Package: neopolars (via r-universe)

January 19, 2025

Title R Bindings for the 'polars' Rust Library

Version 0.0.0.9000

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

License MIT + file LICENSE

Depends R (>= 4.2)

Imports rlang (>= 1.1.0)

Suggests arrow, bit64, blob, cli, clock, data.table, hms, jsonlite, nanoarrow, 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://eitsupi.r-universe.dev

RemoteUrl https://github.com/eitsupi/neo-r-polars

RemoteRef HEAD

RemoteSha 5e287477202e17e2865fda36fb16a6a3db876be0

Contents

`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 `as_polars_df()`.
`int64`	Determine how to convert Polars' Int64, UInt32, or UInt64 type values to R type. One of the followings: • `"double"` (default): Convert to the R's double type. Accuracy may be degraded. • `"character"`: Convert to the R's character type. • `"integer"`: Convert to the R's integer type. If the value is out of the range of R's integer type, export as NA_integer_. • `"integer64"`: Convert to the bit64::integer64 class. The bit64 package must be installed. If the value is out of the range of bit64::integer64, export as bit64::NA_integer64_.
`date`	Determine how to convert Polars' Date type values to R class. One of the followings: • `"Date"` (default): Convert to the R's Date class. • `"IDate"`: Convert to the data.table::IDate class.
`time`	Determine how to convert Polars' Time type values to R class. One of the followings: • `"hms"` (default): Convert to the hms::hms class. • `"ITime"`: Convert to the data.table::ITime class. The data.table package must be installed.
`decimal`	Determine how to convert Polars' Decimal type values to R type. One of the followings:

- `"double"` (default): Convert to the R's double type.
- `"character"`: Convert to the R's character type.

`as_clock_class`
: A logical value indicating whether to export datetimes and duration as the clock package's classes.
  - `FALSE` (default): Duration values are exported as difftime and 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 `as_clock_class` is set to `FALSE` and datetime without timezone values are exported as POSIXct. Character vector or expression containing the followings:
  - `"raise"` (default): Throw an error
  - `"earliest"`: Use the earliest datetime
  - `"latest"`: Use the latest datetime
  - `"null"`: Return a `NA` value

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

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_vec` `= 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 `as_polars_df()`.
`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: • `"double"` (default): Convert to the R's double type. Accuracy may be degraded. • `"character"`: Convert to the R's character type. • `"integer"`: Convert to the R's integer type. If the value is out of the range of R's integer type, export as NA_integer_. • `"integer64"`: Convert to the bit64::integer64 class. The bit64 package must be installed. If the value is out of the range of bit64::integer64, export as bit64::NA_integer64_.

`date` Determine how to convert Polars' Date type values to R class. One of the followings:

- `"Date"` (default): Convert to the R's Date class.
- `"IDate"`: Convert to the data.table::IDate class.

`time` Determine how to convert Polars' Time type values to R class. One of the followings:

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

`struct` Determine how to convert Polars' Struct type values to R class. One of the followings:

- `"dataframe"` (default): Convert to the R's data.frame class.
- `"tibble"`: Convert to the tibble class. If the tibble package is not installed, a warning will be shown.

`decimal` Determine how to convert Polars' Decimal type values to R type. One of the followings:

- `"double"` (default): Convert to the R's double type.
- `"character"`: Convert to the R's character type.

`as_clock_class`

A logical value indicating whether to export datetimes and duration as the clock package's classes.

- `FALSE` (default): Duration values are exported as difftime and 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 `as_clock_class` is set to `FALSE` and datetime without timezone values are exported as POSIXct. Character vector or expression containing the followings:

- `"raise"` (default): Throw an error
- `"earliest"`: Use the earliest datetime
- `"latest"`: Use the latest datetime
- `"null"`: Return a `NA` value

`non_existent` Determine how to deal with non-existent datetimes. Only applicable when `as_clock_class` is set to `FALSE` and datetime without timezone values are exported as POSIXct. One of the followings:

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

Details

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

Value

A list

See Also

- `<DataFrame>$get_columns()`

Examples

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

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

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

---

as_polars_df *Create a Polars DataFrame from an R object*

Description

The `as_polars_df()` function creates a polars DataFrame from various R objects. 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 `<Series>$struct$unnest()` method is used to create a DataFrame from the struct Series. In this case, the `column_name` argument is ignored.
`type_coercion`	A logical, indicats type coercion optimization.
`predicate_pushdown`	A logical, indicats predicate pushdown optimization.
`projection_pushdown`	A logical, indicats projection pushdown optimization.
`simplify_expression`	A logical, indicats simplify expression optimization.
`slice_pushdown`	A logical, indicats slice pushdown optimization.
`comm_subplan_elim`	A logical, indicats tring to cache branching subplans that occur on self-joins or unions.

`comm_subexpr_elim`
: A logical, indicats tring to cache common subexpressions.

`cluster_with_columns`
: A logical, indicats 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.

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 `pl$struct()`. Otherwise (default), done nothing.
`as_lit`	A logical value indicating whether to treat vector as literal values or not. This argument is always set to `TRUE` when calling this function from `pl$lit()`, and expects to return literal values. See examples for details.

Details

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

Default S3 method:

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

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

S3 method for character:

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

Value

A polars expression

Literal scalar mapping

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

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

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

The raw type vector is converted to a Binary scalar.

- raw: Binary

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

- NULL: Null

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

See Also

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

Examples

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

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

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

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

# double
```

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

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

# NULL
as_polars_expr(NULL)

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

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

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

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

`as_polars_lf` *Create a Polars LazyFrame from an R object*

Description

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

Usage

```
as_polars_lf(x, ...)

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

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

Arguments

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

Details

Default S3 method:
Create a DataFrame by calling `as_polars_df()` and then create a LazyFrame from the DataFrame. Additional arguments `...` are passed to `as_polars_df()`.

Value

A polars LazyFrame

---

as_polars_series *Create a Polars Series from an R object*

Description

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

Usage

```
as_polars_series(x, name = NULL, ...)

## Default S3 method:
as_polars_series(x, name = NULL, ...)

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

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

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

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

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

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

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

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

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

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

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

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

## S3 method for class '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:

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

S3 method for POSIXct:

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

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

S3 method for POSIXlt:

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

S3 method for difftime:

Sub-millisecond values will be rounded to milliseconds.

S3 method for hms:

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

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

S3 method for clock_duration:

Calendrical durations (years, quarters, months) are treated as chronologically with the internal representation of seconds. Please check the clock_duration documentation for more details.

S3 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")))

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

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

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

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

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

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

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

# NULL
as_polars_series(NULL)

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

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

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

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

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

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

```
}

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

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

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

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

`as_tibble.polars_data_frame` *Export the polars object as a tibble data frame*

Description

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

Usage

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

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

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

Arguments

`x`	A polars object
`...`	Passed to `as_polars_df()`.
`.name_repair`	Treatment of problematic column names: • `"minimal"`: No name repair or checks, beyond basic existence, • `"unique"`: Make sure names are unique and not empty, • `"check_unique"`: (default value), no name repair, but check they are `unique`, • `"universal"`: Make the names `unique` and syntactic • a function: apply custom name repair (e.g., `.name_repair = make.names` for names in the style of base R). • A purrr-style anonymous function, see `rlang::as_function()` This argument is passed on as `repair` to `vctrs::vec_as_names()`. See there for more details on these terms and the strategies used to enforce them.
`int64`	Determine how to convert Polars' Int64, UInt32, or UInt64 type values to R type. One of the followings: • `"double"` (default): Convert to the R's double type. Accuracy may be degraded. • `"character"`: Convert to the R's character type. • `"integer"`: Convert to the R's integer type. If the value is out of the range of R's integer type, export as NA_integer_. • `"integer64"`: Convert to the bit64::integer64 class. The bit64 package must be installed. If the value is out of the range of bit64::integer64, export as bit64::NA_integer64_.
`date`	Determine how to convert Polars' Date type values to R class. One of the followings:

- `"Date"` (default): Convert to the R's Date class.
- `"IDate"`: Convert to the data.table::IDate class.

`time` Determine how to convert Polars' Time type values to R class. One of the followings:

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

`decimal` Determine how to convert Polars' Decimal type values to R type. One of the followings:

- `"double"` (default): Convert to the R's double type.
- `"character"`: Convert to the R's character type.

`as_clock_class`
A logical value indicating whether to export datetimes and duration as the clock package's classes.

- `FALSE` (default): Duration values are exported as difftime and 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 `as_clock_class` is set to `FALSE` and datetime without timezone values are exported as POSIXct. Character vector or expression containing the followings:

- `"raise"` (default): Throw an error
- `"earliest"`: Use the earliest datetime
- `"latest"`: Use the latest datetime
- `"null"`: Return a `NA` value

`non_existent` Determine how to deal with non-existent datetimes. Only applicable when `as_clock_class` is set to `FALSE` and datetime without timezone values are exported as POSIXct. One of the followings:

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

Value

A tibble

See Also

- `as.data.frame(<polars_object>)`: Export the polars object as a basic data frame.

Examples

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

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

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

---

check_polars *Check if the object is a polars object*

Description

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

- `*_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

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

Details

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

Value

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

Examples

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

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

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

`cs` *Polars column selector function namespace*

Description

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

Usage

```
cs
```

Format

An object of class `polars_object` of length 29.

Supported operators

There are 4 supported operators for selectors:

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

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

Examples

```
cs

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

cs__all *Select all columns*

Description

Select all columns

Usage

```
cs__all()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

Description

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

Usage

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

Arguments

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

Details

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

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

---

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

Description

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

Usage

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

Arguments

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

Details

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

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

---

cs__binary *Select all binary columns*

Description

Select all binary columns

Usage

```
cs__binary()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

---

cs__boolean *Select all boolean columns*

Description

Select all boolean columns

Usage

```
cs__boolean()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

# Select and invert boolean columns:
df$with_columns(inverted = cs$boolean()$not())
```

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

`cs__by_dtype` *Select all columns matching the given dtypes*

Description

Select all columns matching the given dtypes

Usage

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

Arguments

`...`	<dynamic-dots> Data types to select.

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

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

Description

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

Usage

```
cs__by_index(indices)
```

Arguments

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

Details

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

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

`cs__by_name` *Select all columns matching the given names*

Description

Select all columns matching the given names

Usage

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

Arguments

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

Details

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

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

cs__categorical *Select all categorical columns*

Description

Select all categorical columns

Usage

```
cs__categorical()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

Description

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

Usage

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

Arguments

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

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

---

cs__date *Select all date columns*

Description

Select all date columns

Usage

```
cs__date()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

`cs__datetime` *Select all datetime columns*

Description

Select all datetime columns

Usage

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

Arguments

`time_unit`	One (or more) of the allowed time unit precision strings, `"ms"`, `"us"`, and `"ns"`. Default is to select columns with any valid timeunit.
`time_zone`	One of the followings. The value or each element of the vector will be passed to the `time_zone` argument of the `pl$Datetime()` function: • A character vector of one or more timezone strings, as defined in `OlsonNames()`. • `NULL` to select Datetime columns that do not have a timezone. • `"*"` to select Datetime columns that have any timezone. • A list of single timezone strings , `"*"`, and `NULL` to select Datetime columns that do not have a timezone or have the (specific) timezone. For example, the default value `list("*", NULL)` selects all Datetime columns.

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

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

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

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

---

cs__decimal *Select all decimal columns*

Description

Select all decimal columns

Usage

```
cs__decimal()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

---

cs__digit *Select all columns having names consisting only of digits*

Description

Select all columns having names consisting only of digits

Usage

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

Arguments

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

Details

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

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

---

cs__duration *Select all duration columns, optionally filtering by time unit*

Description

Select all duration columns, optionally filtering by time unit

Usage

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

Arguments

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

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

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

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

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

Description

Select columns that end with the given substring(s)

Usage

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

Arguments

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

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

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

Description

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

Usage

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

Arguments

`...`	<dynamic-dots> Column names to exclude.

Details

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

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

cs__first *Select the first column in the current scope*

Description

Select the first column in the current scope

Usage

```
cs__first()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

cs__float *Select all float columns.*

Description

Select all float columns.

Usage

```
cs__float()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

```
df <- pl$DataFrame(
  foo = c("x", "y"),
  bar = c(123L, 456L),
  baz = c(2.0, 5.5),
  zap = c(FALSE, TRUE),
  .schema_overrides = list(baz = pl$Float32, zap = pl$Float64),
)

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

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

---

cs__integer *Select all integer columns.*

Description

Select all integer columns.

Usage

```
cs__integer()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

# Select all integer columns:
df$select(cs$integer())
```

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

`cs__last` *Select the last column in the current scope*

Description

Select the last column in the current scope

Usage

```
cs__last()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

Description

Select all columns that match the given regex pattern

Usage

```
cs__matches(pattern)
```

Arguments

`pattern`	A valid regular expression pattern, compatible with the `regex crate <https://docs.rs/regex/`

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

---

cs__numeric *Select all numeric columns.*

Description

Select all numeric columns.

Usage

```
cs__numeric()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

```
df <- pl$DataFrame(
  foo = c("x", "y"),
  bar = c(123L, 456L),
  baz = c(2.0, 5.5),
  zap = 0:1,
  .schema_overrides = list(bar = pl$Int16, baz = pl$Float32, zap = pl$UInt8),
)

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

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

---

cs__signed_integer *Select all signed integer columns*

Description

Select all signed integer columns

Usage

```
cs__signed_integer()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

# Select signed integer columns:
df$select(cs$signed_integer())

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

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

`cs__starts_with` *Select columns that start with the given substring(s)*

Description

Select columns that start with the given substring(s)

Usage

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

Arguments

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

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

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

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

Description

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

Usage

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

Arguments

`...`	These dots are for future extensions and must be empty.
`include_categorical`	If `TRUE`, also select categorical columns.

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

cs__temporal *Select all temporal columns*

Description

Select all temporal columns

Usage

```
cs__temporal()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

```
df <- pl$DataFrame(
  dtm = as.POSIXct(c("2001-5-7 10:25", "2031-12-31 00:30")),
  dt = as.Date(c("1999-12-31", "2024-8-9")),
  value = 1:2
)

# Match all temporal columns:
df$select(cs$temporal())

# Match all temporal columns except for time columns:
df$select(cs$temporal() - cs$datetime())

# Match all columns except for temporal columns:
df$select(!cs$temporal())
```

cs__time *Select all time columns*

Description

Select all time columns

Usage

```
cs__time()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

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

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

---

cs__unsigned_integer *Select all unsigned integer columns*

Description

Select all unsigned integer columns

Usage

```
cs__unsigned_integer()
```

Value

A Polars selector

See Also

cs for the documentation on operators supported by Polars selectors.

Examples

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

```
)

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

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

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

`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

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

Value

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 *Convert an existing DataFrame to a LazyFrame*

Description

Start a new lazy query from a DataFrame.

Usage

```
dataframe__lazy()
```

Value

A polars LazyFrame

Examples

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

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

Description

Get number of chunks used by the ChunkedArrays of this DataFrame

Usage

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

Arguments

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

Value

An integer vector.

Examples

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

df$n_chunks()
df$n_chunks(strategy = "all")
```

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

Description

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

Usage

```
dataframe__rechunk()
```

Value

A polars DataFrame

dataframe__select *Select and modify columns of a DataFrame*

Description

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

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

Usage

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

Arguments

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

Value

A polars DataFrame

Examples

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

dataframe__slice *Get a slice of the DataFrame.*

Description

Get a slice of the DataFrame.

Usage

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

Arguments

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

Value

A polars DataFrame

Examples

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

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

`dataframe__sort` *Sort a DataFrame*

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 *Select column as Series at index location*

Description

Select column as Series at index location

Usage

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

Arguments

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

Value

Series or NULL

Examples

```
df <- as_polars_df(iris[1:10, ])

# default is to extract the first column
df$to_series()

# Polars is 0-indexed, so we use index = 1 to extract the *2nd* column
df$to_series(index = 1)

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

Description

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

Usage

```
expr_arr_arg_max()
```

Value

A polars expression

Examples

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

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

Description

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

Usage

```
expr_arr_arg_min()
```

Value

A polars expression

Examples

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

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

Description

Check if sub-arrays contain the given item

Usage

```
expr_arr_contains(item)
```

Arguments

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

Value

A polars expression

Examples

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

---

expr_arr_count_matches

Count how often a value occurs in every sub-array

Description

Count how often a value occurs in every sub-array

Usage

```
expr_arr_count_matches(element)
```

Arguments

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

Value

A polars expression

Examples

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

expr_arr_explode *Explode array in separate rows*

Description

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

Usage

```
expr_arr_explode()
```

Value

A polars expression

Examples

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

expr_arr_first *Get the first value of the sub-arrays*

Description

Get the first value of the sub-arrays

Usage

```
expr_arr_first()
```

Value

A polars expression

Examples

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

expr_arr_get *Get the value by index in every sub-array*

Description

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

Usage

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

Arguments

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

Value

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 *Join elements in every sub-array*

Description

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

Usage

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

Arguments

separator	String to separate the items with. Can be an Expr. Strings are not parsed as columns.
...	These dots are for future extensions and must be empty.

Value

A polars expression

Examples

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

---

expr_arr_last *Get the last value of the sub-arrays*

Description

Get the last value of the sub-arrays

Usage

```
expr_arr_last()
```

Value

A polars expression

Examples

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

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

Description

Compute the max value of the sub-arrays

Usage

```
expr_arr_max()
```

Value

A polars expression

Examples

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

expr_arr_median *Compute the median value of the sub-arrays*

Description

Compute the median value of the sub-arrays

Usage

```
expr_arr_median()
```

Value

A polars expression

Examples

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

expr_arr_min *Compute the min value of the sub-arrays*

Description

Compute the min value of the sub-arrays

Usage

```
expr_arr_min()
```

Value

A polars expression

Examples

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

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

Description

Count the number of unique values in every sub-array

Usage

```
expr_arr_n_unique()
```

Value

A polars expression

Examples

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

expr_arr_reverse *Reverse values in every sub-array*

Description

Reverse values in every sub-array

Usage

```
expr_arr_reverse()
```

Value

A polars expression

Examples

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

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

Description

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

Usage

```
expr_arr_shift(n = 1)
```

Value

A polars expression

Examples

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

expr_arr_sort *Sort values in every sub-array*

Description

Sort values in every sub-array

Usage

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

Arguments

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

Examples

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

expr_arr_std *Compute the standard deviation of the sub-arrays*

Description

Compute the standard deviation of the sub-arrays

Usage

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

Value

A polars expression

Examples

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

expr_arr_sum *Compute the sum of the sub-arrays*

Description

Compute the sum of the sub-arrays

Usage

```
expr_arr_sum()
```

Value

A polars expression

Examples

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

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

Description

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

Usage

```
expr_arr_to_list()
```

Value

A polars expression

Examples

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

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

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

Description

Get the unique values in every sub-array

Usage

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

Arguments

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

Value

A polars expression

Examples

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

expr_arr_var *Compute the variance of the sub-arrays*

Description

Compute the variance of the sub-arrays

Usage

```
expr_arr_var(ddof = 1)
```

Value

A polars expression

Examples

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

expr_bin_contains *Check if binaries contain a binary substring*

Description

Check if binaries contain a binary substring

Usage

```
expr_bin_contains(literal)
```

Arguments

literal	The binary substring to look for.

Value

A polars expression

Examples

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

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

expr_bin_decode *Decode values using the provided encoding*

Description

Decode values using the provided encoding

Usage

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

Arguments

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

Value

A polars expression

Examples

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

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

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

`expr_bin_encode` *Encode a value using the provided encoding*

Description

Encode a value using the provided encoding

Usage

```
expr_bin_encode(encoding)
```

Arguments

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

Value

A polars expression

Examples

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

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

---

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

Description

Check if string values end with a binary substring

Usage

```
expr_bin_ends_with(suffix)
```

Arguments

suffix	Suffix substring.

Value

A polars expression

Examples

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

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

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

Value

A polars expression

Examples

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

# You can pass a custom weekend - for example, if you only take Sunday off:
week_mask <- c(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE)
df$with_columns(
```

```
  result = pl$col("start")$dt$add_business_days(5, week_mask = week_mask)
)

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

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

`expr_dt_base_utc_offset` *Base offset from UTC*

Description

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

Usage

```
expr_dt_base_utc_offset()
```

Value

A polars expression

Examples

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

expr_dt_cast_time_unit *Change time unit*

Description

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

Usage

```
expr_dt_cast_time_unit(time_unit)
```

Arguments

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

Value

A polars expression

Examples

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

expr_dt_century *Extract the century from underlying representation*

Description

Returns the century number in the calendar date.

Usage

```
expr_dt_century()
```

Value

A polars expression

Examples

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

---

expr_dt_combine *Combine Date and Time*

Description

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

Usage

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

Arguments

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  dtm = c(
    ISOdatetime(2022, 12, 31, 10, 30, 45),
    ISOdatetime(2023, 7, 5, 23, 59, 59)
  ),
  dt = c(ISOdate(2022, 10, 10), ISOdate(2022, 7, 5)),
  tm = hms::parse_hms(c("1:2:3.456000", "7:8:9.101000"))
)
```

```
df

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

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

Description

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

Usage

```
expr_dt_convert_time_zone(time_zone)
```

Arguments

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

Value

A polars expression

Examples

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

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

expr_dt_date *Extract date from date(time)*

Description

Extract date from date(time)

Usage

```
expr_dt_date()
```

Value

A polars expression

Examples

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

expr_dt_day *Extract day from underlying Date representation*

Description

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

Usage

```
expr_dt_day()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  date = pl$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 *Extract hour from underlying Datetime representation*

Description

Returns the hour number from 0 to 23.

Usage

```
expr_dt_hour()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  date = pl$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 *Extract ISO year from underlying Date representation*

Description

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

Usage

```
expr_dt_iso_year()
```

Value

A polars expression

Examples

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

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

Description

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

Usage

```
expr_dt_is_leap_year()
```

Value

A polars expression

Examples

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

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

expr_dt_microsecond *Extract microseconds from underlying Datetime representation*

Description

Extract microseconds from underlying Datetime representation

Usage

```
expr_dt_microsecond()
```

Value

A polars expression

Examples

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

df$with_columns(
  microsecond = pl$col("datetime")$dt$microsecond()
)
```

expr_dt_millisecond *Extract milliseconds from underlying Datetime representation*

Description

Extract milliseconds from underlying Datetime representation

Usage

```
expr_dt_millisecond()
```

Value

A polars expression

Examples

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

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

---

expr_dt_minute *Extract minute from underlying Datetime representation*

Description

Returns the minute number from 0 to 59.

Usage

```
expr_dt_minute()
```

Value

A polars expression

Examples

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

expr_dt_month — *Extract month from underlying Date representation*

Description

Returns the month number between 1 and 12.

Usage

```
expr_dt_month()
```

Value

A polars expression

Examples

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

expr_dt_month_end — *Roll forward to the last day of the month*

Description

For datetimes, the time of day is preserved.

Usage

```
expr_dt_month_end()
```

Value

A polars expression

Examples

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

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

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

Description

For datetimes, the time of day is preserved.

Usage

```
expr_dt_month_start()
```

Value

A polars expression

Examples

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

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

expr_dt_nanosecond *Extract nanoseconds from underlying Datetime representation*

Description

Extract nanoseconds from underlying Datetime representation

Usage

```
expr_dt_nanosecond()
```

Value

A polars expression

Examples

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

```
  )
)

df$with_columns(
  nanosecond = pl$col("datetime")$dt$nanosecond()
)
```

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

Description

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

Usage

```
expr_dt_offset_by(by)
```

Arguments

`by`	optional string encoding duration see details.

Details

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

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

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

These strings can be combined:

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

Value

A polars expression

Examples

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

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

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

---

expr_dt_ordinal_day *Extract ordinal day from underlying Date representation*

Description

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

Usage

```
expr_dt_ordinal_day()
```

Value

A polars expression

Examples

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

---

expr_dt_quarter *Extract quarter from underlying Date representation*

Description

Returns the quarter ranging from 1 to 4.

Usage

```
expr_dt_quarter()
```

Value

A polars expression

Examples

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

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

Description

Different from dtconvert_time_zone(), this will also modify the underlying timestamp and will ignore the original time zone.

Usage

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

Arguments

`time_zone`	`NULL` or a character time zone from `base::OlsonNames()`. Pass `NULL` to unset time zone.
`...`	These dots are for future extensions and must be empty.
`ambiguous`	Determine how to deal with ambiguous datetimes. Character vector or expression containing the followings: • `"raise"` (default): Throw an error • `"earliest"`: Use the earliest datetime • `"latest"`: Use the latest datetime • `"null"`: Return a null value
`non_existent`	Determine how to deal with non-existent datetimes. One of the followings: • `"raise"` (default): Throw an error • `"null"`: Return a null value

Value

A polars expression

Examples

```
df <- pl$select(
  london_timezone = pl$datetime_range(
    as.Date("2020-03-01"),
    as.Date("2020-07-01"),
    "1mo",
    time_zone = "UTC"
  )$dt$convert_time_zone(time_zone = "Europe/London")
```

```
)
df$with_columns(
  London_to_Amsterdam = pl$col("london_timezone")$dt$replace_time_zone(time_zone="Europe/Amsterdam")
)
# You can use `ambiguous` to deal with ambiguous datetimes:
dates <- c(
  "2018-10-28 01:30",
  "2018-10-28 02:00",
  "2018-10-28 02:30",
  "2018-10-28 02:00"
) |>
  as.POSIXct("UTC")

df2 <- pl$DataFrame(
  ts = as_polars_series(dates),
  ambiguous = c("earliest", "earliest", "latest", "latest"),
)

df2$with_columns(
  ts_localized = pl$col("ts")$dt$replace_time_zone(
    "Europe/Brussels",
    ambiguous = pl$col("ambiguous")
  )
)
```

`expr_dt_round` *Round datetime*

Description

Divide the date/datetime range into buckets. Each date/datetime in the first half of the interval is mapped to the start of its bucket. Each date/datetime in the second half of the interval is mapped to the end of its bucket. Ambiguous results are localised using the DST offset of the original timestamp - for example, rounding `'2022-11-06 01:20:00 CST'` by `'1h'` results in `'2022-11-06 01:00:00 CST'`, whereas rounding `'2022-11-06 01:20:00 CDT'` by `'1h'` results in `'2022-11-06 01:00:00 CDT'`.

Usage

```
expr_dt_round(every)
```

Arguments

`every`	Either an Expr or a string indicating a column name or a duration (see Details).

Details

The `every` and `offset` argument are created with the the following string language:

- 1ns # 1 nanosecond
- 1us # 1 microsecond
- 1ms # 1 millisecond
- 1s # 1 second
- 1m # 1 minute
- 1h # 1 hour
- 1d # 1 day
- 1w # 1 calendar week
- 1mo # 1 calendar month
- 1y # 1 calendar year These strings can be combined:
  - 3d12h4m25s # 3 days, 12 hours, 4 minutes, and 25 seconds

Value

A polars expression

Examples

```
df <- pl$select(
  datetime = pl$datetime_range(
    as.Date("2001-01-01"),
    as.Date("2001-01-02"),
    as.difftime("0:25:0")
  )
)
df$with_columns(round = pl$col("datetime")$dt$round("1h"))

df <- pl$select(
  datetime = pl$datetime_range(
    as.POSIXct("2001-01-01 00:00"),
    as.POSIXct("2001-01-01 01:00"),
    as.difftime("0:10:0")
  )
)
df$with_columns(round = pl$col("datetime")$dt$round("1h"))
```

`expr_dt_second` *Extract seconds from underlying Datetime representation*

Description

Returns the integer second number from 0 to 59, or a floating point number from 0 to 60 if `fractional = TRUE` that includes any milli/micro/nanosecond component.

Usage

```
expr_dt_second(fractional = FALSE)
```

Arguments

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

Value

A polars expression

Examples

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

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

`expr_dt_strftime` *Convert a Date/Time/Datetime/Duration column into a String column with the given format*

Description

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

Usage

```
expr_dt_strftime(format)
```

Arguments

format	Single string of format to use, or NULL. NULL will be treated as "iso". Available formats depend on the column data type: • For Date/Time/Datetime, refer to the chrono strftime documentation for specification. Example: "%y-%m-%d". Special case "iso" will use the ISO8601 format. • For Duration, "iso" or "polars" can be used. The "iso" format string results in ISO8601 duration string output, and "polars" results in the same form seen in the polars print representation.

Value

A polars expression

Examples

```
pl$DataFrame(
  datetime = c(as.POSIXct(c("2021-01-02 00:00:00", "2021-01-03 00:00:00")))
)$
  with_columns(
  datetime_string = pl$col("datetime")$dt$strftime("%Y/%m/%d %H:%M:%S")
)
```

---

expr_dt_time *Extract time*

Description

This only works on Datetime columns, it will error on Date columns.

Usage

```
expr_dt_time()
```

Value

A polars expression

Examples

```
df <- pl$select(dates = pl$datetime_range(
  as.Date("2000-1-1"),
  as.Date("2000-1-2"),
  "1h"
))

df$with_columns(times = pl$col("dates")$dt$time())
```

---

expr_dt_timestamp *Get timestamp in the given time unit*

Description

Get timestamp in the given time unit

Usage

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

Arguments

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

Value

A polars expression

Examples

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

expr_dt_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 a Date/Time/Datetime/Duration column into a String column with the given format*

Description

Similar to `$cast(pl$String)`, but this method allows you to customize the formatting of the resulting string; if no format is provided, the appropriate ISO format for the underlying data type is used.

Usage

```
expr_dt_to_string(format = NULL)
```

Arguments

`format` Single string of format to use, or `NULL` (default). `NULL` will be treated as `"iso"`. Available formats depend on the column data type:

- For Date/Time/Datetime, refer to the chrono strftime documentation for specification. Example: `"%y-%m-%d"`. Special case `"iso"` will use the ISO8601 format.
- For Duration, `"iso"` or `"polars"` can be used. The `"iso"` format string results in ISO8601 duration string output, and `"polars"` results in the same form seen in the polars print representation.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  dt = as.Date(c("1990-03-01", "2020-05-03", "2077-07-05")),
  dtm = as.POSIXct(c("1980-08-10 00:10:20", "2010-10-20 08:25:35", "2040-12-30 16:40:50")),
  tm = hms::as_hms(c("01:02:03.456789", "23:59:09.101", "00:00:00.000100")),
  dur = clock::duration_days(c(-1, 14, 0)) + clock::duration_hours(c(0, -10, 0)) +
    clock::duration_seconds(c(-42, 0, 0)) + clock::duration_microseconds(c(0, 1001, 0)),
)

# Default format for temporal dtypes is ISO8601:
df$select((cs$date() | cs$datetime())$dt$to_string()$name$prefix("s_"))
df$select((cs$time() | cs$duration())$dt$to_string()$name$prefix("s_"))

# All temporal types (aside from Duration) support strftime formatting:
df$select(
  pl$col("dtm"),
  s_dtm = pl$col("dtm")$dt$to_string("%Y/%m/%d (%H.%M.%S)"),
)

# The Polars Duration string format is also available:
df$select(pl$col("dur"), s_dur = pl$col("dur")$dt$to_string("polars"))

# If you're interested in extracting the day or month names,
# you can use the '%A' and '%B' strftime specifiers:
df$select(
  pl$col("dt"),
  day_name = pl$col("dtm")$dt$to_string("%A"),
  month_name = pl$col("dtm")$dt$to_string("%B"),
)
```

`expr_dt_truncate` *Truncate datetime*

Description

Divide the date/datetime range into buckets. Each date/datetime is mapped to the start of its bucket using the corresponding local datetime. Note that weekly buckets start on Monday. Ambiguous results are localised using the DST offset of the original timestamp - for example, truncating `'2022-11-06 01:30:00 CST'` by `'1h'` results in `'2022-11-06 01:00:00 CST'`, whereas truncating `'2022-11-06 01:30:00 CDT'` by `'1h'` results in `'2022-11-06 01:00:00 CDT'`.

