Package 'tplyr2'

August 8, 2026

Title A Grammar of Clinical Summary Tables

Version 0.2.0

Description Implements a grammar of summary data for clinical reports. Clinical summary tables are decomposed into modular layers, each representing an independent summary block. Supports count, descriptive statistics, and shift layer types with a declarative spec-based API built on data.table for performance.

License MIT + file LICENSE

URL https://github.com/atorus-research/tplyr2, https://atorus-research.github.io/tplyr2/

BugReports https://github.com/atorus-research/tplyr2/issues

Encoding UTF-8

Depends R (>= 4.1.0)

Imports data.table, jsonlite, purrr, rlang, stringr

Suggests testthat (>= 3.0.0), withr, yaml, knitr, rmarkdown

VignetteBuilder knitr

Config/testthat/edition 3

LazyData true

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Mike Stackhouse [aut, cre]

Maintainer Mike Stackhouse <mike.stackhouse@atorusresearch.com>

Repository CRAN

Date/Publication 2026-08-08 12:00:02 UTC

Contents

apply_conditional_format *Conditional reformatting of a pre-populated string of numbers*

Description

This function allows you to conditionally re-format a string of numbers based on a numeric value within the string itself. By selecting a "format group", which is targeting a specific number within the string, a user can establish a condition upon which a provided replacement string can be used. Either the entire replacement can be used to replace the entire string, or the replacement text can refill the "format group" while preserving the original width and alignment of the target string.

Usage

```
apply_conditional_format(
  string,
  format_group,
  condition,
  replacement,
  full_string = FALSE
)
```

Arguments

string	Target character vector where text may be replaced
format_group	An integer representing the targeted numeric field within the string, numbered from left to right
condition	An expression, using the variable name x as the target variable within the condition
replacement	A string to use as the replacement value
full_string	TRUE if the full string should be replaced, FALSE if the replacement should be done within the format group

Value

A character vector

Examples

```
string <- c(" 0  (0.0%)", " 8  (9.3%)", "78 (90.7%)")

apply_conditional_format(string, 2, x == 0, " 0        ", full_string = TRUE)

apply_conditional_format(string, 2, x < 1, "(<1%)")
```

apply_formats *Apply format strings to numeric values*

Description

Vectorized formatting function. Takes an f_str object and numeric vectors, returns a character vector of formatted strings.

Usage

```
apply_formats(
  fmt,
  ...,
  precision = NULL,
  lt = NULL,
  gt = NULL,
  lt_gt_group = NULL,
  na = NULL,
  width = NULL,
  pad = c("right", "left")
)
```

Arguments

fmt	An f_str() object. A bare character format string is rejected, since the variable names are what bind ... to the format groups.
...	Numeric vectors, one per variable in the f_str (positional matching)
precision	Optional list of resolved precision per group (for auto-precision)
lt	Optional numeric less-than threshold applied to the group named by lt_gt_group: values in (0, lt) render as "<" lt (see format_number_vec()).
gt	Optional numeric greater-than threshold applied to the group named by lt_gt_group: values in (gt, 100) render as ">" gt.
lt_gt_group	Optional integer index of the format group to which lt/gt apply (used by count layers to target the percent statistic). NULL disables.
na	Optional string substituted for cells whose format-group inputs are all NA, used *instead of* the default blank-width fill. na = "" produces a truly empty cell (nchar 0); na = "NE" renders "NE". The default NULL preserves the blank-width fill. This lets apply_formats() replace hand-rolled fixed-width formatters for externally row-bound statistics.
width	Optional integer total width to pad each formatted token to, using stringr::str_pad(). When the na substitution applies to a cell, na wins and that cell is *not* padded. The default NULL leaves tokens at their natural format width.
pad	Side to pad on when width is set: "right" (default, trailing spaces) or "left" (leading spaces).

Value

Character vector of formatted values

See Also

`f_str()` for the format-string grammar.

Examples

```
# Vectorized: one formatted string per element
apply_formats(f_str("xx (xx.x%)", "n", "pct"),
              n = c(5, 12, 103), pct = c(4.5, 33.333, 99.9))

# `na` replaces the default blank-width fill
apply_formats(f_str("xx.x", "mean"), mean = c(1.2, NA))
apply_formats(f_str("xx.x", "mean"), mean = c(1.2, NA), na = "NE")
apply_formats(f_str("xx.x", "mean"), mean = c(1.2, NA), na = "")

# lt/gt thresholds, targeting the percent group (index 2)
apply_formats(f_str("xx (xx.x%)", "n", "pct"), n = c(1, 199), pct = c(0.4, 99.7),
              lt = 1, gt = 99, lt_gt_group = 2)

# Pad to a fixed width for row-binding against other output
apply_formats(f_str("xx.x", "mean"), mean = c(1.2, 10.75), width = 10)
apply_formats(f_str("xx.x", "mean"), mean = c(1.2, 10.75), width = 10, pad = "left")
```

---

apply_row_masks *Apply row masks to blank repeated row labels*

Description

Walks each `rowlabel*` column top-to-bottom and blanks values that are identical to the previous row, respecting layer boundaries (`ord_layer_index`).

Usage

```
apply_row_masks(result, row_breaks = FALSE)
```

Arguments

`result`	A data.frame produced by `tplyr_build()`
`row_breaks`	Logical. If TRUE, insert a blank row between layers.

Value

A data.frame with repeated labels blanked

Examples

```
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(group_count("AGEGR1", by = "SEX"))
)
built <- tplyr_build(spec, tplyr_adsl)

# SEX repeats on every age-group row
built[, c("rowlabel1", "rowlabel2")]

# Masked: each value prints once, at the top of its block
apply_row_masks(built)[, c("rowlabel1", "rowlabel2")]

# row_breaks inserts a blank row between layers
two <- tplyr_build(
  tplyr_spec(cols = "TRT01P",
             layers = tplyr_layers(group_count("SEX"), group_count("AGEGR1"))),
  tplyr_adsl
)
apply_row_masks(two, row_breaks = TRUE)[, c("rowlabel1", "res1")]
```

---

assoc_test *Association-test column(s) for count, shift, and desc layers*

Description

Configures an association test and lands its formatted result beside the n(\ `comparisons` is supplied:

Usage

```
assoc_test(
  fn,
  format = f_str("x.xxx", "p"),
  label = NULL,
  reference = NULL,
  comparisons = NULL,
  total_row = TRUE
)
```

Arguments

`fn`	A function of one argument. In omnibus mode it is called with the source-data subset (a data.frame) for a single by group; in pairwise mode it is called with a 2x2 numeric matrix (see Details). Its return is rendered into the cell one of two ways: a **numeric** value (or a numeric vector matching the number of variables in `format`) is formatted with `format` – a scalar p-value, or several statistics such as an odds ratio with its confidence interval mapped positionally onto a

multi-variable f_str; or a **character** string is passed through *verbatim*, letting the function that computes an arbitrary test also supply the finished display (a significance flag `"0.031*"`, a ceiling/floor `">.99"`/`"<.0001"`, a sentinel `"NE"`). Return `NA` (numeric or character) to render a blank.

`format` An `f_str` object formatting a **numeric** return; it is ignored when `fn` returns a character string. Its variable count sets how many values `fn` must return: one variable for a scalar (e.g. `f_str("x.xxx", "p")`, the default), or several for a tuple (e.g. `f_str("xx.xx (xx.xx, xx.xx)", "or", "lo", "hi")`). The returned values are passed positionally, so the variable names are free.

