
df$with_columns(  
  diff_minutes = pl$col("date")$diff()$dt$total_minutes()  
)
```

expr_dt_total_nanoseconds

Extract the nanoseconds from a Duration type

Description

Extract the nanoseconds from a Duration type

Usage

```
expr_dt_total_nanoseconds()
```

Value

A polars [expression](#)

Examples

```
df <- pl$select(date = pl$datetime_range(  
  start = as.POSIXct("2020-1-1", tz = "GMT"),  
  end = as.POSIXct("2020-1-1 00:00:01", tz = "GMT"),  
  interval = "1ms"  
)  
)  
df$with_columns(  
  diff_nanosec = pl$col("date")$diff()$dt$total_nanoseconds()  
)
```

expr_dt_total_seconds *Extract the seconds from a Duration type*

Description

Extract the seconds from a Duration type

Usage

```
expr_dt_total_seconds()
```

Value

A polars [expression](#)

Examples

```
df <- pl$select(date = pl$datetime_range(  
  start = as.POSIXct("2020-1-1", tz = "GMT"),  
  end = as.POSIXct("2020-1-1 00:04:00", tz = "GMT"),  
  interval = "1m"  
)  
)  
df$with_columns(  
  diff_sec = pl$col("date")$diff()$dt$total_seconds()  
)
```

expr_dt_to_string *Convert date/time/datetime to string*

Description

Similar to `$cast(pl$string)`, but this method allows you to customize the formatting of the resulting string. This is an alias for `dtstrftime()`.

Usage

```
expr_dt_to_string(format)
```

Arguments

format Format to use. See chrono docs for specifying the format: <https://docs.rs/chrono/latest/chrono/format/strftime/index.html>.

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$to_string("%Y/%m/%d %H:%M:%S")
  )
```

expr_dt_truncate	<i>Truncate datetime</i>
------------------	--------------------------

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_with_time_unit
Set time unit of a Series of dtype Datetime or Duration

Description

This is deprecated. Cast to Int64 and then to Datetime instead.

Usage

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

Arguments

time_unit Time unit, one of 'ns', 'us', or 'ms'.

Value

A polars [expression](#)

Examples

```
df <- pl$select(
  date = pl$datetime_range(
    start = as.Date("2001-1-1"),
    end = as.Date("2001-1-3"),
    interval = "1d1s"
  )
)
df$with_columns(
  with_time_unit_ns = pl$col("date")$dt$with_time_unit(),
  with_time_unit_ms = pl$col("date")$dt$with_time_unit(time_unit = "ms")
)
```

expr_dt_year

Extract year from underlying Date representation

Description

Returns the year number in the calendar date.

Usage

```
expr_dt_year()
```

Value

A polars [expression](#)

Examples

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

`expr_list_all`*Evaluate whether all boolean values in a sub-list are true*

Description

Evaluate whether all boolean values in a sub-list are true

Usage

```
expr_list_all()
```

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(  
  a = list(c(TRUE, TRUE), c(FALSE, TRUE), c(FALSE, FALSE), NA, c())  
)  
df$with_columns(all = pl$col("a")$list$all())
```

`expr_list_any`*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()
```

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)
```

Arguments

item Item that will be checked for membership. Can be an Expr or something coercible to an Expr. Strings are not parsed as columns.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(  
  a = list(3:1, NULL, 1:2),  
  item = 0:2  
)  
df$with_columns(  
  with_expr = pl$col("a")$list$contains(pl$col("item")),  
  with_lit = pl$col("a")$list$contains(1)  
)
```

expr_list_count_matches

Count how often a value produced occurs

Description

Count how often a value produced occurs

Usage

```
expr_list_count_matches(element)
```

Arguments

element An expression that produces a single value.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(a = list(0, 1, c(1, 2, 3, 2), c(1, 2, 1), c(4, 4)))  
  
df$with_columns(  
  number_of_twos = pl$col("a")$list$count_matches(2)  
)
```

expr_list_diff *Compute difference between sub-list values*

Description

This computes the first discrete difference between shifted items of every list. The parameter `n` gives the interval between items to subtract, e.g. if `n = 2` the output will be the difference between the 1st and the 3rd value, the 2nd and 4th value, etc.

Usage

```
expr_list_diff(n = 1, null_behavior = c("ignore", "drop"))
```

Arguments

`n` Number of slots to shift. If negative, then it starts from the end.
`null_behavior` How to handle null values. Either "ignore" (default) or "drop".

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(s = list(1:4, c(10L, 2L, 1L)))
df$with_columns(diff = pl$col("s")$list$diff(2))

# negative value starts shifting from the end
df$with_columns(diff = pl$col("s")$list$diff(-2))
```

expr_list_drop_nulls *Drop all null values in every sub-list*

Description

Drop all null values in every sub-list

Usage

```
expr_list_drop_nulls()
```

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(values = list(c(NA, 0, NA), c(1, NaN), NA))

df$with_columns(
  without_nulls = pl$col("values")$list$drop_nulls()
)
```

expr_list_eval *Run any polars expression on the sub-lists' values*

Description

Run any polars expression on the sub-lists' values

Usage

```
expr_list_eval(expr, ..., parallel = FALSE)
```

Arguments

expr	Expression to run. Note that you can select an element with <code>pl\$element()</code> , <code>pl\$first()</code> , and more. See Examples.
parallel	Run all expressions in parallel. Don't activate this blindly. Parallelism is worth it if there is enough work to do per thread. This likely should not be used in the <code>\$group_by()</code> context, because groups are already executed in parallel.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(
  a = list(c(1, 8, 3), c(3, 2), c(NA, NA, 1)),
  b = list(c("R", "is", "amazing"), c("foo", "bar"), "text")
)

df

# standardize each value inside a list, using only the values in this list
df$select(
  a_stand = pl$col("a")$list$eval(
    (pl$element() - pl$element()$mean()) / pl$element()$std()
  )
)

# count characters for each element in list. Since column "b" is list[str],
# we can apply all `str` functions on elements in the list:
df$select(
```

```

    b_len_chars = pl$col("b")$list$eval(
      pl$element()$str$len_chars()
    )
  )

  # concat strings in each list
  df$select(
    pl$col("b")$list$eval(pl$element()$str$join(" "))$list$first()
  )

```

expr_list_explode *Returns a column with a separate row for every list element*

Description

Returns a column with a separate row for every list element

Usage

```
expr_list_explode()
```

Value

A polars [expression](#)

Examples

```

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

```

expr_list_first *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(index, ..., null_on_oob = FALSE)
```

Arguments

index	An Expr or something coercible to an Expr, that can return several indices. Values are 0-indexed (so index 0 would return the first item of every sub-list) and negative values start from the end (index -1 returns the last item). If the index is out of bounds, it will return a null. Strings are parsed as column names.
...	Dots which should 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(2, null_on_oob = TRUE)
)

# by some column name, must cast to an Int/UInt type to work
df$with_columns(
  gathered = pl$col("a")$list$gather(pl$col("a")$cast(pl$List(pl$UInt64)), null_on_oob = TRUE)
)
```

```
expr_list_gather_every
```

Take every n-th value starting from offset in sub-lists

Description

Take every n-th value starting from offset in sub-lists

Usage

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

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\(\)](#).

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).
...	Dots which should be empty.
null_on_oob	If TRUE, return null if an index is out of bounds. Otherwise, raise an error.

Value

Expr

Examples

```
df <- pl$DataFrame(
  values = list(c(2, 2, NA), c(1, 2, 3), NA, NULL),
  idx = c(1, 2, NA, 3)
)
df$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	<i>Slice the first n values of every sub-list</i>
----------------	---

Description

Slice the first n values of every sub-list

Usage

```
expr_list_head(n = 5L)
```

Arguments

n	Number of values to return for each sub-list. Can be an Expr. Strings are parsed as column names.
---	---

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(
  s = list(1:4, c(10L, 2L, 1L)),
  n = 1:2
)
df$with_columns(
  head_by_expr = pl$col("s")$list$head("n"),
  head_by_lit = pl$col("s")$list$head(2)
)
```

expr_list_join *Join elements of every sub-list*

Description

Join all string items in a sub-list and place a separator between them. This only works if the inner dtype is String.

Usage

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

Arguments

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

... Dots which should be empty.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(
  s = list(c("a", "b", "c"), c("x", "y"), c("e", NA)),
  separator = c("-", "+", "/")
)
df$with_columns(
  join_with_expr = pl$col("s")$list$join(pl$col("separator")),
  join_with_lit = pl$col("s")$list$join(" "),
  join_ignore_null = pl$col("s")$list$join(" ", ignore_nulls = TRUE)
)
```

expr_list_last *Get the last value of the sub-lists*

Description

Get the last value of the sub-lists

Usage

```
expr_list_last()
```


Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(a = list(3:1, NULL, 1:2))
df$with_columns(
  last = pl$col("a")$list$last()
)
```

expr_list_len	<i>Return the number of elements in each sub-list</i>
---------------	---

Description

Null values are counted in the total.