Usage

```
expr_dt_truncate(every)
```

Arguments

`every`	Either an Expr or a string indicating a column name or a duration (see Details).

Details

The `every` and `offset` argument are created with the the following string language:

- 1ns # 1 nanosecond
- 1us # 1 microsecond
- 1ms # 1 millisecond
- 1s # 1 second
- 1m # 1 minute
- 1h # 1 hour
- 1d # 1 day
- 1w # 1 calendar week
- 1mo # 1 calendar month
- 1y # 1 calendar year These strings can be combined:
  - 3d12h4m25s # 3 days, 12 hours, 4 minutes, and 25 seconds

Value

A polars expression

Examples

```
df <- pl$select(
  datetime = pl$datetime_range(
    as.Date("2001-01-01"),
    as.Date("2001-01-02"),
    as.difftime("0:25:0")
  )
)
df$with_columns(truncated = pl$col("datetime")$dt$truncate("1h"))

df <- pl$select(
  datetime = pl$datetime_range(
    as.POSIXct("2001-01-01 00:00"),
    as.POSIXct("2001-01-01 01:00"),
    as.difftime("0:10:0")
  )
)
df$with_columns(truncated = pl$col("datetime")$dt$truncate("30m"))
```

---

expr_dt_week *Extract week from underlying Date representation*

Description

Returns the ISO week number starting from 1. The return value ranges from 1 to 53 (the last week of year differs across years).

Usage

```
expr_dt_week()
```

Value

A polars expression

Examples

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

expr_dt_weekday *Extract weekday from underlying Date representation*

Description

Returns the ISO weekday number where Monday = 1 and Sunday = 7.

Usage

```
expr_dt_weekday()
```

Value

A polars expression

Examples

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

expr_dt_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 `pl$element()`, `pl$first()`, 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 `$group_by()` 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.
...	These dots are for future extensions and must be empty.
null_on_oob	If `TRUE`, return `null` if an index is out of bounds. Otherwise, raise an error.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = list(c(3, 2, 1), 1, c(1, 2)),
  idx = list(0:1, integer(), c(1L, 999L))
)
df$with_columns(
  gathered = pl$col("a")$list$gather("idx", null_on_oob = TRUE)
)

df$with_columns(
  gathered = pl$col("a")$list$gather(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).
`...`	These dots are for future extensions and must be empty.
`null_on_oob`	If `TRUE`, return `null` if an index is out of bounds. Otherwise, raise an error.

Value

Expr

Examples

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

expr_list_head *Slice the first* n *values of every sub-list*

Description

Slice the first n values of every sub-list

Usage

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

Arguments

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

Value

A polars expression

Examples

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

expr_list_join *Join elements of every sub-list*

Description

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

Usage

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

Arguments

separator	String to separate the items with. Can be an Expr. Strings are *not* parsed as columns.
...	These dots are for future extensions and must 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 *Return the number of elements in each sub-list*

Description

Null values are counted in the total.

Usage

```
expr_list_len()
```

Value

A polars expression

Examples

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

---

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

Description

Compute the maximum value in every sub-list

Usage

```
expr_list_max()
```

Value

A polars expression

Examples

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

---

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

Description

Compute the mean value in every sub-list

Usage

```
expr_list_mean()
```

Value

A polars expression

Examples

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

expr_list_median *Compute the median in every sub-list*

Description

Compute the median in every sub-list

Usage

```
expr_list_median()
```

Value

A polars expression

Examples

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

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

expr_list_min *Compute the miminum value in every sub-list*

Description

Compute the miminum value in every sub-list

Usage

```
expr_list_min()
```

Value

A polars expression

Examples

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

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

Description

Count the number of unique values in every sub-lists

Usage

```
expr_list_n_unique()
```

Value

A polars expression

Examples

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

expr_list_reverse *Reverse values in every sub-list*

Description

Reverse values in every sub-list

Usage

```
expr_list_reverse()
```

Value

A polars expression

Examples

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

`expr_list_sample` *Sample values from every sub-list*

Description

Sample values from every sub-list

Usage

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

Arguments

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  values = list(1:3, NA, c(NA, 3L), 5:7),
  n = c(1, 1, 1, 2)
)

df$with_columns(
  sample = pl$col("values")$list$sample(n = pl$col("n"), seed = 1)
)
```

`expr_list_set_difference`

Compute the set difference between elements of a list and other elements

---

Description

This returns the "asymmetric difference", meaning only the elements of the first list that are not in the second list. To get all elements that are in only one of the two lists, use `$set_symmetric_difference()`.

Usage

```
expr_list_set_difference(other)
```

Arguments

`other`	Other list variable. Can be an Expr or something coercible to an Expr.

Details

Note that the datatypes inside the list must have a common supertype. For example, the first column can be `list[i32]` and the second one can be `list[i8]` because it can be cast to `list[i32]`. However, the second column cannot be e.g `list[f32]`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = list(1:3, NA, c(NA, 3L), 5:7),
  b = list(2:4, 3L, c(3L, 4L, NA), c(6L, 8L))
)

df$with_columns(difference = pl$col("a")$list$set_difference("b"))
```

`expr_list_set_intersection`

Compute the intersection between elements of a list and other elements

---

Description

Compute the intersection between elements of a list and other elements

Usage

```
expr_list_set_intersection(other)
```

Arguments

other	Other list variable. Can be an Expr or something coercible to an Expr.

Details

Note that the datatypes inside the list must have a common supertype. For example, the first column can be `list[i32]` and the second one can be `list[i8]` because it can be cast to `list[i32]`. However, the second column cannot be e.g `list[f32]`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = list(1:3, NA, c(NA, 3L), 5:7),
  b = list(2:4, 3L, c(3L, 4L, NA), c(6L, 8L))
)

df$with_columns(intersection = pl$col("a")$list$set_intersection("b"))
```

---

expr_list_set_symmetric_difference

Compute the set symmetric difference between elements of a list and other elements

---

Description

This returns all elements that are in only one of the two lists. To get only elements that are in the first list but not in the second one, use `$set_difference()`.

Usage

```
expr_list_set_symmetric_difference(other)
```

Arguments

other	Other list variable. Can be an Expr or something coercible to an Expr.

Details

Note that the datatypes inside the list must have a common supertype. For example, the first column can be `list[i32]` and the second one can be `list[i8]` because it can be cast to `list[i32]`. However, the second column cannot be e.g `list[f32]`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = list(1:3, NA, c(NA, 3L), 5:7),
  b = list(2:4, 3L, c(3L, 4L, NA), c(6L, 8L))
)

df$with_columns(
  symmetric_difference = pl$col("a")$list$set_symmetric_difference("b")
)
```

---

expr_list_set_union *Compute the union of elements of a list and other elements*

Description

Compute the union of elements of a list and other elements

Usage

```
expr_list_set_union(other)
```

Arguments

other	Other list variable. Can be an Expr or something coercible to an Expr.

Details

Note that the datatypes inside the list must have a common supertype. For example, the first column can be `list[i32]` and the second one can be `list[i8]` because it can be cast to `list[i32]`. However, the second column cannot be e.g `list[f32]`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = list(1:3, NA, c(NA, 3L), 5:7),
  b = list(2:4, 3L, c(3L, 4L, NA), c(6L, 8L))
)

df$with_columns(union = pl$col("a")$list$set_union("b"))
```

expr_list_shift *Shift list values by the given number of indices*

Description

Shift list values by the given number of indices

Usage

```
expr_list_shift(n = 1)
```

Value

A polars expression

Examples

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

expr_list_slice *Slice every sub-list*

Description

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

Usage

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

Arguments

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

Value

A polars expression

Examples

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

---

expr_list_sort *Sort values in every sub-list*

Description

Sort values in every sub-list

Usage

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

Arguments

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

Value

A polars expression

Examples

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

expr_list_std *Compute the standard deviation in every sub-list*

Description

Compute the standard deviation in every sub-list

Usage

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

Arguments

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

Value

A polars expression

Examples

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

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

expr_list_sum *Sum all elements in every sub-list*

Description

Sum all elements in every sub-list

Usage

```
expr_list_sum()
```

Value

A polars expression

Examples

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

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

Description

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

Usage

```
expr_list_tail(n = 5L)
```

Arguments

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

Value

A polars expression

Examples

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

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

Description

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

Usage

```
expr_list_to_array(width)
```

Arguments

`width`	Width of the resulting Array column.

Value

A polars expression

Examples

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

df$with_columns(
  array = pl$col("values")$list$to_array(2)
)
```

---

expr_list_unique *Get unique values in a list*

Description

Get unique values in a list

Usage

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

Arguments

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

Value

A polars expression

Examples

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

expr_list_var *Compute the variance in every sub-list*

Description

Compute the variance in every sub-list

Usage

```
expr_list_var(ddof = 1)
```

Value

A polars expression

Examples

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

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

expr_meta_eq *Indicate if this expression is the same as another expression*

Description

Indicate if this expression is the same as another expression

Usage

```
expr_meta_eq(other)
```

Value

A polars expression

Examples

```
foo_bar <- pl$col("foo")$alias("bar")
foo <- pl$col("foo")
foo_bar$meta$eq(foo)

foo_bar2 <- pl$col("foo")$alias("bar")
foo_bar$meta$eq(foo_bar2)
```

`expr_meta_has_multiple_outputs` *Indicate if this expression expands into multiple expressions*

Description

Indicate if this expression expands into multiple expressions

Usage

```
expr_meta_has_multiple_outputs()
```

Value

A polars expression

Examples

```
e <- pl$col(c("a", "b"))$name$suffix("_foo")
e$meta$has_multiple_outputs()
```

`expr_meta_is_column` *Indicate if this expression is a basic (non-regex) unaliased column*

Description

Indicate if this expression is a basic (non-regex) unaliased column

Usage

```
expr_meta_is_column()
```

Value

A logical value.

Examples

```
e <- pl$col("foo")
e$meta$is_column()

e <- pl.col("foo") * pl.col("bar")
e$meta$is_column()

e <- pl.col(r"^col.*\d+$")
e$meta$is_column()
```

`expr_meta_is_column_selection`
Indicate if this expression only selects columns (optionally with aliasing)

Description

This can include bare columns, column matches by regex or dtype, selectors and exclude ops, and (optionally) column/expression aliasing.

Usage

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

Arguments

`...`	These dots are for future extensions and must be empty.
`allow_aliasing`	If `FALSE` (default), any aliasing is not considered pure column selection. Set `TRUE` to allow for column selection that also includes aliasing.

Value

A logical value.

Examples

```
e <- pl$col("foo")
e$meta$is_column_selection()

e <- pl$col("foo")$alias("bar")
e$meta$is_column_selection()

e$meta$is_column_selection(allow_aliasing = TRUE)

e <- pl$col("foo") * pl$col("bar")
e$meta$is_column_selection()

e <- cs$starts_with("foo")
e$meta$is_column_selection()
```

`expr_meta_is_regex_projection` *Indicate if this expression expands to columns that match a regex pattern*

Description

Indicate if this expression expands to columns that match a regex pattern

Usage

```
expr_meta_is_regex_projection()
```

Value

A logical value.

Examples

```
e <- pl$col("^.*$")$name$prefix("foo_")
e$meta$is_regex_projection()
```

`expr_meta_ne` *Indicate if this expression is not the same as another expression*

Description

Indicate if this expression is not the same as another expression

Usage

```
expr_meta_ne(other)
```

Value

A polars expression

Examples

```
foo_bar <- pl$col("foo")$alias("bar")
foo <- pl$col("foo")
foo_bar$meta$ne(foo)

foo_bar2 <- pl$col("foo")$alias("bar")
foo_bar$meta$ne(foo_bar2)
```

`expr_meta_output_name` *Get the column name that this expression would produce*

Description

It may not always be possible to determine the output name as that can depend on the schema of the context; in that case this will raise an error if `raise_if_undetermined = TRUE` (the default), and return `NA` otherwise.

Usage

```
expr_meta_output_name(..., raise_if_undetermined = TRUE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`raise_if_undetermined`	If `TRUE` (default), raise an error if the output name cannot be determined. Otherwise return `NA`.

Value

A polars expression

Examples

```
e <- pl$col("foo") * pl$col("bar")
e$meta$output_name()

e_filter <- pl$col("foo")$filter(pl$col("bar") == 13)
e_filter$meta$output_name()

e_sum_over <- pl$col("foo")$sum()$over("groups")
e_sum_over$meta$output_name()

e_sum_slice <- pl$col("foo")$sum()$slice(pl$len() - 10, pl$col("bar"))
e_sum_slice$meta$output_name()

pl$len()$meta$output_name()
```

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

Description

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

Usage

```
expr_meta_pop()
```

Value

A polars expression

Examples

```
e <- pl$col("foo")$alias("bar")
pop <- e$meta$pop()
pop

pop[[1]]$meta$eq(pl$col("foo"))
pop[[1]]$meta$eq(pl$col("foo"))
```

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

Description

Get a list with the root column name

Usage

```
expr_meta_root_names()
```

Value

A polars expression

Examples

```
e <- pl$col("foo") * pl$col("bar")
e$meta$root_names()

e_filter <- pl$col("foo")$filter(pl$col("bar") == 13)
e_filter$meta$root_names()

e_sum_over <- pl$sum("foo")$over("groups")
e_sum_over$meta$root_names()

e_sum_slice <- pl$sum("foo")$slice(pl$len() - 10, pl$col("bar"))
e_sum_slice$meta$root_names()
```

---

expr_meta_serialize *Serialize this expression to a string in binary or JSON format*

Description

Serialize this expression to a string in binary or JSON format

Usage

```
expr_meta_serialize(..., format = c("binary", "json"))
```

Arguments

`...`	These dots are for future extensions and must be empty.
`format`	The format in which to serialize. Must be one of: • `"binary"` (default): serialize to binary format (bytes). • `"json"`: serialize to JSON format (string).

Details

Serialization is not stable across Polars versions: a LazyFrame serialized in one Polars version may not be deserializable in another Polars version.

Value

A polars expression

Examples

```
# Serialize the expression into a binary representation.
expr <- pl$col("foo")$sum()$over("bar")
bytes <- expr$meta$serialize()
rawToChar(bytes)
```

```
pl$deserialize_expr(bytes)

# Serialize into json
expr$meta$serialize(format = "json") |>
  jsonlite::prettify()
```

`expr_meta_tree_format`

Format the expression as a tree

Description

Format the expression as a tree

Usage

```
expr_meta_tree_format()
```

Value

A character vector

Examples

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

`expr_meta_undo_aliases`

Undo any renaming operation like `alias` *or* `name$keep`

Description

Undo any renaming operation like `alias` or `name$keep`

Usage

```
expr_meta_undo_aliases()
```

Value

A polars expression

Examples

```
e <- pl$col("foo")$alias("bar")
e$meta$undo_aliases()$meta$eq(pl$col("foo"))

e <- pl$col("foo")$sum()$over("bar")
e$name$keep()$meta$undo_aliases()$meta$eq(e)
```

---

expr_struct_field *Retrieve one or multiple Struct field(s) as a new Series*

Description

Retrieve one or multiple Struct field(s) as a new Series

Usage

```
expr_struct_field(...)
```

Arguments

...	<dynamic-dots> Names of struct fields to retrieve.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  aaa = c(1, 2),
  bbb = c("ab", "cd"),
  ccc = c(TRUE, NA),
  ddd = list(1:2, 3)
)$select(struct_col = pl$struct("aaa", "bbb", "ccc", "ddd"))
df

# Retrieve struct field(s) as Series:
df$select(pl$col("struct_col")$struct$field("bbb"))

df$select(
  pl$col("struct_col")$struct$field("bbb"),
  pl$col("struct_col")$struct$field("ddd")
)

# Use wildcard expansion:
df$select(pl$col("struct_col")$struct$field("*"))

# Retrieve multiple fields by name:
df$select(pl$col("struct_col")$struct$field("aaa", "bbb"))
```

```
# Retrieve multiple fields by regex expansion:
df$select(pl$col("struct_col")$struct$field("^a.*|b.*$"))
```

`expr_struct_json_encode` *Convert this struct to a string column with json values*

Description

Convert this struct to a string column with json values

Usage

```
expr_struct_json_encode()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = list(1:2, c(9, 1, 3)),
  b = list(45, NA)
)$select(a = pl$struct("a", "b"))

df

df$with_columns(encoded = pl$col("a")$struct$json_encode())
```

`expr_struct_rename_fields` *Rename the fields of the struct*

Description

Rename the fields of the struct

Usage

```
expr_struct_rename_fields(names)
```

Arguments

`names`	New names, given in the same order as the struct's fields.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  aaa = c(1, 2),
  bbb = c("ab", "cd"),
  ccc = c(TRUE, NA),
  ddd = list(1:2, 3)
)$select(struct_col = pl$struct("aaa", "bbb", "ccc", "ddd"))
df

df <- df$select(
  pl$col("struct_col")$struct$rename_fields(c("www", "xxx", "yyy", "zzz"))
)
df$select(pl$col("struct_col")$struct$field("*"))

# Following a rename, the previous field names cannot be referenced:
tryCatch(
  {
    df$select(pl$col("struct_col")$struct$field("aaa"))
  },
  error = function(e) print(e)
)
```

---

expr_struct_unnest *Expand the struct into its individual fields*

Description

This is an alias for `Expr$struct$field("*")`.

Usage

```
expr_struct_unnest()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  aaa = c(1, 2),
  bbb = c("ab", "cd"),
  ccc = c(TRUE, NA),
  ddd = list(1:2, 3)
)$select(struct_col = pl$struct("aaa", "bbb", "ccc", "ddd"))
df
```

```
df$select(pl$col("struct_col")$struct$unnest())
```

`expr_struct_with_fields` *Add or overwrite fields of this struct*

Description

This is similar to `with_columns()` on DataFrame and LazyFrame.

Usage

```
expr_struct_with_fields(...)
```

Arguments

`...`	<dynamic-dots> Field(s) to add. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  x = c(1, 4, 9),
  y = c(4, 9, 16),
  multiply = c(10, 2, 3)
)$select(coords = pl$struct("x", "y"), "multiply")
df

df <- df$with_columns(
  pl$col("coords")$struct$with_fields(
    pl$field("x")$sqrt(),
    y_mul = pl$field("y") * pl$col("multiply")
  )
)

df
df$select(pl$col("coords")$struct$field("*"))
```

`expr_str_contains` *Check if string contains a substring that matches a pattern*

Description

Check if string contains a substring that matches a pattern

Usage

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

Arguments

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

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

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

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

Description

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

Usage

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

Arguments

`patterns`	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` *Encode a value using the provided encoding*

Description

Encode a value using the provided encoding

Usage

```
expr_str_encode(encoding)
```

Arguments

`encoding`	Either 'hex' or 'base64'.

Value

A polars expression

Examples

```
df <- pl$DataFrame(strings = c("foo", "bar", NA))
df$select(pl$col("strings")$str$encode("hex"))
df$with_columns(
  pl$col("strings")$str$encode("base64")$alias("base64"), # notice DataType is not encoded
  pl$col("strings")$str$encode("hex")$alias("hex") # ... and must restored with cast
)$with_columns(
  pl$col("base64")$str$decode("base64")$alias("base64_decoded")$cast(pl$String),
  pl$col("hex")$str$decode("hex")$alias("hex_decoded")$cast(pl$String)
)
```

---

expr_str_ends_with *Check if string ends with a regex*

Description

Check if string values end with a substring.

Usage

```
expr_str_ends_with(suffix)
```

Arguments

suffix	Suffix substring or Expr.

Details

See also `$str$starts_with()` and `$str$contains()`.

Value

A polars expression

Examples

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

expr_str_extract *Extract the target capture group from provided patterns*

Description

Extract the target capture group from provided patterns

Usage

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

Arguments

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

Value

A polars expression

Examples

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

expr_str_extract_all *Extract all matches for the given regex pattern*

Description

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

Usage

```
expr_str_extract_all(pattern)
```

Arguments

pattern	A valid regex pattern

Value

A polars expression

Examples

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

---

expr_str_extract_groups

Extract all capture groups for the given regex pattern

Description

Extract all capture groups for the given regex pattern

Usage

```
expr_str_extract_groups(pattern)
```

Arguments

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

Details

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

Value

A polars expression

Examples

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

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

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

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

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

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

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

Description

Use the aho-corasick algorithm to extract matches

Usage

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

Arguments

`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

`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

- strstart_with(): Check if string values start with a substring.
- strends_with(): Check if string values end with a substring.
- strcontains(): 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` *Vertically concatenate the string values in the column to a single string value.*

Description

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

Usage

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

Arguments

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

Value

A polars expression

Examples

```
# concatenate a Series of strings to a single string
df <- pl$DataFrame(foo = c(1, NA, 2))

df$select(pl$col("foo")$str$join("-"))

df$select(pl$col("foo")$str$join("-", ignore_nulls = FALSE))
```

`expr_str_json_decode` *Parse string values as JSON.*

Description

Parse string values as JSON.

Usage

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

Arguments

`dtype`	The dtype to cast the extracted value to. If `NULL`, the dtype will be inferred from the JSON value.
`...`	These dots are for future extensions and must be empty.
`infer_schema_length`	How many rows to parse to determine the schema. If `NULL`, all rows are used.

Details

Throw errors if encounter invalid json strings.

Value

A polars expression

Examples

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

`expr_str_json_path_match` *Extract the first match of JSON string with the provided JSONPath expression*

Description

Extract the first match of JSON string with the provided JSONPath expression

Usage

```
expr_str_json_path_match(json_path)
```

Arguments

`json_path`	A valid JSON path query string.

Details

Throw errors if encounter invalid JSON strings. All return value will be cast to String regardless of the original value.

Documentation on JSONPath standard can be found here: https://goessner.net/articles/JsonPath/.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  json_val = c('{"a":"1"}', NA, '{"a":2}', '{"a":2.1}', '{"a":true}')
)
df$select(pl$col("json_val")$str$json_path_match("$.a"))
```

---

expr_str_len_bytes *Get the number of bytes in strings*

Description

Get length of the strings as UInt32 (as number of bytes). Use `$str$len_chars()` to get the number of characters.

Usage

```
expr_str_len_bytes()
```

Details

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

Value

A polars expression

Examples

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

---

expr_str_len_chars *Get the number of characters in strings*

Description

Get length of the strings as UInt32 (as number of characters). Use `$str$len_bytes()` to get the number of bytes.

Usage

```
expr_str_len_chars()
```

Details

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

Value

A polars expression

Examples

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

expr_str_pad_end *Left justify strings*

Description

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

Usage

```
expr_str_pad_end(length, fill_char = " ")
```

Arguments

`length`	Justify left to this length.
`fill_char`	Fill with this ASCII character.

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c("cow", "monkey", NA, "hippopotamus"))
df$select(pl$col("a")$str$pad_end(8, "*"))
```

expr_str_pad_start *Right justify strings*

Description

Return the string right justified in a string of length `length`.

Usage

```
expr_str_pad_start(length, fill_char = " ")
```

Arguments

`length`	Justify right to this length.
`fill_char`	Fill with this ASCII character.