`label` Character string used as the result column's header label. In pairwise mode it may be a vector with one entry per comparison (or a single value recycled across comparisons); `NULL` generates a default `"<reference> vs <comparison>"` label per comparison. In omnibus mode defaults to `"p-value"`.

`reference` Pairwise mode only. Character(1) naming the reference arm level of the first `cols` variable. `NULL` (default) uses that variable's first level at build time.

`comparisons` Pairwise mode only. A character vector (or list of single levels) of arm levels, each compared to `reference` (e.g. `c("Low", "High")`). Supplying this switches on pairwise/per-level mode; `NULL` (default) keeps omnibus mode.

`total_row` Pairwise mode only. Logical(1); when the layer also emits a total row (`layer_settings(total_row = TRUE)`), `TRUE` (default) computes the pairwise p-value on that row too – for a nested AE layer the grand-total ("any event anywhere") 2x2 – while `FALSE` leaves it blank. Missing rows are always left blank.

Details

Omnibus mode (`comparisons = NULL`, the default). Runs `fn` once per `by` group, *across* the treatment columns, and lands its result as a single trailing column. The supplied function receives the raw source-data subset for the `by` group (all `cols` levels and all target/row levels), so a caller can tabulate and test naturally (e.g. `fisher.test(table(.data$TRT, .data$RESP))` or `coin::cmh_test(...)`). When the layer has no `by` variable the test runs once over the whole layer; otherwise once per `by` group, with the value placed on that group's first output row. This mode works on **count**, **shift**, and **desc** layers – on a desc layer it is the natural home for a continuous-variable comparison across arms (ANOVA / Kruskal-Wallis / t-test), e.g. `fn = function(.data) anova(lm(AGE ~ TRT, .data))[["Pr(>F)"]][1]`, with one p-value per `by` group on that group's first statistic row.

Pairwise / per-level mode (`comparisons` non-`NULL`). Count layers only. Emits one `pval` column per comparison, each comparing an arm level of the first `cols` variable to `reference`, with a value on *every* target-level row (like `risk_diff`'s `rdiff` columns). On a **nested** count layer it emits a value on every row of every level – each inner (e.g. preferred-term) row and each outer (e.g. system-organ-class subtotal) row, each row's 2x2 built from that row's own counts – and, when the layer has a total row, on the grand-total row too (see `total_row`). Here `fn` receives, for one (row, comparison) pair, a 2x2 contingency **matrix** `matrix(c(n_ref, n_cmp, N_ref - n_ref, N_cmp - n_cmp), nrow = 2)` – rows are (reference, comparison) arm, columns are (event, no event) – where `n` is the cell count and `N` the population denominator for that arm. When the layer sets `distinct_by`, the distinct counts/denominators are used. `fn` returns either a numeric value (a scalar, or a vector of several statistics matching a multi-variable `format`) or a verbatim character display string (`NA` renders a blank).

Attach it to a layer via `layer_settings(assoc_test = assoc_test(...))`.

Value

A tplyr_assoc_test object.

Examples

```
# Omnibus
at <- assoc_test(
  fn = function(.data) fisher.test(table(.data$TRT, .data$RESP))$p.value,
  format = f_str("x.xxx", "p"),
  label = "p-value [1]"
)

# Pairwise per-level (count layer): Fisher on an incidence 2x2
at2 <- assoc_test(
  fn = function(m) fisher.test(m)$p.value,
  reference = "Placebo",
  comparisons = c("Low", "High"),
  format = f_str("x.xxx", "p")
)

# Omnibus on a desc layer: continuous comparison across arms (ANOVA)
at3 <- assoc_test(
  fn = function(.data) anova(lm(AGE ~ TRT, .data))[["Pr(>F)"]][1],
  format = f_str("x.xxx", "p")
)

# Multiple statistics in one cell: odds ratio with a confidence interval
at4 <- assoc_test(
  fn = function(m) {
    ft <- fisher.test(m)
    c(ft$estimate, ft$conf.int[1], ft$conf.int[2])
  },
  reference = "Placebo",
  comparisons = c("Low", "High"),
  format = f_str("xx.xx (xx.xx, xx.xx)", "or", "lo", "hi"),
  label = "OR (95% CI)"
)
```

---

as_display *Extract a display-ready frame from a build result*

Description

Returns just the display content of a tplyr_build result, dropping the internal ordering helpers (ord_layer_index, ord_layer_*) and the row_id metadata column, and giving a frame ready to hand to a table-rendering package (clinify, flextable, gt, ...). The build output is already ordered, so no re-sorting is performed.

Usage

```
as_display(x, labels = FALSE)
```

Arguments

x	A data.frame produced by tplyr_build.
labels	Logical. When TRUE, the res* / rdiff* / pval* columns are renamed to their header labels (their label attribute, as returned by get_data_labels); the row-label columns keep their names. Defaults to FALSE.

Details

Everything that is not an internal helper is kept. That is normally the rowlabel*, res*, rdiff*, and pval* columns, but a stats_as_columns desc layer with no by variable names its result columns after the statistics themselves, and those are retained too.

Value

A data.frame of display columns.

Examples

```
spec <- tplyr_spec(cols = "TRT", layers = tplyr_layers(group_count("SEX")))
b <- tplyr_build(spec, data.frame(TRT = rep(c("A", "B"), 3),
                                  SEX = rep(c("F", "M"), 3)))
as_display(b)
```

---

collapse_row_labels *Collapse row labels into a single column*

Description

This is a generalized post processing function that allows you to take groups of by variables and collapse them into a single column. Repeating values are split into separate rows, and for each level of nesting, a specified indentation level can be applied.

Usage

```
collapse_row_labels(
  x,
  ...,
  indent = "  ",
  target_col = "row_label",
  nest = FALSE
)
```

Arguments

x	Input data frame
...	Column names (as character strings) to be collapsed, must be 2 or more
indent	Indentation string to be used, which is multiplied at each indentation level
target_col	Character string naming the output column containing collapsed row labels
nest	Logical. If TRUE, collapse row labels in-place without inserting stub rows for repeating values. Allows a single column to be passed. Default is FALSE.

Value

data.frame with row labels collapsed into a single column

Examples

```
x <- data.frame(
  row_label1 = c("A", "A", "A", "B", "B"),
  row_label2 = c("C", "C", "D", "E", "F"),
  var1 = 1:5,
  stringsAsFactors = FALSE
)

collapse_row_labels(x, "row_label1", "row_label2")

collapse_row_labels(x, "row_label1", "row_label2", indent = "    ",
                    target_col = "rl")
```

---

custom_group *Create a custom column group configuration*

Description

Combines existing column levels into a custom group. Rows matching any of the source levels are duplicated with the column variable set to the group name.

Usage

```
custom_group(col_var, ...)
```

Arguments

col_var	Character string naming the column variable
...	Named arguments where names are group labels and values are character vectors of source levels to combine. Example: `"High Dose" = c("Dose 1", "Dose 2")`

Value

A tplyr_custom_group object

Examples

```
# Pool the two dose arms into one "Xanomeline (All)" column, kept alongside
# the arms it is built from
spec <- tplyr_spec(
  cols = "TRT01P",
  custom_groups = list(custom_group(
    "TRT01P",
    "Xanomeline (All)" = c("Xanomeline High Dose", "Xanomeline Low Dose")
  )),
  layers = tplyr_layers(group_count("AGEGR1"))
)
tplyr_build(spec, tplyr_adsl)

# Several groups at once
custom_group(
  "TRT01P",
  "Active"  = c("Xanomeline High Dose", "Xanomeline Low Dose"),
  "Control" = "Placebo"
)
```

---

f_str *Create a format string object*

Description

Create a format string object

Usage

```
f_str(format_string, ..., empty = NULL)
```

Arguments

format_string	Character string defining the display template
...	Character strings naming the variables that populate the template
empty	Value to display when data is NA/missing. Supplied as `c(.overall = "...")`, it replaces the entire cell, but only once *every* format group in the string is NA. Supplied unnamed (e.g. `empty = "NA"`), it instead fills each NA format group in place, right-justified to the width that group would have occupied, so a partially missing cell keeps its alignment – `f_str("xx (xxx)", "n", "pct", empty = "NA")` renders `"NA ( NA)"`. The default (`NULL`) leaves NA groups as blanks of the field width.