Usage

```
expr_list_len()
```

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(list_of_strs = list(c("a", "b", NA), "c"))
df$with_columns(len_list = pl$col("list_of_strs")$list$len())
```

expr_list_max	<i>Compute the maximum value in every sub-list</i>
---------------	--

Description

Compute the maximum value in every sub-list

Usage

```
expr_list_max()
```

Value

A polars [expression](#)

Examples

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

expr_list_mean *Compute the mean value in every sub-list*

Description

Compute the mean value in every sub-list

Usage

```
expr_list_mean()
```

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA))
df$with_columns(mean = pl$col("values")$list$mean())
```

expr_list_median *Compute the median in every sub-list*

Description

Compute the median in every sub-list

Usage

```
expr_list_median()
```

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(values = list(c(-1, 0, 1), c(1, 10)))

df$with_columns(
  median = pl$col("values")$list$median()
)
```

expr_list_min	<i>Compute the minimum value in every sub-list</i>
---------------	--

Description

Compute the minimum value in every sub-list

Usage

```
expr_list_min()
```

Value

A polars [expression](#)

Examples

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

expr_list_n_unique	<i>Count the number of unique values in every sub-lists</i>
--------------------	---

Description

Count the number of unique values in every sub-lists

Usage

```
expr_list_n_unique()
```

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(values = list(c(2, 2, NA), c(1, 2, 3), NA))
df$with_columns(unique = pl$col("values")$list$n_unique())
```

expr_list_reverse *Reverse values in every sub-list*

Description

Reverse values in every sub-list

Usage

```
expr_list_reverse()
```

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA))
df$with_columns(reverse = pl$col("values")$list$reverse())
```

expr_list_sample *Sample values from every sub-list*

Description

Sample values from every sub-list

Usage

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

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)
```

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(  
  s = list(1:4, c(10L, 2L, 1L)),  
  idx = 1:2  
)  
df$with_columns(  
  shift_by_expr = pl$col("s")$list$shift(pl$col("idx")),  
  shift_by_lit = pl$col("s")$list$shift(2),  
  shift_by_negative_lit = pl$col("s")$list$shift(-2)  
)
```

expr_list_slice	<i>Slice every sub-list</i>
-----------------	-----------------------------

Description

This extracts length values at most, starting at index offset. This can return less than length values if length is larger than the number of values.

Usage

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

Arguments

offset	Start index. Negative indexing is supported. Can be an Expr. Strings are parsed as column names.
length	Length of the slice. If NULL (default), the slice is taken to the end of the list. Can be an Expr. Strings are parsed as column names.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(  
  s = list(1:4, c(10L, 2L, 1L)),  
  idx_off = 1:2,  
  len = c(4, 1)  
)  
df$with_columns(  
  slice_by_expr = pl$col("s")$list$slice("idx_off", "len"),  
  slice_by_lit = pl$col("s")$list$slice(2, 3)  
)
```

expr_list_sort	<i>Sort values in every sub-list</i>
----------------	--------------------------------------

Description

Sort values in every sub-list

Usage

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

Arguments

... Dots which should be empty.
 descending Sort values in descending order.
 nulls_last Place null values last.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(values = list(c(NA, 2, 1, 3), c(Inf, 2, 3, NaN), NA))
df$with_columns(sort = pl$col("values")$list$sort())
```

expr_list_std	<i>Compute the standard deviation in every sub-list</i>
---------------	---

Description

Compute the standard deviation in every sub-list

Usage

```
expr_list_std(ddof = 1)
```

Arguments

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

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(values = list(c(-1, 0, 1), c(1, 10)))
df$with_columns(
  std = pl$col("values")$list$std()
)
```

expr_list_sum	<i>Sum all elements in every sub-list</i>
---------------	---

Description

Sum all elements in every sub-list

Usage

```
expr_list_sum()
```

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(values = list(c(1, 2, 3, NA), c(2, 3), NA))
df$with_columns(sum = pl$col("values")$list$sum())
```

expr_list_tail	<i>Slice the last n values of every sub-list</i>
----------------	--

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

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_unique	<i>Get unique values in a list</i>
------------------	------------------------------------

Description

Get unique values in a list

Usage

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

Arguments

... Dots which should 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	<i>Compute the variance in every sub-list</i>
---------------	---

Description

Compute the variance in every sub-list

Usage

```
expr_list_var(ddof = 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_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 (default), treat pattern as a literal string, not as a regular expression.
strict	Logical. If TRUE (default), raise an error if the underlying pattern is not a valid regex, otherwise mask out with a null value.

Details

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

Value

A polars [expression](#)

See Also

- [\\$str\\$start_with\(\)](#): Check if string values start with a substring.
- [\\$str\\$ends_with\(\)](#): Check if string values end with a substring.
- [\\$str\\$find\(\)](#): Return the index position of the first substring matching a pattern.

Examples

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

df <- pl$DataFrame(txt = c("Crab", "cat and dog", "rab$bit", NA))
df$with_columns(
```

```

  regex = pl$col("txt")$str$contains("cat|bit"),
  literal = pl$col("txt")$str$contains("rab$", literal = TRUE)
)

```

expr_str_contains_any *Use the aho-corasick algorithm to find matches*

Description

This function determines if any of the patterns find a match.

Usage

```
expr_str_contains_any(patterns, ..., ascii_case_insensitive = FALSE)
```

Arguments

patterns	Character vector or something can be coerced to strings Expr of a valid regex pattern, compatible with the regex crate .
...	These dots are for future extensions and must be empty.
ascii_case_insensitive	Enable ASCII-aware case insensitive matching. When this option is enabled, searching will be performed without respect to case for ASCII letters (a-z and A-Z) only.

Value

A polars [expression](#)

See Also

- [<Expr>\\$str\\$contains\(\)](#)

Examples

```

df <- pl$DataFrame(
  lyrics = c(
    "Everybody wants to rule the world",
    "Tell me what you want, what you really really want",
    "Can you feel the love tonight"
  )
)

df$with_columns(
  contains_any = pl$col("lyrics")$str$contains_any(c("you", "me"))
)

```

 expr_str_count_matches

Count all successive non-overlapping regex matches

Description

Count all successive non-overlapping regex matches

Usage

```
expr_str_count_matches(pattern, ..., literal = FALSE)
```

Arguments

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 (default), 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	<i>Encode a value using the provided encoding</i>
-----------------	---

Description

Encode a value using the provided encoding

Usage

```
expr_str_encode(encoding)
```

Arguments

encoding	Either 'hex' or 'base64'.
----------	---------------------------

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(strings = c("foo", "bar", NA))
df$select(pl$col("strings")$str$encode("hex"))
df$with_columns(
  pl$col("strings")$str$encode("base64")$alias("base64"), # notice DataType is not encoded
  pl$col("strings")$str$encode("hex")$alias("hex") # ... and must 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	<i>Check if string ends with a regex</i>
--------------------	--

Description

Check if string values end with a substring.

Usage

```
expr_str_ends_with(sub)
```

Arguments

sub	Suffix substring or Expr.
-----	---------------------------

Details

See also `strstarts_with()` and `strcontains()`.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(fruits = c("apple", "mango", NA))
df$select(
  pl$col("fruits"),
  pl$col("fruits")$str$ends_with("go")$alias("has_suffix")
)
```

expr_str_extract *Extract the target capture group from provided patterns*

Description

Extract the target capture group from provided patterns

Usage

```
expr_str_extract(pattern, group_index)
```

Arguments

pattern	A valid regex pattern. Can be an Expr or something coercible to an Expr. Strings are parsed as column names.
group_index	Index of the targeted capture group. Group 0 means the whole pattern, first group begin at index 1 (default).

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(  
  a = c(  
    "http://vote.com/ballon_dor?candidate=messi&ref=polars",  
    "http://vote.com/ballon_dor?candidate=jorginho&ref=polars",  
    "http://vote.com/ballon_dor?candidate=ronaldo&ref=polars"  
  )  
)  
df$with_columns(  
  extracted = pl$col("a")$str$extract(pl$lit(r"(candidate=(\w+))"), 1)  
)
```

expr_str_extract_all *Extract all matches for the given regex pattern*

Description

Extracts all matches for the given regex pattern. Extracts each successive non-overlapping regex match in an individual string as an array.

Usage

```
expr_str_extract_all(pattern)
```

Arguments

pattern A valid regex pattern

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(foo = c("123 bla 45 asd", "xyz 678 910t"))
df$select(
  pl$col("foo")$str$extract_all(r"((\d+))")$alias("extracted_nrs")
)
```

expr_str_extract_groups

Extract all capture groups for the given regex pattern

Description

Extract all capture groups for the given regex pattern

Usage

```
expr_str_extract_groups(pattern)
```

Arguments

pattern A character of a valid regular expression pattern containing at least one capture group, compatible with the [regex crate](#).