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c("cow", "monkey", NA, "hippopotamus"))
df$select(pl$col("a")$str$pad_start(8, "*"))
```

---

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

Description

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

Usage

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

Arguments

`pattern`	A character or something can be coerced to a string Expr of a valid regex pattern, compatible with the regex crate.
`value`	A character or an Expr of string that will replace the matched substring.
`...`	These dots are for future extensions and must be empty.
`literal`	Logical. If `TRUE` (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 `literal = FALSE`.

Details

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

Value

A polars expression

Capture groups

The dollar sign ($) is a special character related to capture groups. To refer to a literal dollar sign, use $$ instead or set `literal` to `TRUE`.

See Also

- <Expr>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` *Returns string values in reversed order*

Description

Returns string values in reversed order

Usage

```
expr_str_reverse()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(text = c("foo", "bar", NA))
df$with_columns(reversed = pl$col("text")$str$reverse())
```

`expr_str_slice` *Create subslices of the string values of a String Series*

Description

Create subslices of the string values of a String Series

Usage

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

Arguments

`offset`	Start index. Negative indexing is supported.
`length`	Length of the slice. If `NULL` (default), the slice is taken to the end of the string.

Value

A polars expression

Examples

```
df <- pl$DataFrame(s = c("pear", NA, "papaya", "dragonfruit"))
df$with_columns(
  pl$col("s")$str$slice(-3)$alias("s_sliced")
)
```

expr_str_split — *Split the string by a substring*

Description

Split the string by a substring

Usage

```
expr_str_split(by, ..., inclusive = FALSE)
```

Arguments

by	Substring to split by. Can be an Expr.
...	These dots are for future extensions and must be empty.
inclusive	If TRUE, include the split character/string in the results.

Value

A polars expression

Examples

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

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

expr_str_splitn — *Split the string by a substring, restricted to returning at most* n *items*

Description

If the number of possible splits is less than n-1, the remaining field elements will be null. If the number of possible splits is n-1 or greater, the last (nth) substring will contain the remainder of the string.

Usage

```
expr_str_splitn(by, n)
```

Arguments

`by`	Substring to split by. Can be an Expr.
`n`	Number of splits to make.

Value

A polars expression

Examples

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

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

Description

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

Usage

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

Arguments

`by`	Substring to split by. Can be an Expr.
`n`	Number of splits to make.
`...`	These dots are for future extensions and must be empty.
`inclusive`	If `TRUE`, include the split character/string in the results.

Value

A polars expression

Examples

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

expr_str_starts_with *Check if string starts with a regex*

Description

Check if string values starts with a substring.

Usage

```
expr_str_starts_with(prefix)
```

Arguments

prefix	Prefix substring or Expr.

Details

See also `$str$contains()` and `$str$ends_with()`.

Value

A polars expression

Examples

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

expr_str_strip_chars *Strip leading and trailing characters*

Description

Remove leading and trailing characters.

Usage

```
expr_str_strip_chars(characters = NULL)
```

Arguments

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

Details

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

Value

A polars expression

Examples

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

---

expr_str_strip_chars_end

Strip trailing characters

Description

Remove trailing characters.

Usage

```
expr_str_strip_chars_end(characters = NULL)
```

Arguments

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

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(foo = c(" hello", "\tworld"))
df$select(pl$col("foo")$str$strip_chars_end(" hel\trld"))
df$select(pl$col("foo")$str$strip_chars_end("rldhel\t "))
```

`expr_str_strip_chars_start`
Strip leading characters

Description

Remove leading characters.

Usage

```
expr_str_strip_chars_start(characters = NULL)
```

Arguments

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

Details

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

Value

A polars expression

Examples

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

`expr_str_strip_prefix`
Strip prefix

Description

The prefix will be removed from the string exactly once, if found.

Usage

```
expr_str_strip_prefix(prefix = NULL)
```

Arguments

`prefix`	The prefix to be removed.

Details

This method strips the exact character sequence provided in `prefix` from the start of the input. To strip a set of characters in any order, use `$strip_chars_start()` instead.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c("foobar", "foofoobar", "foo", "bar"))
df$with_columns(
  stripped = pl$col("a")$str$strip_prefix("foo")
)
```

---

expr_str_strip_suffix

Strip suffix

Description

The suffix will be removed from the string exactly once, if found.

Usage

```
expr_str_strip_suffix(suffix = NULL)
```

Arguments

`suffix`	The suffix to be removed.

Details

This method strips the exact character sequence provided in `suffix` from the end of the input. To strip a set of characters in any order, use `$strip_chars_end()` instead.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c("foobar", "foobarbar", "foo", "bar"))
df$with_columns(
  stripped = pl$col("a")$str$strip_suffix("bar")
)
```

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

Description

Similar to the `strptime()` function.

Usage

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

Arguments

`dtype`	The data type to convert into. Can be either `pl$Date`, `pl$Datetime`, or `pl$Time`.
`format`	Format to use for conversion. Refer to the chrono crate documentation for the full specification. Example: `"%Y-%m-%d %H:%M:%S"`. If `NULL` (default), the format is inferred from the data. Notice that time zone `%Z` is not supported and will just ignore timezones. Numeric time zones like `%z` or `%:z` are supported.
`...`	These dots are for future extensions and must be empty.
`strict`	If `TRUE` (default), raise an error if a single string cannot be parsed. If `FALSE`, produce a polars `null`.
`exact`	If `TRUE` (default), require an exact format match. If `FALSE`, allow the format to match anywhere in the target string. Conversion to the Time type is always exact. Note that using `exact = FALSE` introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.
`cache`	Use a cache of unique, converted dates to apply the datetime conversion.
`ambiguous`	Determine how to deal with ambiguous datetimes. Character vector or expression containing the followings: • `"raise"` (default): Throw an error • `"earliest"`: Use the earliest datetime • `"latest"`: Use the latest datetime • `"null"`: Return a null value

Details

When parsing a Datetime the column precision will be inferred from the format string, if given, e.g.: `"%F %T%.3f"` => `pl$Datetime("ms")`. If no fractional second component is found then the default is `"us"` (microsecond).

Value

A polars expression

See Also

- `<Expr>$str$to_date()`
- `<Expr>$str$to_datetime()`
- `<Expr>$str$to_time()`

Examples

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

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

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

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

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

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

# Ignore invalid time
df <- pl$DataFrame(
  x = c(
```

```
    "2023-01-01 11:22:33 -0100",
    "2023-01-01 11:22:33 +0300",
    "invalid time"
  )
)

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

`expr_str_tail` *Return the last n characters of each string*

Description

Return the last n characters of each string

Usage

```
expr_str_tail(n)
```

Arguments

`n`	Length of the slice (integer or expression). Strings are parsed as column names. Negative indexing is supported.

Details

The `n` input is defined in terms of the number of characters in the (UTF-8) string. A character is defined as a Unicode scalar value. A single character is represented by a single byte when working with ASCII text, and a maximum of 4 bytes otherwise.

When the `n` input is negative, `tail()` returns characters starting from the `n`th from the beginning of the string. For example, if `n = -3`, then all characters except the first three are returned.

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  s = c("pear", NA, "papaya", "dragonfruit"),
  n = c(3, 4, -2, -5)
)
```

```
df$with_columns(
  s_tail_5 = pl$col("s")$str$tail(5),
  s_tail_n = pl$col("s")$str$tail("n")
)
```

`expr_str_to_date` *Convert a String column into a Date column*

Description

Convert a String column into a Date column

Usage

```
expr_str_to_date(format = NULL, ..., strict = TRUE, exact = TRUE, cache = TRUE)
```

Arguments

`format`	Format to use for conversion. Refer to the chrono crate documentation for the full specification. Example: `"%Y-%m-%d %H:%M:%S"`. If `NULL` (default), the format is inferred from the data. Notice that time zone `%Z` is not supported and will just ignore timezones. Numeric time zones like `%z` or `%:z` are supported.
`...`	These dots are for future extensions and must be empty.
`strict`	If `TRUE` (default), raise an error if a single string cannot be parsed. If `FALSE`, produce a polars `null`.
`exact`	If `TRUE` (default), require an exact format match. If `FALSE`, allow the format to match anywhere in the target string. Conversion to the Time type is always exact. Note that using `exact = FALSE` introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.
`cache`	Use a cache of unique, converted dates to apply the datetime conversion.

Value

A polars expression

See Also

- `<Expr>$str$strptime()`

Examples

```
df <- pl$DataFrame(x = c("2020/01/01", "2020/02/01", "2020/03/01"))

df$select(pl$col("x")$str$to_date())

# by default, this errors if some values cannot be converted
df <- pl$DataFrame(x = c("2020/01/01", "2020 02 01", "2020-03-01"))
try(df$select(pl$col("x")$str$to_date()))
df$select(pl$col("x")$str$to_date(strict = FALSE))
```

---

expr_str_to_datetime *Convert a String column into a Datetime column*

Description

Convert a String column into a Datetime column

Usage

```
expr_str_to_datetime(
  format = NULL,
  ...,
  time_unit = NULL,
  time_zone = NULL,
  strict = TRUE,
  exact = TRUE,
  cache = TRUE,
  ambiguous = c("raise", "earliest", "latest", "null")
)
```

Arguments

format	Format to use for conversion. Refer to the chrono crate documentation for the full specification. Example: `"%Y-%m-%d %H:%M:%S"`. If `NULL` (default), the format is inferred from the data. Notice that time zone `%Z` is not supported and will just ignore timezones. Numeric time zones like `%z` or `%:z` are supported.
...	These dots are for future extensions and must be empty.
time_unit	Unit of time for the resulting Datetime column. If `NULL` (default), the time unit is inferred from the format string if given, e.g.: `"%F %T%.3f"` => `pl$Datetime("ms")`. If no fractional second component is found, the default is `"us"` (microsecond).
time_zone	for the resulting Datetime column.
strict	If `TRUE` (default), raise an error if a single string cannot be parsed. If `FALSE`, produce a polars `null`.

exact — If TRUE (default), require an exact format match. If FALSE, allow the format to match anywhere in the target string. Note that using exact = FALSE introduces a performance penalty - cleaning your data beforehand will almost certainly be more performant.

cache — Use a cache of unique, converted dates to apply the datetime conversion.

ambiguous — Determine how to deal with ambiguous datetimes. Character vector or expression containing the followings:

- "raise" (default): Throw an error
- "earliest": Use the earliest datetime
- "latest": Use the latest datetime
- "null": Return a null value

Value

A polars expression

See Also

- <Expr>strstrptime()

Examples

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

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

---

expr_str_to_decimal *Convert a String column into a Decimal column*

Description

This method infers the needed parameters precision and scale.

Usage

```
expr_str_to_decimal(..., inference_length = 100)
```

Arguments

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

inference_length — Number of elements to parse to determine the precision and scale.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  numbers = c(
    "40.12", "3420.13", "120134.19", "3212.98",
    "12.90", "143.09", "143.9"
  )
)
df$with_columns(numbers_decimal = pl$col("numbers")$str$to_decimal())
```

---

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

Description

Convert a String column into an Int64 column with base radix

Usage

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

Arguments

...	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>strstrptime()

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(length)
```

Arguments

`length`	Pad the string until it reaches this length. Strings with length equal to or greater than this value are returned as-is. This can be an Expr or something coercible to an Expr. Strings are parsed as column names.

Details

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

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(-1L, 123L, 999999L, NA))
df$with_columns(zfill = pl$col("a")$cast(pl$String)$str$zfill(4))
```

---

expr__abs *Compute absolute values*

Description

Compute absolute values

Usage

```
expr__abs()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = -1:2)
df$with_columns(abs = pl$col("a")$abs())
```

---

expr__add *Add two expressions*

Description

Method equivalent of addition operator `expr + other`.

Usage

```
expr__add(other)
```

Arguments

`other`	Element to add. Can be a string (only if `expr` is a string), a numeric value or an other expression.

Value

A polars expression

See Also

- Arithmetic operators

Examples

```
df <- pl$DataFrame(x = 1:5)

df$with_columns(
  `x+int` = pl$col("x")$add(2L),
  `x+expr` = pl$col("x")$add(pl$col("x")$cum_prod())
)

df <- pl$DataFrame(
  x = c("a", "d", "g"),
  y = c("b", "e", "h"),
  z = c("c", "f", "i")
)

df$with_columns(
  pl$col("x")$add(pl$col("y"))$add(pl$col("z"))$alias("xyz")
)
```

---

expr__agg_groups *Get the group indexes of the group by operation*

Description

Should be used in aggregation context only.

Usage

```
expr__agg_groups()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  group = rep(c("one", "two"), each = 3),
  value = c(94, 95, 96, 97, 97, 99)
)

df$group_by("group", maintain_order = TRUE)$agg(pl$col("value")$agg_groups())
```

---

expr__alias *Rename the expression*

Description

Rename the expression

Usage

```
expr__alias(name)
```

Arguments

`name`	The new name.

Value

A polars expression

Examples

```
# Rename an expression to avoid overwriting an existing column
df <- pl$DataFrame(a = 1:3, b = c("x", "y", "z"))
df$with_columns(
  pl$col("a") + 10,
  pl$col("b")$str$to_uppercase()$alias("c")
)

# Overwrite the default name of literal columns to prevent errors due to
# duplicate column names.
df$with_columns(
  pl$lit(TRUE)$alias("c"),
  pl$lit(4)$alias("d")
)
```

`expr__all` *Check if all boolean values in a column are true*

Description

This method is an expression - not to be confused with `pl$all()` which is a function to select all columns.

Usage

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

Arguments

`...`	These dots are for future extensions and must be empty.
`ignore_nulls`	If `TRUE` (default), ignore null values. If `FALSE`, Kleene logic is used to deal with nulls: if the column contains any null values and no `TRUE` values, the output is null.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(TRUE, TRUE),
  b = c(TRUE, FALSE),
  c = c(NA, TRUE),
  d = c(NA, NA)
)

# By default, ignore null values. If there are only nulls, then all() returns
# TRUE.
df$select(pl$col("*")$all())

# If we set ignore_nulls = FALSE, then we don't know if all values in column
# "c" are TRUE, so it returns null
df$select(pl$col("*")$all(ignore_nulls = FALSE))
```

expr__and — *Apply logical AND on two expressions*

Description

Combine two boolean expressions with AND.

Usage

```
expr__and(other)
```

Arguments

other	Element to add. Can be a string (only if expr is a string), a numeric value or an other expression.

Value

A polars expression

Examples

```
pl$lit(TRUE) & TRUE
pl$lit(TRUE)$and(pl$lit(TRUE))
```

expr__any — *Check if any boolean value in a column is true*

Description

Check if any boolean value in a column is true

Usage

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

Arguments

...	These dots are for future extensions and must be empty.
ignore_nulls	If TRUE (default), ignore null values. If FALSE, Kleene logic is used to deal with nulls: if the column contains any null values and no TRUE values, the output is null.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(TRUE, FALSE),
  b = c(FALSE, FALSE),
  c = c(NA, FALSE)
)

df$select(pl$col("*")$any())

# If we set ignore_nulls = FALSE, then we don't know if any values in column
# "c" is TRUE, so it returns null
df$select(pl$col("*")$any(ignore_nulls = FALSE))
```

---

expr__append *Append expressions*

Description

Append expressions

Usage

```
expr__append(other, ..., upcast = TRUE)
```

Arguments

other	Expression to append.
...	These dots are for future extensions and must be empty.
upcast	If TRUE (default), cast both Series to the same supertype.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 8:10, b = c(NA, 4, 4))
df$select(pl$all()$head(1)$append(pl$all()$tail(1)))
```

expr__approx_n_unique

Approximate count of unique values

Description

This is done using the HyperLogLog++ algorithm for cardinality estimation.

Usage

```
expr__approx_n_unique()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(n = c(1, 1, 2))
df$select(pl$col("n")$approx_n_unique())

df <- pl$DataFrame(n = 0:1000)
df$select(
  exact = pl$col("n")$n_unique(),
  approx = pl$col("n")$approx_n_unique()
)
```

expr__arccos

Compute inverse cosine

Description

Compute inverse cosine

Usage

```
expr__arccos()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(-1, cos(0.5), 0, 1, NA))$
  with_columns(arccos = pl$col("a")$arccos())
```

expr__arccosh *Compute inverse hyperbolic cosine*

Description

Compute inverse hyperbolic cosine

Usage

```
expr__arccosh()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(-1, cosh(0.5), 0, 1, NA))$
  with_columns(arccosh = pl$col("a")$arccosh())
```

expr__arcsin *Compute inverse sine*

Description

Compute inverse sine

Usage

```
expr__arcsin()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(-1, sin(0.5), 0, 1, NA))$
  with_columns(arcsin = pl$col("a")$arcsin())
```

expr__arcsinh *Compute inverse hyperbolic sine*

Description

Compute inverse hyperbolic sine

Usage

```
expr__arcsinh()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(-1, sinh(0.5), 0, 1, NA))$
  with_columns(arcsinh = pl$col("a")$arcsinh())
```

expr__arctan *Compute inverse tangent*

Description

Compute inverse tangent

Usage

```
expr__arctan()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(-1, tan(0.5), 0, 1, NA_real_))$
  with_columns(arctan = pl$col("a")$arctan())
```

expr__arctanh *Compute inverse hyperbolic tangent*

Description

Compute inverse hyperbolic tangent

Usage

```
expr__arctanh()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(-1, tanh(0.5), 0, 1, NA))$
  with_columns(arctanh = pl$col("a")$arctanh())
```

expr__arg_max *Get the index of the maximal value*

Description

Get the index of the maximal value

Usage

```
expr__arg_max()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(20, 10, 30))
df$select(pl$col("a")$arg_max())
```

expr__arg_min *Get the index of the minimal value*

Description

Get the index of the minimal value

Usage

```
expr__arg_min()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(20, 10, 30))
df$select(pl$col("a")$arg_min())
```

expr__arg_sort *Index of a sort*

Description

Get the index values that would sort this column.

Usage

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

Arguments

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

Value

A polars expression

See Also

pl$arg_sort_by() to find the row indices that would sort multiple columns.

Examples

```
pl$DataFrame(
  a = c(6, 1, 0, NA, Inf, NaN)
)$with_columns(arg_sorted = pl$col("a")$arg_sort())
```

---

expr__arg_true *Return indices where expression is true*

Description

Return indices where expression is true

Usage

```
expr__arg_true()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, 1, 2, 1))
df$select((pl$col("a") == 1)$arg_true())
```

---

expr__arg_unique *Get the index of the first unique value*

Description

Get the index of the first unique value

Usage

```
expr__arg_unique()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3, b = c(NA, 4, 4))
df$select(pl$col("a")$arg_unique())
df$select(pl$col("b")$arg_unique())
```

expr__backward_fill *Fill missing values with the next non-null value*

Description

Fill missing values with the next non-null value

Usage

```
expr__backward_fill(limit = NULL)
```

Arguments

fill	The number of consecutive null values to backward fill.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 2, NA),
  b = c(4, NA, 6),
  c = c(NA, NA, 2)
)
df$select(pl$all()$backward_fill())
df$select(pl$all()$backward_fill(limit = 1))
```

expr__bottom_k *Return the* k *smallest elements*

Description

Non-null elements are always preferred over null elements. The output is not guaranteed to be in any particular order, call $sort() after this function if you wish the output to be sorted. This has time complexity $O(n)$.

Usage

```
expr__bottom_k(k = 5)
```

Arguments

k	Number of elements to return.

Value

A polars expression

Examples

```
df <- pl$DataFrame(value = c(1, 98, 2, 3, 99, 4))
df$select(
  top_k = pl$col("value")$top_k(k = 3),
  bottom_k = pl$col("value")$bottom_k(k = 3)
)
```

---

expr__bottom_k_by — *Return the elements corresponding to the* `k` *smallest elements of the* `by` *column(s)*

Description

Non-null elements are always preferred over null elements. The output is not guaranteed to be in any particular order, call $sort() after this function if you wish the output to be sorted. This has time complexity $O(n)$.

Usage

```
expr__bottom_k_by(by, k = 5, ..., reverse = FALSE)
```

Arguments

`by`	Column(s) used to determine the smallest elements. Accepts expression input. Strings are parsed as column names.
`k`	Number of elements to return.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = 1:6,
  b = 6:1,
  c = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")
)

# Get the bottom 2 rows by column a or b:
df$select(
  pl$all()$bottom_k_by("a", 2)$name$suffix("_btm_by_a"),
  pl$all()$bottom_k_by("b", 2)$name$suffix("_btm_by_b")
)
```

```
# Get the bottom 2 rows by multiple columns with given order.
df$select(
  pl$all()$
    bottom_k_by(c("c", "a"), 2, reverse = c(FALSE, TRUE))$
    name$suffix("_btm_by_ca"),
  pl$all()$
    bottom_k_by(c("c", "b"), 2, reverse = c(FALSE, TRUE))$
    name$suffix("_btm_by_cb"),
)

# Get the bottom 2 rows by column a in each group
df$group_by("c", maintain_order = TRUE)$agg(
  pl$all()$bottom_k_by("a", 2)
)$explode(pl$all()$exclude("c"))
```

`expr__cast` *Cast between DataType*

Description

Cast between DataType

Usage

```
expr__cast(dtype, ..., strict = TRUE, wrap_numerical = FALSE)
```

Arguments

`dtype`	DataType to cast to.
`...`	These dots are for future extensions and must be empty.
`strict`	If `TRUE` (default), an error will be thrown if cast failed at resolve time.
`wrap_numerical`	If `TRUE`, numeric casts wrap overflowing values instead of marking the cast as invalid.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3, b = c(1, 2, 3))
df$with_columns(
  pl$col("a")$cast(pl$dtypes$Float64),
  pl$col("b")$cast(pl$dtypes$Int32)
)

# strict FALSE, inserts null for any cast failure
pl$lit(c(100, 200, 300))$cast(pl$dtypes$UInt8, strict = FALSE)$to_series()
```

```
# strict TRUE, raise any failure as an error when query is executed.
tryCatch(
  {
    pl$lit("a")$cast(pl$dtypes$Float64, strict = TRUE)$to_series()
  },
  error = function(e) e
)
```

`expr__cbrt` *Compute cube root*

Description

Compute cube root

Usage

```
expr__cbrt()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(1, 2, 4))$
  with_columns(cbrt = pl$col("a")$cbrt())
```

`expr__ceil` *Rounds up to the nearest integer value*

Description

This only works on floating point Series.

Usage

```
expr__ceil()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(0.3, 0.5, 1.0, 1.1))
df$with_columns(
  ceil = pl$col("a")$ceil()
)
```

`expr__clip` *Set values outside the given boundaries to the boundary value*

Description

This method only works for numeric and temporal columns. To clip other data types, consider writing a when-then-otherwise expression.

Usage

```
expr__clip(lower_bound = NULL, upper_bound = NULL)
```

Arguments

`lower_bound`	Lower bound. Accepts expression input. Non-expression inputs are parsed as literals.
`upper_bound`	Upper bound. Accepts expression input. Non-expression inputs are parsed as literals.

Details

This method only works for numeric and temporal columns. To clip other data types, consider writing a when-then-otherwise expression.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(-50, 5, 50, NA))

# Specifying both a lower and upper bound:
df$with_columns(
  clip = pl$col("a")$clip(1, 10)
)

# Specifying only a single bound:
df$with_columns(
  clip = pl$col("a")$clip(upper_bound = 10)
)
```

expr__cos *Compute cosine*

Description

Compute cosine

Usage

```
expr__cos()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(0, pi / 2, pi, NA))$
  with_columns(cosine = pl$col("a")$cos())
```

expr__cosh *Compute hyperbolic cosine*

Description

Compute hyperbolic cosine

Usage

```
expr__cosh()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(-1, acosh(2), 0, 1, NA))$
  with_columns(cosh = pl$col("a")$cosh())
```

expr__cot *Compute cotangent*

Description

Compute cotangent

Usage

```
expr__cot()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(0, pi / 2, -5, NA))$
  with_columns(cotangent = pl$col("a")$cot())
```

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

Description

Get the number of non-null elements in the column

Usage

```
expr__count()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3, b = c(NA, 4, 4))
df$select(pl$all()$count())
```

`expr__cumulative_eval` *Return the cumulative count of the non-null values in the column*

Description

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

Usage

```
expr__cumulative_eval(expr, ..., min_periods = 1, parallel = FALSE)
```

Arguments

`expr`	Expression to evaluate.
`...`	These dots are for future extensions and must be empty.
`min_periods`	Number of valid values (i.e. `length - null_count`) there should be in the window before the expression is evaluated.
`parallel`	Run in parallel. Don't do this in a group by or another operation that already has much parallelization.

Details

This can be really slow as it can have `O(n^2)` complexity. Don't use this for operations that visit all elements.

Value

A polars expression

Examples

```
df <- pl$DataFrame(values = 1:5)
df$with_columns(
  pl$col("values")$cumulative_eval(
    pl$element()$first() - pl$element()$last()**2
  )
)
```

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

Description

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

Usage

```
expr__cum_count(..., reverse = FALSE)
```

Arguments

...	These dots are for future extensions and must be empty.
reverse	If TRUE, reverse the count.

Value

A polars expression

Examples

```
pl$DataFrame(a = 1:4)$with_columns(
  cum_count = pl$col("a")$cum_count(),
  cum_count_reversed = pl$col("a")$cum_count(reverse = TRUE)
)
```

expr__cum_max *Return the cumulative max computed at every element.*

Description

Return the cumulative max computed at every element.

Usage

```
expr__cum_max(..., reverse = FALSE)
```

Arguments

...	These dots are for future extensions and must be empty.
reverse	If TRUE, start from the last value.

Details

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

Value

A polars expression

Examples

```
pl$DataFrame(a = c(1:4, 2L))$with_columns(
  cum_max = pl$col("a")$cum_max(),
  cum_max_reversed = pl$col("a")$cum_max(reverse = TRUE)
)
```

---

expr__cum_min *Return the cumulative min computed at every element.*

Description

Return the cumulative min computed at every element.

Usage

```
expr__cum_min(..., reverse = FALSE)
```

Arguments

...	These dots are for future extensions and must be empty.
reverse	If TRUE, start from the last value.

Details

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

Value

A polars expression

Examples

```
pl$DataFrame(a = c(1:4, 2L))$with_columns(
  cum_min = pl$col("a")$cum_min(),
  cum_min_reversed = pl$col("a")$cum_min(reverse = TRUE)
)
```

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

Description

Return the cumulative product computed at every element.

Usage

```
expr__cum_prod(..., reverse = FALSE)
```

Arguments

...	These dots are for future extensions and must be empty.
reverse	If TRUE, start with the total product of elements and divide each row one by one.

Details

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

Value

A polars expression

Examples

```
pl$DataFrame(a = 1:4)$with_columns(
  cum_prod = pl$col("a")$cum_prod(),
  cum_prod_reversed = pl$col("a")$cum_prod(reverse = TRUE)
)
```

expr__cum_sum *Return the cumulative sum computed at every element.*

Description

Return the cumulative sum computed at every element.

Usage

```
expr__cum_sum(..., reverse = FALSE)
```

Arguments

...	These dots are for future extensions and must be empty.
reverse	If TRUE, start with the total sum of elements and substract each row one by one.

Details

The Dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

Value

A polars expression

Examples

```
pl$DataFrame(a = 1:4)$with_columns(
  cum_sum = pl$col("a")$cum_sum(),
  cum_sum_reversed = pl$col("a")$cum_sum(reverse = TRUE)
)
```

---

expr__cut *Bin continuous values into discrete categories*

Description

[**Experimental**]

Usage

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

Arguments

breaks	List of unique cut points.
...	These dots are for future extensions and must be empty.
labels	Names of the categories. The number of labels must be equal to the number of cut points plus one.
left_closed	Set the intervals to be left-closed instead of right-closed.
include_breaks	Include a column with the right endpoint of the bin each observation falls in. This will change the data type of the output from a Categorical to a Struct.

Value

A polars expression

Examples

```
# Divide a column into three categories.
df <- pl$DataFrame(foo = -2:2)
df$with_columns(
  cut = pl$col("foo")$cut(c(-1, 1), labels = c("a", "b", "c"))
)

# Add both the category and the breakpoint.
df$with_columns(
  cut = pl$col("foo")$cut(c(-1, 1), include_breaks = TRUE)
)$unnest()
```

---

expr__degrees *Convert from radians to degrees*

Description

Convert from radians to degrees

Usage

```
expr__degrees()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(1, 2, 4) * pi)$
  with_columns(degrees = pl$col("a")$degrees())
```

---

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

Description

Calculate the n-th discrete difference between elements

Usage

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

Arguments

`n`	Integer indicating the number of slots to shift.
`null_behavior`	How to handle null values. Must be `"ignore"` (default), or `"drop"`.

Value

A polars expression

Examples

```
pl$DataFrame(a = c(20, 10, 30, 25, 35))$with_columns(
  diff_default = pl$col("a")$diff(),
  diff_2_ignore = pl$col("a")$diff(2, "ignore")
)
```

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

Description

Compute the dot/inner product between two Expressions

Usage

```
expr__dot(expr)
```

Arguments

`other`	Expression to compute dot product with.

Value

A polars expression

Examples

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

expr__drop_nans *Drop all floating point NaN values*

Description

The original order of the remaining elements is preserved. A `NaN` value is not the same as a `null` value. To drop `null` values, use $drop_nulls().

Usage

```
expr__drop_nans()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, NA, 3, NaN))
df$select(pl$col("a")$drop_nans())
```

expr__drop_nulls *Drop all floating point null values*

Description

The original order of the remaining elements is preserved. A `null` value is not the same as a `NaN` value. To drop `NaN` values, use $drop_nans().

Usage

```
expr__drop_nulls()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, NA, 3, NaN))
df$select(pl$col("a")$drop_nulls())
```

expr__entropy *Compute entropy*

Description

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

Usage

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

Arguments

`base`	Numeric value used as base, defaults to `exp(1)`.
`...`	These dots are for future extensions and must be empty.
`normalize`	Normalize `pk` if it doesn't sum to 1.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a")$entropy(base = 2))
df$select(pl$col("a")$entropy(base = 2, normalize = FALSE))
```

expr__eq *Check equality*

Description

This propagates null values, i.e. any comparison involving `null` will return `null`. Use `$eq_missing()` to consider null values as equal.