Details

Each run of `x` characters is one format group, and each group is filled by the correspondingly-positioned variable in `...`. The count of `x`s sets the field width, so `"xx.x"` renders two integer digits and one decimal. Literal text between groups is preserved verbatim. `a` (and `A`) request auto-precision, where the decimal count comes from the data.

Value

A tplyr_f_str object

See Also

`apply_formats()` to render values outside a build.

Examples

```
# Two format groups filled by n and pct
fmt <- f_str("xx (xx.x%)", "n", "pct")
fmt
apply_formats(fmt, n = c(5, 12), pct = c(4.5, 33.33))

# Width is set by the number of x's
apply_formats(f_str("xxx", "n"), n = 7)
apply_formats(f_str("x", "n"), n = 7)

# `empty` fills each NA group in place, preserving alignment
apply_formats(f_str("xx (xxx)", "n", "pct", empty = "NA"),
              n = NA, pct = NA)

# `.overall` replaces the whole cell, but only when every group is NA
both_na <- f_str("xx (xxx)", "n", "pct", empty = c(.overall = "Not est."))
apply_formats(both_na, n = NA, pct = NA)

# Used in a layer
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(
    group_count("AGEGR1", settings = layer_settings(
      format_strings = list(n_counts = f_str("xx (xx.x%)", "n", "pct"))))
  )
)
tplyr_build(spec, tplyr_adsl)
```

`generate_row_ids` *Generate unique row IDs for output rows*

Description

Creates a character ID for each row by combining the layer index and row label values. These IDs can be used with `tplyr_meta_result()` and `tplyr_meta_subset()` to look up cell metadata.

Usage

```
generate_row_ids(result)
```

Arguments

`result`	A data.frame produced by `tplyr_build()`

Value

Character vector of row IDs (same length as `nrow(result)`)

Examples

```
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(group_count("AGEGR1"))
)
built <- tplyr_build(spec, tplyr_adsl)
generate_row_ids(built)

# IDs are derived from the row labels, so generate them from an unmodified
# build. tplyr_build(metadata = TRUE) attaches a row_id column that survives
# post-processing, which is the safer route.
with_meta <- tplyr_build(spec, tplyr_adsl, metadata = TRUE)
with_meta$row_id
```

`get_data_labels` *Extract variable labels from a data.frame*

Description

Returns a named character vector of variable labels. Labels are extracted from the `"label"` attribute of each column (standard for Haven-imported CDISC data).

Usage

```
get_data_labels(data)
```

Arguments

`data`	A data.frame

Value

Named character vector where names are column names and values are labels. Columns without labels return `NA_character_`.

Examples

```
# CDISC data imported via haven carries a "label" attribute per column
labs <- get_data_labels(tplyr_adsl)
head(labs)

# Columns with no label attribute come back NA
get_data_labels(data.frame(a = 1, b = 2))
```

`group_analyze` *Create a custom analysis layer*

Description

Allows a user-defined function to compute summary statistics. The function receives a data subset and the target variable name for each group combination, and returns a data.frame of results.

Usage

```
group_analyze(
  target_var,
  by = NULL,
  where = NULL,
  analyze_fn,
  settings = layer_settings()
)
```

Arguments

`target_var`	Character string naming the target variable(s)
`by`	Character string or vector for row grouping
`where`	Expression for filtering data for this layer
`analyze_fn`	A function with signature `function(.data, .target_var)` that returns a data.frame. See Details.
`settings`	A layer_settings object

Details

The `analyze_fn` is called once per group combination (defined by `cols` and `by` data variables). It receives:

- `.data`: A data.frame subset for the current group
- `.target_var`: Character string with the target variable name(s)

If `format_strings` are provided in settings, `analyze_fn` should return a single-row data.frame of named numeric values. Each format string entry becomes one output row, with its name used as the row label.

If no `format_strings` are provided, `analyze_fn` must return a data.frame with `row_label` and `formatted` columns.

Note that `analyze_fn` is called once per `cols` x `by` combination, so it only ever sees a single treatment column at a time — it cannot compute a statistic *across* the treatment columns. For an omnibus association test that spans the columns (e.g. Fisher's exact or CMH on a count/shift layer), see `assoc_test`.

Value

A tplyr_analyze_layer object

See Also

`assoc_test` for cross-column association tests.

Examples

```
# format_strings mode: the function returns one row of named numbers, and
# each format string becomes an output row.
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(
    group_analyze("AGE",
      analyze_fn = function(.data, .target_var) {
        v <- .data[[.target_var]]
        data.frame(gmean = exp(mean(log(v))), rng = diff(range(v)))
      },
      settings = layer_settings(format_strings = list(
        "Geometric mean" = f_str("xx.xx", "gmean"),
        "Range"          = f_str("xx", "rng")
      )))
  )
)
tplyr_build(spec, tplyr_adsl)

# Pre-formatted mode: the function supplies row_label and formatted itself.
pre <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(
    group_analyze("AGE",
```

```
      analyze_fn = function(.data, .target_var) {
        v <- .data[[.target_var]]
        data.frame(
          row_label = "Median [IQR]",
          formatted = sprintf("%.1f [%.1f]", median(v), IQR(v))
        )
      })
  )
)
tplyr_build(pre, tplyr_adsl)
```

`group_count` *Create a count layer*

Description

Create a count layer

Usage

```
group_count(target_var, by = NULL, where = NULL, settings = layer_settings())
```

Arguments

`target_var`	Character string or vector naming the target variable(s). Multiple variables create nested/hierarchical counts.
`by`	Character string or vector for row grouping. Strings that don't match column names are treated as text labels. Use `label()` for explicit disambiguation.
`where`	Expression for filtering data for this layer
`settings`	A layer_settings object

Value

A tplyr_count_layer object

See Also

`layer_settings()` for denominators, sorting, and special rows.

Examples

```
# Counts of a categorical variable within each column group
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(group_count("AGEGR1"))
)
tplyr_build(spec, tplyr_adsl)
```

```
# Distinct subject counts with a total row, filtered to serious events
ae <- tplyr_spec(
  cols = "TRTA",
  layers = tplyr_layers(
    group_count("AEBODSYS",
      where = AESER == "Y",
      settings = layer_settings(distinct_by = "USUBJID", total_row = TRUE))
  )
)
head(tplyr_build(ae, tplyr_adae))

# Two target variables nest: preferred term within body system
nested <- tplyr_spec(
  cols = "TRTA",
  layers = tplyr_layers(group_count(c("AEBODSYS", "AEDECOD")))
)
head(tplyr_build(nested, tplyr_adae))
```

`group_desc` *Create a descriptive statistics layer*

Description

Create a descriptive statistics layer

Usage

```
group_desc(target_var, by = NULL, where = NULL, settings = layer_settings())
```

Arguments

`target_var`	Character string or vector naming the target variable(s)
`by`	Character string or vector for row grouping
`where`	Expression for filtering data for this layer
`settings`	A layer_settings object

Value

A tplyr_desc_layer object

See Also

`layer_settings()` for auto-precision and custom summaries.