Details

All group names are strings. If your pattern contains unnamed groups, their numerical position is converted to a string. See examples.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(
  url = c(
    "http://vote.com/ballon_dor?candidate=messi&ref=python",
    "http://vote.com/ballon_dor?candidate=weghorst&ref=polars",
    "http://vote.com/ballon_dor?error=404&ref=rust"
  )
)
```

```

pattern <- r"(candidate=(?<candidate>\w*)&ref=(?<ref>\w*))"

df$with_columns(
  captures = pl$col("url")$str$extract_groups(pattern)
)$unnest("captures")

# If the groups are unnamed, their numerical position (as a string) is used:

pattern <- r"(candidate=(\w*)&ref=(\w*))"

df$with_columns(
  captures = pl$col("url")$str$extract_groups(pattern)
)$unnest("captures")

```

expr_str_extract_many *Use the aho-corasick algorithm to extract matches*

Description

Use the aho-corasick algorithm to extract matches

Usage

```

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

```

Arguments

patterns	String patterns to search. This can be an Expr or something coercible to an Expr. Strings are parsed as column names.
...	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.

Value

A polars [expression](#)

Examples

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

df$with_columns(
  matches = pl$col("values")$str$extract_many(patterns),
  matches_overlap = pl$col("values")$str$extract_many(patterns, overlapping = TRUE)
)

df <- pl$DataFrame(
  values = c("discontent", "rhapsody"),
  patterns = list(c("winter", "disco", "onte", "discontent"), c("rhap", "ody", "coalesce"))
)

df$select(pl$col("values")$str$extract_many("patterns"))
```

`expr_str_find`*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

<code>pattern</code>	A character or something can be coerced to a string Expr of a valid regex pattern, compatible with the regex crate .
<code>...</code>	These dots are for future extensions and must be empty.
<code>literal</code>	Logical. If TRUE (default), treat <code>pattern</code> as a literal string, not as a regular expression.
<code>strict</code>	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_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 **nth** from the end of the string. For example, if **n** = -3, then all characters except the last three are returned.

If the length of the string has fewer than **n** characters, the full string is returned.

Value

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	<i>Vertically concatenate the string values in the column to a single string value.</i>
---------------	---

Description

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

Usage

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

Arguments

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.

Usage

```
expr_str_json_decode(dtype, infer_schema_length = 100)
```

Arguments

`dtype` The dtype to cast the extracted value to. If NULL, the dtype will be inferred from the JSON value.

`infer_schema_length` How many rows to parse to determine the schema. If NULL, all rows are used.

Details

Throw errors if encounter invalid json strings.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(  
  json_val = c('{\"a\":1, \"b\": true}', NA, '{\"a\":2, \"b\": false}')  
)  
dtype <- pl$Struct(pl$field("a", pl$Int64), pl$field("b", pl$Boolean))  
df$select(pl$col("json_val")$str$json_decode(dtype))
```

expr_str_json_path_match

Extract the first match of JSON string with the provided 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 `strlen_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 `strlen_bytes()` to get the number of bytes.

Usage

```
expr_str_len_chars()
```

Details

If you know that you are working with ASCII text, lengths will be equivalent, and faster (returns length in terms of the number of bytes).

Value

A polars [expression](#)

Examples

```
pl$DataFrame(  
  s = c("Café", NA, "345", "æøå")  
)$select(  
  pl$col("s"),  
  pl$col("s")$str$len_bytes()$alias("lengths"),  
  pl$col("s")$str$len_chars()$alias("n_chars")  
)
```

expr_str_pad_end *Left justify strings*

Description

Return the string left justified in a string of length width.

Usage

```
expr_str_pad_end(width, fillchar = " ")
```

Arguments

width	Justify left to this length.
fillchar	Fill with this ASCII character.

Details

Padding is done using the specified fillchar. The original string is returned if width is less than or equal to len(s).

Value

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 width.

Usage

```
expr_str_pad_start(width, fillchar = " ")
```

Arguments

width	Justify right to this length.
fillchar	Fill with this ASCII character.

Details

Padding is done using the specified fillchar. The original string is returned if width is less than or equal to len(s).

Value

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	<i>Replace first matching regex/literal substring with a new string value</i>
------------------	---

Description

Replace first matching regex/literal substring with a new string value

Usage

```
expr_str_replace(pattern, value, ..., literal = FALSE, n = 1L)
```

Arguments

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 (default), treat pattern as a literal string, not as a regular expression.
n	A number of matches to replace. Note that regex replacement with $n > 1$ not yet supported, so raise an error if $n > 1$ and pattern includes regex pattern and <code>literal = FALSE</code> .

Details

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

Value

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>strreplace_all()`

Examples

```
df <- pl$DataFrame(id = 1L:2L, text = c("123abc", "abc456"))
df$with_columns(pl$col("text")$str$replace(r"(abc\b)", "ABC"))

# Capture groups are supported.
# Use `${1}` in the value string to refer to the first capture group in the pattern,
# `${2}` to refer to the second capture group, and so on.
# You can also use named capture groups.
df <- pl$DataFrame(word = c("hat", "hut"))
df$with_columns(
  positional = pl$col("word")$str$replace("h(.)t", "b${1}d"),
  named = pl$col("word")$str$replace("h(<vowel>.)t", "b{vowel}d")
)

# Apply case-insensitive string replacement using the `(?i)` flag.
df <- pl$DataFrame(
  city = "Philadelphia",
  season = c("Spring", "Summer", "Autumn", "Winter"),
  weather = c("Rainy", "Sunny", "Cloudy", "Snowy")
)
df$with_columns(
  pl$col("weather")$str$replace("(?i)foggy|rainy|cloudy|snowy", "Sunny")
)
```

`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 (default), treat pattern as a literal string, not as a regular expression.

Details

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

Value

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$DataFrame(
  city = "Philadelphia",
  season = c("Spring", "Summer", "Autumn", "Winter"),
  weather = c("Rainy", "Sunny", "Cloudy", "Snowy")
)
df$with_columns(
  pl$col("weather")$str$replace_all(
```

```

    "(?i)foggy|rainy|cloudy|snowy", "Sunny"
  )
)

```

expr_str_replace_many *Use the aho-corasick algorithm to replace many matches*

Description

This function replaces several matches at once.

Usage

```
expr_str_replace_many(patterns, replace_with, ascii_case_insensitive = FALSE)
```

Arguments

patterns	String patterns to search. Can be an Expr.
replace_with	A vector of strings used as replacements. If this is of length 1, then it is applied to all matches. Otherwise, it must be of same length as the patterns argument.
ascii_case_insensitive	Enable ASCII-aware case insensitive matching. When this option is enabled, searching will be performed without respect to case for ASCII letters (a-z and A-Z) only.

Value

A polars [expression](#)

Examples

```

df <- pl$DataFrame(
  lyrics = c(
    "Everybody wants to rule the world",
    "Tell me what you want, what you really really want",
    "Can you feel the love tonight"
  )
)

# a replacement of length 1 is applied to all matches
df$with_columns(
  remove_pronouns = pl$col("lyrics")$str$replace_many(c("you", "me"), "")
)

# if there are more than one replacement, the patterns and replacements are
# matched
df$with_columns(
  fake_pronouns = pl$col("lyrics")$str$replace_many(c("you", "me"), c("foo", "bar"))
)

```

expr_str_reverse	<i>Returns string values in reversed order</i>
------------------	--

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	<i>Create subslices of the string values of a String Series</i>
----------------	---

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)
```

Arguments

by Substring to split by. Can be an Expr.
inclusive If TRUE, include the split character/string in the results.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(s = c("foo bar", "foo-bar", "foo bar baz"))
df$select(pl$col("s")$str$split(by = " "))

df <- pl$DataFrame(
  s = c("foo^bar", "foo_bar", "foo*bar*baz"),
  by = c("_", "_", "*")
)
df
df$select(pl$col("s")$str$split(by = pl$col("by"))$alias("split"))
```

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.
n Number of splits to make.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(s = c("a_1", NA, "c", "d_4_e"))
df$with_columns(
  s1 = pl$col("s")$str$splitn(by = "_", 1),
  s2 = pl$col("s")$str$splitn(by = "_", 2),
  s3 = pl$col("s")$str$splitn(by = "_", 3)
)
```

expr_str_split_exact *Split the string by a substring using n splits*

Description

This results in a struct of n+1 fields. If it cannot make n splits, the remaining field elements will be null.

Usage

```
expr_str_split_exact(by, n, inclusive = FALSE)
```

Arguments

by Substring to split by.
n Number of splits to make.
inclusive If TRUE, include the split character/string in the results.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(s = c("a_1", NA, "c", "d_4"))
df$with_columns(
  split = pl$col("s")$str$split_exact(by = "_", 1),
  split_inclusive = pl$col("s")$str$split_exact(by = "_", 1, inclusive = TRUE)
)
```

expr_str_starts_with *Check if string starts with a regex*

Description

Check if string values starts with a substring.