Usage

```
expr__eq(other)
```

Arguments

`other`	A literal or expression value to compare with.

Value

A polars expression

See Also

expr___eq_missing

Examples

```
df <- pl$DataFrame(x = c(NA, FALSE, TRUE), y = c(TRUE, TRUE, TRUE))
df$with_columns(
  eq = pl$col("x")$eq(pl$col("y")),
  eq_missing = pl$col("x")$eq_missing(pl$col("y"))
)
```

---

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

Description

This considers that null values are equal. It differs from `$eq()` where null values are propagated.

Usage

```
expr__eq_missing(other)
```

Arguments

`other`	A literal or expression value to compare with.

Value

A polars expression

See Also

expr___eq

Examples

```
df <- pl$DataFrame(x = c(NA, FALSE, TRUE), y = c(TRUE, TRUE, TRUE))
df$with_columns(
  eq = pl$col("x")$eq("y"),
  eq_missing = pl$col("x")$eq_missing("y")
)
```

`expr__ewm_mean` *Compute exponentially-weighted moving mean*

Description

Compute exponentially-weighted moving mean

Usage

```
expr__ewm_mean(
  ...,
  com,
  span,
  half_life,
  alpha,
  adjust = TRUE,
  min_periods = 1,
  ignore_nulls = FALSE
)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`com`	Specify decay in terms of center of mass, γ, with $$\alpha = \frac{1}{1+\gamma} \; \forall \; \gamma \geq 0$$.
`span`	Specify decay in terms of span, θ, with $$\alpha = \frac{2}{\theta+1} \; \forall \; \theta \geq 1$$
`half_life`	Specify decay in terms of half-life, λ, with $$\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \; \forall \; \lambda > 0$$
`alpha`	Specify smoothing factor alpha directly, $0 < \alpha \leq 1$.
`adjust`	Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings: • when `TRUE` (default), the EW function is calculated using weights $w_i = (1-\alpha)^i$; • when `FALSE`, the EW function is calculated recursively by $$y_0 = x_0$$ $$y_t = (1-\alpha)y_{t-1} + \alpha x_t$$

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

`ignore_nulls` Ignore missing values when calculating weights.

- when `FALSE` (default), weights are based on absolute positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of (x_0, null, x_2) are $(1-\alpha)^2$ and 1 if `adjust = TRUE`, and $(1-\alpha)^2$ and α if `adjust = FALSE`.
- when `TRUE`, weights are based on relative positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of (x_0, null, x_2) are $1-\alpha$ and 1 if `adjust = TRUE`, and $1-\alpha$ and α if `adjust = FALSE`.

Value

A polars expression

Examples

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

`expr__ewm_mean_by` *Compute time-based exponentially weighted moving average*

Description

Given observations x_0, x_1, ..., x_{n-1} at times t_0, t_1, ..., t_{n-1}, the EWMA is calculated as

$$y_0 = x_0$$

$$\alpha_i = 1 - \exp\left\{\frac{-\ln(2)(t_i - t_{i-1})}{\tau}\right\}$$

$$y_i = \alpha_i x_i + (1-\alpha_i) y_{i-1}; \quad i > 0$$

where τ is the `half_life`.

Usage

```
expr__ewm_mean_by(by, ..., half_life)
```

Arguments

`by` Times to calculate average by. Should be DateTime, Date, UInt64, UInt32, Int64, or Int32 data type.

`half_life` Unit over which observation decays to half its value. Can be created either from a timedelta, or by using the following string language:

- 1ns (1 nanosecond)

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

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

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  values = c(0, 1, 2, NA, 4),
  times = as.Date(
    c("2020-01-01", "2020-01-03", "2020-01-10", "2020-01-15", "2020-01-17")
  )
)
df$with_columns(
  result = pl$col("values")$ewm_mean_by("times", half_life = "4d")
)
```

`expr__ewm_std` *Compute exponentially-weighted moving standard deviation*

Description

Compute exponentially-weighted moving standard deviation

Usage

```
expr__ewm_std(
  ...,
  com,
  span,
  half_life,
```

```
  alpha,
  adjust = TRUE,
  bias = FALSE,
  min_periods = 1,
  ignore_nulls = FALSE
)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`com`	Specify decay in terms of center of mass, γ, with $$\alpha = \frac{1}{1+\gamma} \; \forall \; \gamma \geq 0$$.
`span`	Specify decay in terms of span, θ, with $$\alpha = \frac{2}{\theta+1} \; \forall \; \theta \geq 1$$
`half_life`	Specify decay in terms of half-life, λ, with $$\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \; \forall \; \lambda > 0$$
`alpha`	Specify smoothing factor alpha directly, $0 < \alpha \leq 1$.
`adjust`	Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings: • when `TRUE` (default), the EW function is calculated using weights $w_i = (1-\alpha)^i$; • when `FALSE`, the EW function is calculated recursively by $$y_0 = x_0$$ $$y_t = (1-\alpha)y_{t-1} + \alpha x_t$$
`bias`	If `FALSE` (default), apply a correction to make the estimate statistically unbiased.
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`ignore_nulls`	Ignore missing values when calculating weights. • when `FALSE` (default), weights are based on absolute positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of (x_0, null, x_2) are $(1-\alpha)^2$ and 1 if `adjust = TRUE`, and $(1-\alpha)^2$ and α if `adjust = FALSE`. • when `TRUE`, weights are based on relative positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of (x_0, null, x_2) are $1-\alpha$ and 1 if `adjust = TRUE`, and $1-\alpha$ and α if `adjust = FALSE`.

Value

A polars expression

Examples

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

---

expr__ewm_var *Compute exponentially-weighted moving variance*

Description

Compute exponentially-weighted moving variance

Usage

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

Arguments

...	These dots are for future extensions and must be empty.
com	Specify decay in terms of center of mass, γ, with $$\alpha = \frac{1}{1+\gamma} \; \forall \; \gamma \geq 0$$.
span	Specify decay in terms of span, θ, with $$\alpha = \frac{2}{\theta+1} \; \forall \; \theta \geq 1$$
half_life	Specify decay in terms of half-life, λ, with $$\alpha = 1 - \exp\left\{\frac{-\ln(2)}{\lambda}\right\} \; \forall \; \lambda > 0$$
alpha	Specify smoothing factor alpha directly, $0 < \alpha \leq 1$.

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

- when `TRUE` (default), the EW function is calculated using weights $w_i = (1 - \alpha)^i$;
- when `FALSE`, the EW function is calculated recursively by

$$y_0 = x_0$$

$$y_t = (1 - \alpha)y_{t-1} + \alpha x_t$$

`bias` If `FALSE` (default), apply a correction to make the estimate statistically unbiased.

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

`ignore_nulls` Ignore missing values when calculating weights.

- when `FALSE` (default), weights are based on absolute positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of (x_0, null, x_2) are $(1 - \alpha)^2$ and 1 if `adjust = TRUE`, and $(1 - \alpha)^2$ and α if `adjust = FALSE`.
- when `TRUE`, weights are based on relative positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of (x_0, null, x_2) are $1 - \alpha$ and 1 if `adjust = TRUE`, and $1 - \alpha$ and α if `adjust = FALSE`.

Value

A polars expression

Examples

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

---

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

Description

Exclude columns from a multi-column expression.

Usage

```
expr__exclude(...)
```

Arguments

`...` The name or datatype of the column(s) to exclude. Accepts regular expression input. Regular expressions should start with `^` and end with `$`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(aa = 1:2, ba = c("a", NA), cc = c(NA, 2.5))
df

# Exclude by column name(s):
df$select(pl$all()$exclude("ba"))

# Exclude by regex, e.g. removing all columns whose names end with the
# letter "a":
df$select(pl$all()$exclude("^.*a$"))

# Exclude by dtype(s), e.g. removing all columns of type Int64 or Float64:
df$select(pl$all()$exclude(pl$Int64, pl$Float64))
```

---

expr__exp *Compute the exponential*

Description

Compute the exponential

Usage

```
expr__exp()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(1, 2, 4))$
  with_columns(exp = pl$col("a")$exp())
```

expr__explode *Explode a list expression*

Description

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

Usage

```
expr__explode()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  groups = c("a", "b"),
  values = list(1:2, 3:4)
)

df$select(pl$col("values")$explode())
```

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

Description

Extend the Series with `n` copies of a value

Usage

```
expr__extend_constant(value, n)
```

Arguments

`value`	A constant literal value or a unit expression with which to extend the expression result Series. This can be `NA` to extend with nulls.
`n`	The number of additional values that will be added.

Value

A polars expression

Examples

```
df <- pl$DataFrame(values = 1:3)
df$select(pl$col("values")$extend_constant(99, n = 2))
```

---

expr__fill_nan *Fill floating point* NaN *value with a fill value*

Description

Fill floating point NaN value with a fill value

Usage

```
expr__fill_nan(value)
```

Arguments

value	Value used to fill NaN values.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, NA, 2, NaN))
df$with_columns(
  filled_nan = pl$col("a")$fill_nan(99)
)
```

---

expr__fill_null *Fill floating point null value with a fill value*

Description

Fill floating point null value with a fill value

Usage

```
expr__fill_null(value, strategy = NULL, limit = NULL)
```

Arguments

value	Value used to fill null values. Can be missing if strategy is specified. Accepts expression input, strings are parsed as column names.
strategy	Strategy used to fill null values. Must be one of "forward", "backward", "min", "max", "mean", "zero", "one".
limit	Number of consecutive null values to fill when using the "forward" or "backward" strategy.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, NA, 2, NaN))
df$with_columns(
  filled_null_zero = pl$col("a")$fill_null(strategy = "zero"),
  filled_null_99 = pl$col("a")$fill_null(99),
  filled_null_forward = pl$col("a")$fill_null(strategy = "forward"),
  filled_null_expr = pl$col("a")$fill_null(pl$col("a")$median())
)
```

---

expr__filter *Filter the expression based on one or more predicate expressions*

Description

Elements where the filter does not evaluate to `TRUE` are discarded, including nulls. This is mostly useful in an aggregation context. If you want to filter on a DataFrame level, use `DataFrame$filter()` or `LazyFrame$filter()`.

Usage

```
expr__filter(...)
```

Arguments

...	<dynamic-dots> Expression(s) that evaluate to a boolean Series.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  group_col = c("g1", "g1", "g2"),
  b = c(1, 2, 3)
)
df

df$group_by("group_col")$agg(
  lt = pl$col("b")$filter(pl$col("b") < 2),
  gte = pl$col("b")$filter(pl$col("b") >= 2)
)
```

expr__first *Get the first value*

Description

Get the first value

Usage

```
expr__first()
```

Value

A polars expression

Examples

```
pl$DataFrame(x = 3:1)$with_columns(first = pl$col("x")$first())
```

expr__flatten *Flatten a list or string column*

Description

This is an alias for $explode().

Usage

```
expr__flatten()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  group = c("a", "b", "b"),
  values = list(1:2, 2:3, 4)
)

df$group_by("group")$agg(pl$col("values")$flatten())
```

expr__floor *Rounds down to the nearest integer value*

Description

This only works on floating point Series.

Usage

```
expr__floor()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(0.3, 0.5, 1.0, 1.1))
df$with_columns(
  floor = pl$col("a")$floor()
)
```

expr__floor_div *Floor divide using two expressions*

Description

Method equivalent of floor division operator `expr %/% other`. `$floordiv()` is an alias for `$floor_div()`, which exists for compatibility with Python Polars.

Usage

```
expr__floor_div(other)

expr__floordiv(other)
```

Arguments

`other`	Numeric literal or expression value.

Value

A polars expression

See Also

- Arithmetic operators
- <Expr>$true_div()
- <Expr>$mod()

Examples

```
df <- pl$DataFrame(x = 1:5)

df$with_columns(
  `x/2` = pl$col("x")$true_div(2),
  `x%/%2` = pl$col("x")$floor_div(2)
)
```

---

expr__forward_fill *Fill missing values with the last non-null value*

Description

Fill missing values with the last non-null value

Usage

```
expr__forward_fill(limit = NULL)
```

Arguments

fill	The number of consecutive null values to forward fill.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 2, NA),
  b = c(4, NA, 6),
  c = c(2, NA, NA)
)
df$select(pl$all()$forward_fill())
df$select(pl$all()$forward_fill(limit = 1))
```

expr__gather *Take values by index*

Description

Take values by index

Usage

```
expr__gather(indices)
```

Arguments

indices	An expression that leads to a UInt32 dtyped Series.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  group = c("one", "one", "one", "two", "two", "two"),
  value = c(1, 98, 2, 3, 99, 4)
)
df$group_by("group", maintain_order = TRUE)$agg(
  pl$col("value")$gather(c(2, 1))
)
```

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

Description

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

Usage

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

Arguments

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(foo = 1:9)
df$select(pl$col("foo")$gather_every(3))
df$select(pl$col("foo")$gather_every(3, offset = 1))
```

---

expr__ge *Check greater or equal inequality*

Description

Check greater or equal inequality

Usage

```
expr__ge(other)
```

Arguments

other	A literal or expression value to compare with.

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = 1:3)
df$with_columns(
  with_ge = pl$col("x")$ge(pl$lit(2)),
  with_symbol = pl$col("x") >= pl$lit(2)
)
```

---

expr__get *Return a single value by index*

Description

Return a single value by index

Usage

```
expr__get(index)
```

Arguments

index	An expression that leads to a UInt32 dtyped Series.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  group = c("one", "one", "one", "two", "two", "two"),
  value = c(1, 98, 2, 3, 99, 4)
)
df$group_by("group", maintain_order = TRUE)$agg(
  pl$col("value")$get(1)
)
```

---

expr__gt *Check greater or equal inequality*

Description

Check greater or equal inequality

Usage

```
expr__gt(other)
```

Arguments

other	A literal or expression value to compare with.

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = 1:3)
df$with_columns(
  with_gt = pl$col("x")$gt(pl$lit(2)),
  with_symbol = pl$col("x") > pl$lit(2)
)
```

expr__hash *Hash elements*

Description

Hash elements

Usage

```
expr__hash(seed = 0, seed_1 = NULL, seed_2 = NULL, seed_3 = NULL)
```

Arguments

seed	Integer, random seed parameter. Defaults to 0.
seed_1, seed_2, seed_3	Integer, random seed parameters. Default to seed if not set.

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, 2, NA), b = c("x", NA, "z"))
df$with_columns(pl$all()$hash(10, 20, 30, 40))
```

expr__has_nulls *Check whether the expression contains one or more null values*

Description

Check whether the expression contains one or more null values

Usage

```
expr__has_nulls()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(NA, 1, NA),
  b = c(10, NA, 300),
  c = c(350, 650, 850)
)
df$select(pl$all()$has_nulls())
```

`expr__head` *Get the first n elements*

Description

Get the first n elements

Usage

```
expr__head(n = 10)
```

Arguments

`n`	Number of elements to take.

Value

A polars expression

Examples

```
pl$DataFrame(x = 1:11)$select(pl$col("x")$head(3))
```

`expr__hist` *Bin values into buckets and count their occurrences*

Description

[**Experimental**]

Usage

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

Arguments

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, 3, 8, 8, 2, 1, 3))
df$select(pl$col("a")$hist(bins = 1:3))
df$select(
  pl$col("a")$hist(
    bins = 1:3, include_category = TRUE, include_breakpoint = TRUE
  )
)
```

`expr__implode` *Aggregate values into a list*

Description

Aggregate values into a list

Usage

```
expr__implode()
```

Value

A polars expression

Examples

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

expr__interpolate *Fill null values using interpolation*

Description

Fill null values using interpolation

Usage

```
expr__interpolate(method = c("linear", "nearest"))
```

Arguments

method	Interpolation method. Must be one of "linear" or "nearest".

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, NA, 3), b = c(1, NaN, 3))
df$with_columns(
  a_interpolated = pl$col("a")$interpolate(),
  b_interpolated = pl$col("b")$interpolate()
)
```

expr__interpolate_by *Fill null values using interpolation based on another column*

Description

Fill null values using interpolation based on another column

Usage

```
expr__interpolate_by(by)
```

Arguments

by	Column to interpolate values based on.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, NA, NA, 3), b = c(1, 2, 7, 8))
df$with_columns(
  a_interpolated = pl$col("a")$interpolate_by("b")
)
```

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

Description

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

Usage

```
expr__is_between(
  lower_bound,
  upper_bound,
  closed = c("both", "left", "right", "none")
)
```

Arguments

`lower_bound`	Lower bound value. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals.
`upper_bound`	Upper bound value. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals.
`closed`	Define which sides of the interval are closed (inclusive). Must be one of `"left"`, `"right"`, `"both"` or `"none"`.

Details

If the value of the `lower_bound` is greater than that of the `upper_bound` then the result will be `FALSE`, as no value can satisfy the condition.

Value

A polars expression

Examples

```
df <- pl$DataFrame(num = 1:5)
df$with_columns(
  is_between = pl$col("num")$is_between(2, 4)
)

# Use the closed argument to include or exclude the values at the bounds:
```

```
df$with_columns(
  is_between = pl$col("num")$is_between(2, 4, closed = "left")
)

# You can also use strings as well as numeric/temporal values (note: ensure
# that string literals are wrapped with lit so as not to conflate them with
# column names):
df <- pl$DataFrame(a = letters[1:5])
df$with_columns(
  is_between = pl$col("a")$is_between(pl$lit("a"), pl$lit("c"))
)

# Use column expressions as lower/upper bounds, comparing to a literal value:
df <- pl$DataFrame(a = 1:5, b = 5:1)
df$with_columns(
  between_ab = pl$lit(3)$is_between(pl$col("a"), pl$col("b"))
)
```

---

expr__is_duplicated *Return a boolean mask indicating duplicated values*

Description

Return a boolean mask indicating duplicated values

Usage

```
expr__is_duplicated()
```

Value

A polars expression

Examples

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

---

expr__is_finite *Check if elements are finite*

Description

Check if elements are finite

Usage

```
expr__is_finite()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, 2), b = c(3, Inf))
df$with_columns(
  a_finite = pl$col("a")$is_finite(),
  b_finite = pl$col("b")$is_finite()
)
```

---

expr__is_first_distinct

Return a boolean mask indicating the first occurrence of each distinct value

---

Description

Return a boolean mask indicating the first occurrence of each distinct value

Usage

```
expr__is_first_distinct()
```

Value

A polars expression

Examples

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

---

expr__is_in

Check if elements of an expression are present in another expression

---

Description

Check if elements of an expression are present in another expression

Usage

```
expr__is_in(other)
```

Arguments

other	Accepts expression input. Strings are parsed as column names.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  sets = list(1:3, 1:2, 9:10),
  optional_members = 1:3
)
df$with_columns(
  contains = pl$col("optional_members")$is_in("sets")
)
```

---

expr__is_infinite *Check if elements are infinite*

Description

Check if elements are infinite

Usage

```
expr__is_infinite()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, 2), b = c(3, Inf))
df$with_columns(
  a_infinite = pl$col("a")$is_infinite(),
  b_infinite = pl$col("b")$is_infinite()
)
```

`expr__is_last_distinct` *Return a boolean mask indicating the last occurrence of each distinct value*

Description

Return a boolean mask indicating the last occurrence of each distinct value

Usage

```
expr__is_last_distinct()
```

Value

A polars expression

Examples

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

`expr__is_nan` *Check if elements are NaN*

Description

Floating point `NaN` (Not A Number) should not be confused with missing data represented as `NA` (in R) or `null` (in Polars).

Usage

```
expr__is_nan()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 2, NA, 1, 5),
  b = c(1, 2, NaN, 1, 5)
)
df$with_columns(
  a_nan = pl$col("a")$is_nan(),
  b_nan = pl$col("b")$is_nan()
)
```

---

expr__is_not_nan *Check if elements are not NaN*

Description

Floating point `NaN` (Not A Number) should not be confused with missing data represented as `NA` (in R) or `null` (in Polars).

Usage

```
expr__is_not_nan()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 2, NA, 1, 5),
  b = c(1, 2, NaN, 1, 5)
)
df$with_columns(
  a_not_nan = pl$col("a")$is_not_nan(),
  b_not_nan = pl$col("b")$is_not_nan()
)
```

---

expr__is_not_null *Check if elements are not NULL*

Description

Check if elements are not NULL

Usage

```
expr__is_not_null()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 2, NA, 1, 5),
  b = c(1, 2, NaN, 1, 5)
)
df$with_columns(
  a_not_null = pl$col("a")$is_not_null(),
  b_not_null = pl$col("b")$is_not_null()
)
```

---

expr__is_null *Check if elements are NULL*

Description

Check if elements are NULL

Usage

```
expr__is_null()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 2, NA, 1, 5),
  b = c(1, 2, NaN, 1, 5)
)
df$with_columns(
  a_null = pl$col("a")$is_null(),
  b_null = pl$col("b")$is_null()
)
```

expr__is_unique *Return a boolean mask indicating unique values*

Description

Return a boolean mask indicating unique values

Usage

```
expr__is_unique()
```

Value

A polars expression

Examples

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

expr__kurtosis *Compute the kurtosis (Fisher or Pearson)*

Description

Kurtosis is the fourth central moment divided by the square of the variance. If Fisher's definition is used, then 3.0 is subtracted from the result to give 0.0 for a normal distribution. If `bias` is `FALSE` then the kurtosis is calculated using `k` statistics to eliminate bias coming from biased moment estimators.

Usage

```
expr__kurtosis(..., fisher = TRUE, bias = TRUE)
```

Arguments

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = c(1, 2, 3, 2, 1))
df$select(pl$col("x")$kurtosis())
```

---

expr__last *Get the last value*

Description

Get the last value

Usage

```
expr__last()
```

Value

A polars expression

Examples

```
pl$DataFrame(x = 3:1)$with_columns(last = pl$col("x")$last())
```

---

expr__le *Check lower or equal inequality*

Description

Check lower or equal inequality

Usage

```
expr__le(other)
```

Arguments

other	A literal or expression value to compare with.

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = 1:3)
df$with_columns(
  with_le = pl$col("x")$le(pl$lit(2)),
  with_symbol = pl$col("x") <= pl$lit(2)
)
```

expr__len *Return the number of elements in the column*

Description

Null values are counted in the total.

Usage

```
expr__len()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3, b = c(NA, 4, 4))
df$select(pl$all()$len())
```

expr__limit *Get the first n rows*

Description

This is an alias for $head().

Usage

```
expr__limit(n = 10)
```

Arguments

n	Number of rows to return.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:9)
df$select(pl$col("a")$limit(3))
```

expr__log — *Compute the logarithm*

Description

Compute the logarithm

Usage

```
expr__log(base = exp(1))
```

Arguments

base	Numeric value used as base, defaults to `exp(1)`.

Value

A polars expression

Examples

```
pl$DataFrame(a = c(1, 2, 4))$
  with_columns(
  log = pl$col("a")$log(),
  log_base_2 = pl$col("a")$log(base = 2)
)
```

expr__log10 — *Compute the base-10 logarithm*

Description

Compute the base-10 logarithm

Usage

```
expr__log10()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(1, 2, 4))$
  with_columns(log10 = pl$col("a")$log10())
```

expr__log1p *Compute the natural logarithm plus one*

Description

This computes `log(1 + x)` but is more numerically stable for `x` close to zero.

Usage

```
expr__log1p()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(1, 2, 4))$
  with_columns(log1p = pl$col("a")$log1p())
```

expr__lower_bound *Calculate the lower bound*

Description

Returns a unit Series with the lowest value possible for the dtype of this expression.

Usage

```
expr__lower_bound()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a")$lower_bound())
```

expr__lt *Check strictly lower inequality*

Description

Check strictly lower inequality

Usage

```
expr__lt(other)
```

Arguments

other	A literal or expression value to compare with.

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = 1:3)
df$with_columns(
  with_lt = pl$col("x")$lt(pl$lit(2)),
  with_symbol = pl$col("x") < pl$lit(2)
)
```

expr__max *Get the maximum value*

Description

Get the maximum value

Usage

```
expr__max()
```

Value

A polars expression

Examples

```
pl$DataFrame(x = c(1, NaN, 3))$
  with_columns(max = pl$col("x")$max())
```

expr__mean *Get mean value*

Description

Get mean value

Usage

```
expr__mean()
```

Value

A polars expression

Examples

```
pl$DataFrame(x = c(1, 3, 4, NA))$
  with_columns(mean = pl$col("x")$mean())
```

expr__median *Get median value*

Description

Get median value

Usage

```
expr__median()
```

Value

A polars expression

Examples

```
pl$DataFrame(x = c(1, 3, 4, NA))$
  with_columns(median = pl$col("x")$median())
```

expr__min *Get the minimum value*

Description

Get the minimum value

Usage

```
expr__min()
```

Value

A polars expression

Examples

```
pl$DataFrame(x = c(1, NaN, 3))$
  with_columns(min = pl$col("x")$min())
```

expr__mod *Modulo using two expressions*

Description

Method equivalent of modulus operator `expr %% other`.

Usage

```
expr__mod(other)
```

Arguments

`other`	Numeric literal or expression value.

Value

A polars expression

See Also

- Arithmetic operators
- `<Expr>$floor_div()`

Examples

```
df <- pl$DataFrame(x = -5L:5L)

df$with_columns(
  `x%%2` = pl$col("x")$mod(2)
)
```

---

expr__mode *Compute the most occurring value(s)*

Description

Compute the most occurring value(s)

Usage

```
expr__mode()
```

Value

A polars expression

Examples

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

---

expr__mul *Multiply two expressions*

Description

Method equivalent of multiplication operator `expr * other`.

Usage

```
expr__mul(other)
```

Arguments

`other`	Numeric literal or expression value.

Value

A polars expression

See Also

- Arithmetic operators

Examples

```
df <- pl$DataFrame(x = c(1, 2, 4, 8, 16))

df$with_columns(
  `x*2` = pl$col("x")$mul(2),
  `x * xlog2` = pl$col("x")$mul(pl$col("x")$log(2))
)
```

---

expr__nan_max *Get the maximum value with NaN*

Description

This returns `NaN` if there are any.

Usage

```
expr__nan_max()
```

Value

A polars expression

Examples

```
pl$DataFrame(x = c(1, NA, 3, NaN, Inf))$
  with_columns(nan_max = pl$col("x")$nan_max())
```

---

expr__nan_min *Get the minimum value with NaN*

Description

This returns `NaN` if there are any.

Usage

```
expr__nan_min()
```

Value

A polars expression

Examples

```
pl$DataFrame(x = c(1, NA, 3, NaN, Inf))$
  with_columns(nan_min = pl$col("x")$nan_min())
```

`expr__ne` *Check inequality*

Description

This propagates null values, i.e. any comparison involving `null` will return `null`. Use `$ne_missing()` to consider null values as equal.

Usage

```
expr__ne(other)
```

Arguments

`other`	A literal or expression value to compare with.

Value

A polars expression

See Also

expr___ne_missing

Examples

```
df <- pl$DataFrame(x = c(NA, FALSE, TRUE), y = c(TRUE, TRUE, TRUE))
df$with_columns(
  ne = pl$col("x")$ne(pl$col("y")),
  ne_missing = pl$col("x")$ne_missing(pl$col("y"))
)
```

expr__ne_missing *Check inequality without* null *propagation*

Description

Method equivalent of addition operator `expr + other`.

Usage

```
expr__ne_missing(other)
```

Arguments

other	Element to add. Can be a string (only if `expr` is a string), a numeric value or an other expression.