Examples

```
# Default summary: n, Mean (SD), Median, Q1/Q3, Min/Max, Missing
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(group_desc("AGE"))
)
tplyr_build(spec, tplyr_adsl)

# Choose the statistics and their formats
custom <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(
    group_desc("AGE", settings = layer_settings(
      format_strings = list(
        "n"         = f_str("xx", "n"),
        "Mean (SD)" = f_str("xx.x (xx.xx)", "mean", "sd")
      )))
  )
)
tplyr_build(custom, tplyr_adsl)

# Several target variables in one layer, grouped by visit
multi <- tplyr_spec(
  cols = "TRTA",
  layers = tplyr_layers(group_desc(c("AVAL", "CHG"), by = "AVISIT"))
)
head(tplyr_build(multi, tplyr_adlb))
```

---

group_shift *Create a shift layer*

Description

Create a shift layer

Usage

```
group_shift(target_var, by = NULL, where = NULL, settings = layer_settings())
```

Arguments

target_var	Named character vector with row and column elements
by	Character string or vector for row grouping
where	Expression for filtering data for this layer
settings	A layer_settings object

Value

A tplyr_shift_layer object

Examples

```
# Baseline (rows) against post-baseline (columns) reference ranges
spec <- tplyr_spec(
  cols = "TRTA",
  layers = tplyr_layers(
    group_shift(c(row = "BNRIND", column = "ANRIND"))
  )
)
head(tplyr_build(spec, tplyr_adlb))

# Percentages relative to each baseline (column) group rather than the arm
by_col <- tplyr_spec(
  cols = "TRTA",
  layers = tplyr_layers(
    group_shift(c(row = "BNRIND", column = "ANRIND"),
                settings = layer_settings(shift_denom = "column"))
  )
)
head(tplyr_build(by_col, tplyr_adlb))
```

---

is_pop_data *Check if an object is a tplyr_pop_data*

Description

Check if an object is a tplyr_pop_data

Usage

```
is_pop_data(x)
```

Arguments

x	An object to check

Value

Logical

Examples

```
is_pop_data(pop_data(cols = "TRT01P"))
is_pop_data("TRT01P")
```

is_tplyr_layer *Check if an object is a tplyr_layer*

Description

Check if an object is a tplyr_layer

Usage

```
is_tplyr_layer(x)
```

Arguments

x	Object to check

Value

Logical

Examples

```
is_tplyr_layer(group_count("SEX"))
is_tplyr_layer(group_desc("AGE"))
is_tplyr_layer("SEX")
```

is_tplyr_spec *Check if an object is a tplyr_spec*

Description

Check if an object is a tplyr_spec

Usage

```
is_tplyr_spec(x)
```

Arguments

x	An object to check

Value

Logical

Examples

```
spec <- tplyr_spec(cols = "TRT01P", layers = tplyr_layers(group_count("SEX")))
is_tplyr_spec(spec)
is_tplyr_spec(mtcars)
```

`label` *Create a text label for use in by parameters*

Description

Explicitly marks a string as a text label (not a data variable name). Useful when a label string might coincidentally match a column name.

Usage

```
label(x)
```

Arguments

`x`	Character string to use as a label

Value

A tplyr_label object

Examples

```
# A `by` string that matches no column is already treated as a label, but
# label() is explicit -- and necessary when the text matches a column name.
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(
    group_count("AGEGR1", by = label("Age Group (y)"))
  )
)
head(tplyr_build(spec, tplyr_adsl))
```

`layer_settings` *Create layer settings*

Description

Configuration object for all layer options. Unused parameters default to NULL and are ignored during build. Type-specific validation happens at build time.

Usage

```
layer_settings(
  format_strings = NULL,
  stat_columns = NULL,
  denoms_by = NULL,
  shift_denom = "total",
  denom_row = FALSE,
  denom_row_label = "n",
  denom_row_format = NULL,
  denom_where = NULL,
  denom_ignore = NULL,
  distinct_by = NULL,
  total_row = FALSE,
  total_row_label = "Total",
  total_row_count_missings = TRUE,
  missing_count = NULL,
  missing_subjects = FALSE,
  missing_subjects_label = "Missing",
  keep_levels = NULL,
  limit_data_by = NULL,
  custom_summaries = NULL,
  stats_as_columns = FALSE,
  precision_by = NULL,
  precision_on = NULL,
  precision_data = NULL,
  precision_cap = NULL,
  order_count_method = NULL,
  ordering_cols = NULL,
  result_order_var = NULL,
  outer_sort_position = NULL,
  risk_diff = NULL,
 ci_method = c("clopper_pearson", "wilson", "wald", "agresti_coull", "jeffreys"),
  ci_level = 0.95,
  assoc_test = NULL,
  pct_lt = NULL,
  pct_gt = NULL,
  zero_count_display = "full",
  name = NULL
```

)

Arguments

format_strings	Named list of f_str objects
stat_columns	Named list of f_str objects for count layers. Each entry produces its own result column per column group (e.g. one "n (\ name used as the column sublabel. Column label attributes follow the pattern "<column group> (N=n) | <stat name>". When set, it takes precedence over format_strings. Names may not contain " | " or "(N=", which are reserved by the label grammar.
denoms_by	Character vector of variable names for denominator grouping. This **replaces** (does not augment) the default denominator grouping, which is the column (cols) variable(s). To get per-column denominators that also break down by a by variable, you must list the cols variable(s) explicitly alongside the by variable(s) — e.g. denoms_by = c("TRT", "SEX"), not denoms_by = "SEX". Passing only the by variable collapses the denominator across the columns.
shift_denom	Denominator basis for shift layers. "total" (default) computes percentages out of the column (cols) total — i.e. the treatment arm. "column" computes them column-wise, out of each shift *column* group (the "from"/baseline group) within the arm, which is the standard "% within the from group" shift display; the header (N=) labels then reflect the per-column-group denominators. Ignored when denoms_by is set (which specifies the grouping explicitly).
denom_row	Logical, shift layers only. When TRUE, emit the per-column-group denominator (the same total used for the percentages) as an integer row above the shift-to rows — the "n" row of a threshold/normal-range shift table. Pairs naturally with shift_denom = "column". Defaults to FALSE.
denom_row_label	Character string, the row label for the denom_row row. Defaults to "n".
denom_row_format	An f_str object formatting the denom_row cells, shift layers only. Lets the denominator row carry its own width independent of the n_counts format (e.g. f_str("xx", "n") for a plain narrow integer). The f_str must reference a single variable (the denominator count is passed positionally). NULL (default) pads the integer to the width of the shift cells. An absent baseline group renders as 0 either way.
denom_where	Expression for separate denominator filter
denom_ignore	Character vector of values to exclude from denominators
distinct_by	Character string naming the variable for distinct counting
total_row	Logical, whether to add a total row
total_row_label	Character string for total row label
total_row_count_missings	Logical, include missing in total
missing_count	List configuring the Missing row. Recognized keys: missing_values Character vector of target values to fold into the Missing row alongside NA.

label Row label; defaults to "Missing".

sort_value Numeric sort key placing the row; defaults to Inf (last).

format An f_str() overriding the layer's count format for this row.

denom_exclude Logical. When TRUE, the rows counted as missing leave the layer's percentage denominator, so percentages are of the non-missing population. This applies to every row in the layer, including the Missing row itself and any total row. Defaults to FALSE.