Usage

```
expr_str_starts_with(sub)
```

Arguments

sub Prefix substring or Expr.

Details

See also `strcontains()` and `strends_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(matches = NULL)
```

Arguments

matches The set of characters to be removed. All combinations of this set of characters will be stripped. If NULL (default), all whitespace is removed instead. This can be an Expr.

Details

This function will not strip any chars beyond the first char not matched. `strip_chars()` removes characters at the beginning and the end of the string. Use `strip_chars_start()` and `strip_chars_end()` to remove characters only from left and right respectively.

Value

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(matches = NULL)
```

Arguments

matches	The set of characters to be removed. All combinations of this set of characters will be stripped. If NULL (default), all whitespace is removed instead. This can be an Expr.
---------	--

Details

This function will not strip any chars beyond the first char not matched. `strip_chars_end()` removes characters at the end of the string only. Use `strip_chars()` and `strip_chars_start()` to remove characters from the left and right or only from the left respectively.

Value

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(matches = NULL)
```

Arguments

matches	The set of characters to be removed. All combinations of this set of characters will be stripped. If NULL (default), all whitespace is removed instead. This can be an Expr.
---------	--

Details

This function will not strip any chars beyond the first char not matched. `strip_chars_start()` removes characters at the beginning of the string only. Use `strip_chars()` and `strip_chars_end()` to remove characters from the left and right or only from the right respectively.

Value

A polars [expression](#)

Examples

```
df <- pl$DataFrame(foo = c(" hello", "\tworld"))
df$select(pl$col("foo")$str$strip_chars_start(" hel rld"))
```

expr_str_strptime *Convert a String column into a Date/Datetime/Time column.*

Description

Similar to the [strptime\(\)](#) function.

Usage

```

expr_str_strptime(
  dtype,
  format = NULL,
  ...,
  strict = TRUE,
  exact = TRUE,
  cache = TRUE,
  ambiguous = c("raise", "earliest", "latest", "null")
)

```

Arguments

<code>dtype</code>	The data type to convert into. Can be either <code>pl\$Date</code> , <code>pl\$Datetime</code> , or <code>pl\$Time</code> .
<code>format</code>	Format to use for conversion. Refer to the chrono crate documentation for the full specification. Example: <code>"%Y-%m-%d %H:%M:%S"</code> . If <code>NULL</code> (default), the format is inferred from the data. Notice that time zone <code>%Z</code> is not supported and will just ignore timezones. Numeric time zones like <code>%z</code> or <code>:%:z</code> are supported.
<code>...</code>	These dots are for future extensions and must be empty.
<code>strict</code>	If <code>TRUE</code> (default), raise an error if a single string cannot be parsed. If <code>FALSE</code> , produce a polars <code>null</code> .
<code>exact</code>	If <code>TRUE</code> (default), require an exact format match. If <code>FALSE</code> , allow the format to match anywhere in the target string. Conversion to the <code>Time</code> type is always exact. Note that using <code>exact = FALSE</code> introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.
<code>cache</code>	Use a cache of unique, converted dates to apply the datetime conversion.
<code>ambiguous</code>	Determine how to deal with ambiguous datetimes. Character vector or expression containing the followings: <ul style="list-style-type: none"> • <code>"raise"</code> (default): Throw an error • <code>"earliest"</code>: Use the earliest datetime • <code>"latest"</code>: Use the latest datetime • <code>"null"</code>: 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\(\)](#)

Examples

```
# Dealing with a consistent format
df <- pl$DataFrame(x = c("2020-01-01 01:00Z", "2020-01-01 02:00Z"))

df$select(pl$col("x")$str$strptime(pl$Datetime(), "%Y-%m-%d %H:%M%#Z"))

# Auto infer format
df$select(pl$col("x")$str$strptime(pl$Datetime()))

# Datetime with timezone is interpreted as UTC timezone
df <- pl$DataFrame(x = c("2020-01-01T01:00:00+09:00"))
df$select(pl$col("x")$str$strptime(pl$Datetime()))

# Dealing with different formats.
df <- pl$DataFrame(
  date = c(
    "2021-04-22",
    "2022-01-04 00:00:00",
    "01/31/22",
    "Sun Jul 8 00:34:60 2001"
  )
)

df$select(
  pl$coalesce(
    pl$col("date")$str$strptime(pl$Date, "%F", strict = FALSE),
    pl$col("date")$str$strptime(pl$Date, "%F %T", strict = FALSE),
    pl$col("date")$str$strptime(pl$Date, "%D", strict = FALSE),
    pl$col("date")$str$strptime(pl$Date, "%c", strict = FALSE)
  )
)

# Ignore invalid time
df <- pl$DataFrame(
  x = c(
    "2023-01-01 11:22:33 -0100",
    "2023-01-01 11:22:33 +0300",
    "invalid time"
  )
)

df$select(pl$col("x")$str$strptime(
  pl$Datetime("ns"),
  format = "%Y-%m-%d %H:%M:%S %Z",
  strict = FALSE
))
```



```
))
```

expr_str_tail	<i>Return the last n characters of each string</i>
---------------	--

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 nth from the beginning of the string. For example, if n = -3, then all characters except the first three are returned.

If the length of the string has fewer than n characters, the full string is returned.

Value

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" => <code>pl\$Datetime("ms")</code> . 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: <ul style="list-style-type: none"> "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\(\)](#)

Examples

```
df <- pl$DataFrame(x = c("2020-01-01 01:00Z", "2020-01-01 02:00Z"))

df$select(pl$col("x")$str$to_datetime("%Y-%m-%d %H:%M%Z"))
df$select(pl$col("x")$str$to_datetime(time_unit = "ms"))
```

expr_str_to_integer *Convert a String column into an Int64 column with base radix*

Description

Convert a String column into an Int64 column with base radix

Usage

```
expr_str_to_integer(..., base = 10L, strict = TRUE)
```

Arguments

...	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.
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, 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$lit(c("A", "b", "c", "1", NA))$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_uppercase *Convert a string to uppercase*

Description

Transform to uppercase variant.

Usage

```
expr_str_to_uppercase()
```

Value

A polars [expression](#)

Examples

```
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(alignment)
```

Arguments

alignment Fill the value up to this length. This can be an Expr or something coercible to an Expr. Strings are parsed as column names.

Details

Return a copy of the string left filled with ASCII '0' digits to make a string of length width.

A leading sign prefix ('+'/'-') is handled by inserting the padding after the sign character rather than before. The original string is returned if width is less than or equal to len(s).

Value

A polars [expression](#)

Examples

```
some_floats_expr <- pl$lit(c(0, 10, -5, 5))

# cast to String and ljust alignment = 5, and view as R char vector
some_floats_expr$cast(pl$String)$str$zfill(5)$to_r()

# cast to int and the to utf8 and then ljust alignment = 5, and view as R
# char vector
some_floats_expr$cast(pl$Int64)$cast(pl$String)$str$zfill(5)$to_r()
```

lazyframe__collect *Materialize this LazyFrame into a DataFrame*

Description

By default, all query optimizations are enabled. Individual optimizations may be disabled by setting the corresponding parameter to FALSE.

Usage

```
lazyframe__collect(
  ...,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  comm_subplan_elim = TRUE,
  comm_subexpr_elim = TRUE,
  cluster_with_columns = TRUE,
  no_optimization = FALSE,
  streaming = FALSE,
  `_eager` = FALSE
)
```

Arguments

...	These dots are for future extensions and must be empty.
type_coercion	A logical, indicates type coercion optimization.
predicate_pushdown	A logical, indicates predicate pushdown optimization.
projection_pushdown	A logical, indicates projection pushdown optimization.
simplify_expression	A logical, indicates simplify expression optimization.
slice_pushdown	A logical, indicates slice pushdown optimization.
comm_subplan_elim	A logical, indicates try to cache branching subplans that occur on self-joins or unions.
comm_subexpr_elim	A logical, indicates try to cache common subexpressions.
cluster_with_columns	A logical, indicates to combine sequential independent calls to with_columns.
no_optimization	A logical. If TRUE, turn off (certain) optimizations.
streaming	A logical. If TRUE, process the query in batches to handle larger-than-memory data. If FALSE (default), the entire query is processed in a single batch. Note that streaming mode is considered unstable. It may be changed at any point without it being considered a breaking change.
_eager	A logical, indicates to turn off multi-node optimizations and the other optimizations. This option is intended for internal use only.

Value

A polars [DataFrame](#)

Examples