Value

A polars expression

See Also

expr___ne

Examples

```
df <- pl$DataFrame(x = c(NA, FALSE, TRUE), y = c(TRUE, TRUE, TRUE))
df$with_columns(
  ne = pl$col("x")$ne("y"),
  ne_missing = pl$col("x")$ne_missing("y")
)
```

expr__not *Negate a boolean expression*

Description

Negate a boolean expression

Usage

```
expr__not()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(TRUE, FALSE, FALSE, NA))

df$with_columns(a_not = pl$col("a")$not())

# Same result with "!"
df$with_columns(a_not = !pl$col("a"))
```

---

expr__null_count *Count null values*

Description

Count null values

Usage

```
expr__null_count()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(NA, 1, NA),
  b = c(10, NA, 300),
  c = c(1, 2, 2)
)
df$select(pl$all()$null_count())
```

---

expr__n_unique *Count unique values*

Description

`null` is considered to be a unique value for the purposes of this operation.

Usage

```
expr__n_unique()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  x = c(1, 1, 2, 2, 3),
  y = c(1, 1, 1, NA, NA)
)
df$select(
  x_unique = pl$col("x")$n_unique(),
  y_unique = pl$col("y")$n_unique()
)
```

---

expr__or *Apply logical OR on two expressions*

Description

Combine two boolean expressions with OR.

Usage

```
expr__or(other)
```

Arguments

other	Element to add. Can be a string (only if expr is a string), a numeric value or an other expression.

Value

A polars expression

Examples

```
pl$lit(TRUE) | FALSE
pl$lit(TRUE)$or(pl$lit(TRUE))
```

---

expr__over *Compute expressions over the given groups*

Description

This expression is similar to performing a group by aggregation and joining the result back into the original DataFrame. The outcome is similar to how window functions work in PostgreSQL.

Usage

```
expr__over(
  ...,
  order_by = NULL,
  mapping_strategy = c("group_to_rows", "join", "explode")
)
```

Arguments

`...`	dynamic-dots> Column(s) to group by. Accepts expression input. Characters are parsed as column names.
`order_by`	Order the window functions/aggregations with the partitioned groups by the result of the expression passed to `order_by`. Accepts expression input. Strings are parsed as column names.
`mapping_strategy`	One of the following: • `"group_to_rows"` (default): if the aggregation results in multiple values, assign them back to their position in the DataFrame. This can only be done if the group yields the same elements before aggregation as after. • `"join"`: join the groups as `List<group_dtype>` to the row positions. Note that this can be memory intensive. • `"explode"`: don't do any mapping, but simply flatten the group. This only makes sense if the input data is sorted.

Value

A polars expression

Examples

```
# Pass the name of a column to compute the expression over that column.
df <- pl$DataFrame(
  a = c("a", "a", "b", "b", "b"),
  b = c(1, 2, 3, 5, 3),
  c = c(5, 4, 2, 1, 3)
)

df$with_columns(
  pl$col("c")$max()$over("a")$name$suffix("_max")
)

# Expression input is supported.
df$with_columns(
  pl$col("c")$max()$over(pl$col("b") %/% 2)$name$suffix("_max")
)

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

```
df$with_columns(
  pl$col("c")$min()$over("a", "b")$name$suffix("_min")
)

group_vars <- list(pl$col("a"), pl$col("b"))
df$with_columns(
  pl$col("c")$min()$over(!!!group_vars)$name$suffix("_min")
)

# Or use positional arguments to group by multiple columns in the same way.
df$with_columns(
  pl$col("c")$min()$over("a", pl$col("b") %% 2)$name$suffix("_min")
)

# Alternative mapping strategy: join values in a list output
df$with_columns(
  top_2 = pl$col("c")$top_k(2)$over("a", mapping_strategy = "join")
)

# order_by specifies how values are sorted within a group, which is
# essential when the operation depends on the order of values
df <- pl$DataFrame(
  g = c(1, 1, 1, 1, 2, 2, 2, 2),
  t = c(1, 2, 3, 4, 4, 1, 2, 3),
  x = c(10, 20, 30, 40, 10, 20, 30, 40)
)

# without order_by, the first and second values in the second group would
# be inverted, which would be wrong
df$with_columns(
  x_lag = pl$col("x")$shift(1)$over("g", order_by = "t")
)
```

---

expr__pct_change *Computes percentage change between values*

Description

Computes the percentage change (as fraction) between current element and most-recent non-null element at least `n` period(s) before the current element. By default it computes the change from the previous row.

Usage

```
expr__pct_change(n = 1)
```

Arguments

`n`	Integer or Expr indicating the number of periods to shift for forming percent change.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(10:12, NA, 12))
df$with_columns(
  pct_change = pl$col("a")$pct_change()
)
```

---

expr__peak_max *Get a boolean mask of the local maximum peaks*

Description

Get a boolean mask of the local maximum peaks

Usage

```
expr__peak_max()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = c(1, 2, 3, 2, 3, 4, 5, 2))
df$with_columns(peak_max = pl$col("x")$peak_max())
```

---

expr__peak_min *Get a boolean mask of the local minimum peaks*

Description

Get a boolean mask of the local minimum peaks

Usage

```
expr__peak_min()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = c(1, 2, 3, 2, 3, 4, 5, 2))
df$with_columns(peak_min = pl$col("x")$peak_min())
```

expr__pow — *Exponentiation using two expressions*

Description

Method equivalent of exponentiation operator `expr ^ exponent`.

Usage

```
expr__pow(other)
```

Arguments

`exponent`	Numeric literal or expression value.

Value

A polars expression

See Also

- Arithmetic operators

Examples

```
df <- pl$DataFrame(x = c(1, 2, 4, 8))

df$with_columns(
  cube = pl$col("x")$pow(3),
  `x^xlog2` = pl$col("x")$pow(pl$col("x")$log(2))
)
```

expr__product — *Compute the product of an expression.*

Description

Compute the product of an expression.

Usage

```
expr__product()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = 1:3, b = c(NA, 4, 4))$
  select(pl$all()$product())
```

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

Description

[**Experimental**]

Usage

```
expr__qcut(
  quantiles,
  ...,
  labels = NULL,
  left_closed = FALSE,
  allow_duplicates = FALSE,
  include_breaks = FALSE
)
```

Arguments

`quantiles`	Either a vector of quantile probabilities between 0 and 1 or a positive integer determining the number of bins with uniform probability.
`...`	These dots are for future extensions and must be empty.
`labels`	Names of the categories. The number of labels must be equal to the number of categories.
`left_closed`	Set the intervals to be left-closed instead of right-closed.
`allow_duplicates`	If `TRUE`, duplicates in the resulting quantiles are dropped, rather than raising an error. This can happen even with unique probabilities, depending on the data.
`include_breaks`	Include a column with the right endpoint of the bin each observation falls in. This will change the data type of the output from a Categorical to a Struct.

Value

A polars expression

Examples

```
# Divide a column into three categories according to pre-defined quantile
# probabilities.
df <- pl$DataFrame(foo = -2:2)
df$with_columns(
  qcut = pl$col("foo")$qcut(c(0.25, 0.75), labels = c("a", "b", "c"))
)

# Divide a column into two categories using uniform quantile probabilities.
df$with_columns(
  qcut = pl$col("foo")$qcut(2, labels = c("low", "high"), left_closed = TRUE)
)

# Add both the category and the breakpoint.
df$with_columns(
  qcut = pl$col("foo")$qcut(c(0.25, 0.75), include_breaks = TRUE)
)$unnest()
```

---

expr__quantile *Get quantile value(s)*

Description

Get quantile value(s)

Usage

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

Arguments

`quantile`	Quantile between 0.0 and 1.0.
`interpolation`	Interpolation method. Must be one of `"nearest"`, `"higher"`, `"lower"`, `"midpoint"`, `"linear"`.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 0:5)
df$select(pl$col("a")$quantile(0.3))
df$select(pl$col("a")$quantile(0.3, interpolation = "higher"))
df$select(pl$col("a")$quantile(0.3, interpolation = "lower"))
df$select(pl$col("a")$quantile(0.3, interpolation = "midpoint"))
df$select(pl$col("a")$quantile(0.3, interpolation = "linear"))
```

expr__radians *Convert from degrees to radians*

Description

Convert from degrees to radians

Usage

```
expr__radians()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(-720, -540, -360, -180, 0, 180, 360, 540, 720))$
  with_columns(radians = pl$col("a")$radians())
```

expr__rank *Assign ranks to data, dealing with ties appropriately*

Description

Assign ranks to data, dealing with ties appropriately

Usage

```
expr__rank(
  method = c("average", "min", "max", "dense", "ordinal", "random"),
  ...,
  descending = FALSE,
  seed = NULL
)
```

Arguments

`method`	The method used to assign ranks to tied elements. Must be one of the following: • `"average"` (default): The average of the ranks that would have been assigned to all the tied values is assigned to each value. • `"min"`: The minimum of the ranks that would have been assigned to all the tied values is assigned to each value. (This is also referred to as "competition" ranking.)

- `"max"` : The maximum of the ranks that would have been assigned to all the tied values is assigned to each value.
- `"dense"`: Like 'min', but the rank of the next highest element is assigned the rank immediately after those assigned to the tied elements.
- `"ordinal"` : All values are given a distinct rank, corresponding to the order that the values occur in the Series.
- `"random"` : Like 'ordinal', but the rank for ties is not dependent on the order that the values occur in the Series.

`...`	These dots are for future extensions and must be empty.
`descending`	Rank in descending order.
`seed`	Integer. Only used if `method = "random"`.

Value

A polars expression

Examples

```
# Default is to use the "average" method to break ties
df <- pl$DataFrame(a = c(3, 6, 1, 1, 6))
df$with_columns(rank = pl$col("a")$rank())

# Ordinal method
df$with_columns(rank = pl$col("a")$rank("ordinal"))

# Use "rank" with "over" to rank within groups:
df <- pl$DataFrame(
  a = c(1, 1, 2, 2, 2),
  b = c(6, 7, 5, 14, 11)
)
df$with_columns(
  rank = pl$col("b")$rank()$over("a")
)
```

---

expr__rechunk *Create a single chunk of memory for this Series*

Description

Create a single chunk of memory for this Series

Usage

```
expr__rechunk()
```

Value

A polars expression

Examples

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

# Create a Series with 3 nulls, append column a then rechunk
df$select(pl$repeat(NA, 3)$append(pl$col("a"))$rechunk())
```

`expr__reinterpret` *Reinterpret the underlying bits as a signed/unsigned integer*

Description

This operation is only allowed for 64-bit integers. For lower bits integers, you can safely use the $cast() operation.

Usage

```
expr__reinterpret(..., signed = TRUE)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`signed`	If `TRUE` (default), reinterpret as pl$Int64. Otherwise, reinterpret as pl$UInt64.

Value

A polars expression

Examples

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

# Create a Series with 3 nulls, append column a then rechunk
df$with_columns(
  reinterpreted = pl$col("a")$reinterpret()
)
```

`expr__repeat_by` *Repeat the elements in this Series as specified in the given expression*

Description

The repeated elements are expanded into a List dtype.

Usage

```
expr__repeat_by(by)
```

Arguments

by	Numeric column that determines how often the values will be repeated. The column will be coerced to UInt32. Give this dtype to make the coercion a no-op. Accepts expression input, strings are parsed as column names.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c("x", "y", "z"), n = 1:3)

df$with_columns(
  repeated = pl$col("a")$repeat_by("n")
)
```

---

expr__replace *Replace the given values by different values of the same data type.*

Description

This allows one to recode values in a column, leaving all other values unchanged. See `$replace_strict()` to give a default value to all other values and to specify the output datatype.

Usage

```
expr__replace(old, new)
```

Arguments

old	Value or vector of values to replace. Accepts expression input. Vectors are parsed as Series, other non-expression inputs are parsed as literals. Also accepts a list of values like `list(old = new)`.
new	Value or vector of values to replace by. Accepts expression input. Vectors are parsed as Series, other non-expression inputs are parsed as literals. Length must match the length of `old` or have length 1.

Details

The global string cache must be enabled when replacing categorical values.

Value

A polars expression

Examples

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

# "old" and "new" can take vectors of length 1 or of same length
df$with_columns(replaced = pl$col("a")$replace(2, 100))
df$with_columns(replaced = pl$col("a")$replace(c(2, 3), c(100, 200)))

# "old" can be a named list where names are values to replace, and values are
# the replacements
mapping <- list(`2` = 100, `3` = 200)
df$with_columns(replaced = pl$col("a")$replace(mapping))

# The original data type is preserved when replacing by values of a
# different data type. Use $replace_strict() to replace and change the
# return data type.
df <- pl$DataFrame(a = c("x", "y", "z"))
mapping <- list(x = 1, y = 2, z = 3)
df$with_columns(replaced = pl$col("a")$replace(mapping))

# "old" and "new" can take Expr
df <- pl$DataFrame(a = c(1, 2, 2, 3), b = c(1.5, 2.5, 5, 1))
df$with_columns(
  replaced = pl$col("a")$replace(
    old = pl$col("a")$max(),
    new = pl$col("b")$sum()
  )
)
```

---

expr__replace_strict *Replace all values by different values*

Description

This changes all the values in a column, either using a specific replacement or a default one. See `$replace()` to replace only a subset of values.

Usage

```
expr__replace_strict(old, new, ..., default = NULL, return_dtype = NULL)
```

Arguments

`old`	Value or vector of values to replace. Accepts expression input. Vectors are parsed as Series, other non-expression inputs are parsed as literals. Also accepts a list of values like `list(old = new)`.
`new`	Value or vector of values to replace by. Accepts expression input. Vectors are parsed as Series, other non-expression inputs are parsed as literals. Length must match the length of `old` or have length 1.

`...`	These dots are for future extensions and must be empty.
`default`	Set values that were not replaced to this value. If `NULL` (default), an error is raised if any values were not replaced. Accepts expression input. Non-expression inputs are parsed as literals.
`return_dtype`	The data type of the resulting expression. If `NULL` (default), the data type is determined automatically based on the other inputs.

Details

The global string cache must be enabled when replacing categorical values.

Value

A polars expression

Examples

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

# "old" and "new" can take vectors of length 1 or of same length
df$with_columns(replaced = pl$col("a")$replace_strict(2, 100, default = 1))
df$with_columns(
  replaced = pl$col("a")$replace_strict(c(2, 3), c(100, 200), default = 1)
)

# "old" can be a named list where names are values to replace, and values are
# the replacements
mapping <- list(`2` = 100, `3` = 200)
df$with_columns(replaced = pl$col("a")$replace_strict(mapping, default = -1))

# By default, an error is raised if any non-null values were not replaced.
# Specify a default to set all values that were not matched.
tryCatch(
  df$with_columns(replaced = pl$col("a")$replace_strict(mapping)),
  error = function(e) print(e)
)

# one can specify the data type to return instead of automatically
# inferring it
df$with_columns(
  replaced = pl$col("a")$replace_strict(
    mapping, default = 1, return_dtype = pl$Int32
  )
)

# "old", "new", and "default" can take Expr
df <- pl$DataFrame(a = c(1, 2, 2, 3), b = c(1.5, 2.5, 5, 1))
df$with_columns(
  replaced = pl$col("a")$replace_strict(
    old = pl$col("a")$max(),
    new = pl$col("b")$sum(),
    default = pl$col("b"),
```

```
  )
)
```

`expr__reshape` *Reshape this Expr to a flat Series or a Series of Lists*

Description

Reshape this Expr to a flat Series or a Series of Lists

Usage

```
expr__reshape(dimensions)
```

Arguments

`dimensions`	A integer vector of length of the dimension size. If `-1` is used in any of the dimensions, that dimension is inferred. Currently, more than two dimensions not supported.
`nested_type`	The nested data type to create. List only supports 2 dimensions, whereas Array supports an arbitrary number of dimensions.

Details

If a single dimension is given, results in an expression of the original data type. If a multiple dimensions are given, results in an expression of data type List with shape equal to the dimensions.

Value

A polars expression

Examples

```
df <- pl$DataFrame(foo = 1:9)

df$select(pl$col("foo")$reshape(9))
df$select(pl$col("foo")$reshape(c(3, 3)))

# Use `-1` to infer the other dimension
df$select(pl$col("foo")$reshape(c(-1, 3)))
df$select(pl$col("foo")$reshape(c(3, -1)))

# One can specify more than 2 dimensions by using the Array type
df <- pl$DataFrame(foo = 1:12)
df$select(
  pl$col("foo")$reshape(c(3, 2, 2), nested_type = pl$Array(pl$Float32, 2))
)
```

expr__reverse *Reverse an expression*

Description

Reverse an expression

Usage

```
expr__reverse()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = 1:5,
  fruits = c("banana", "banana", "apple", "apple", "banana"),
  b = 5:1
)

df$with_columns(
  pl$all()$reverse()$name$suffix("_reverse")
)
```

expr__rle *Compress the column data using run-length encoding*

Description

Run-length encoding (RLE) encodes data by storing each run of identical values as a single value and its length.

Usage

```
expr__rle()
```

Value

A polars expression

Examples

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

df$select(pl$col("a")$rle())$unnest("a")
```

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

Description

The ID starts at 0 and increases by one each time the value of the column changes.

Usage

```
expr__rle_id()
```

Details

This functionality is especially useful for defining a new group for every time a column's value changes, rather than for every distinct value of that column.

Value

A polars expression

Examples

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

df$with_columns(
  rle_id_a = pl$col("a")$rle_id(),
  rle_id_ab = pl$struct("a", "b")$rle_id()
)
```

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

Description

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

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

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

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

Usage

```
expr__rolling(index_column, ..., period, offset = NULL, closed = "right")
```

Arguments

`index_column`	Character. Name of the column used to group based on the time window. Often of type Date/Datetime. This column must be sorted in ascending order. In case of a rolling group by on indices, dtype needs to be one of UInt32, UInt64, Int32, Int64. Note that the first three get cast to Int64, so if performance matters use an Int64 column.
`...`	These dots are for future extensions and must be empty.
`period`	Length of the window - must be non-negative.
`offset`	Offset of the window. Default is `-period`.
`closed`	Define which sides of the range are closed (inclusive). One of the following: `"both"` (default), `"left"`, `"right"`, `"none"`.

Details

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

Value

A polars expression

Examples

```
dates <- as.POSIXct(
  c(
    "2020-01-01 13:45:48", "2020-01-01 16:42:13", "2020-01-01 16:45:09",
    "2020-01-02 18:12:48", "2020-01-03 19:45:32","2020-01-08 23:16:43"
  )
)
df <- pl$DataFrame(dt = dates, a = c(3, 7, 5, 9, 2, 1))

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

`expr__rolling_max` *Apply a rolling max over values*

Description

[Experimental]

A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the `weights` vector. The resulting values will be aggregated.

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

Usage

```
expr__rolling_max(
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`center`	If `TRUE`, set the labels at the center of the window.

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_max = pl$col("a")$rolling_max(window_size = 2)
)
```

```
# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_max = pl$col("a")$rolling_max(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

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

`expr__rolling_max_by` *Apply a rolling max based on another column*

Description

[Experimental]

Given a `by` column `<t_0, t_1, ..., t_n>`, then `closed = "right"` (the default) means the windows will be:

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

Usage

```
expr__rolling_max_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = c("right", "both", "left", "none")
)
```

Arguments

`by`	Should be DateTime, Date, UInt64, UInt32, Int64, or Int32 data type after conversion by `as_polars_expr()`. Note that the integer ones require using `"i"` in `window_size`. Accepts expression input. Strings are parsed as column names.
`window_size`	The length of the window. Can be a dynamic temporal size indicated by a timedelta or the following string language: • 1ns (1 nanosecond) • 1us (1 microsecond) • 1ms (1 millisecond)

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

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

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

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

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

Details

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

Value

A polars expression

Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# Compute the rolling max with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_max = pl$col("index")$rolling_max_by(
    "date",
    window_size = "2h"
  )
)

# Compute the rolling max with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_max = pl$col("index")$rolling_max_by(
```

```
    "date",
    window_size = "2h",
    closed = "both"
  )
)
```

`expr__rolling_mean` *Apply a rolling mean over values*

Description

[Experimental]

A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the `weights` vector. The resulting values will be aggregated.

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

Usage

```
expr__rolling_mean(
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`center`	If `TRUE`, set the labels at the center of the window.

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_mean = pl$col("a")$rolling_mean(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_mean = pl$col("a")$rolling_mean(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

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

`expr__rolling_mean_by`

Apply a rolling mean based on another column

Description

[**Experimental**]

Given a `by` column `<t_0, t_1, ..., t_n>`, then `closed = "right"` (the default) means the windows will be:

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

Usage

```
expr__rolling_mean_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = c("right", "both", "left", "none")
)
```

Arguments

`by`	Should be DateTime, Date, UInt64, UInt32, Int64, or Int32 data type after conversion by `as_polars_expr()`. Note that the integer ones require using `"i"` in `window_size`. Accepts expression input. Strings are parsed as column names.

`window_size` The length of the window. Can be a dynamic temporal size indicated by a timedelta or the following string language:

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

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

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

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

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

Details

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

Value

A polars expression

Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# Compute the rolling mean with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_mean = pl$col("index")$rolling_mean_by(
    "date",
    window_size = "2h"
```

```
  )
)

# Compute the rolling mean with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_mean = pl$col("index")$rolling_mean_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)
```

`expr__rolling_median` *Apply a rolling median over values*

Description

[Experimental]

A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the `weights` vector. The resulting values will be aggregated.

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

Usage

```
expr__rolling_median(
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`center`	If `TRUE`, set the labels at the center of the window.

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_median = pl$col("a")$rolling_median(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_median = pl$col("a")$rolling_median(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

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

---

expr__rolling_median_by

Apply a rolling median based on another column

Description

[**Experimental**]

Given a `by` column `<t_0, t_1, ..., t_n>`, then `closed = "right"` (the default) means the windows will be:

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

Usage

```
expr__rolling_median_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = c("right", "both", "left", "none")
)
```

Arguments

`by`	Should be DateTime, Date, UInt64, UInt32, Int64, or Int32 data type after conversion by `as_polars_expr()`. Note that the integer ones require using `"i"` in `window_size`. Accepts expression input. Strings are parsed as column names.
`window_size`	The length of the window. Can be a dynamic temporal size indicated by a timedelta or the following string language: • 1ns (1 nanosecond) • 1us (1 microsecond) • 1ms (1 millisecond) • 1s (1 second) • 1m (1 minute) • 1h (1 hour) • 1d (1 calendar day) • 1w (1 calendar week) • 1mo (1 calendar month) • 1q (1 calendar quarter) • 1y (1 calendar year) Or combine them: `"3d12h4m25s"` # 3 days, 12 hours, 4 minutes, and 25 seconds By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`closed`	Define which sides of the interval are closed (inclusive). Default is `"right"`.

Details

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

Value

A polars expression

Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)
```

```
# Compute the rolling median with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_median = pl$col("index")$rolling_median_by(
    "date",
    window_size = "2h"
  )
)

# Compute the rolling median with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_median = pl$col("index")$rolling_median_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)
```

`expr__rolling_min` *Apply a rolling min over values*

Description

[Experimental]

A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the `weights` vector. The resulting values will be aggregated.

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

Usage

```
expr__rolling_min(
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`center`	If `TRUE`, set the labels at the center of the window.

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_min = pl$col("a")$rolling_min(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_min = pl$col("a")$rolling_min(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

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

---

expr__rolling_min_by *Apply a rolling min based on another column*

Description

[**Experimental**]

Given a `by` column `<t_0, t_1, ..., t_n>`, then `closed = "right"` (the default) means the windows will be:

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

Usage

```
expr__rolling_min_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = c("right", "both", "left", "none")
)
```

Arguments

`by`	Should be DateTime, Date, UInt64, UInt32, Int64, or Int32 data type after conversion by `as_polars_expr()`. Note that the integer ones require using `"i"` in `window_size`. Accepts expression input. Strings are parsed as column names.
`window_size`	The length of the window. Can be a dynamic temporal size indicated by a timedelta or the following string language: • 1ns (1 nanosecond) • 1us (1 microsecond) • 1ms (1 millisecond) • 1s (1 second) • 1m (1 minute) • 1h (1 hour) • 1d (1 calendar day) • 1w (1 calendar week) • 1mo (1 calendar month) • 1q (1 calendar quarter) • 1y (1 calendar year) Or combine them: `"3d12h4m25s"` # 3 days, 12 hours, 4 minutes, and 25 seconds By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`closed`	Define which sides of the interval are closed (inclusive). Default is `"right"`.

Details

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

Value

A polars expression

Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)
```

```
# Compute the rolling min with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_min = pl$col("index")$rolling_min_by(
    "date",
    window_size = "2h"
  )
)

# Compute the rolling min with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_min = pl$col("index")$rolling_min_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)
```

`expr__rolling_quantile` *Apply a rolling quantile over values*

Description

[Experimental]

A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the `weights` vector. The resulting values will be aggregated.

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

Usage

```
expr__rolling_quantile(
  quantile,
  interpolation = c("nearest", "higher", "lower", "midpoint", "linear"),
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE
)
```

Arguments

`quantile`	Quantile between 0.0 and 1.0.

`interpolation`	Interpolation method. Must be one of `"nearest"`, `"higher"`, `"lower"`, `"midpoint"`, `"linear"`.
`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`center`	If `TRUE`, set the labels at the center of the window.

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_quantile = pl$col("a")$rolling_quantile(
    quantile = 0.25, window_size = 4
  )
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_quantile = pl$col("a")$rolling_quantile(
    quantile = 0.25, window_size = 4, weights = c(0.2, 0.4, 0.4, 0.2)
  )
)

# Specify weights and interpolation method:
df$with_columns(
  rolling_quantile = pl$col("a")$rolling_quantile(
    quantile = 0.25, window_size = 4, weights = c(0.2, 0.4, 0.4, 0.2),
    interpolation = "linear"
  )
)

# Center the values in the window
df$with_columns(
  rolling_quantile = pl$col("a")$rolling_quantile(
    quantile = 0.25, window_size = 5, center = TRUE
  )
)
```

`expr__rolling_quantile_by` *Apply a rolling quantile based on another column*

Description

[Experimental]

Given a `by` column `<t_0, t_1, ..., t_n>`, then `closed = "right"` (the default) means the windows will be:

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

Usage

```
expr__rolling_quantile_by(
  by,
  window_size,
  ...,
  quantile,
  interpolation = c("nearest", "higher", "lower", "midpoint", "linear"),
  min_periods = 1,
  closed = c("right", "both", "left", "none")
)
```

Arguments

`by`	Should be DateTime, Date, UInt64, UInt32, Int64, or Int32 data type after conversion by `as_polars_expr()`. Note that the integer ones require using `"i"` in `window_size`. Accepts expression input. Strings are parsed as column names.
`window_size`	The length of the window. Can be a dynamic temporal size indicated by a timedelta or the following string language: • 1ns (1 nanosecond) • 1us (1 microsecond) • 1ms (1 millisecond) • 1s (1 second) • 1m (1 minute) • 1h (1 hour) • 1d (1 calendar day) • 1w (1 calendar week) • 1mo (1 calendar month) • 1q (1 calendar quarter)

- 1y (1 calendar year)

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

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

`quantile`	Quantile between 0.0 and 1.0.
`interpolation`	Interpolation method. Must be one of `"nearest"`, `"higher"`, `"lower"`, `"midpoint"`, `"linear"`.
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`closed`	Define which sides of the interval are closed (inclusive). Default is `"right"`.

Details

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

Value

A polars expression

Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# Compute the rolling quantile with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_quantile = pl$col("index")$rolling_quantile_by(
    "date",
    window_size = "2h"
  )
)

# Compute the rolling quantile with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_quantile = pl$col("index")$rolling_quantile_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)
```

`expr__rolling_skew` *Apply a rolling skew over values*

Description

[Experimental]

A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the `weights` vector. The resulting values will be aggregated.

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

Usage

```
expr__rolling_skew(window_size, ..., bias = TRUE)
```

Arguments

`window_size`	The length of the window in number of elements.
`...`	These dots are for future extensions and must be empty.
`bias`	If `FALSE`, the calculations are corrected for statistical bias.

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, 4, 2, 9))
df$with_columns(
  rolling_skew = pl$col("a")$rolling_skew(3)
)
```

`expr__rolling_std` *Apply a rolling standard deviation over values*

Description

[Experimental]

A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the `weights` vector. The resulting values will be aggregated.

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

Usage

```
expr__rolling_std(
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE,
  ddof = 1
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`center`	If `TRUE`, set the labels at the center of the window.