Any other key is an error.

missing_subjects
: Logical, add missing subjects row

missing_subjects_label
: Character string for missing subjects label

keep_levels
: Character vector of levels to keep

limit_data_by
: Character vector for data limiting

custom_summaries
: Named list of expressions for custom summaries

stats_as_columns
: Logical, transpose stats to columns

precision_by
: Character vector for precision grouping

precision_on
: Character string for precision variable

precision_data
: Data frame with external precision values

precision_cap
: Named numeric vector c(int=, dec=)

order_count_method
: Character, ordering method

ordering_cols
: Character, which column drives ordering

result_order_var
: Character, which result variable for ordering

outer_sort_position
: Character, outer sort direction

risk_diff
: List with risk difference configuration

ci_method
: Method for the single-proportion confidence interval exposed through the ci_lower/ci_upper (and distinct_ci_lower/distinct_ci_upper) count-layer format keywords. One of "clopper_pearson" (default, exact / SAS PROC FREQ EXACT parity), "wilson" (score, matching stats::prop.test(correct = FALSE)), "wald", "agresti_coull", or "jeffreys". See proportion_ci.

ci_level
: Numeric coverage probability for the single-proportion confidence interval keywords. Defaults to 0.95.

assoc_test
: A assoc_test object attaching an association-test p-value column. Omnibus mode works on count, shift, and desc layers (a desc layer's continuous comparison, e.g. ANOVA/Kruskal); pairwise/per-level mode is count layers only.

pct_lt
: Numeric less-than threshold for count-layer percents. A cell with a nonzero count whose percent would display below this value renders the percent as "<" followed by the threshold (e.g. pct_lt = 1 shows 1 (<1%) instead of 1 (0%)). NULL disables.

`pct_gt` Numeric greater-than threshold for count-layer percents. A cell whose percent is below 100 but would display above this value renders the percent as `">"` followed by the threshold (e.g. `pct_gt = 99` shows `>99` for `99.6%`). NULL disables.

`zero_count_display` How to display count-layer cells whose count is zero: `"full"` (default) keeps the usual `"0 ( 0%)"`; `"count_only"` shows only the count field (e.g. `" 0"`); `"blank"` shows an empty string.

`name` Character string, layer name for identification

Value

A tplyr_layer_settings object

Settings by Layer Type

Not all settings apply to every layer type. The table below shows which settings are applicable for each of the four layer types:

Setting	Count	Desc	Shift	Analyze
format_strings	X	X	X	X
stat_columns	X			
denoms_by	X	X	X	
shift_denom			X	
denom_row			X	
denom_row_label			X	
denom_row_format			X	
denom_where	X	X	X	
denom_ignore	X		X	
distinct_by	X		X	
total_row	X			
total_row_label	X			
total_row_count_missings	X			
missing_count	X			
missing_subjects	X			
missing_subjects_label	X			
keep_levels	X			
limit_data_by	X			
custom_summaries		X		
stats_as_columns		X		
precision_by		X		
precision_on		X		
precision_data		X		
precision_cap		X		
order_count_method	X			
ordering_cols	X			
result_order_var	X			
outer_sort_position	X			
risk_diff	X			

ci_method	X			
ci_level	X			
assoc_test	X	X	X	
pct_lt	X			
pct_gt	X			
zero_count_display	X			
name	X	X	X	X

Settings provided for an inapplicable layer type are silently ignored.

Examples

```
# Formats and special rows on a count layer
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(
    group_count("AGEGR1", settings = layer_settings(
      format_strings = list(n_counts = f_str("xx (xx.x%)", "n", "pct")),
      total_row = TRUE,
      total_row_label = "Total subjects"
    ))
  )
)
tplyr_build(spec, tplyr_adsl)

# Denominators: percentages within each arm-by-sex cell rather than the arm
denom <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(
    group_count("AGEGR1", by = "SEX",
                settings = layer_settings(denoms_by = c("TRT01P", "SEX")))
  )
)
tplyr_build(denom, tplyr_adsl)

# Auto-precision on a desc layer: 'a' takes decimals from the data, and
# precision_cap bounds them.
prec <- tplyr_spec(
  cols = "TRTA",
  layers = tplyr_layers(
    group_desc("AVAL", by = "PARAMCD", settings = layer_settings(
      format_strings = list("Mean (SD)" = f_str("a.a+1 (a.a+2)", "mean", "sd")),
      precision_by = "PARAMCD",
      precision_cap = c(int = 3, dec = 2)
    ))
  )
)
head(tplyr_build(prec, tplyr_adlb))

# A custom summary adds a statistic the built-ins do not provide
cv <- tplyr_spec(
```

```
  cols = "TRT01P",
  layers = tplyr_layers(
    group_desc("AGE", settings = layer_settings(
      custom_summaries = list(cv = quote(sd(.var) / mean(.var) * 100)),
      format_strings = list("CV%" = f_str("xx.x", "cv"))
    ))
  )
)
tplyr_build(cv, tplyr_adsl)
```

---

pop_data *Create a population data configuration*

Description

Configuration object specifying how population data maps to the spec. The actual population data.frame is provided at build time via `tplyr_build(spec, data, pop_data = ...)`.

Usage

```
pop_data(cols, where = NULL)
```

Arguments

cols	Character vector of column variable names in the population data. If named, names are the spec column names and values are the pop_data column names (e.g., `c("TRTA" = "TRT01P")`). If unnamed, maps positionally to spec cols.
where	Expression for filtering the population data (optional)

Value

A tplyr_pop_data object

Examples

```
# The AE data's TRTA maps to the subject-level TRT01P. Denominators and the
# header N then come from the population, not from the AE records.
spec <- tplyr_spec(
  cols = "TRTA",
  pop_data = pop_data(cols = c("TRTA" = "TRT01P")),
  layers = tplyr_layers(
    group_count("AEBODSYS",
                settings = layer_settings(distinct_by = "USUBJID"))
  )
)
built <- tplyr_build(spec, tplyr_adae, pop_data = tplyr_adsl)
head(built)
```

```
# Header N reflects the 254 enrolled subjects, not the 200 AE records
tplyr_header_n(built)

# Restrict the population to the safety set
saf <- pop_data(cols = c("TRTA" = "TRT01P"), where = SAFFL == "Y")
saf
```

`proportion_ci` *Confidence interval for a single binomial proportion*

Description

Vectorized computation of a two-sided confidence interval for a single proportion `x / n`, using one of five standard methods. All methods are computed in closed form or via `qbeta` (no per-row `binom.test` loop), so the function scales to a full count table's worth of cells at once.

Usage

```
proportion_ci(
  x,
  n,
  method = c("clopper_pearson", "wilson", "wald", "agresti_coull", "jeffreys"),
  level = 0.95
)
```

Arguments

`x`	Numeric vector of event counts (numerators).
`n`	Numeric vector of trial counts (denominators). Recycled against `x` following the usual R rules.
`method`	One of `"clopper_pearson"` (default), `"wilson"`, `"wald"`, `"agresti_coull"`, or `"jeffreys"`.
`level`	Numeric coverage probability (default `0.95`).

Details

The methods and their references:

`clopper_pearson` Exact interval based on the beta distribution. Matches `stats::binom.test()` and SAS `PROC FREQ ... EXACT` (the clinical convention). This is the default.

`wilson` Wilson score interval without continuity correction. Matches `stats::prop.test(correct = FALSE)`.