```
lf <- pl$LazyFrame(
  a = c("a", "b", "a", "b", "b", "c"),
  b = 1:6,
  c = 6:1,
)
lf$group_by("a")$agg(pl$all()$sum())$collect()

# Collect in streaming mode
lf$group_by("a")$agg(pl$all()$sum())$collect(
  streaming = TRUE
)
```

lazyframe__select	<i>Select and modify columns of a LazyFrame</i>
-------------------	---

Description

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

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

Usage

```
lazyframe__select(...)
```

Arguments

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

Value

A polars [LazyFrame](#)

Examples

```
# Pass the name of a column to select that column.
lf <- pl$LazyFrame(
  foo = 1:3,
  bar = 6:8,
  ham = letters[1:3]
)
lf$select("foo")$collect()

# Multiple columns can be selected by passing a list of column names.
lf$select("foo", "bar")$collect()

# Expressions are also accepted.
lf$select(pl$col("foo"), pl$col("bar") + 1)$collect()

# Name expression (used as the column name of the output DataFrame)
lf$select(
  threshold = pl$when(pl$col("foo") > 2)$then(10)$otherwise(0)
)$collect()

# Expressions with multiple outputs can be automatically instantiated
# as Structs by setting the `POLARS_AUTO_STRUCTIFY` environment variable.
```

```
# (Experimental)
if (requireNamespace("withr", quietly = TRUE)) {
  withr::with_envvar(c(POLARS_AUTO_STRUCTIFY = "1"), {
    lf$select(
      is_odd = ((pl$col(pl$Int32) %% 2) == 1)$name$suffix("_is_odd"),
    )$collect()
  })
}
```

lazyframe__with_columns

Modify/append column(s) of a LazyFrame

Description

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

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

Usage

```
lazyframe__with_columns(...)
```

Arguments

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

Value

A polars [LazyFrame](#)

Examples

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

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

```

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

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

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

```

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
```

Format

An object of class `polars_object` of length 43.

Examples

```

pl

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

# Create a polars DataFrame
# In Python:

```

```
# ```python
# >>> import polars as pl
# >>> df = pl.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
# ```
# In R:
df <- pl$DataFrame(a = c(1, 2, 3), b = c(4, 5, 6))
df
```

```
pl_api_register_series_namespace
```

Registering custom functionality with a polars Series

Description

Registering custom functionality with a polars Series

Usage

```
pl_api_register_series_namespace(name, ns_fn)
```

Arguments

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")

```

pl_col

Create an expression representing column(s) in a DataFrame

Description

Create an expression representing column(s) in a DataFrame

Usage

```
pl_col(...)
```

Arguments

... [<dynamic-dots>](#) The name or datatype of the column(s) to represent. Unnamed objects one of the following:

- character vectors
 - Single wildcard "*" has a special meaning: check the examples.
- (lists of) polars_data_type objects

Value

A polars [expression](#)

Examples

```

# a single column by a character
pl$col("foo")

# multiple columns by characters
pl$col("foo", "bar")

# multiple columns by polars data types
pl$col(pl$Float64, pl$String)

# Single "*" is converted to a wildcard expression
pl$col("*")

# multiple character vectors and a list of polars data types are also allowed
pl$col(c("foo", "bar"), "baz")

```

```

pl$col("foo", c("bar", "baz"))
pl$col(list(pl$Float64, pl$String), list(pl$UInt32))

# 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__DataFrame	<i>Polars DataFrame class</i> (polars_data_frame)
---------------	---

Description

DataFrames are two-dimensional data structure representing data as a table with rows and columns. Polars DataFrames are similar to [R Data Frames](#). R Data Frame's columns are [R vectors](#), while Polars DataFrame's columns are [Polars Series](#).

Usage

```
pl__DataFrame(..., .schema_overrides = NULL, .strict = TRUE)
```

Arguments

... [<dynamic-dots>](#) Name-value pairs of objects to be converted to polars [Series](#) by the [as_polars_series\(\)](#) function. Each [Series](#) will be used as a column of the [DataFrame](#). All values must be the same length. Each name will be used as the column name. If the name is empty, the original name of the [Series](#) will be used.

.schema_overrides **[Experimental]** A list of polars data types or NULL (default). Passed to the [\\$cast\(\)](#) method as dynamic-dots.

.strict **[Experimental]** A logical value. Passed to the [\\$cast\(\)](#) method's .strict argument.

Details

The `pl$DataFrame()` function mimics the constructor of the `DataFrame` class of Python Polars. This function is basically a shortcut for `as_polars_df(list(...))$cast(!!!.schema_overrides, .strict = .strict)`, so each argument in ... is converted to a Polars Series by [as_polars_series\(\)](#) and then passed to [as_polars_df\(\)](#).

Value

A polars [DataFrame](#)

Active bindings

- `columns`: `$columns` returns a character vector with the names of the columns.
- `dtypes`: `$dtypes` returns a nameless list of the data type of each column.
- `schema`: `$schema` returns a named list with the column names as names and the data types as values.
- `shape`: `$shape` returns a integer vector of length two with the number of rows and columns of the DataFrame.
- `height`: `$height` returns a integer with the number of rows of the DataFrame.
- `width`: `$width` returns a integer with the number of columns of the DataFrame.

Examples

```
# Constructing a DataFrame from vectors:
pl$DataFrame(a = 1:2, b = 3:4)

# Constructing a DataFrame from Series:
pl$DataFrame(pl$Series("a", 1:2), pl$Series("b", 3:4))

# Constructing a DataFrame from a list:
data <- list(a = 1:2, b = 3:4)

## Using the as_polars_df function (recommended)
as_polars_df(data)

## Using dynamic dots feature
pl$DataFrame(!!!data)

# Active bindings:
df <- pl$DataFrame(a = 1:3, b = c("foo", "bar", "baz"))

df$columns
df$dtypes
df$schema
df$shape
df$height
df$width
```

pl_datetime_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.
...	Dots which should 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.
...	Dots which should 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"
  )
)
```

<code>pl__date_range</code>	<i>Generate a date range</i>
-----------------------------	------------------------------

Description

If both start and end are passed as the Date types (not Datetime), and the interval granularity is no finer than "1d", the returned range is also of type Date. All other permutations return a Datetime.

Usage

```
pl__date_range(
  start,
  end,
  interval = "1d",
  ...,
  closed = 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.
...	Dots which should be empty.
closed	Define which sides of the range are closed (inclusive). One of the following: "both" (default), "left", "right", "none".

Value

An [Expr](#) of data type Date or [Datetime](#)

Polars duration string language

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

It has the following format:

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

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

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

See Also

[pl\\$date_ranges\(\)](#) to create a simple Series of data type list(Date) based on column values.

Examples

```
# Using Polars duration string to specify the interval:
pl$select(
  date = pl$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

<code>start</code>	Lower bound of the date range. Something that can be coerced to a <code>Date</code> or a Datetime expression. See examples for details.
<code>end</code>	Upper bound of the date range. Something that can be coerced to a <code>Date</code> or a Datetime expression. See examples for details.
<code>interval</code>	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.
<code>...</code>	Dots which should be empty.
<code>closed</code>	Define which sides of the range are closed (inclusive). One of the following: <code>"both"</code> (default), <code>"left"</code> , <code>"right"</code> , <code>"none"</code> .

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__LazyFrame	<i>Polars LazyFrame class (polars_lazy_frame)</i>
---------------	---

Description

Representation of a Lazy computation graph/query against a [DataFrame](#). This allows for whole-query optimisation in addition to parallelism, and is the preferred (and highest-performance) mode of operation for polars.

Usage

```
pl__LazyFrame(..., .schema_overrides = NULL, .strict = TRUE)
```

Arguments

...	<dynamic-dots> Name-value pairs of objects to be converted to polars Series by the as_polars_series() function. Each Series will be used as a column of the DataFrame . All values must be the same length. Each name will be used as the column name. If the name is empty, the original name of the Series will be used.
.schema_overrides	[Experimental] A list of polars data types or NULL (default). Passed to the \$cast() method as dynamic-dots.
.strict	[Experimental] A logical value. Passed to the \$cast() method's .strict argument.

Details

The `pl$LazyFrame(...)` function is a shortcut for `pl$DataFrame(...)$lazy()`.

Value

A polars [LazyFrame](#)

See Also

- [<LazyFrame>\\$collect\(\)](#): Materialize a [LazyFrame](#) into a [DataFrame](#).

Examples