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_std = pl$col("a")$rolling_std(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_std = pl$col("a")$rolling_std(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

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

---

expr__rolling_std_by *Apply a rolling standard deviation based on another column*

Description

[**Experimental**]

Given a `by` column `<t_0, t_1, ..., t_n>`, then `closed = "right"` (the default) means the windows will be:

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

Usage

```
expr__rolling_std_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = c("right", "both", "left", "none"),
  ddof = 1
)
```

Arguments

`by`	Should be DateTime, Date, UInt64, UInt32, Int64, or Int32 data type after conversion by `as_polars_expr()`. Note that the integer ones require using `"i"` in `window_size`. Accepts expression input. Strings are parsed as column names.

`window_size` The length of the window. Can be a dynamic temporal size indicated by a timedelta or the following string language:

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

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

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

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

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

Details

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

Value

A polars expression

Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# Compute the rolling std with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_std = pl$col("index")$rolling_std_by(
    "date",
    window_size = "2h"
```

```
  )
)

# Compute the rolling std with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_std = pl$col("index")$rolling_std_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)
```

`expr__rolling_sum` *Apply a rolling sum over values*

Description

[Experimental]

A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the `weights` vector. The resulting values will be aggregated.

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

Usage

```
expr__rolling_sum(
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`center`	If `TRUE`, set the labels at the center of the window.

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_sum = pl$col("a")$rolling_sum(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_sum = pl$col("a")$rolling_sum(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

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

---

expr__rolling_sum_by *Apply a rolling sum based on another column*

Description

[Experimental]

Given a `by` column `<t_0, t_1, ..., t_n>`, then `closed = "right"` (the default) means the windows will be:

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

Usage

```
expr__rolling_sum_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = c("right", "both", "left", "none")
)
```

Arguments

`by`	Should be DateTime, Date, UInt64, UInt32, Int64, or Int32 data type after conversion by `as_polars_expr()`. Note that the integer ones require using `"i"` in `window_size`. Accepts expression input. Strings are parsed as column names.
`window_size`	The length of the window. Can be a dynamic temporal size indicated by a timedelta or the following string language: • 1ns (1 nanosecond) • 1us (1 microsecond) • 1ms (1 millisecond) • 1s (1 second) • 1m (1 minute) • 1h (1 hour) • 1d (1 calendar day) • 1w (1 calendar week) • 1mo (1 calendar month) • 1q (1 calendar quarter) • 1y (1 calendar year) Or combine them: `"3d12h4m25s"` # 3 days, 12 hours, 4 minutes, and 25 seconds By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".
`min_periods`	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
`closed`	Define which sides of the interval are closed (inclusive). Default is `"right"`.

Details

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

Value

A polars expression

Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)
```

```
# Compute the rolling sum with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_sum = pl$col("index")$rolling_sum_by(
    "date",
    window_size = "2h"
  )
)

# Compute the rolling sum with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_sum = pl$col("index")$rolling_sum_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)
```

`expr__rolling_var` *Apply a rolling variance over values*

Description

[Experimental]

A window of length `window_size` will traverse the array. The values that fill this window will (optionally) be multiplied with the weights given by the `weights` vector. The resulting values will be aggregated.

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

Usage

```
expr__rolling_var(
  window_size,
  weights = NULL,
  ...,
  min_periods = NULL,
  center = FALSE,
  ddof = 1
)
```

Arguments

`window_size`	The length of the window in number of elements.
`weights`	An optional slice with the same length as the window that will be multiplied elementwise with the values in the window.

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

Details

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:6)
df$with_columns(
  rolling_var = pl$col("a")$rolling_var(window_size = 2)
)

# Specify weights to multiply the values in the window with:
df$with_columns(
  rolling_var = pl$col("a")$rolling_var(
    window_size = 2, weights = c(0.25, 0.75)
  )
)

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

---

expr__rolling_var_by *Apply a rolling variance based on another column*

Description

[**Experimental**]

Given a `by` column `<t_0, t_1, ..., t_n>`, then `closed = "right"` (the default) means the windows will be:

- `(t_0 - window_size, t_0]`
- `(t_1 - window_size, t_1]`
- ...
- `(t_n - window_size, t_n]`

Usage

```
expr__rolling_var_by(
  by,
  window_size,
  ...,
  min_periods = 1,
  closed = c("right", "both", "left", "none"),
  ddof = 1
)
```

Arguments

by	Should be DateTime, Date, UInt64, UInt32, Int64, or Int32 data type after conversion by `as_polars_expr()`. Note that the integer ones require using `"i"` in `window_size`. Accepts expression input. Strings are parsed as column names.
window_size	The length of the window. Can be a dynamic temporal size indicated by a timedelta or the following string language: • 1ns (1 nanosecond) • 1us (1 microsecond) • 1ms (1 millisecond) • 1s (1 second) • 1m (1 minute) • 1h (1 hour) • 1d (1 calendar day) • 1w (1 calendar week) • 1mo (1 calendar month) • 1q (1 calendar quarter) • 1y (1 calendar year) Or combine them: `"3d12h4m25s"` # 3 days, 12 hours, 4 minutes, and 25 seconds By "calendar day", we mean the corresponding time on the next day (which may not be 24 hours, due to daylight savings). Similarly for "calendar week", "calendar month", "calendar quarter", and "calendar year".
min_periods	The number of values in the window that should be non-null before computing a result. If `NULL` (default), it will be set equal to `window_size`.
closed	Define which sides of the interval are closed (inclusive). Default is `"right"`.

Details

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

Value

A polars expression

Examples

```
df_temporal <- pl$select(
  index = 0:24,
  date = pl$datetime_range(
    as.POSIXct("2001-01-01"),
    as.POSIXct("2001-01-02"),
    "1h"
  )
)

# Compute the rolling var with the temporal windows closed on the right
# (default)
df_temporal$with_columns(
  rolling_row_var = pl$col("index")$rolling_var_by(
    "date",
    window_size = "2h"
  )
)

# Compute the rolling var with the closure of windows on both sides
df_temporal$with_columns(
  rolling_row_var = pl$col("index")$rolling_var_by(
    "date",
    window_size = "2h",
    closed = "both"
  )
)
```

---

expr__round *Round underlying floating point data by decimals digits*

Description

Round underlying floating point data by decimals digits

Usage

```
expr__round(decimals)
```

Arguments

decimals	Number of decimals to round by.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(0.33, 0.52, 1.02, 1.17))

df$with_columns(
  rounded = pl$col("a")$round(1)
)
```

---

expr__round_sig_figs *Round to a number of significant figures*

Description

Round to a number of significant figures

Usage

```
expr__round_sig_figs(digits)
```

Arguments

digits	Number of significant figures to round to.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(0.01234, 3.333, 1234))

df$with_columns(
  rounded = pl$col("a")$round_sig_figs(2)
)
```

---

expr__sample *Sample from this expression*

Description

Sample from this expression

Usage

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

Arguments

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a")$sample(
  fraction = 1, with_replacement = TRUE, seed = 1
))
```

`expr__search_sorted` *Find indices where elements should be inserted to maintain order*

Description

This returns -1 if x is lower than 0, 0 if x == 0, and 1 if x is greater than 0.

Usage

```
expr__search_sorted(element, side = c("any", "left", "right"))
```

Arguments

`element`	Expression or scalar value.
`side`	Must be one of the following: • `"any"`: the index of the first suitable location found is given; • `"left"`: the index of the leftmost suitable location found is given; • `"right"`: the index the rightmost suitable location found is given.

Value

A polars expression

Examples

```
df <- pl$DataFrame(values = c(1, 2, 3, 5))
df$select(
  zero = pl$col("values")$search_sorted(0),
  three = pl$col("values")$search_sorted(3),
  six = pl$col("values")$search_sorted(6),
)
```

`expr__set_sorted` *Flags the expression as "sorted"*

Description

Enables downstream code to user fast paths for sorted arrays.

Warning: This can lead to incorrect results if the data is NOT sorted!! Use with care!

Usage

```
expr__set_sorted(..., descending = FALSE)
```

Arguments

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a")$set_sorted()$max())
```

expr__shift *Shift values by the given number of indices*

Description

Shift values by the given number of indices

Usage

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

Arguments

n	Number of indices to shift forward. If a negative value is passed, values are shifted in the opposite direction instead.
...	These dots are for future extensions and must be empty.
fill_value	Fill the resulting null values with this value.

Value

A polars expression

Examples

```
# By default, values are shifted forward by one index.
df <- pl$DataFrame(a = 1:4)
df$with_columns(shift = pl$col("a")$shift())

# Pass a negative value to shift in the opposite direction instead.
df$with_columns(shift = pl$col("a")$shift(-2))

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

expr__shrink_dtype *Shrink numeric columns to the minimal required datatype*

Description

Shrink to the dtype needed to fit the extrema of this Series. This can be used to reduce memory pressure.

Usage

```
expr__shrink_dtype()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(-112, 2, 112))$cast(pl$Int64)
df$with_columns(
  shrunk = pl$col("a")$shrink_dtype()
)
```

---

expr__shuffle *Shuffle the contents of this expression*

Description

Note this is shuffled independently of any other column or Expression. If you want each row to stay the same use `df$sample(shuffle = TRUE)`.

Usage

```
expr__shuffle(seed = NULL)
```

Arguments

`seed`	Integer indicating the seed for the random number generator. If `NULL` (default), a random seed is generated each time the shuffle is called.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3)
df$with_columns(
  shuffled = pl$col("a")$shuffle(seed = 1)
)
```

expr__sign *Compute the sign*

Description

This returns -1 if x is lower than 0, 0 if x == 0, and 1 if x is greater than 0.

Usage

```
expr__sign()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(-9, 0, 0, 4, NA))
df$with_columns(sign = pl$col("a")$sign())
```

expr__sin *Compute sine*

Description

Compute sine

Usage

```
expr__sin()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(0, pi / 2, pi, NA))$
  with_columns(sine = pl$col("a")$sin())
```

expr__sinh *Compute hyperbolic sine*

Description

Compute hyperbolic sine

Usage

```
expr__sinh()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(-1, asinh(0.5), 0, 1, NA))$
  with_columns(sinh = pl$col("a")$sinh())
```

expr__skew *Compute the skewness*

Description

For normally distributed data, the skewness should be about zero. For unimodal continuous distributions, a skewness value greater than zero means that there is more weight in the right tail of the distribution.

Usage

```
expr__skew(..., bias = TRUE)
```

Arguments

...	These dots are for future extensions and must be empty.
bias	If FALSE, the calculations are corrected for statistical bias.

Details

The sample skewness is computed as the Fisher-Pearson coefficient of skewness, i.e.

$$g_1 = \frac{m_3}{m_2^{3/2}}$$

where

$$m_i = \frac{1}{N} \sum_{n=1}^{N} (x[n] - \bar{x})^i$$

is the biased sample i`th` central moment, and $\bar{x}$ is the sample mean. If `bias = FALSE`, the calculations are corrected for bias and the value computed is the adjusted Fisher-Pearson standardized moment coefficient, i.e.

$$G_1 = \frac{k_3}{k_2^{3/2}} = \frac{\sqrt{N(N-1)}}{N-2} \frac{m_3}{m_2^{3/2}}$$

Value

A polars expression

Examples

```
df <- pl$DataFrame(x = c(1, 2, 3, 2, 1))
df$select(pl$col("x")$skew())
```

---

expr__slice *Get a slice of this expression*

Description

Get a slice of this expression

Usage

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

Arguments

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

Value

A polars expression

Examples

```
# as head
pl$DataFrame(a = 0:100)$select(
  pl$all()$slice(0, 6)
)

# as tail
pl$DataFrame(a = 0:100)$select(
  pl$all()$slice(-6, 6)
)

pl$DataFrame(a = 0:100)$select(
  pl$all()$slice(80)
)
```

---

expr__sort *Sort this expression*

Description

If used in a groupby context, values within each group are sorted.

Usage

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

Arguments

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(6, 1, 0, NA, Inf, NaN))

df$with_columns(
  sorted = pl$col("a")$sort(),
  sorted_desc = pl$col("a")$sort(descending = TRUE),
  sorted_nulls_last = pl$col("a")$sort(nulls_last = TRUE)
)

# When sorting in a group by context, values in each group are sorted.
df <- pl$DataFrame(
  group = c("one", "one", "one", "two", "two", "two"),
```

```
  value = c(1, 98, 2, 3, 99, 4)
)

df$group_by("group")$agg(pl$col("value")$sort())
```

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

Description

If used in a groupby context, values within each group are sorted.

Usage

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

Arguments

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

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  group = c("a", "a", "b", "b"),
  value1 = c(1, 3, 4, 2),
  value2 = c(8, 7, 6, 5)
)

# by one column/expression
```

```
df$with_columns(
  sorted = pl$col("group")$sort_by("value1")
)

# by two columns/expressions
df$with_columns(
  sorted = pl$col("group")$sort_by(
    "value2", pl$col("value1"),
    descending = c(TRUE, FALSE)
  )
)

# by some expression
df$with_columns(
  sorted = pl$col("group")$sort_by(pl$col("value1") + pl$col("value2"))
)

# in an aggregation context, values are sorted within groups
df$group_by("group")$agg(
  pl$col("value1")$sort_by("value2")
)
```

---

expr__sqrt *Compute square root*

Description

Compute square root

Usage

```
expr__sqrt()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(1, 2, 4))$
  with_columns(sqrt = pl$col("a")$sqrt())
```

`expr__std` *Compute the standard deviation*

Description

Compute the standard deviation

Usage

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

Value

A polars expression

Examples

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

`expr__sub` *Substract two expressions*

Description

Method equivalent of subtraction operator `expr - other`.

Usage

```
expr__sub(other)
```

Arguments

`other`	Numeric literal or expression value.

Value

A polars expression

See Also

- Arithmetic operators

Examples

```
df <- pl$DataFrame(x = 0:4)

df$with_columns(
  `x-2` = pl$col("x")$sub(2),
  `x-expr` = pl$col("x")$sub(pl$col("x")$cum_sum())
)
```

---

expr__sum *Get sum value*

Description

Get sum value

Usage

```
expr__sum()
```

Details

The dtypes Int8, UInt8, Int16 and UInt16 are cast to Int64 before summing to prevent overflow issues.

Value

A polars expression

Examples

```
pl$DataFrame(x = c(1L, NA, 2L))$
  with_columns(sum = pl$col("x")$sum())
```

---

expr__tail *Get the last n elements*

Description

Get the last n elements

Usage

```
expr__tail(n = 10)
```

Arguments

n	Number of elements to take.

Value

A polars expression

Examples

```
pl$DataFrame(x = 1:11)$select(pl$col("x")$tail(3))
```

`expr__tan` *Compute tangent*

Description

Compute tangent

Usage

```
expr__tan()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(0, pi / 2, pi, NA))$
  with_columns(tangent = pl$col("a")$tan())
```

`expr__tanh` *Compute hyperbolic tangent*

Description

Compute hyperbolic tangent

Usage

```
expr__tanh()
```

Value

A polars expression

Examples

```
pl$DataFrame(a = c(-1, atanh(0.5), 0, 1, NA))$
  with_columns(tanh = pl$col("a")$tanh())
```

expr__top_k — *Return the* k *largest elements*

Description

Non-null elements are always preferred over null elements. The output is not guaranteed to be in any particular order, call $sort() after this function if you wish the output to be sorted. This has time complexity $O(n)$.

Usage

```
expr__top_k(k = 5)
```

Arguments

k	Number of elements to return.

Value

A polars expression

Examples

```
df <- pl$DataFrame(value = c(1, 98, 2, 3, 99, 4))
df$select(
  top_k = pl$col("value")$top_k(k = 3),
  bottom_k = pl$col("value")$bottom_k(k = 3)
)
```

expr__top_k_by — *Return the* k *largest elements*

Description

Non-null elements are always preferred over null elements. The output is not guaranteed to be in any particular order, call $sort() after this function if you wish the output to be sorted. This has time complexity $O(n)$.

Usage

```
expr__top_k_by(by, k = 5, ..., reverse = FALSE)
```

Arguments

by	Column(s) used to determine the smallest elements. Accepts expression input. Strings are parsed as column names.
k	Number of elements to return.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = 1:6,
  b = 6:1,
  c = c("Apple", "Orange", "Apple", "Apple", "Banana", "Banana")
)

# Get the top 2 rows by column a or b:
df$select(
  pl$all()$top_k_by("a", 2)$name$suffix("_btm_by_a"),
  pl$all()$top_k_by("b", 2)$name$suffix("_btm_by_b")
)

# Get the top 2 rows by multiple columns with given order.
df$select(
  pl$all()$
    top_k_by(c("c", "a"), 2, reverse = c(FALSE, TRUE))$
    name$suffix("_btm_by_ca"),
  pl$all()$
    top_k_by(c("c", "b"), 2, reverse = c(FALSE, TRUE))$
    name$suffix("_btm_by_cb"),
)

# Get the top 2 rows by column a in each group
df$group_by("c", maintain_order = TRUE)$agg(
  pl$all()$top_k_by("a", 2)
)$explode(pl$all()$exclude("c"))
```

---

expr__to_physical *Cast to physical representation of the logical dtype*

Description

The following data types will be changed:

- Date -> Int32
- Datetime -> Int64
- Time -> Int64
- Duration -> Int64
- Categorical -> UInt32
- List(inner) -> List(physical of inner)

Other data types will be left unchanged.

Usage

```
expr__to_physical()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = factor(c("a", "x", NA, "a")))
df$with_columns(
  phys = pl$col("a")$to_physical()
)
```

---

expr__true_div *Divide two expressions*

Description

Method equivalent of float division operator `expr / other`. `$truediv()` is an alias for `$true_div()`, which exists for compatibility with Python Polars.

Usage

```
expr__true_div(other)

expr__truediv(other)
```

Arguments

`other`	Numeric literal or expression value.

Details

Zero-division behaviour follows IEEE-754:

- `0/0`: Invalid operation - mathematically undefined, returns `NaN`.
- `n/0`: On finite operands gives an exact infinite result, e.g.: ±infinity.

Value

A polars expression

See Also

- Arithmetic operators
- `<Expr>$floor_div()`

Examples

```
df <- pl$DataFrame(
  x = -2:2,
  y = c(0.5, 0, 0, -4, -0.5)
)

df$with_columns(
  `x/2` = pl$col("x")$true_div(2),
  `x/y` = pl$col("x")$true_div(pl$col("y"))
)
```

---

expr__unique *Get unique values*

Description

This method differs from `$value_counts()` in that it does not return the values, only the counts and might be faster.

Usage

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

Arguments

`maintain_order`
: Maintain order of data. This requires more work.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = c(1, 1, 2))
df$select(pl$col("a")$unique())
```

expr__unique_counts *Count unique values in the order of appearance*

Description

This method differs from `$value_counts()` in that it does not return the values, only the counts and might be faster.

Usage

```
expr__unique_counts()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(id = c("a", "b", "b", "c", "c", "c"))
df$select(pl$col("id")$unique_counts())
```

expr__upper_bound *Calculate the upper bound*

Description

Returns a unit Series with the highest value possible for the dtype of this expression.

Usage

```
expr__upper_bound()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:3)
df$select(pl$col("a")$upper_bound())
```

`expr__value_counts` *Count the occurrences of unique values*

Description

Count the occurrences of unique values

Usage

```
expr__value_counts(
  ...,
  sort = FALSE,
  parallel = FALSE,
  name = "count",
  normalize = FALSE
)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`sort`	Sort the output by count in descending order. If `FALSE` (default), the order of the output is random.
`parallel`	Execute the computation in parallel. This option should likely not be enabled in a group by context, as the computation is already parallelized per group.
`name`	Give the resulting count field a specific name. Default is `"count"`.
`normalize`	If `TRUE`, gives relative frequencies of the unique values.

Value

A polars expression

Examples

```
df <- pl$DataFrame(color = c("red", "blue", "red", "green", "blue", "blue"))
df$select(pl$col("color")$value_counts())

# Sort the output by (descending) count and customize the count field name.
df <- df$select(pl$col("color")$value_counts(sort = TRUE, name = "n"))
df

df$unnest()
```

expr__var *Compute the variance*

Description

Compute the variance

Usage

```
expr__var(ddof = 1)
```

Value

A polars expression

Examples

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

expr__xor *Apply logical XOR on two expressions*

Description

Combine two boolean expressions with XOR.

Usage

```
expr__xor(other)
```

Arguments

`other`	Element to add. Can be a string (only if `expr` is a string), a numeric value or an other expression.

Value

A polars expression

Examples

```
pl$lit(TRUE)$xor(pl$lit(FALSE))
```

`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, indicats type coercion optimization.
`predicate_pushdown`	A logical, indicats predicate pushdown optimization.
`projection_pushdown`	A logical, indicats projection pushdown optimization.
`simplify_expression`	A logical, indicats simplify expression optimization.
`slice_pushdown`	A logical, indicats slice pushdown optimization.
`comm_subplan_elim`	A logical, indicats tring to cache branching subplans that occur on self-joins or unions.
`comm_subexpr_elim`	A logical, indicats tring to cache common subexpressions.
`cluster_with_columns`	A logical, indicats 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 *Select and modify columns of a LazyFrame*

Description

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

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

Usage

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

Arguments

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

Value

A polars LazyFrame

Examples

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

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

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

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

# Expressions with multiple outputs can be automatically instantiated
# as Structs by setting the `POLARS_AUTO_STRUCTIFY` environment variable.
# (Experimental)
if (requireNamespace("withr", quietly = TRUE)) {
  withr::with_envvar(c(POLARS_AUTO_STRUCTIFY = "1"), {
    lf$select(
      is_odd = ((pl$col(pl$Int32) %% 2) == 1)$name$suffix("_is_odd"),
    )$collect()
  })
}
```

---

lazyframe__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 77.

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__all — *Either return an expression representing all columns, or evaluate a bitwise AND operation*

Description

If no arguments are passed, this function is syntactic sugar for `col("*")`. Otherwise, this function is syntactic sugar for `col(names)$all()`.

Usage

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

Arguments

...	Name(s) of the columns to use in the aggregation.
ignore_nulls	If TRUE (default), ignore null values. If FALSE, Kleene logic is used to deal with nulls: if the column contains any null values and no TRUE values, the output is null.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(TRUE, FALSE, TRUE),
  b = c(FALSE, FALSE, FALSE)
)

# Selecting all columns
df$select(pl$all()$sum())

# Evaluate bitwise AND for a column.
df$select(pl$all("a"))
```

---

pl__all_horizontal *Apply the AND logical horizontally across columns*

Description

Apply the AND logical horizontally across columns

Usage

```
pl__all_horizontal(...)
```

Arguments

...	<dynamic-dots> Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

Details

Kleene logic is used to deal with nulls: if the column contains any null values and no FALSE values, the output is null.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(FALSE, FALSE, TRUE, TRUE, FALSE, NA),
  b = c(FALSE, TRUE, TRUE, NA, NA, NA),
  c = c("u", "v", "w", "x", "y", "z")
)

df$with_columns(
  all = pl$all_horizontal("a", "b", "c")
)
```

---

pl__any *Evaluate a bitwise OR operation*

Description

This function is syntactic sugar for `col(names)$any()`.

Usage

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

Arguments

`...`	Name(s) of the columns to use in the aggregation.
`ignore_nulls`	If `TRUE` (default), ignore null values. If `FALSE`, Kleene logic is used to deal with nulls: if the column contains any null values and no `TRUE` values, the output is null.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(TRUE, FALSE, TRUE),
  b = c(FALSE, FALSE, FALSE)
)

df$select(pl$any("a"))
```

pl__any_horizontal *Apply the OR logical horizontally across columns*

Description

Apply the OR logical horizontally across columns

Usage

```
pl__any_horizontal(...)
```

Arguments

...	<dynamic-dots> Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

Details

Kleene logic is used to deal with nulls: if the column contains any null values and no `FALSE` values, the output is null.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(FALSE, FALSE, TRUE, TRUE, FALSE, NA),
  b = c(FALSE, TRUE, TRUE, NA, NA, NA),
  c = c("u", "v", "w", "x", "y", "z")
)

df$with_columns(
  any = pl$any_horizontal("a", "b", "c")
)
```

pl__arg_where *Return indices where* `condition` *evaluates to* `TRUE`

Description

Return indices where `condition` evaluates to `TRUE`

Usage

```
pl__arg_where(condition)
```

Arguments

condition	Boolean expression to evaluate.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = 1:5)
df$select(
  pl$arg_where(pl$col("a") %% 2 == 0)
)
```

---

pl__coalesce *Folds the columns from left to right, keeping the first non-null value*

Description

Folds the columns from left to right, keeping the first non-null value

Usage

```
pl__coalesce(...)
```

Arguments

...	<dynamic-dots> Non-named objects can be referenced as columns. Each object will be converted to expression by as_polars_expr(). Strings are parsed as column names, other non-expression inputs are parsed as literals.

Value

A polars expression

Examples

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

df$with_columns(d = pl$coalesce("a", "b", "c", 10))

df$with_columns(d = pl$coalesce(pl$col("a", "b", "c"), 10))
```

pl__col *Create an expression representing column(s) in a DataFrame*

Description

Create an expression representing column(s) in a DataFrame

Usage

```
pl__col(...)
```

Arguments

...	<dynamic-dots> The name or data type of the column(s) to represent. Unnamed objects one of the following: • Single string(s) representing column names – Regular expressions starting with ^ and ending with $ are allowed. – Single wildcard "*" has a special meaning: check the examples. • Polars DataType(s)

Value

A polars expression

Examples

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

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

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

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

# Character vectors with length > 1 should be used with `!!!`
pl$col(!!!c("foo", "bar"), "baz")
pl$col("foo", !!!c("bar", "baz"))

# there are some special notations for selecting columns
df <- pl$DataFrame(foo = 1:3, bar = 4:6, baz = 7:9)

## select all columns with a wildcard `"*"`
df$select(pl$col("*"))

## select multiple columns by a regular expression
```

```
## starts with `^` and ends with `$`
df$select(pl$col("^ba.*$"))
```

`pl__concat` *Combine multiple DataFrames, LazyFrames, or Series into a single object*

Description

Combine multiple DataFrames, LazyFrames, or Series into a single object

Usage

```
pl__concat(
  ...,
  how = c("vertical", "vertical_relaxed", "diagonal", "diagonal_relaxed", "horizontal",
    "align"),
  rechunk = FALSE,
  parallel = TRUE
)
```

Arguments

`...`	<`dynamic-dots`> DataFrames, LazyFrames, Series. All elements must have the same class.
`how`	Strategy to concatenate items. Must be one of: • `"vertical"`: applies multiple vstack operations; • `"vertical_relaxed"`: same as `"vertical"`, but additionally coerces columns to their common supertype if they are mismatched (eg: Int32 to Int64); • `"diagonal"`: finds a union between the column schemas and fills missing column values with `null`; • `"diagonal_relaxed"`: same as `"diagonal"`, but additionally coerces columns to their common supertype if they are mismatched (eg: Int32 to Int64); • `"horizontal"`: stacks Series from DataFrames horizontally and fills with `null` if the lengths don't match; • `"align"`: Combines frames horizontally, auto-determining the common key columns and aligning rows using the same logic as `align_frames`; this behaviour is patterned after a full outer join, but does not handle column-name collision. (If you need more control, you should use a suitable join method instead). Series only support the `"vertical"` strategy.
`rechunk`	Make sure that the result data is in contiguous memory.
`parallel`	Only relevant for LazyFrames. This determines if the concatenated lazy computations may be executed in parallel.

Value

The same class (`polars_data_frame`, `polars_lazy_frame` or `polars_series`) as the input.