`wald` Normal-approximation ("simple asymptotic") interval, clamped to `[0, 1]`.

`agresti_coull` Adds `z^2/2` pseudo-successes and pseudo-failures, then applies the Wald interval to the adjusted counts.

`jeffreys` Bayesian interval using the Jeffreys `Beta(0.5, 0.5)` prior, with the standard boundary adjustments at `x == 0` and `x == n`.

Edge cases: when `n == 0` (or either input is `NA`) both bounds are `NA`; when `x == 0` the lower bound is exactly `0`; when `x == n` the upper bound is exactly `1`.

Value

A `data.table` with two columns, `lower` and `upper`, giving the interval bounds as proportions in `[0, 1]` (multiply by 100 for the percentage scale).

Examples

```
proportion_ci(c(0, 12, 40), c(40, 40, 40), method = "clopper_pearson")
```

---

replace_leading_whitespace

Replace leading whitespace with a specified string

Description

Useful for HTML rendering where leading spaces are collapsed.

Usage

```
replace_leading_whitespace(x, replace_with = " ")
```

Arguments

`x`	Character vector
`replace_with`	Replacement string for each leading space

Value

Character vector with leading spaces replaced

Examples

```
# Indentation survives an HTML renderer that would collapse plain spaces
terms <- c("CARDIAC DISORDERS", "   ATRIAL FIBRILLATION")
out <- replace_leading_whitespace(terms)
out

# One replacement per leading space; interior spacing is untouched
replace_leading_whitespace("  A B", replace_with = "-")
```

str_extract_num *Extract numeric values from formatted strings*

Description

Extracts the Nth numeric value from a formatted tplyr2 string.

Usage

```
str_extract_num(x, index = 1L)
```

Arguments

x	Character vector of formatted strings
index	Integer, which numeric value to extract (1-based)

Value

Numeric vector

Examples

```
cells <- c(" 22 (25.6%)", "  9 (10.5%)", " 55 (64.0%)")

# Default pulls the count; index = 2 pulls the percent
str_extract_num(cells)
str_extract_num(cells, index = 2)

# Asking past the available numbers gives NA, as does an NA input
str_extract_num(c(" 5", NA), index = 2)

# Recover a sort key from an already-formatted table
built <- tplyr_build(
  tplyr_spec(cols = "TRT01P", layers = tplyr_layers(group_count("AGEGR1"))),
  tplyr_adsl
)
built[order(-str_extract_num(built$res1)), c("rowlabel1", "res1")]
```

str_indent_wrap *Wrap strings to a specific width with hyphenation while preserving indentation*

Description

Leverages `stringr::str_wrap()` under the hood, but takes extra steps to preserve any indentation that has been applied to a character element, and use hyphenated wrapping of single words that run longer than the allotted wrapping width.

Usage

```
str_indent_wrap(x, width = 10, tab_width = 5)
```

Arguments

x	An input character vector
width	The desired width of elements within the output character vector
tab_width	The number of spaces to which tabs should be converted

Details

`stringr::str_wrap()` is highly efficient, but in the context of table creation there are two features missing — hyphenation for long running strings that overflow width, and respect for pre-indentation of a character element. For example, in an adverse event table, you may have body system rows as an un-indented column, and preferred terms as indented columns. These strings may run long and require wrapping to not surpass the column width. Furthermore, for crowded tables a single word may be longer than the column width itself.

This function resolves these two issues, while minimizing additional overhead required to apply the wrapping of strings.

Note: This function automatically converts tabs to spaces. Tab width varies depending on font, so width cannot automatically be determined within a data frame. Users can specify the width via `tab_width`.

Value

A character vector with string wrapping applied

Examples

```
ex_text1 <- c("RENAL AND URINARY DISORDERS", "   NEPHROLITHIASIS")
ex_text2 <- c("RENAL AND URINARY DISORDERS", "\tNEPHROLITHIASIS")

cat(paste(str_indent_wrap(ex_text1, width = 8), collapse = "\n\n"), "\n")
cat(paste(str_indent_wrap(ex_text2, tab_width = 4), collapse = "\n\n"), "\n")
```

---

total_group *Create a total group configuration*

Description

Specifies that a synthetic "Total" column level should be added by duplicating all rows with the specified column variable set to the label.

Usage

```
total_group(col_var, label = "Total")
```

Arguments

col_var	Character string naming the column variable to totalize
label	Character string for the total group label (default: "Total")

Value

A tplyr_total_group object

Examples

```
# Adds a "Total" column spanning every arm, alongside the individual arms
spec <- tplyr_spec(
  cols = "TRT01P",
  total_groups = list(total_group("TRT01P")),
  layers = tplyr_layers(group_count("AGEGR1"))
)
tplyr_build(spec, tplyr_adsl)

# Rename the total column
all_pts <- tplyr_spec(
  cols = "TRT01P",
  total_groups = list(total_group("TRT01P", label = "All Patients")),
  layers = tplyr_layers(group_count("SEX"))
)
tplyr_build(all_pts, tplyr_adsl)
```

---

tplyr2_options *Get or set tplyr2 package options*

Description

View and modify tplyr2-specific options. When called with no arguments, returns all current tplyr2 options with their defaults. When called with named arguments, sets those options.

Usage

```
tplyr2_options(...)
```

Arguments

...	Named arguments to set (e.g., IBMRounding = TRUE). Option names are automatically prefixed with tplyr2..

Details

Available options:

tplyr2.IBMRounding Logical. Use round-half-away-from-zero instead of R's default banker's rounding. Default: `FALSE`.

tplyr2.quantile_type Integer. Quantile algorithm type passed to `quantile()`. Default: `7`.

tplyr2.precision_cap Named numeric vector `c(int=, dec=)`. Maximum int/dec widths for auto-precision. Default: `NULL`.

tplyr2.custom_summaries Named list of expressions for global custom summary functions. Default: `NULL`.

tplyr2.scipen Integer. scipen value used during `tplyr_build()` to prevent scientific notation. Default: `9999`.

An unrecognized option name is an error rather than a silent no-op, so a misspelling cannot quietly leave the build on the default behavior.

Value

When called with no arguments, a named list of current option values. When called with arguments, invisibly returns the previous values.

Examples

```
# Inspect the current values
tplyr2_options()

# Setting returns the previous values, so the change can be undone
old <- tplyr2_options(IBMRounding = TRUE)
getOption("tplyr2.IBMRounding")
do.call(options, old)
getOption("tplyr2.IBMRounding")

# IBM (half-away-from-zero) rounding vs R's banker's rounding
fmt <- f_str("xx", "n")
apply_formats(fmt, n = 2.5)
old <- tplyr2_options(IBMRounding = TRUE)
apply_formats(fmt, n = 2.5)
do.call(options, old)

# A misspelled name errors instead of setting a dead option
try(tplyr2_options(IBMrounding = TRUE))
```

tplyr_adae *Adverse events analysis dataset*

Description

A sample CDISC ADaM ADAE dataset from the PHUSE Test Data Factory. Contains adverse event records.

Usage

```
tplyr_adae
```

Format

A data.frame

tplyr_adlb *Laboratory data analysis dataset*

Description

A sample CDISC ADaM ADLB dataset from the PHUSE Test Data Factory. Contains laboratory test results.

Usage

```
tplyr_adlb
```

Format

A data.frame

tplyr_adsl *Subject-level analysis dataset*

Description

A sample CDISC ADaM ADSL dataset from the PHUSE Test Data Factory. Contains subject-level demographics and baseline characteristics.