```
# Constructing a LazyFrame from vectors:
pl$LazyFrame(a = 1:2, b = 3:4)

# Constructing a LazyFrame from Series:
pl$LazyFrame(pl$Series("a", 1:2), pl$Series("b", 3:4))

# Constructing a LazyFrame from a list:
data <- list(a = 1:2, b = 3:4)
```

```
## Using dynamic dots feature
pl$LazyFrame(!!!data)
```

pl_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 <code>as_polars_expr()</code> .
dtype	A polars data type or NULL (default). If not NULL, casted to the specified data type.

Value

A polars [expression](#)

Literal scalar mapping

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

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

These types' NA is converted to a null literal with casting to the corresponding Polars type.

The [raw](#) type vector is converted to a Binary scalar.

- raw: Binary

NULL is converted to a Null type null literal.

- NULL: Null

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

See Also

- [as_polars_series\(\)](#): R -> Polars type mapping is mostly defined by this function.
- [as_polars_expr\(\)](#): Internal implementation of pl\$lit().

Examples

```
# Literal scalar values
pl$lit(1L)
pl$lit(5.5)
pl$lit(NULL)
pl$lit("foo_bar")

## Generally, for a vector (an R object) becomes a Series with length 1,
## it is converted to a Series and then get the first value to become a scalar literal.
pl$lit(as.Date("2021-01-20"))
pl$lit(as.POSIXct("2023-03-31 10:30:45"))
pl$lit(data.frame(a = 1, b = "foo"))

# Literal Series data
pl$lit(1:3)
pl$lit(pl$Series("x", 1:3))
```

pl__Series	<i>Polars Series class (polars_series)</i>
------------	--

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

These packages may be used for exporting [Series](#) to R. See `<Series>$to_r_vector()` for details.

- [bit64](#)
- [blob](#)
- [clock](#)
- [data.table](#)
- [hms](#)
- [tibble](#)
- [vctrs](#)

Value

NULL invisibly.

Examples

```
pl$show_versions()
```

pl__struct	<i>Collect columns into a struct column</i>
------------	---

Description

Collect columns into a struct column

Usage

```
pl__struct(...)
```

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 [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"))

# Name each struct field.
df$select(pl$struct(p = "int", q = "bool")$alias("my_struct"))$schema
```

polars_duration_string

The Polars duration string language

Description

The Polars duration string language

Polars duration string language

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

It has the following format:

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

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

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

polars_expr

Polars expression class (polars_expr)

Description

An expression is a tree of operations that describe how to construct one or more [Series](#). As the outputs are [Series](#), it is straightforward to apply a sequence of expressions each of which transforms the output from the previous step. See examples for details.

See Also

- `pl$lit()`: Create a literal expression.
- `pl$col()`: Create an expression representing column(s) in a [DataFrame](#).

Examples

```
# An expression:
# 1. Select column `foo`,
# 2. Then sort the column (not in reversed order)
# 3. Then take the first two values of the sorted output
pl$col("foo")$sort()$head(2)

# Expressions will be evaluated inside a context, such as `<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
)
```

series_struct_unnest *Convert this struct Series to a DataFrame with a separate column for each field*

Description

Convert this struct Series to a DataFrame with a separate column for each field

Usage

```
series_struct_unnest()
```

Value

A polars [DataFrame](#)

See Also

- `as_polars_df()`

Examples

```
s <- as_polars_series(data.frame(a = c(1, 3), b = c(2, 4)))
s$struct$unnest()
```

series__to_frame *Cast this Series to a DataFrame*

Description

Cast this Series to a DataFrame

Usage

```
series__to_frame(name = NULL)
```

Arguments

name A character or NULL. If not NULL, name/rename the [Series](#) column in the new [DataFrame](#). If NULL, the column name is taken from the [Series](#) name.

Value

A polars [DataFrame](#)

See Also

- [as_polars_df\(\)](#)

Examples

```
s <- pl$Series("a", c(123, 456))
df <- s$to_frame()
df

df <- s$to_frame("xyz")
df
```

series__to_r_vector *Export the Series as an R vector*

Description

Export the [Series](#) as an R [vector](#). But note that the Struct data type is exported as a [data.frame](#) by default for consistency, and a [data.frame](#) is not a vector. If you want to ensure the return value is a [vector](#), please set `ensure_vector = TRUE`, or use the [as.vector\(\)](#) function instead.

Usage

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

Arguments

...	These dots are for future extensions and must be empty.
ensure_vector	A logical value indicating whether to ensure the return value is a vector . When the Series has the Struct data type and this argument is FALSE (default), the return value is a data.frame , not a vector (<code>is.vector(<data.frame>)</code> is FALSE). If TRUE, return a named list instead of a data.frame .
int64	Determine how to convert Polars' Int64, UInt32, or UInt64 type values to R type. One of the followings: <ul style="list-style-type: none"> "double" (default): Convert to the R's double type. Accuracy may be degraded. "character": Convert to the R's character type. "integer": Convert to the R's integer type. If the value is out of the range of R's integer type, export as NA_integer_. "integer64": Convert to the bit64::integer64 class. The bit64 package must be installed. If the value is out of the range of bit64::integer64, export as bit64::NA_integer64_.
date	Determine how to convert Polars' Date type values to R class. One of the followings: <ul style="list-style-type: none"> "Date" (default): Convert to the R's Date class. "IDate": Convert to the data.table::IDate class.
time	Determine how to convert Polars' Time type values to R class. One of the followings: <ul style="list-style-type: none"> "hms" (default): Convert to the hms::hms class. "ITime": Convert to the data.table::ITime class. The data.table package must be installed.
struct	Determine how to convert Polars' Struct type values to R class. One of the followings: <ul style="list-style-type: none"> "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	Determine how to convert Polars' Decimal type values to R type. One of the followings: <ul style="list-style-type: none"> • "double" (default): Convert to the R's <code>double</code> type. • "character": Convert to the R's <code>character</code> type.
as_clock_class	A logical value indicating whether to export datetimes and duration as the <code>clock</code> package's classes. <ul style="list-style-type: none"> • FALSE (default): Duration values are exported as <code>difftime</code> and datetime values are exported as <code>POSIXct</code>. Accuracy may be degraded. • TRUE: Duration values are exported as <code>clock_duration</code>, datetime without timezone values are exported as <code>clock_naive_time</code>, and datetime with timezone values are exported as <code>clock_zoned_time</code>. For this case, the <code>clock</code> package must be installed. Accuracy will be maintained.
ambiguous	Determine how to deal with ambiguous datetimes. Only applicable when <code>as_clock_class</code> is set to FALSE and datetime without timezone values are exported as <code>POSIXct</code> . Character vector or <code>expression</code> containing the followings: <ul style="list-style-type: none"> • "raise" (default): Throw an error • "earliest": Use the earliest datetime • "latest": Use the latest datetime • "null": Return a NA value
non_existent	Determine how to deal with non-existent datetimes. Only applicable when <code>as_clock_class</code> is set to FALSE and datetime without timezone values are exported as <code>POSIXct</code> . One of the followings: <ul style="list-style-type: none"> • "raise" (default): Throw an error • "null": Return a NA value

Details

The class/type of the exported object depends on the data type of the Series as follows:

- Boolean: `logical`.
- UInt8, UInt16, Int8, Int16, Int32: `integer`.
- Int64, UInt32, UInt64: `double`, `character`, `integer`, or `bit64::integer64`, depending on the `int64` argument.
- Float32, Float64: `double`.
- Decimal: `double`.
- String: `character`.
- Categorical: `factor`.
- Date: `Date` or `data.table::IDate`, depending on the `date` argument.
- Time: `hms::hms` or `data.table::ITime`, depending on the `time` argument.
- Datetime (without timezone): `POSIXct` or `clock_naive_time`, depending on the `as_clock_class` argument.
- Datetime (with timezone): `POSIXct` or `clock_zoned_time`, depending on the `as_clock_class` argument.

- Duration: `difftime` or `clock_duration`, depending on the `as_clock_class` argument.
- Binary: `blob::blob`.
- Null: `vctrs::unspecified`.
- List, Array: `vctrs::list_of`.
- Struct: `data.frame` or `tibble`, depending on the `struct` argument. If `ensure_vector = TRUE`, the top-level Struct is exported as a named `list` for to ensure the return value is a `vector`.