Examples

```
# default is 'vertical' strategy
df1 <- pl$DataFrame(a = 1L, b = 3L)
df2 <- pl$DataFrame(a = 2L, b = 4L)
pl$concat(df1, df2)

# 'a' is coerced to float64
df1 <- pl$DataFrame(a = 1L, b = 3L)
df2 <- pl$DataFrame(a = 2, b = 4L)
pl$concat(df1, df2, how = "vertical_relaxed")

df_h1 <- pl$DataFrame(l1 = 1:2, l2 = 3:4)
df_h2 <- pl$DataFrame(r1 = 5:6, r2 = 7:8, r3 = 9:10)
pl$concat(df_h1, df_h2, how = "horizontal")

# use 'diagonal' strategy to fill empty column values with nulls
df1 <- pl$DataFrame(a = 1L, b = 3L)
df2 <- pl$DataFrame(a = 2L, c = 4L)
pl$concat(df1, df2, how = "diagonal")

df_a1 <- pl$DataFrame(id = 1:2, x = 3:4)
df_a2 <- pl$DataFrame(id = 2:3, y = 5:6)
df_a3 <- pl$DataFrame(id = c(1L, 3L), z = 7:8)
pl$concat(df_a1, df_a2, df_a3, how = "align")
```

---

pl__concat_list *Horizontally concatenate columns into a single list column*

Description

Horizontally concatenate columns into a single list column

Usage

```
pl__concat_list(...)
```

Arguments

...	<dynamic-dots> Columns to concatenate into a single list column. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals.

Value

A polars expression

Examples

```
df <- pl$DataFrame(a = list(1:2, 3, 4:5), b = list(4, integer(0), NULL))

# Concatenate two existing list columns. Null values are propagated.
df$with_columns(concat_list = pl$concat_list("a", "b"))

# Non-list columns are cast to a list before concatenation. The output data
# type is the supertype of the concatenated columns.
df$select("a", concat_list = pl$concat_list("a", pl$lit("x")))

# Create lagged columns and collect them into a list. This mimics a rolling
# window.
df <- pl$DataFrame(A = c(1, 2, 9, 2, 13))
df <- df$select(
  A_lag_1 = pl$col("A")$shift(1),
  A_lag_2 = pl$col("A")$shift(2),
  A_lag_3 = pl$col("A")$shift(3)
)
df$select(A_rolling = pl$concat_list("A_lag_1", "A_lag_2", "A_lag_3"))
```

---

pl__concat_str *Horizontally concatenate columns into a single string column*

Description

Operates in linear time.

Usage

```
pl__concat_str(..., separator = "", ignore_nulls = FALSE)
```

Arguments

...	<dynamic-dots> Columns to concatenate into a single string column. Accepts expression input. Strings are parsed as column names, other non-expression inputs are parsed as literals. Non-`String` columns are cast to `String`.
separator	String that will be used to separate the values of each column.
ignore_nulls	If `FALSE` (default), null values will be propagated, i.e. if the row contains any null values, the output is null.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = 1:3,
  b = c("dogs", "cats", NA),
  c = c("play", "swim", "walk")
)
df$with_columns(
  full_sentence = pl$concat_str(
    pl$col("a") * 2L,
    pl$col("b"),
    pl$col("c"),
    separator = " ",
  )
)
```

---

pl__cum_sum *Cumulatively sum all values*

Description

This function is syntactic sugar for `col(names)$cum_sum()`.

Usage

```
pl__cum_sum(...)
```

Arguments

...	Name(s) of the columns to use in the aggregation.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

# Get the cum_sum of a column
df$select(pl$cum_sum("a"))

# Get the cum_sum of multiple columns
df$select(pl$cum_sum("a", "b"))
```

`pl__DataFrame` *Polars DataFrame class (*`polars_data_frame`*)*

Description

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

Usage

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

Arguments

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

Details

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

Value

A polars DataFrame

Active bindings

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

- `width`: `$width` returns a integer with the number of columns of the DataFrame.
- `flags`: `$flags` returns a list with column names as names and a named logical vector with the flags as values.

Flags

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

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

Examples

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

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

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

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

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

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

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

---

pl__datetime *Create a Polars literal expression of type Datetime*

Description

Create a Polars literal expression of type Datetime

Usage

```
pl__datetime(
  year,
  month,
  day,
  hour = NULL,
  minute = NULL,
  second = NULL,
  microsecond = NULL,
  ...,
  time_unit = c("us", "ns", "ms"),
  time_zone = NULL,
  ambiguous = c("raise", "earliest", "latest", "null")
)
```

Arguments

`year`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of year.
`month`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of month. Range: 1-12.
`day`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of day. Range: 1-31.
`hour`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of hour. Range: 0-23.
`minute`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of minute. Range: 0-59.
`second`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of second. Range: 0-59.
`microsecond`	An polars expression or something can be coerced to an polars expression by `as_polars_expr()`, which represents a column or literal number of microsecond. Range: 0-999999.
`...`	These dots are for future extensions and must be empty.
`time_unit`	One of `"us"` (default, microseconds), `"ns"` (nanoseconds) or `"ms"`(milliseconds). Representing the unit of time.
`time_zone`	A string or `NULL` (default). Representing the timezone.
`ambiguous`	Determine how to deal with ambiguous datetimes. Character vector or expression containing the followings: • `"raise"` (default): Throw an error

- `"earliest"`: Use the earliest datetime
- `"latest"`: Use the latest datetime
- `"null"`: Return a null value

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  month = c(1, 2, 3),
  day = c(4, 5, 6),
  hour = c(12, 13, 14),
  minute = c(15, 30, 45)
)

df$with_columns(
  pl$datetime(
    2024,
    pl$col("month"),
    pl$col("day"),
    pl$col("hour"),
    pl$col("minute"),
    time_zone = "Australia/Sydney"
  )
)

# We can also use `pl$datetime()` for filtering:
df <- pl$select(
  start = ISOdatetime(2024, 1, 1, 0, 0, 0),
  end = c(
    ISOdatetime(2024, 5, 1, 20, 15, 10),
    ISOdatetime(2024, 7, 1, 21, 25, 20),
    ISOdatetime(2024, 9, 1, 22, 35, 30)
  )
)

df$filter(pl$col("end") > pl$datetime(2024, 6, 1))
```

---

pl__datetime_range *Generate a datetime range*

Description

Generate a datetime range

Usage

```
pl__datetime_range(
  start,
  end,
  interval = "1d",
  ...,
  closed = c("both", "left", "none", "right"),
  time_unit = NULL,
  time_zone = NULL
)
```

Arguments

`start`	Lower bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`end`	Upper bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`interval`	Interval of the range periods, specified as a difftime object or using the Polars duration string language. See the `Polars duration string language` section for details. Must consist of full days.
`...`	These dots are for future extensions and must be empty.
`closed`	Define which sides of the range are closed (inclusive). One of the following: `"both"` (default), `"left"`, `"right"`, `"none"`.
`time_unit`	Time unit of the resulting the Datetime data type. One of `"ns"`, `"us"`, `"ms"` or `NULL`
`time_zone`	Time zone of the resulting Datetime data type.

Value

A polars expression

Polars duration string language

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

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)

- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

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

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

See Also

`pl$datetime_ranges()` to create a simple Series of data type list(Datetime) based on column values.

Examples

```
# Using Polars duration string to specify the interval:
pl$select(
  datetime = pl$datetime_range(as.Date("2022-01-01"), as.Date("2022-03-01"), "1mo")
)

# Using `difftime` object to specify the interval:
pl$select(
  datetime = pl$datetime_range(
    as.Date("1985-01-01"),
    as.Date("1985-01-10"),
    as.difftime(1, units = "days") + as.difftime(12, units = "hours")
  )
)

# Specifying a time zone:
pl$select(
  datetime = pl$datetime_range(
    as.Date("2022-01-01"),
    as.Date("2022-03-01"),
    "1mo",
    time_zone = "America/New_York"
  )
)
```

---

pl__datetime_ranges *Generate a list containing a datetime range*

Description

Generate a list containing a datetime range

Usage

```
pl__datetime_ranges(
  start,
  end,
  interval = "1d",
  ...,
  closed = c("both", "left", "none", "right"),
  time_unit = NULL,
  time_zone = NULL
)
```

Arguments

`start`	Lower bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`end`	Upper bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`interval`	Interval of the range periods, specified as a difftime object or using the Polars duration string language. See the `Polars duration string language` section for details. Must consist of full days.
`...`	These dots are for future extensions and must be empty.
`closed`	Define which sides of the range are closed (inclusive). One of the following: `"both"` (default), `"left"`, `"right"`, `"none"`.
`time_unit`	Time unit of the resulting the Datetime data type. One of `"ns"`, `"us"`, `"ms"` or `NULL`
`time_zone`	Time zone of the resulting Datetime data type.

Value

A polars expression

Polars duration string language

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

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)

- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

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

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

See Also

`pl$datetime_range()` to create a simple Series of data type Datetime.

Examples

```
df <- pl$DataFrame(
  start = as.POSIXct(c("2022-01-01 10:00", "2022-01-01 11:00", NA)),
  end = rep(as.POSIXct("2022-01-01 12:00"), 3)
)

df$with_columns(
  dt_range = pl$datetime_ranges("start", "end", interval = "1h"),
  dt_range_cr = pl$datetime_ranges("start", "end", closed = "right", interval = "1h")
)

# provide a custom "end" value
df$with_columns(
  dt_range_lit = pl$datetime_ranges(
    "start", pl$lit(as.POSIXct("2022-01-01 11:00")),
    interval = "1h"
  )
)
```

---

pl__date_range *Generate a date range*

Description

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

Usage

```
pl__date_range(
  start,
  end,
  interval = "1d",
```

```
  ...,
  closed = c("both", "left", "none", "right")
)
```

Arguments

`start`	Lower bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`end`	Upper bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`interval`	Interval of the range periods, specified as a difftime object or using the Polars duration string language. See the `Polars duration string language` section for details. Must consist of full days.
`...`	These dots are for future extensions and must be empty.
`closed`	Define which sides of the range are closed (inclusive). One of the following: `"both"` (default), `"left"`, `"right"`, `"none"`.

Value

A polars expression

Polars duration string language

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

It has the following format:

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

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

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

See Also

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

Examples

```
# Using Polars duration string to specify the interval:
pl$select(
  date = pl$date_range(as.Date("2022-01-01"), as.Date("2022-03-01"), "1mo")
)

# Using `difftime` object to specify the interval:
pl$select(
  date = pl$date_range(
    as.Date("1985-01-01"),
    as.Date("1985-01-10"),
    as.difftime(2, units = "days")
  )
)
```

---

pl__date_ranges *Create a column of date ranges*

Description

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

Usage

```
pl__date_ranges(
  start,
  end,
  interval = "1d",
  ...,
  closed = c("both", "left", "none", "right")
)
```

Arguments

`start`	Lower bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`end`	Upper bound of the date range. Something that can be coerced to a Date or a Datetime expression. See examples for details.
`interval`	Interval of the range periods, specified as a difftime object or using the Polars duration string language. See the `Polars duration string language` section for details. Must consist of full days.

...	These dots are for future extensions and must be empty.
closed	Define which sides of the range are closed (inclusive). One of the following: "both" (default), "left", "right", "none".

Value

A polars expression

Polars duration string language

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

It has the following format:

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

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

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

See Also

pl$date_range() to create a simple Series of data type Date.

Examples

```
df <- pl$DataFrame(
  start = as.Date(c("2022-01-01", "2022-01-02", NA)),
  end = rep(as.Date("2022-01-03"), 3)
)

df$with_columns(
  date_range = pl$date_ranges("start", "end"),
  date_range_cr = pl$date_ranges("start", "end", closed = "right")
)
```

```
# provide a custom "end" value
df$with_columns(
  date_range_lit = pl$date_ranges("start", pl$lit(as.Date("2022-01-02")))
)
```

`pl__duration` *Create polars Duration from distinct time components*

Description

A Duration represents a fixed amount of time. For example, `pl$duration(days = 1)` means "exactly 24 hours". By contrast, `<expr>$dt$offset_by("1d")` means "1 calendar day", which could sometimes be 23 hours or 25 hours depending on Daylight Savings Time. For non-fixed durations such as "calendar month" or "calendar day", please use `<expr>$dt$offset_by()` instead.

Usage

```
pl__duration(
  ...,
  weeks = NULL,
  days = NULL,
  hours = NULL,
  minutes = NULL,
  seconds = NULL,
  milliseconds = NULL,
  microseconds = NULL,
  nanoseconds = NULL,
  time_unit = NULL
)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`weeks`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of weeks, or `NULL` (default).
`days`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of days, or `NULL` (default).
`hours`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of hours, or `NULL` (default).
`minutes`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of minutes, or `NULL` (default).
`seconds`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of seconds, or `NULL` (default).
`milliseconds`	Something can be coerced to an polars expression by `as_polars_expr()` which represents a column or literal number of milliseconds, or `NULL` (default).

microseconds	Something can be coerced to an polars expression by as_polars_expr() which represents a column or literal number of microseconds, or NULL (default).
nanoseconds	Something can be coerced to an polars expression by as_polars_expr() which represents a column or literal number of nanoseconds, or NULL (default).
time_unit	One of NULL, "us" (microseconds), "ns" (nanoseconds) or "ms"(milliseconds). Representing the unit of time. If NULL (default), the time unit will be inferred from the other inputs: "ns" if nanoseconds was specified, "us" otherwise.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  dt = as.POSIXct(c("2022-01-01", "2022-01-02")),
  add = c(1, 2)
)
df

df$select(
  add_weeks = pl$col("dt") + pl$duration(weeks = pl$col("add")),
  add_days = pl$col("dt") + pl$duration(days = pl$col("add")),
  add_seconds = pl$col("dt") + pl$duration(seconds = pl$col("add")),
  add_millis = pl$col("dt") + pl$duration(milliseconds = pl$col("add")),
  add_hours = pl$col("dt") + pl$duration(hours = pl$col("add"))
)
```

---

pl__element *Alias for an element being evaluated in an eval expression*

Description

Alias for an element being evaluated in an eval expression

Usage

```
pl__element()
```

Value

A polars expression

Examples

```
# A horizontal rank computation by taking the elements of a list:
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2)
)
df$with_columns(
  rank = pl$concat_list(c("a", "b"))$list$eval(pl$element()$rank())
)

# A mathematical operation on array elements:
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2)
)
df$with_columns(
  a_b_doubled = pl$concat_list(c("a", "b"))$list$eval(pl$element() * 2)
)
```

---

pl__first *Get the first column of the context*

Description

Get the first column of the context

Usage

```
pl__first()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "baz")
)

df$select(pl$first())
```

`pl__int_range` *Generate a range of integers*

Description

Generate a range of integers

Usage

```
pl__int_range(start = 0, end = NULL, step = 1, ..., dtype = pl$Int64)
```

Arguments

`start`	Start of the range (inclusive). Defaults to 0.
`end`	End of the range (exclusive). If `NULL` (default), the value of `start` is used and `start` is set to 0.
`step`	Step size of the range.
`...`	These dots are for future extensions and must be empty.
`dtype`	Data type of the range.

Value

A polars expression

Examples

```
pl$select(int = pl$int_range(0, 3))

# end can be omitted for a shorter syntax.
pl$select(int = pl$int_range(3))

# Generate an index column by using int_range in conjunction with len().
df <- pl$DataFrame(a = c(1, 3, 5), b = c(2, 4, 6))
df$select(
  index = pl$int_range(pl$len(), dtype = pl$UInt32),
  pl$all()
)
```

pl__int_ranges *Generate a range of integers for each row of the input columns*

Description

Generate a range of integers for each row of the input columns

Usage

```
pl__int_ranges(start = 0, end = NULL, step = 1, ..., dtype = pl$Int64)
```

Arguments

`start`	Start of the range (inclusive). Defaults to 0.
`end`	End of the range (exclusive). If `NULL` (default), the value of `start` is used and `start` is set to 0.
`step`	Step size of the range.
`...`	These dots are for future extensions and must be empty.
`dtype`	Data type of the range.

Value

A polars expression

Examples

```
df <- pl$DataFrame(start = c(1, -1), end = c(3, 2))
df$with_columns(int_range = pl$int_ranges("start", "end"))

# end can be omitted for a shorter syntax$
df$select("end", int_range = pl$int_ranges("end"))
```

pl__last *Get the last column of the context*

Description

Get the last column of the context

Usage

```
pl__last()
```

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "baz")
)

df$select(pl$last())
```

---

pl__LazyFrame *Polars LazyFrame class (*polars_lazy_frame*)*

Description

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

Usage

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

Arguments

`...`	<dynamic-dots> Name-value pairs of objects to be converted to polars Series by the `as_polars_series()` function. Each Series will be used as a column of the DataFrame. All values must be the same length. 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 `as_polars_expr()`.
`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__max *Get the maximum value*

Description

This function is syntactic sugar for `col(names)$max()`.

Usage

```
pl__max(...)
```

Arguments

...	Name(s) of the columns to use in the aggregation.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

# Get the maximum value of a column
df$select(pl$max("a"))

# Get the maximum value of multiple columns
df$select(pl$max("a", "b"))
```

---

pl__max_horizontal *Get the maximum value horizontally across columns*

Description

Get the maximum value horizontally across columns

Usage

```
pl__max_horizontal(...)
```

Arguments

...	<dynamic-dots> Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3)
  b = c(4, 5, NA),
  c = c(1, 2, NA, Inf)
)
df$with_columns(
  max = pl$max_horizontal("a", "b")
)
```

pl__mean_horizontal *Compute the mean horizontally across columns*

Description

Compute the mean horizontally across columns

Usage

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

Arguments

...	<dynamic-dots> Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.
ignore_nulls	A logical. If TRUE, ignore null values (default). If FALSE, any null value in the input will lead to a null output.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3)
  b = c(4, 5, NA),
  c = c("x", "y", "z")
)

df$with_columns(
  mean = pl$mean_horizontal("a", "b")
)
```

pl__min *Get the minimum value*

Description

This function is syntactic sugar for `col(names)$min()`.

Usage

```
pl__min(...)
```

Arguments

...	Name(s) of the columns to use in the aggregation.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

# Get the minimum value of a column
df$select(pl$min("a"))

# Get the minimum value of multiple columns
df$select(pl$min("a", "b"))
```

---

pl__min_horizontal *Get the minimum value horizontally across columns*

Description

Get the minimum value horizontally across columns

Usage

```
pl__min_horizontal(...)
```

Arguments

...	<dynamic-dots> Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3)
  b = c(4, 5, NA),
  c = c("x", "y", "z")
)
df$with_columns(
  min = pl$min_horizontal("a", "b")
)
```

---

pl__nth *Get the nth column(s) of the context*

Description

Get the nth column(s) of the context

Usage

```
pl__nth(indices)
```

Arguments

indices	One or more indices representing the columns to retrieve.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "baz")
)

df$select(pl$nth(1))
df$select(pl$nth(c(2, 0)))
```

`pl__read_csv` *New DataFrame from CSV*

Description

New DataFrame from CSV

Usage

```
pl__read_csv(
  source,
  ...,
  has_header = TRUE,
  separator = ",",
  comment_prefix = NULL,
  quote_char = "\"",
  skip_rows = 0,
  schema = NULL,
  schema_overrides = NULL,
  null_values = NULL,
  missing_utf8_is_empty_string = FALSE,
  ignore_errors = FALSE,
  cache = FALSE,
  infer_schema = TRUE,
  infer_schema_length = 100,
  n_rows = NULL,
  encoding = c("utf8", "utf8-lossy"),
  low_memory = FALSE,
  rechunk = FALSE,
  skip_rows_after_header = 0,
  row_index_name = NULL,
  row_index_offset = 0,
  try_parse_dates = FALSE,
  eol_char = "\n",
  raise_if_empty = TRUE,
  truncate_ragged_lines = FALSE,
  decimal_comma = FALSE,
  glob = TRUE,
  storage_options = NULL,
  retries = 2,
  file_cache_ttl = NULL,
  include_file_paths = NULL
)
```

Arguments

`source`	Path to a file or URL. It is possible to provide multiple paths provided that all CSV files have the same schema. It is not possible to provide

several URLs.

`...` Dots which should be empty.

`has_header` Indicate if the first row of dataset is a header or not.If `FALSE`, column names will be autogenerated in the following format: `"column_x"` with `x` being an enumeration over every column in the dataset starting at 1.

`separator` Single byte character to use as separator in the file.

`comment_prefix`
A string, which can be up to 5 symbols in length, used to indicate the start of a comment line. For instance, it can be set to `#` or `//`.

`quote_char` Single byte character used for quoting. Set to `NULL` to turn off special handling and escaping of quotes.

`skip_rows` Start reading after a particular number of rows. The header will be parsed at this offset.

`schema` Provide the schema. This means that polars doesn't do schema inference. This argument expects the complete schema, whereas `schema_overrides` can be used to partially overwrite a schema. This must be a list. Names of list elements are used to match to inferred columns.

`schema_overrides`
Overwrite dtypes during inference. This must be a list. Names of list elements are used to match to inferred columns.

`null_values` Character vector specifying the values to interpret as `NA` values. It can be named, in which case names specify the columns in which this replacement must be made (e.g. `c(col1 = "a")`).

`missing_utf8_is_empty_string`
By default, a missing value is considered to be `NA`. Setting this parameter to `TRUE` will consider missing UTF8 values as an empty character.

`ignore_errors` Keep reading the file even if some lines yield errors. You can also use `infer_schema = FALSE` to read all columns as UTF8 to check which values might cause an issue.

`cache` Cache the result after reading.

`infer_schema` If `TRUE` (default), the schema is inferred from the data using the first `infer_schema_length` rows. When `FALSE`, the schema is not inferred and will be `pl$String` if not specified in `schema` or `schema_overrides`.

`infer_schema_length`
The maximum number of rows to scan for schema inference. If `NULL`, the full data may be scanned (this is slow). Set `infer_schema = FALSE` to read all columns as `pl$String`.

`n_rows` Stop reading from CSV file after reading `n_rows`.

`encoding` Either `"utf8"` or `"utf8-lossy"`. Lossy means that invalid UTF8 values are replaced with "?" characters.

`low_memory` Reduce memory pressure at the expense of performance.

`rechunk` Reallocate to contiguous memory when all chunks / files are parsed.

`skip_rows_after_header`
Skip this number of rows when the header is parsed.

`row_index_name`
: If not `NULL`, this will insert a row index column with the given name into the DataFrame.

`row_index_offset`
: Offset to start the row index column (only used if the name is set).

`try_parse_dates`
: Try to automatically parse dates. Most ISO8601-like formats can be inferred, as well as a handful of others. If this does not succeed, the column remains of data type `pl$String`.

`eol_char`
: Single byte end of line character (default: `\n`). When encountering a file with Windows line endings (`\r\n`), one can go with the default `\n`. The extra `\r` will be removed when processed.

`raise_if_empty`
: If `FALSE`, parsing an empty file returns an empty DataFrame or LazyFrame.

`truncate_ragged_lines`
: Truncate lines that are longer than the schema.

`decimal_comma`
: Parse floats using a comma as the decimal separator instead of a period.

`glob`
: Expand path given via globbing rules.

`storage_options`
: Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here:

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

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

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

`file_cache_ttl`
: Amount of time to keep downloaded cloud files since their last access time, in seconds. Uses the `POLARS_FILE_CACHE_TTL` environment variable (which defaults to 1 hour) if not given.

`include_file_paths`
: Include the path of the source file(s) as a column with this name.

Value

A polars DataFrame

Examples

```
my_file <- tempfile()
write.csv(iris, my_file)
pl$read_csv(my_file)
unlink(my_file)
```

`pl__read_ipc` *Read into a DataFrame from Arrow IPC (Feather v2) file*

Description

Read into a DataFrame from Arrow IPC (Feather v2) file

Usage

```
pl__read_ipc(
  source,
  ...,
  n_rows = NULL,
  cache = TRUE,
  rechunk = FALSE,
  row_index_name = NULL,
  row_index_offset = 0L,
  storage_options = NULL,
  retries = 2,
  file_cache_ttl = NULL,
  hive_partitioning = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  include_file_paths = NULL
)
```

Arguments

`source`	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.
`...`	These dots are for future extensions and must be empty.
`n_rows`	Stop reading from parquet file after reading `n_rows`.
`cache`	Cache the result after reading.
`rechunk`	In case of reading multiple files via a glob pattern rechunk the final DataFrame into contiguous memory chunks.
`row_index_name`	If not `NULL`, this will insert a row index column with the given name into the DataFrame.
`row_index_offset`	Offset to start the row index column (only used if the name is set).
`storage_options`	Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here: • aws

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

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

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

`hive_partitioning`
Infer statistics and schema from Hive partitioned sources and use them to prune reads. If `NULL` (default), it is automatically enabled when a single directory is passed, and otherwise disabled.

`hive_schema` A list containing the column names and data types of the columns by which the data is partitioned, e.g. `list(a = pl$String, b = pl$Float32)`. If `NULL` (default), the schema of the Hive partitions is inferred.

`try_parse_hive_dates`
Whether to try parsing hive values as date / datetime types.

`include_file_paths`
Character value indicating the column name that will include the path of the source file(s).

Value

A polars DataFrame

Examples

```
temp_dir <- tempfile()
# Write a hive-style partitioned arrow file dataset
arrow::write_dataset(
  mtcars,
  temp_dir,
  partitioning = c("cyl", "gear"),
  format = "arrow",
  hive_style = TRUE
)
list.files(temp_dir, recursive = TRUE)

# If the path is a folder, Polars automatically tries to detect partitions
# and includes them in the output
pl$read_ipc(temp_dir)

# We can also impose a schema to the partition
pl$read_ipc(temp_dir, hive_schema = list(cyl = pl$String, gear = pl$Int32))
```

`pl__read_ipc_stream` *Read into a DataFrame from Arrow IPC stream format*

Description

Read into a DataFrame from Arrow IPC stream format

Usage

```
pl__read_ipc_stream(
  source,
  ...,
  columns = NULL,
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  rechunk = TRUE
)
```

Arguments

`source`	A character of the path to an Arrow IPC stream file.
`...`	These dots are for future extensions and must be empty.
`columns`	A character vector of column names to read.
`n_rows`	Stop reading from parquet file after reading `n_rows`.
`row_index_name`	If not `NULL`, this will insert a row index column with the given name into the DataFrame.
`row_index_offset`	Offset to start the row index column (only used if the name is set).
`rechunk`	A logical value to indicate whether to make sure that all data is contiguous.

Value

A polars DataFrame

Examples

```
temp_file <- tempfile(fileext = ".arrows")

mtcars |>
  nanoarrow::write_nanoarrow(temp_file)

pl$read_ipc_stream(temp_file, columns = c("cyl", "am"))
```

`pl__read_ndjson` *Read into a DataFrame from NDJSON file*

Description

Read into a DataFrame from NDJSON file

Usage

```
pl__read_ndjson(
  source,
  ...,
  schema = NULL,
  schema_overrides = NULL,
  infer_schema_length = 100,
  batch_size = 1024,
  n_rows = NULL,
  low_memory = FALSE,
  rechunk = FALSE,
  row_index_name = NULL,
  row_index_offset = 0L,
  ignore_errors = FALSE,
  storage_options = NULL,
  retries = 2,
  file_cache_ttl = NULL,
  include_file_paths = NULL
)
```

Arguments

`source`	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.
`...`	These dots are for future extensions and must be empty.
`schema`	Specify the datatypes of the columns. The datatypes must match the datatypes in the file(s). If there are extra columns that are not in the file(s), consider also enabling `allow_missing_columns`.
`schema_overrides`	Overwrite dtypes during inference. This must be a list. Names of list elements are used to match to inferred columns.
`infer_schema_length`	The maximum number of rows to scan for schema inference. If `NULL`, the full data may be scanned (this is slow). Set `infer_schema = FALSE` to read all columns as `pl$String`.
`n_rows`	Stop reading from parquet file after reading `n_rows`.
`low_memory`	Reduce memory pressure at the expense of performance

`rechunk` In case of reading multiple files via a glob pattern rechunk the final DataFrame into contiguous memory chunks.

`row_index_name`
If not `NULL`, this will insert a row index column with the given name into the DataFrame.

`row_index_offset`
Offset to start the row index column (only used if the name is set).

`ignore_errors` Keep reading the file even if some lines yield errors. You can also use `infer_schema = FALSE` to read all columns as UTF8 to check which values might cause an issue.

`storage_options`
Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here:

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

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

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

`file_cache_ttl`
Amount of time to keep downloaded cloud files since their last access time, in seconds. Uses the `POLARS_FILE_CACHE_TTL` environment variable (which defaults to 1 hour) if not given.

`include_file_paths`
Character value indicating the column name that will include the path of the source file(s).