Usage

```
tplyr_adsl
```

Format

A data.frame

tplyr_build *Build a tplyr2 table from a spec and data*

Description

Executes the table specification against the provided data, producing a formatted output data frame.

Usage

```
tplyr_build(spec, data, pop_data = NULL, metadata = FALSE, ...)
```

Arguments

spec	A tplyr_spec object (or path to a JSON/YAML spec file)
data	A data.frame to process
pop_data	Optional population data.frame (overrides spec pop_data)
metadata	If TRUE, attach cell-level metadata enabling traceability back to source data rows via `tplyr_meta_result()` and `tplyr_meta_subset()`.
...	Additional named arguments overriding spec-level parameters. Names must match a field of `spec` (or `where`/`pop_data`); an unrecognized name is an error rather than a silent no-op. Because `...` is evaluated eagerly, a `where` override must be a character string or a quoted expression, not the bare expression `tplyr_spec()` accepts.

Value

A data.frame with rowlabel, res, and ord columns

See Also

`tplyr_spec()` to build the specification, and `tplyr_numeric_data()` for the unformatted values behind the cells.

Examples

```
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(group_count("AGEGR1"))
)
tplyr_build(spec, tplyr_adsl)

# Override spec fields at build time without editing the spec. Overrides go
# through `...`, which evaluates eagerly, so a `where` must be a string or
# quoted -- not the bare expression tplyr_spec() accepts.
tplyr_build(spec, tplyr_adsl, where = "SEX == 'F'")
tplyr_build(spec, tplyr_adsl, where = quote(SEX == "F"))
```

```
# Population data supplies the denominators and the header N
pop_spec <- tplyr_spec(
  cols = "TRTA",
  pop_data = pop_data(cols = c("TRTA" = "TRT01P")),
  layers = tplyr_layers(
    group_count("AEBODSYS",
                settings = layer_settings(distinct_by = "USUBJID"))
  )
)
head(tplyr_build(pop_spec, tplyr_adae, pop_data = tplyr_adsl))
```

`tplyr_from_ard` *Reconstruct a formatted table from ARD and a spec*

Description

Takes Analysis Results Data (long format) and a `tplyr_spec`, then applies the spec's formatting rules to produce a formatted output table.

Usage

```
tplyr_from_ard(ard, spec)
```

Arguments

`ard`	A data.frame in ARD format (as produced by `tplyr_to_ard()`)
`spec`	A `tplyr_spec` object defining the table structure

Value

A data.frame with the same structure as `tplyr_build()` output

See Also

`tplyr_to_ard()` to produce the ARD.

Examples

```
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(group_desc("AGE"))
)
built <- tplyr_build(spec, tplyr_adsl)

# Round-tripping through the ARD reproduces the formatted cells, so a table
# can be rebuilt from stored results without the subject-level data
ard <- tplyr_to_ard(built)
recon <- tplyr_from_ard(ard, spec)
```

```
recon[, c("rowlabel1", "res1")]
identical(as.vector(recon$res1), as.vector(built$res1))

# Changing only the spec's formats re-renders the same numbers differently
respec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(
    group_desc("AGE", settings = layer_settings(
      format_strings = list("Mean (SD)" = f_str("xx.xx (xx.xxx)", "mean", "sd"))))
  )
)
tplyr_from_ard(ard, respec)[, c("rowlabel1", "res1")]
```

`tplyr_header_n` *Extract header N from a tplyr2 build result*

Description

Returns the population-based header N values that were computed during `tplyr_build()`. Only available when population data was provided.

Usage

```
tplyr_header_n(result)
```

Arguments

`result`	A data.frame produced by `tplyr_build()`

Value

A data.frame with column variable levels and their N values, or NULL if no population data was used.

Examples

```
spec <- tplyr_spec(
  cols = "TRTA",
  pop_data = pop_data(cols = c("TRTA" = "TRT01P")),
  layers = tplyr_layers(group_count("AEBODSYS"))
)
built <- tplyr_build(spec, tplyr_adae, pop_data = tplyr_adsl)
tplyr_header_n(built)

# NULL when the build had no population data to draw an N from
no_pop <- tplyr_build(
  tplyr_spec(cols = "TRT01P", layers = tplyr_layers(group_count("SEX"))),
  tplyr_adsl
)
```

```
tplyr_header_n(no_pop)
```

`tplyr_layers` *Create a list of layers*

Description

Wraps one or more layer objects into a validated list for use in tplyr_spec().

Usage

```
tplyr_layers(...)
```

Arguments

`...`	Layer objects created by group_count(), group_desc(), group_shift(), or group_analyze()

Value

A list of tplyr_layer objects

Examples

```
# Layers stack in the order given, and may mix types freely
layers <- tplyr_layers(
  group_desc("AGE"),
  group_count("SEX"),
  group_count("AGEGR1")
)
length(layers)

spec <- tplyr_spec(cols = "TRT01P", layers = layers)
tplyr_build(spec, tplyr_adsl)
```

`tplyr_meta` *Metadata object for a tplyr output cell*

Description

Contains filter expressions that, when evaluated against the original data, reproduce the subset of rows that contributed to a specific cell in the output table.

Usage

```
tplyr_meta(
  names = character(0),
  filters = list(),
  layer_index = integer(0),
  anti_join = NULL,
  statistic = NULL
)
```

Arguments

names	Character vector of variable names relevant to this cell
filters	List of R language objects (call expressions) representing filter conditions
layer_index	Integer layer index (1-based)
anti_join	NULL or a tplyr_meta_anti_join object for missing subjects rows
statistic	NULL or a character string naming which statistic the cell displays (set for stat_columns layers, where the stat sub-columns of a column group share the same source-data filters)

Value

A tplyr_meta object

See Also

tplyr_meta_result() to retrieve one from a build.

Examples

```
# Usually obtained from a build rather than constructed by hand
m <- tplyr_meta(
  names = c("TRT01P", "AGEGR1"),
  filters = list(quote(TRT01P == "Placebo"), quote(AGEGR1 == "65-80")),
  layer_index = 1L
)
m

# The filters are ordinary language objects, so they can be applied directly
subset(tplyr_adsl, TRT01P == "Placebo" & AGEGR1 == "65-80")[1:3, c("USUBJID", "AGEGR1")]
```

`tplyr_meta_result` *Get metadata for a specific output cell*

Description

Returns a `tplyr_meta` object containing the filter expressions that describe the source data for the specified cell.

Usage

```
tplyr_meta_result(result, row_id, column)
```

Arguments

`result`	A data.frame from `tplyr_build()` built with `metadata = TRUE`
`row_id`	Character row ID (from `result$row_id` or `generate_row_ids()`)
`column`	Character column name (e.g., `"res1"`)

Value

A `tplyr_meta` object, or NULL if no metadata for that cell

See Also

`tplyr_meta_subset()` to fetch the source rows themselves.

Examples

```
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(group_count("AGEGR1"))
)
built <- tplyr_build(spec, tplyr_adsl, metadata = TRUE)
built[, c("row_id", "rowlabel1", "res1")]

# The filters behind the Placebo / 65-80 cell
tplyr_meta_result(built, built$row_id[1], "res1")
```

`tplyr_meta_subset` *Get source data rows for a specific output cell*

Description

Evaluates the stored filter expressions against the original data to return the rows that contributed to the specified output cell.