Value

A `vector`

Examples

```
# Struct values handling
series_struct <- as_polars_series(
  data.frame(
    a = 1:2,
    b = I(list(data.frame(c = "foo"), data.frame(c = "bar")))
  )
)
series_struct

## Export Struct as data.frame
series_struct$to_r_vector()

## Export Struct as data.frame,
## but the top-level Struct is exported as a named list
series_struct$to_r_vector(ensure_vector = TRUE)

## Export Struct as tibble
series_struct$to_r_vector(struct = "tibble")

## Export Struct as tibble,
## but the top-level Struct is exported as a named list
series_struct$to_r_vector(struct = "tibble", ensure_vector = TRUE)

# Integer values handling
series_uint64 <- as_polars_series(
  c(NA, "0", "4294967295", "18446744073709551615")
)$cast(pl$UInt64)
series_uint64

## Export UInt64 as double
series_uint64$to_r_vector(int64 = "double")

## Export UInt64 as character
series_uint64$to_r_vector(int64 = "character")

## Export UInt64 as integer (overflow occurs)
series_uint64$to_r_vector(int64 = "integer")
```

```
## Export UInt64 as bit64::integer64 (overflow occurs)
if (requireNamespace("bit64", quietly = TRUE)) {
  series_uint64$to_r_vector(int64 = "integer64")
}

# Duration values handling
series_duration <- as_polars_series(
  c(NA, -1000000000, -10, -1, 1000000000)
)$cast(pl$Duration("ns"))
series_duration

## Export Duration as difftime
series_duration$to_r_vector(as_clock_class = FALSE)

## Export Duration as clock_duration
if (requireNamespace("clock", quietly = TRUE)) {
  series_duration$to_r_vector(as_clock_class = TRUE)
}

# Datetime values handling
series_datetime <- as_polars_series(
  as.POSIXct(
    c(NA, "1920-01-01 00:00:00", "1970-01-01 00:00:00", "2020-01-01 00:00:00"),
    tz = "UTC"
  )
)$cast(pl$Datetime("ns", "UTC"))
series_datetime

## Export zoned datetime as POSIXct
series_datetime$to_r_vector(as_clock_class = FALSE)

## Export zoned datetime as clock_zoned_time
if (requireNamespace("clock", quietly = TRUE)) {
  series_datetime$to_r_vector(as_clock_class = TRUE)
}
```

Index

* datasets

- cs, [27](#)
- pl, [147](#)
- <DataFrame>\$get_columns(), [10](#)
- <DataFrame>\$partition_by(), [32](#)
- <DataFrame>\$to_struct(), [20](#)
- <Expr>\$str\$contains(), [111](#)
- <Expr>\$str\$replace(), [127](#)
- <Expr>\$str\$replace_all(), [126](#)
- <Expr>\$str\$strptime(), [138, 140, 142](#)
- <Expr>\$str\$to_date(), [136](#)
- <Expr>\$str\$to_datetime(), [136](#)
- <Expr>\$str\$to_time(), [136](#)
- <LazyFrame>\$collect(), [12, 159](#)
- <Series>\$struct\$unnest(), [11, 12](#)
- <Series>\$to_frame(), [12](#)
- <Series>\$to_r_vector(), [10, 20, 162](#)
- \$cast(), [150, 159](#)
- \$dt\$base_utc_offset(), [62](#)
- \$dt\$convert_time_zone(), [71](#)
- \$dt\$dst_offset(), [57](#)
- \$list\$gather(), [94](#)
- \$list\$get(), [93](#)
- \$set_difference(), [102](#)
- \$set_symmetric_difference(), [101](#)
- \$str\$contains(), [119](#)
- \$str\$ends_with(), [110, 119](#)
- \$str\$find(), [110](#)
- \$str\$start_with(), [110, 119](#)
- \$to_frame(), [12](#)

- abort(), [26](#)
- as.data.frame(<polars_data_frame>), [13](#)
- as.data.frame(<polars_object>), [24](#)
- as.data.frame.polars_data_frame, [6](#)
- as.data.frame.polars_lazy_frame
(as.data.frame.polars_data_frame),
[6](#)
- as.list(<polars_data_frame>), [13, 31](#)
- as.list.polars_data_frame, [8](#)

- as.list.polars_lazy_frame
(as.list.polars_data_frame), [8](#)
- as.vector(), [166](#)
- as_polars_df, [10](#)
- as_polars_df(), [6, 9, 10, 12, 17, 20, 22, 150,](#)
[165, 166](#)
- as_polars_df(x, ...), [16](#)
- as_polars_expr, [13](#)
- as_polars_expr(), [13, 33, 37, 145, 146, 160,](#)
[161, 163](#)
- as_polars_lf, [16](#)
- as_polars_lf(), [16](#)
- as_polars_series, [17](#)
- as_polars_series(), [10, 12, 14, 15, 17, 19,](#)
[36, 150, 159–162](#)
- as_tibble.polars_data_frame, [21](#)
- as_tibble.polars_lazy_frame
(as_tibble.polars_data_frame),
[21](#)

- base::OlsonNames(), [60, 72](#)
- bit64, [7, 9, 23, 162, 167](#)
- bit64::integer64, [7, 9, 23, 167, 168](#)
- bit64::NA_integer64_, [7, 9, 23, 167](#)
- blob, [162](#)
- blob::blob, [169](#)

- character, [7, 9, 14, 23, 167, 168](#)
- check_list_of_polars_dtype
(check_polars), [24](#)
- check_polars, [24](#)
- check_polars_df (check_polars), [24](#)
- check_polars_dtype (check_polars), [24](#)
- check_polars_expr (check_polars), [24](#)
- check_polars_lf (check_polars), [24](#)
- check_polars_selector (check_polars), [24](#)
- check_polars_series (check_polars), [24](#)
- clock, [7, 9, 23, 162, 168](#)
- clock_duration, [7, 9, 23, 168, 169](#)
- clock_naive_time, [7, 9, 23, 168](#)

- clock_zoned_time, [7](#), [9](#), [23](#), [168](#)
- cs, [27](#)
- data.frame, [9](#), [12](#), [19](#), [166](#), [167](#), [169](#)
- data.table, [7](#), [9](#), [23](#), [162](#), [167](#)
- data.table::IDate, [7](#), [9](#), [23](#), [167](#), [168](#)
- data.table::ITime, [7](#), [9](#), [23](#), [167](#), [168](#)
- DataFrame, [11–13](#), [17](#), [22](#), [28–30](#), [33–35](#), [37](#), [144](#), [150](#), [159](#), [165](#), [166](#)
- DataFrame (pl__DataFrame), [150](#)
- dataframe__cast, [27](#)
- dataframe__clone, [28](#)
- dataframe__drop, [29](#)
- dataframe__equals, [30](#)
- dataframe__filter, [30](#)
- dataframe__get_columns, [31](#)
- dataframe__group_by, [32](#)
- dataframe__lazy, [33](#)
- dataframe__select, [33](#)
- dataframe__slice, [34](#)
- dataframe__sort, [35](#)
- dataframe__to_series, [35](#)
- dataframe__to_struct, [36](#)
- dataframe__with_columns, [37](#)
- Date, [7](#), [9](#), [19](#), [23](#), [167](#), [168](#)
- Datetime, [139](#), [152](#), [154](#), [156](#), [157](#)
- difftime, [7](#), [9](#), [23](#), [152](#), [154](#), [156](#), [157](#), [168](#), [169](#)
- double, [6](#), [7](#), [9](#), [23](#), [167](#), [168](#)
- environment, [148](#)
- environment class, [27](#), [147](#)
- Expr, [14](#), [42](#), [95](#), [110–112](#), [118](#), [125](#), [127](#), [156](#)
- Expr (polars_expr), [164](#)
- expr_arr_all, [38](#)
- expr_arr_any, [38](#)
- expr_arr_arg_max, [39](#)
- expr_arr_arg_min, [39](#)
- expr_arr_contains, [40](#)
- expr_arr_count_matches, [40](#)
- expr_arr_explode, [41](#)
- expr_arr_first, [41](#)
- expr_arr_get, [42](#)
- expr_arr_join, [43](#)
- expr_arr_last, [43](#)
- expr_arr_max, [44](#)
- expr_arr_median, [44](#)
- expr_arr_min, [45](#)
- expr_arr_n_unique, [45](#)
- expr_arr_reverse, [46](#)
- expr_arr_shift, [46](#)
- expr_arr_sort, [47](#)
- expr_arr_std, [47](#)
- expr_arr_sum, [48](#)
- expr_arr_to_list, [48](#)
- expr_arr_unique, [49](#)
- expr_arr_var, [49](#)
- expr_bin_contains, [50](#)
- expr_bin_decode, [50](#)
- expr_bin_encode, [51](#)
- expr_bin_ends_with, [52](#)
- expr_bin_size, [53](#)
- expr_bin_starts_with, [53](#)
- expr_cat_get_categories, [54](#)
- expr_cat_set_ordering, [55](#)
- expr_dt_add_business_days, [56](#)
- expr_dt_base_utc_offset, [57](#)
- expr_dt_cast_time_unit, [57](#)
- expr_dt_century, [58](#)
- expr_dt_combine, [59](#)
- expr_dt_convert_time_zone, [60](#)
- expr_dt_date, [60](#)
- expr_dt_day, [61](#)
- expr_dt_dst_offset, [62](#)
- expr_dt_epoch, [62](#)
- expr_dt_hour, [63](#)
- expr_dt_is_leap_year, [64](#)
- expr_dt_iso_year, [63](#)
- expr_dt_microsecond, [64](#)
- expr_dt_millisecond, [65](#)
- expr_dt_minute, [66](#)
- expr_dt_month, [66](#)
- expr_dt_month_end, [67](#)
- expr_dt_month_start, [67](#)
- expr_dt_nanosecond, [68](#)
- expr_dt_offset_by, [69](#)
- expr_dt_ordinal_day, [70](#)
- expr_dt_quarter, [71](#)
- expr_dt_replace_time_zone, [71](#)
- expr_dt_round, [73](#)
- expr_dt_second, [74](#)
- expr_dt_strftime, [75](#)
- expr_dt_time, [76](#)
- expr_dt_timestamp, [76](#)
- expr_dt_to_string, [81](#)
- expr_dt_total_days, [77](#)
- expr_dt_total_hours, [78](#)