Value

A polars DataFrame

Examples

```
ndjson_filename <- tempfile()
jsonlite::stream_out(iris, file(ndjson_filename), verbose = FALSE)
pl$read_ndjson(ndjson_filename)
```

`pl__read_parquet` *Read into a DataFrame from Parquet file*

Description

Read into a DataFrame from Parquet file

Usage

```
pl__read_parquet(
  source,
  ...,
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  parallel = c("auto", "columns", "row_groups", "prefiltered", "none"),
  use_statistics = TRUE,
  hive_partitioning = NULL,
  glob = TRUE,
  schema = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  rechunk = FALSE,
  low_memory = FALSE,
  cache = TRUE,
  storage_options = NULL,
  retries = 2,
  include_file_paths = NULL,
  allow_missing_columns = FALSE
)
```

Arguments

`source`	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.
`...`	These dots are for future extensions and must be empty.
`n_rows`	Stop reading from parquet file after reading `n_rows`.
`row_index_name`	If not `NULL`, this will insert a row index column with the given name into the DataFrame.
`row_index_offset`	Offset to start the row index column (only used if the name is set).
`parallel`	This determines the direction and strategy of parallelism. `"auto"` will try to determine the optimal direction. The `"prefiltered"` strategy first evaluates the pushed-down predicates in parallel and determines a mask of which rows to read. Then, it parallelizes over both the columns and the

row groups while filtering out rows that do not need to be read. This can provide significant speedups for large files (i.e. many row-groups) with a predicate that filters clustered rows or filters heavily. In other cases, prefiltered may slow down the scan compared other strategies.

The prefiltered settings falls back to auto if no predicate is given.

`use_statistics`
: Use statistics in the parquet to determine if pages can be skipped from reading.

`hive_partitioning`
: Infer statistics and schema from Hive partitioned sources and use them to prune reads.

`glob`
: Expand path given via globbing rules.

`schema`
: Specify the datatypes of the columns. The datatypes must match the datatypes in the file(s). If there are extra columns that are not in the file(s), consider also enabling `allow_missing_columns`.

`hive_schema`
: The column names and data types of the columns by which the data is partitioned. If `NULL` (default), the schema of the hive partitions is inferred.

`try_parse_hive_dates`
: Whether to try parsing hive values as date / datetime types.

`rechunk`
: In case of reading multiple files via a glob pattern rechunk the final DataFrame into contiguous memory chunks.

`low_memory`
: Reduce memory pressure at the expense of performance

`cache`
: Cache the result after reading.

`storage_options`
: Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here:

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

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

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

`include_file_paths`
: Character value indicating the column name that will include the path of the source file(s).

`allow_missing_columns`
: When reading a list of parquet files, if a column existing in the first file cannot be found in subsequent files, the default behavior is to raise an error. However, if `allow_missing_columns` is set to `TRUE`, a full-NULL column is returned instead of erroring for the files that do not contain the column.

Value

A polars DataFrame

Examples

```
# Write a Parquet file than we can then import as DataFrame
temp_file <- withr::local_tempfile(fileext = ".parquet")
as_polars_df(mtcars)$write_parquet(temp_file)

pl$read_parquet(temp_file)

# Write a hive-style partitioned parquet dataset
temp_dir <- withr::local_tempdir()
as_polars_df(mtcars)$write_parquet(temp_dir, partition_by = c("cyl", "gear"))
list.files(temp_dir, recursive = TRUE)

# If the path is a folder, Polars automatically tries to detect partitions
# and includes them in the output
pl$read_parquet(temp_dir)
```

---

pl__scan_csv *Lazily read from a CSV file or multiple files via glob patterns*

Description

This allows the query optimizer to push down predicates and projections to the scan level, thereby potentially reducing memory overhead.

Usage

```
pl__scan_csv(
  source,
  ...,
  has_header = TRUE,
  separator = ",",
  comment_prefix = NULL,
  quote_char = "\"",
  skip_rows = 0,
  schema = NULL,
  schema_overrides = NULL,
  null_values = NULL,
  missing_utf8_is_empty_string = FALSE,
  ignore_errors = FALSE,
  cache = FALSE,
  infer_schema = TRUE,
  infer_schema_length = 100,
```

```
  n_rows = NULL,
  encoding = c("utf8", "utf8-lossy"),
  low_memory = FALSE,
  rechunk = FALSE,
  skip_rows_after_header = 0,
  row_index_name = NULL,
  row_index_offset = 0,
  try_parse_dates = FALSE,
  eol_char = "\n",
  raise_if_empty = TRUE,
  truncate_ragged_lines = FALSE,
  decimal_comma = FALSE,
  glob = TRUE,
  storage_options = NULL,
  retries = 2,
  file_cache_ttl = NULL,
  include_file_paths = NULL
)
```

Arguments

`source`	Path to a file or URL. It is possible to provide multiple paths provided that all CSV files have the same schema. It is not possible to provide several URLs.
`...`	Dots which should be empty.
`has_header`	Indicate if the first row of dataset is a header or not.If `FALSE`, column names will be autogenerated in the following format: `"column_x"` with `x` being an enumeration over every column in the dataset starting at 1.
`separator`	Single byte character to use as separator in the file.
`comment_prefix`	A string, which can be up to 5 symbols in length, used to indicate the start of a comment line. For instance, it can be set to `#` or `//`.
`quote_char`	Single byte character used for quoting. Set to `NULL` to turn off special handling and escaping of quotes.
`skip_rows`	Start reading after a particular number of rows. The header will be parsed at this offset.
`schema`	Provide the schema. This means that polars doesn't do schema inference. This argument expects the complete schema, whereas `schema_overrides` can be used to partially overwrite a schema. This must be a list. Names of list elements are used to match to inferred columns.
`schema_overrides`	Overwrite dtypes during inference. This must be a list. Names of list elements are used to match to inferred columns.
`null_values`	Character vector specifying the values to interpret as `NA` values. It can be named, in which case names specify the columns in which this replacement must be made (e.g. `c(col1 = "a")`).

`missing_utf8_is_empty_string`
: By default, a missing value is considered to be `NA`. Setting this parameter to `TRUE` will consider missing UTF8 values as an empty character.

`ignore_errors`
: Keep reading the file even if some lines yield errors. You can also use `infer_schema = FALSE` to read all columns as UTF8 to check which values might cause an issue.

`cache`
: Cache the result after reading.

`infer_schema`
: If `TRUE` (default), the schema is inferred from the data using the first `infer_schema_length` rows. When `FALSE`, the schema is not inferred and will be `pl$String` if not specified in `schema` or `schema_overrides`.

`infer_schema_length`
: The maximum number of rows to scan for schema inference. If `NULL`, the full data may be scanned (this is slow). Set `infer_schema = FALSE` to read all columns as `pl$String`.

`n_rows`
: Stop reading from CSV file after reading `n_rows`.

`encoding`
: Either `"utf8"` or `"utf8-lossy"`. Lossy means that invalid UTF8 values are replaced with "?" characters.

`low_memory`
: Reduce memory pressure at the expense of performance.

`rechunk`
: Reallocate to contiguous memory when all chunks / files are parsed.

`skip_rows_after_header`
: Skip this number of rows when the header is parsed.

`row_index_name`
: If not `NULL`, this will insert a row index column with the given name into the DataFrame.

`row_index_offset`
: Offset to start the row index column (only used if the name is set).

`try_parse_dates`
: Try to automatically parse dates. Most ISO8601-like formats can be inferred, as well as a handful of others. If this does not succeed, the column remains of data type `pl$String`.

`eol_char`
: Single byte end of line character (default: `\n`). When encountering a file with Windows line endings (`\r\n`), one can go with the default `\n`. The extra `\r` will be removed when processed.

`raise_if_empty`
: If `FALSE`, parsing an empty file returns an empty DataFrame or LazyFrame.

`truncate_ragged_lines`
: Truncate lines that are longer than the schema.

`decimal_comma`
: Parse floats using a comma as the decimal separator instead of a period.

`glob`
: Expand path given via globbing rules.

`storage_options`
: Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here:

  - aws

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

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

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

`file_cache_ttl`
: Amount of time to keep downloaded cloud files since their last access time, in seconds. Uses the `POLARS_FILE_CACHE_TTL` environment variable (which defaults to 1 hour) if not given.

`include_file_paths`
: Include the path of the source file(s) as a column with this name.

`credential_provider`
: Provide a function that can be called to provide cloud storage credentials. The function is expected to return a dictionary of credential keys along with an optional credential expiry time.

Value

A polars LazyFrame

Examples

```
my_file <- tempfile()
write.csv(iris, my_file)
lazy_frame <- pl$scan_csv(my_file)
lazy_frame$collect()
unlink(my_file)
```

---

pl__scan_ipc *Lazily read from an Arrow IPC (Feather v2) file or multiple files via glob patterns*

Description

This allows the query optimizer to push down predicates and projections to the scan level, thereby potentially reducing memory overhead.

Usage

```
pl__scan_ipc(
  source,
  ...,
  n_rows = NULL,
  cache = TRUE,
```

```
  rechunk = FALSE,
  row_index_name = NULL,
  row_index_offset = 0L,
  storage_options = NULL,
  retries = 2,
  file_cache_ttl = NULL,
  hive_partitioning = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  include_file_paths = NULL
)
```

Arguments

`source`	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.
`...`	These dots are for future extensions and must be empty.
`n_rows`	Stop reading from parquet file after reading `n_rows`.
`cache`	Cache the result after reading.
`rechunk`	In case of reading multiple files via a glob pattern rechunk the final DataFrame into contiguous memory chunks.
`row_index_name`	If not `NULL`, this will insert a row index column with the given name into the DataFrame.
`row_index_offset`	Offset to start the row index column (only used if the name is set).
`storage_options`	Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here: • aws • gcp • azure • Hugging Face (`hf://`): Accepts an API key under the token parameter `c(token = YOUR_TOKEN)` or by setting the `HF_TOKEN` environment variable. If `storage_options` is not provided, Polars will try to infer the information from environment variables.
`retries`	Number of retries if accessing a cloud instance fails.
`hive_partitioning`	Infer statistics and schema from Hive partitioned sources and use them to prune reads. If `NULL` (default), it is automatically enabled when a single directory is passed, and otherwise disabled.
`hive_schema`	A list containing the column names and data types of the columns by which the data is partitioned, e.g. `list(a = pl$String, b = pl$Float32)`. If `NULL` (default), the schema of the Hive partitions is inferred.

try_parse_hive_dates
: Whether to try parsing hive values as date / datetime types.

include_file_paths
: Character value indicating the column name that will include the path of the source file(s).

Value

A polars LazyFrame

Examples

```
temp_dir <- tempfile()
# Write a hive-style partitioned arrow file dataset
arrow::write_dataset(
  mtcars,
  temp_dir,
  partitioning = c("cyl", "gear"),
  format = "arrow",
  hive_style = TRUE
)
list.files(temp_dir, recursive = TRUE)

# If the path is a folder, Polars automatically tries to detect partitions
# and includes them in the output
pl$scan_ipc(temp_dir)$collect()

# We can also impose a schema to the partition
pl$scan_ipc(temp_dir, hive_schema = list(cyl = pl$String, gear = pl$Int32))$collect()
```

---

pl__scan_ndjson *Lazily read from a local or cloud-hosted NDJSON file(s)*

Description

This allows the query optimizer to push down predicates and projections to the scan level, thereby potentially reducing memory overhead.

Usage

```
pl__scan_ndjson(
  source,
  ...,
  schema = NULL,
  schema_overrides = NULL,
  infer_schema_length = 100,
  batch_size = 1024,
```

```
  n_rows = NULL,
  low_memory = FALSE,
  rechunk = FALSE,
  row_index_name = NULL,
  row_index_offset = 0L,
  ignore_errors = FALSE,
  storage_options = NULL,
  retries = 2,
  file_cache_ttl = NULL,
  include_file_paths = NULL
)
```

Arguments

`source`	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.
`...`	These dots are for future extensions and must be empty.
`schema`	Specify the datatypes of the columns. The datatypes must match the datatypes in the file(s). If there are extra columns that are not in the file(s), consider also enabling `allow_missing_columns`.
`schema_overrides`	Overwrite dtypes during inference. This must be a list. Names of list elements are used to match to inferred columns.
`infer_schema_length`	The maximum number of rows to scan for schema inference. If `NULL`, the full data may be scanned (this is slow). Set `infer_schema = FALSE` to read all columns as `pl$String`.
`n_rows`	Stop reading from parquet file after reading `n_rows`.
`low_memory`	Reduce memory pressure at the expense of performance
`rechunk`	In case of reading multiple files via a glob pattern rechunk the final DataFrame into contiguous memory chunks.
`row_index_name`	If not `NULL`, this will insert a row index column with the given name into the DataFrame.
`row_index_offset`	Offset to start the row index column (only used if the name is set).
`ignore_errors`	Keep reading the file even if some lines yield errors. You can also use `infer_schema = FALSE` to read all columns as UTF8 to check which values might cause an issue.
`storage_options`	Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here: • aws • gcp • azure

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

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

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

`file_cache_ttl`
Amount of time to keep downloaded cloud files since their last access time, in seconds. Uses the `POLARS_FILE_CACHE_TTL` environment variable (which defaults to 1 hour) if not given.

`include_file_paths`
Character value indicating the column name that will include the path of the source file(s).

Value

A polars LazyFrame

Examples

```
ndjson_filename <- tempfile()
jsonlite::stream_out(iris, file(ndjson_filename), verbose = FALSE)
pl$scan_ndjson(ndjson_filename)$collect()
```

---

pl__scan_parquet *Lazily read from a local or cloud-hosted parquet file (or files)*

Description

This allows the query optimizer to push down predicates and projections to the scan level, thereby potentially reducing memory overhead.

Usage

```
pl__scan_parquet(
  source,
  ...,
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  parallel = c("auto", "columns", "row_groups", "prefiltered", "none"),
  use_statistics = TRUE,
  hive_partitioning = NULL,
  glob = TRUE,
  schema = NULL,
```

```
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  rechunk = FALSE,
  low_memory = FALSE,
  cache = TRUE,
  storage_options = NULL,
  retries = 2,
  include_file_paths = NULL,
  allow_missing_columns = FALSE
)
```

Arguments

`source`	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the `storage_options` parameter.
`...`	These dots are for future extensions and must be empty.
`n_rows`	Stop reading from parquet file after reading `n_rows`.
`row_index_name`	If not `NULL`, this will insert a row index column with the given name into the DataFrame.
`row_index_offset`	Offset to start the row index column (only used if the name is set).
`parallel`	This determines the direction and strategy of parallelism. `"auto"` will try to determine the optimal direction. The `"prefiltered"` strategy first evaluates the pushed-down predicates in parallel and determines a mask of which rows to read. Then, it parallelizes over both the columns and the row groups while filtering out rows that do not need to be read. This can provide significant speedups for large files (i.e. many row-groups) with a predicate that filters clustered rows or filters heavily. In other cases, prefiltered may slow down the scan compared other strategies. The prefiltered settings falls back to auto if no predicate is given.
`use_statistics`	Use statistics in the parquet to determine if pages can be skipped from reading.
`hive_partitioning`	Infer statistics and schema from Hive partitioned sources and use them to prune reads.
`glob`	Expand path given via globbing rules.
`schema`	Specify the datatypes of the columns. The datatypes must match the datatypes in the file(s). If there are extra columns that are not in the file(s), consider also enabling `allow_missing_columns`.
`hive_schema`	The column names and data types of the columns by which the data is partitioned. If `NULL` (default), the schema of the hive partitions is inferred.
`try_parse_hive_dates`	Whether to try parsing hive values as date / datetime types.

`rechunk` In case of reading multiple files via a glob pattern rechunk the final DataFrame into contiguous memory chunks.

`low_memory` Reduce memory pressure at the expense of performance

`cache` Cache the result after reading.

`storage_options`
Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here:

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

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

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

`include_file_paths`
Character value indicating the column name that will include the path of the source file(s).

`allow_missing_columns`
When reading a list of parquet files, if a column existing in the first file cannot be found in subsequent files, the default behavior is to raise an error. However, if `allow_missing_columns` is set to `TRUE`, a full-NULL column is returned instead of erroring for the files that do not contain the column.

Value

A polars LazyFrame

Examples

```
# Write a Parquet file than we can then import as DataFrame
temp_file <- withr::local_tempfile(fileext = ".parquet")
as_polars_df(mtcars)$write_parquet(temp_file)

pl$scan_parquet(temp_file)$collect()

# Write a hive-style partitioned parquet dataset
temp_dir <- withr::local_tempdir()
as_polars_df(mtcars)$write_parquet(temp_dir, partition_by = c("cyl", "gear"))
list.files(temp_dir, recursive = TRUE)

# If the path is a folder, Polars automatically tries to detect partitions
# and includes them in the output
pl$scan_parquet(temp_dir)$collect()
```

`pl__Series` *Polars Series class (*`polars_series`*)*

Description

Series are a 1-dimensional data structure, which are similar to R vectors. Within a series all elements have the same Data Type.

Usage

```
pl__Series(name = NULL, values = NULL)
```

Arguments

`name`	A single string or `NULL`. Name of the Series. Will be used as a column name when used in a polars DataFrame. When not specified, name is set to an empty string.
`values`	An R object. Passed as the `x` param of `as_polars_series()`.

Details

The `pl$Series()` function mimics the constructor of the Series class of Python Polars. This function calls `as_polars_series()` internally to convert the input object to a Polars Series.

Active bindings

- `dtype`: `$dtype` returns the data type of the Series.
- `name`: `$name` returns the name of the Series.
- `shape`: `$shape` returns a integer vector of length two with the number of length of the Series and width of the Series (always 1).

See Also

- `as_polars_series()`

Examples

```
# Constructing a Series by specifying name and values positionally:
s <- pl$Series("a", 1:3)
s

# Active bindings:
s$dtype
s$name
s$shape
```

pl__show_versions *Print out the version of Polars and its optional dependencies*

Description

[**Experimental**] Print out the version of Polars and its optional dependencies.

Usage

```
pl__show_versions()
```

Details

cli enhances the terminal output, especially error messages.

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

- bit64
- blob
- clock
- data.table
- hms
- tibble
- vctrs

Value

`NULL` invisibly.

Examples

```
pl$show_versions()
```

pl__struct *Collect columns into a struct column*

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
```

---

pl__sum *Sum all values*

Description

This function is syntactic sugar for `col(names)$sum()`.

Usage

```
pl__sum(...)
```

Arguments

...	Name(s) of the columns to use in the aggregation.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3),
  b = c(4, 5, 2),
  c = c("foo", "bar", "foo")
)

# Get the sum of a column
df$select(pl$sum("a"))

# Get the sum of multiple columns
df$select(pl$sum("a", "b"))
```

---

pl__sum_horizontal *Compute the sum horizontally across columns*

Description

Compute the sum horizontally across columns

Usage

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

Arguments

...	<dynamic-dots> Columns to aggregate horizontally. Accepts expressions. Strings are parsed as column names, other non-expression inputs are parsed as literals.
ignore_nulls	A logical. If TRUE, ignore null values (default). If FALSE, any null value in the input will lead to a null output.

Value

A polars expression

Examples

```
df <- pl$DataFrame(
  a = c(1, 8, 3)
  b = c(4, 5, NA),
  c = c("x", "y", "z")
)
df$with_columns(
  sum = pl$sum_horizontal("a", "b")
)
```

`pl__time_range` *Generate a time range*

Description

Generate a time range

Usage

```
pl__time_range(
  start = NULL,
  end = NULL,
  interval = "1h",
  ...,
  closed = c("both", "left", "none", "right")
)
```

Arguments

`start`	Lower bound of the time range. If omitted, defaults to `00:00:00.000`.
`end`	Upper bound of the time range. If omitted, defaults to `23:59:59.999`
`interval`	Interval of the range periods, specified as a difftime or using the Polars duration string language (see details).
`...`	These dots are for future extensions and must be empty.
`closed`	Define which sides of the range are closed (inclusive). One of the following: `"both"` (default), `"left"`, `"right"`, `"none"`.

Value

A polars expression

Polars duration string language

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

It has the following format:

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)

- 1w (1 calendar week)
- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

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

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

Examples

```
pl$select(
  time = pl$time_range(
    start = hms::parse_hms("14:00:00"),
    interval = as.difftime("3:15:00")
  )
)
```

---

pl__time_ranges *Create a column of time ranges*

Description

Create a column of time ranges

Usage

```
pl__time_ranges(
  start = NULL,
  end = NULL,
  interval = "1h",
  ...,
  closed = c("both", "left", "none", "right")
)
```

Arguments

`start`	Lower bound of the time range. If omitted, defaults to `00:00:00.000`.
`end`	Upper bound of the time range. If omitted, defaults to `23:59:59.999`
`interval`	Interval of the range periods, specified as a difftime or using the Polars duration string language (see details).
`...`	These dots are for future extensions and must be empty.
`closed`	Define which sides of the range are closed (inclusive). One of the following: `"both"` (default), `"left"`, `"right"`, `"none"`.

Value

A polars expression

Polars duration string language

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

It has the following format:

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

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

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

Examples

```
df <- pl$DataFrame(
  start = hms::parse_hms(c("09:00:00", "10:00:00")),
  end = hms::parse_hms(c("11:00:00", "11:00:00"))
)
df$with_columns(time_range = pl$time_ranges("start", "end"))
```

`polars_dtype` *Polars DataType class (*`polars_dtype`*)*

Description

Polars supports a variety of data types that fall broadly under the following categories:

- Numeric data types: signed integers, unsigned integers, floating point numbers, and decimals.
- Nested data types: lists, structs, and arrays.
- Temporal: dates, datetimes, times, and time deltas.
- Miscellaneous: strings, binary data, Booleans, categoricals, and enums.

All types support missing values represented by the special value `null`. This is not to be conflated with the special value `NaN` in floating number data types; see the section about floating point numbers for more information.

Usage

```
pl__Decimal(precision = NULL, scale = 0L)

pl__Datetime(time_unit = c("us", "ns", "ms"), time_zone = NULL)

pl__Duration(time_unit = c("us", "ns", "ms"))

pl__Categorical(ordering = c("physical", "lexical"))

pl__Enum(categories)

pl__Array(inner, shape)

pl__List(inner)

pl__Struct(...)
```

Arguments

`precision`	Single integer or `NULL` (default), maximum number of digits in each number. If `NULL`, the precision is inferred.
`scale`	Single integer or `NULL`. Number of digits to the right of the decimal point in each number. The default is `0`.
`time_unit`	One of `"us"` (default, microseconds), `"ns"` (nanoseconds) or `"ms"`(milliseconds). Representing the unit of time.
`time_zone`	A string or `NULL` (default). Representing the timezone.
`ordering`	One of `"physical"` (default) or `"lexical"`. Ordering by order of appearance (`"physical"`) or string value (`"lexical"`).

`categories`	A character vector. Should not contain `NA` values and all values should be unique.
`inner`	A polars data type object.
`shape`	A integer-ish vector, representing the shape of the Array.
`...`	<dynamic-dots> Name-value pairs of polars data type. Each pair represents a field of the Struct.

Details

Full data types table:

Type(s)	Details
`Boolean`	Boolean type that is bit packed efficiently.
`Int8`, `Int16`, `Int32`, `Int64`	Varying-precision signed integer types.
`UInt8`, `UInt16`, `UInt32`, `UInt64`	Varying-precision unsigned integer types.
`Float32`, `Float64`	Varying-precision signed floating point numbers.
`Decimal` [**Experimental**]	Decimal 128-bit type with optional precision and non-negative scale.
`String`	Variable length UTF-8 encoded string data, typically Human-readable.
`Binary`	Stores arbitrary, varying length raw binary data.
`Date`	Represents a calendar date.
`Time`	Represents a time of day.
`Datetime`	Represents a calendar date and time of day.
`Duration`	Represents a time duration.
`Array`	Arrays with a known, fixed shape per series; akin to numpy arrays.
`List`	Homogeneous 1D container with variable length.
`Categorical`	Efficient encoding of string data where the categories are inferred at runtime.
`Enum` [**Experimental**]	Efficient ordered encoding of a set of predetermined string categories.
`Struct`	Composite product type that can store multiple fields.
`Null`	Represents null values.

Examples

```
pl$Int8
pl$Int16
pl$Int32
pl$Int64
pl$UInt8
pl$UInt16
pl$UInt32
pl$UInt64
pl$Float32
pl$Float64
pl$Decimal(scale = 2)
pl$String
pl$Binary
pl$Date
pl$Time
pl$Datetime()
pl$Duration()
```

```
pl$Array(pl$Int32, c(2, 3))
pl$List(pl$Int32)
pl$Categorical()
pl$Enum(c("a", "b", "c"))
pl$Struct(a = pl$Int32, b = pl$String)
pl$Null
```

`polars_duration_string` *The Polars duration string language*

Description

The Polars duration string language

Polars duration string language

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

It has the following format:

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

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

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

polars_expr *Polars expression class (*polars_expr*)*

Description

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

See Also

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

Examples

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

# Expressions will be evaluated inside a context, such as `<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__n_chunks` *Get the number of chunks that this Series contains*

Description

Get the number of chunks that this Series contains

Usage

```
series__n_chunks()
```

Value

An integer value

Examples

```
s <- pl$Series("a", c(1, 2, 3))
s$n_chunks()

s2 <- pl$Series("a", c(4, 5, 6))

# Concatenate Series with rechunk = TRUE
pl$concat(c(s, s2), rechunk = TRUE)$n_chunks()

# Concatenate Series with rechunk = FALSE
pl$concat(c(s, s2), rechunk = FALSE)$n_chunks()
```

`series__to_frame` *Cast this Series to a DataFrame*

Description

Cast this Series to a DataFrame

Usage

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

Arguments

`name`	A character or `NULL`. If not `NULL`, name/rename the Series column in the new DataFrame. If `NULL`, the column name is taken from the Series name.

Value

A polars DataFrame

See Also

- `as_polars_df()`

Examples

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

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

---

series__to_r_vector *Export the Series as an R vector*

Description

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

Usage

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

Arguments

`...`	These dots are for future extensions and must be empty.
`ensure_vector`	A logical value indicating whether to ensure the return value is a vector. When the Series has the Struct data type and this argument is `FALSE` (default), the return value is a data.frame, not a vector (`is.vector(<data.frame>)` is `FALSE`). If `TRUE`, return a named list instead of a data.frame.
`int64`	Determine how to convert Polars' Int64, UInt32, or UInt64 type values to R type. One of the followings: • `"double"` (default): Convert to the R's double type. Accuracy may be degraded. • `"character"`: Convert to the R's character type. • `"integer"`: Convert to the R's integer type. If the value is out of the range of R's integer type, export as NA_integer_. • `"integer64"`: Convert to the bit64::integer64 class. The bit64 package must be installed. If the value is out of the range of bit64::integer64, export as bit64::NA_integer64_.
`date`	Determine how to convert Polars' Date type values to R class. One of the followings: • `"Date"` (default): Convert to the R's Date class. • `"IDate"`: Convert to the data.table::IDate class.
`time`	Determine how to convert Polars' Time type values to R class. One of the followings: • `"hms"` (default): Convert to the hms::hms class. • `"ITime"`: Convert to the data.table::ITime class. The data.table package must be installed.
`struct`	Determine how to convert Polars' Struct type values to R class. One of the followings: • `"dataframe"` (default): Convert to the R's data.frame class. • `"tibble"`: Convert to the tibble class. If the tibble package is not installed, a warning will be shown.
`decimal`	Determine how to convert Polars' Decimal type values to R type. One of the followings: • `"double"` (default): Convert to the R's double type. • `"character"`: Convert to the R's character type.
`as_clock_class`	A logical value indicating whether to export datetimes and duration as the clock package's classes. • `FALSE` (default): Duration values are exported as difftime and 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 `as_clock_class` is set to `FALSE` and datetime without timezone values are exported as POSIXct. Character vector or expression containing the followings:

- `"raise"` (default): Throw an error
- `"earliest"`: Use the earliest datetime
- `"latest"`: Use the latest datetime
- `"null"`: Return a `NA` value

`non_existent` Determine how to deal with non-existent datetimes. Only applicable when `as_clock_class` is set to `FALSE` and datetime without timezone values are exported as POSIXct. One of the followings:

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

Details

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