Usage

```
tplyr_meta_subset(result, row_id, column, data, pop_data = NULL)
```

Arguments

`result`	A data.frame from `tplyr_build()` built with `metadata = TRUE`
`row_id`	Character row ID
`column`	Character column name (e.g., `"res1"`)
`data`	The original data.frame that was passed to `tplyr_build()`
`pop_data`	Optional population data.frame, required when the cell represents a missing subjects row (anti-join)

Value

A data.frame subset of the original data, or NULL if no metadata

Examples

```
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(group_count("AGEGR1"))
)
built <- tplyr_build(spec, tplyr_adsl, metadata = TRUE)
built[, c("row_id", "rowlabel1", "res1")]

# Trace the first cell back to the subjects it counted
src <- tplyr_meta_subset(built, built$row_id[1], "res1", tplyr_adsl)
nrow(src)
head(src[, c("USUBJID", "TRT01P", "AGEGR1")])

# The row count matches the number displayed in the cell
built$res1[1]
```

`tplyr_numeric_data` *Retrieve raw numeric data from a tplyr_build result*

Description

Returns the unformatted numeric data that was computed during the build process, before formatting and pivoting to wide format.

Usage

```
tplyr_numeric_data(result, layer = NULL)
```

Arguments

`result`	A data.frame produced by `tplyr_build()`
`layer`	Integer layer index (1-based), or NULL for all layers

Value

If `layer` is specified, a data.frame of raw statistics for that layer. If `layer` is NULL, a named list of data.frames keyed by layer index. Returns NULL if numeric data was not retained.

See Also

`tplyr_stats_data()` for a single statistic with its grouping columns.

Examples

```
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(
    group_count("AGEGR1"),
    group_desc("AGE")
  )
)
built <- tplyr_build(spec, tplyr_adsl)

# One data.frame per layer, keyed by layer index
names(tplyr_numeric_data(built))

# The counts behind the formatted cells, before rounding and padding
head(tplyr_numeric_data(built, 1))

# Every statistic the desc layer computed, including unused ones
names(tplyr_numeric_data(built, 2))
```

`tplyr_read_spec` *Read a tplyr_spec from JSON or YAML*

Description

Deserializes a spec from a file. Expressions are reconstructed from their string representations.

Usage

```
tplyr_read_spec(path)
```

Arguments

`path`	File path to a JSON or YAML spec file

Value

A tplyr_spec object

See Also

`tplyr_write_spec()` to write one.

Examples

```
spec <- tplyr_spec(
  cols = "TRT01P",
  where = SAFFL == "Y",
  layers = tplyr_layers(group_count("AGEGR1", by = "SEX"))
)

path <- file.path(tempdir(), "spec.json")
tplyr_write_spec(spec, path)

# The round trip reproduces the spec, and the same table
spec2 <- tplyr_read_spec(path)
spec2
identical(tplyr_build(spec, tplyr_adsl), tplyr_build(spec2, tplyr_adsl))

# tplyr_build() also accepts a spec file path directly
head(tplyr_build(path, tplyr_adsl))

unlink(path)
```

tplyr_spec *Create a tplyr2 table specification*

Description

The spec is a pure configuration object describing what to compute. No data processing occurs until `tplyr_build()` is called.

Usage

```
tplyr_spec(
  cols,
  where = NULL,
  pop_data = NULL,
  total_groups = NULL,
  custom_groups = NULL,
  layers = tplyr_layers(),
  settings = NULL
)
```

Arguments

`cols`	Character vector of column variable names
`where`	Expression for global data filter (optional)
`pop_data`	A pop_data() object for population-based features (optional)
`total_groups`	List of total_group() objects (optional)
`custom_groups`	List of custom_group() objects (optional)
`layers`	A list of layer objects from tplyr_layers()
`settings`	Additional spec-level settings (optional)

Value

A tplyr_spec object

See Also

`tplyr_build()` to execute a spec, `tplyr_layers()` to assemble layers, and `layer_settings()` for per-layer configuration.

Examples

```
# A spec is inert configuration -- nothing is computed until tplyr_build()
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(
    group_count("AGEGR1"),
```

```
    group_desc("AGE")
  )
)
spec

tplyr_build(spec, tplyr_adsl)

# A global `where` filter applies to every layer
safety <- tplyr_spec(
  cols = "TRT01P",
  where = SAFFL == "Y",
  layers = tplyr_layers(group_count("SEX"))
)
tplyr_build(safety, tplyr_adsl)
```

`tplyr_stats_data` *Retrieve raw statistic values from a tplyr_build result*

Description

Filters the raw numeric data for a specific layer and statistic. Use `tplyr_numeric_data()` to get every statistic for the layer.

Usage

```
tplyr_stats_data(result, layer, statistic)
```

Arguments

`result`	A data.frame produced by `tplyr_build()`
`layer`	Integer layer index (1-based)
`statistic`	Character string naming the statistic column to extract (e.g., "n", "pct", "mean", "sd")

Value

A data.frame with the layer's grouping columns and the requested statistic. Returns NULL if the layer has no numeric data or does not compute the statistic.

Examples

```
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(group_desc("AGE"))
)
built <- tplyr_build(spec, tplyr_adsl)
tplyr_stats_data(built, 1, "mean")
```

`tplyr_to_ard` *Convert tplyr_build output to Analysis Results Data (ARD) format*

Description

Transforms the numeric data attached to a `tplyr_build()` result into a long-format data frame with one row per statistic per group combination. This is compatible with the CDISC Analysis Results Data standard.

Usage

```
tplyr_to_ard(result)
```

Arguments

`result`	A data.frame produced by `tplyr_build()`

Value

A data.frame in long format with columns:

analysis_id Integer layer index

stat_name Character name of the statistic

stat_value Numeric value of the statistic

... Grouping columns from the original data

See Also

`tplyr_from_ard()` to rebuild a formatted table from an ARD.

Examples

```
spec <- tplyr_spec(
  cols = "TRT01P",
  layers = tplyr_layers(
    group_count("AGEGR1"),
    group_desc("AGE")
  )
)
built <- tplyr_build(spec, tplyr_adsl)

ard <- tplyr_to_ard(built)
head(ard)

# One row per statistic per group; analysis_id identifies the layer
table(ard$analysis_id)
unique(ard$stat_name[ard$analysis_id == 2])
```

tplyr_write_spec *Write a tplyr_spec to JSON or YAML*

Description

Serializes a spec object to a file. The format is determined by the file extension: `.json` for JSON, `.yaml` or `.yml` for YAML.

Usage

```
tplyr_write_spec(spec, path)
```

Arguments

`spec`	A tplyr_spec object
`path`	File path. Extension determines format.

Details

Expressions (e.g., `where` clauses) are deparsed to strings and reconstructed on read. Format string objects (`f_str`) are stored as their component parts and regenerated on read.

Value

Invisible file path

See Also

`tplyr_read_spec()` to read one back.

Examples

```
spec <- tplyr_spec(
  cols = "TRT01P",
  where = SAFFL == "Y",
  layers = tplyr_layers(
    group_count("AGEGR1", by = "SEX", settings = layer_settings(
      denoms_by = c("TRT01P", "SEX"),
      format_strings = list(n_counts = f_str("xx (xx.x%)", "n", "pct"))))
  )
)

path <- file.path(tempdir(), "spec.json")
tplyr_write_spec(spec, path)
cat(readLines(path), sep = "\n")

# YAML is chosen by the file extension
if (requireNamespace("yaml", quietly = TRUE)) {
  ypath <- file.path(tempdir(), "spec.yaml")
```

```
  tplyr_write_spec(spec, ypath)
  cat(readLines(ypath), sep = "\n")
  unlink(ypath)
}
unlink(path)
```

Index