- expr_dt_total_microseconds, 78
- expr_dt_total_milliseconds, 79
- expr_dt_total_minutes, 79
- expr_dt_total_nanoseconds, 80
- expr_dt_total_seconds, 81
- expr_dt_truncate, 82
- expr_dt_week, 83
- expr_dt_weekday, 84
- expr_dt_with_time_unit, 84
- expr_dt_year, 85
- expr_list_all, 86
- expr_list_any, 86
- expr_list_arg_max, 87
- expr_list_arg_min, 87
- expr_list_concat, 88
- expr_list_contains, 88
- expr_list_count_matches, 89
- expr_list_diff, 90
- expr_list_drop_nulls, 90
- expr_list_eval, 91
- expr_list_explode, 92
- expr_list_first, 92
- expr_list_gather, 93
- expr_list_gather_every, 94
- expr_list_get, 94
- expr_list_head, 95
- expr_list_join, 96
- expr_list_last, 96
- expr_list_len, 97
- expr_list_max, 97
- expr_list_mean, 98
- expr_list_median, 98
- expr_list_min, 99
- expr_list_n_unique, 99
- expr_list_reverse, 100
- expr_list_sample, 100
- expr_list_set_difference, 101
- expr_list_set_intersection, 102
- expr_list_set_symmetric_difference, 102
- expr_list_set_union, 103
- expr_list_shift, 104
- expr_list_slice, 105
- expr_list_sort, 105
- expr_list_std, 106
- expr_list_sum, 107
- expr_list_tail, 107
- expr_list_to_array, 108
- expr_list_unique, 109
- expr_list_var, 109
- expr_str_contains, 110
- expr_str_contains_any, 111
- expr_str_count_matches, 112
- expr_str_decode, 112
- expr_str_encode, 113
- expr_str_ends_with, 114
- expr_str_extract, 115
- expr_str_extract_all, 115
- expr_str_extract_groups, 116
- expr_str_extract_many, 117
- expr_str_find, 118
- expr_str_head, 119
- expr_str_join, 120
- expr_str_json_decode, 121
- expr_str_json_path_match, 121
- expr_str_len_bytes, 122
- expr_str_len_chars, 123
- expr_str_pad_end, 124
- expr_str_pad_start, 124
- expr_str_replace, 125
- expr_str_replace_all, 126
- expr_str_replace_many, 128
- expr_str_reverse, 129
- expr_str_slice, 129
- expr_str_split, 130
- expr_str_split_exact, 131
- expr_str_splitn, 130
- expr_str_starts_with, 132
- expr_str_strip_chars, 132
- expr_str_strip_chars_end, 133
- expr_str_strip_chars_start, 134
- expr_str_strptime, 134
- expr_str_tail, 137
- expr_str_to_date, 138
- expr_str_to_datetime, 139
- expr_str_to_integer, 140
- expr_str_to_lowercase, 141
- expr_str_to_time, 141
- expr_str_to_uppercase, 142
- expr_str_zfill, 142
- expression, 7, 10, 13, 15, 23, 38–41, 43–72, 74–125, 127–135, 137–143, 149, 152, 154, 158, 160, 163, 168
- expression (polars_expr), 164
- expressions, 13, 33, 37, 145, 146, 163
- factor, 168

- GroupBy, [32](#)
- hms, [20](#), [162](#)
- hms::hms, [7](#), [9](#), [23](#), [167](#), [168](#)
- integer, [7](#), [9](#), [23](#), [167](#), [168](#)
- is_list_of_polars_dtype (check_polars), [24](#)
- is_polars (check_polars), [24](#)
- is_polars_df (check_polars), [24](#)
- is_polars_dtype (check_polars), [24](#)
- is_polars_expr (check_polars), [24](#)
- is_polars_lf (check_polars), [24](#)
- is_polars_selector (check_polars), [24](#)
- is_polars_series (check_polars), [24](#)
- LazyFrame, [16](#), [17](#), [33](#), [145](#), [146](#), [159](#)
- LazyFrame (pl__LazyFrame), [159](#)
- lazyframe__collect, [143](#)
- lazyframe__select, [145](#)
- lazyframe__with_columns, [146](#)
- list, [10](#), [12](#), [19](#), [167](#), [169](#)
- literals, [33](#), [37](#), [145](#), [146](#), [163](#)
- logical, [168](#)
- NA_integer_, [7](#), [9](#), [23](#), [167](#)
- pl, [147](#)
- pl\$col(), [15](#), [29](#), [165](#)
- pl\$date_range(), [158](#)
- pl\$date_ranges(), [156](#)
- pl\$Datetime(ms), [135](#), [139](#)
- pl\$datetime_range(), [155](#)
- pl\$datetime_ranges(), [153](#)
- pl\$lit(), [13](#), [14](#), [165](#)
- pl\$struct(), [14](#)
- pl__col, [149](#)
- pl__DataFrame, [150](#)
- pl__date_range, [155](#)
- pl__date_ranges, [157](#)
- pl__datetime_range, [151](#)
- pl__datetime_ranges, [153](#)
- pl__LazyFrame, [159](#)
- pl__lit, [160](#)
- pl__Series, [161](#)
- pl__show_versions, [162](#)
- pl__struct, [163](#)
- pl_api_register_series_namespace, [148](#)
- polars data frames, [24](#)
- Polars DataFrame, [10](#)
- polars DataFrame, [10](#), [19](#), [161](#)
- polars expressions, [24](#)
- polars lazy frames, [24](#)
- polars selectors, [24](#)
- Polars Series, [10](#), [150](#)
- polars Series, [17](#), [20](#)
- polars series, [24](#)
- polars_data_frame, [20](#)
- polars_data_frame (pl__DataFrame), [150](#)
- polars_duration_string, [164](#)
- polars_expr, [164](#)
- polars_lazy_frame, [12](#)
- polars_lazy_frame (pl__LazyFrame), [159](#)
- polars_series, [12](#)
- polars_series (pl__Series), [161](#)
- POSIXct, [7](#), [9](#), [10](#), [23](#), [168](#)
- R data frame, [7](#)
- R Data Frames, [150](#)
- R vector, [9](#)
- R vectors, [150](#), [161](#)
- raw, [15](#), [160](#)
- rlang, [26](#)
- rlang::abort(), [26](#)
- rlang::as_function(), [23](#)
- Series, [9](#), [11](#), [12](#), [14](#), [19](#), [31](#), [36](#), [150](#), [159](#), [162](#), [164](#), [166](#)
- Series (pl__Series), [161](#)
- series__to_frame, [166](#)
- series__to_r_vector, [166](#)
- series_struct_unnest, [165](#)
- strptime(), [134](#)
- tibble, [9](#), [24](#), [162](#), [167](#), [169](#)
- vctrs, [162](#)
- vctrs::list_of, [169](#)
- vctrs::list_of(), [19](#)
- vctrs::unspecified, [169](#)
- vctrs::vec_as_names(), [23](#)
- vctrs_rcrd, [19](#)
- vector, [166](#), [167](#), [169](#)
- vectors, [9](#)