

Package ‘tabular’

June 11, 2026

Title Render Tables and Listings for Clinical Submissions

Version 0.1.0

Description Render clinical submission tables and listings to 'RTF', 'LaTeX', 'HTML', 'PDF', and 'DOCX' from pre-summarised data frames, with no external 'Java' or 'SAS' dependency. Features include decimal alignment via font metrics, multi-level column headers with passthrough leaves, predicate-targeted cell styling, footnotes, and group-aware pagination. Built for Clinical Data Interchange Standards Consortium (CDISC) Analysis Data Model (ADaM) workflows and regulatory submissions to agencies such as the Food and Drug Administration (FDA), European Medicines Agency (EMA), and Pharmaceuticals and Medical Devices Agency (PMDA).

License MIT + file LICENSE

URL <https://vthanik.github.io/tabular/>,
<https://github.com/vthanik/tabular>

BugReports <https://github.com/vthanik/tabular/issues>

Encoding UTF-8

Depends R (>= 4.3.0)

Imports S7, cli, commonmark, rlang, xml2

Suggests testthat (>= 3.0.0), withr, digest, ggplot2, htmltools, knitr, quarto, pkgdown, rstudioapi, systemfonts, tibble, tinytex, webshot2, yaml

VignetteBuilder quarto

SystemRequirements Quarto command line tool
(<<https://github.com/quarto-dev/quarto-cli>>), needed only to build the package vignettes.

Config/testthat/edition 3

Config/Needs/website pkgdown

LazyData true

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Vignesh Thanikachalam [aut, cre, cph]

Maintainer Vignesh Thanikachalam <about.vignesh@gmail.com>

Repository CRAN

Date/Publication 2026-06-11 12:30:02 UTC

Contents

as.tags.tabular_spec	3
as_grid	4
brdr	8
cdisc_eff_estimates	11
cdisc_eff_n	12
cdisc_eff_resp	13
cdisc_saf_ae	15
cdisc_saf_aesocpt	16
cdisc_saf_aesocpt_ard	18
cdisc_saf_demo	19
cdisc_saf_demo_ard	21
cdisc_saf_n	22
cdisc_saf_subgroup	23
cdisc_saf_vital	25
cells	26
check_fonts	29
check_latex	31
cols	33
cols_apply	37
col_spec	39
emit	49
footnote	54
get_preset	56
headers	58
html	62
md	63
paginate	65
pivot_across	70
preset	78
preset_minimal	86
print.tabular_spec	88
set_preset	90
sort_rows	94
style	97
style_template	100
subgroup	101
tabular	106

Index

111

as.tags.tabular_spec *Convert a tabular_spec to an htmltools tagList*

Description

Renders the spec to a self-contained HTML fragment and wraps it in an `htmltools::tagList` suitable for inline embedding in Quarto / Rmd chunks, RStudio / Positron viewer panes, pkgdown reference pages, and Shiny UIs.

Usage

```
## S3 method for class 'tabular_spec'
as.tags(x, ..., id = NULL)
```

Arguments

x	<i>The tabular_spec to convert.</i> <code><tabular_spec></code> : required.
...	<i>Reserved.</i> Ignored.
id	<i>Wrapping div id.</i> <code><character(1) NULL></code> : default NULL (auto-generate). Pass an explicit id when you need to target the table from external CSS or JavaScript.

Details

Fragment extraction. Tabular's HTML backend emits a full `<!DOCTYPE html>` document with a `<style>` block in the head and the table inside `<body>`. For inline embedding we extract the `<style>` and `<body>` content separately and re-wrap them in an `htmltools::tagList`:

```
<style>...table CSS...</style>
<div id="..." style="overflow-x:auto;max-width:100%;">
  ...table content...
</div>
```

The wrapping `<div>` gets a random unique id (so multiple tables on the same page have CSS-scopable hooks) and `overflow-x: auto` so wide tables get a horizontal scrollbar instead of overflowing their container.

Value

An `htmltools::tagList` containing a `<style>` block plus a wrapping `<div>` containing the table. Knitr, htmltools, and RStudio / Positron viewer panes all know how to render it.

See Also

Renders via: `print.tabular_spec`, `knit_print()`.

Terminal verb: `emit()`.

Examples

```
# `as.tags()` converts a spec into an htmltools tagList you can drop into
# a custom HTML page, a Shiny UI, or a Quarto / Rmd chunk. `print()` and
# `knit_print()` call it under the hood, so you seldom call it directly --
# but it is the seam for composing several tables into one container.
s1 <- tabular(cdisc_saf_demo, titles = "Demographics")
s2 <- tabular(cdisc_saf_ae, titles = "AE overall")

# Compose two tables into one parent tagList. Autoprinting `tables` in a
# Quarto / Rmd chunk renders both inline (via knit_print); embed it with
# htmltools::save_html() or a Shiny renderUI().
tables <- htmltools::tagList(
  htmltools::as.tags(s1),
  htmltools::as.tags(s2)
)

# The common path is autoprinting a spec: the viewer at an interactive
# prompt, an inline live table under pkgdown / knitr, and HTML source
# under R CMD check. This is the gt / flextable / tinytable convention --
# end on a bare table object and let the registered print method choose,
# with no browsable() / if (interactive()) wrapper, so R CMD check never
# launches a browser.
s1
```

as_grid

Resolve a tabular_spec into a tabular_grid

Description

Runs the full engine pipeline against spec and returns the resolved tabular_grid — the same intermediate object `emit()` hands to a backend. Pure function: no files written, no global state touched. Use this during development to inspect what `emit()` will pass downstream, when building a custom backend, or when piping the resolved grid into a non-file consumer (e.g. an inline preview chunk in a Quarto notebook).

Usage

```
as_grid(.spec)
```

Arguments

```
.spec The tabular_spec to resolve. <tabular_spec>: required. Built by the verb
chain (tabular() -> cols() -> headers() -> sort_rows() -> style() ->
paginate() -> preset()).
```

Details

Engine pipeline order is load-bearing. Phases run in this fixed order; the order matters because each phase reads the post- previous-phase state of the spec:

1. `engine_sort()` — apply the sort spec.
2. `engine_headers()` — validate the header tree and flatten it to a band grid.
3. `engine_style()` — evaluate every style predicate against the post-sort data grid. A predicate may reference any column in `spec@data`.
4. `engine_format()` — apply per-column formats, substitute `na_text`, and parse every cell / title / footnote / label through `parse_inline()` to its `inline_ast`.
5. `engine_decimal()` — column-wide decimal alignment for any column flagged `col_spec(aligned = "decimal")`. Operates on the formatted text; output is the same character matrix with NBSB padding inserted so the decimal marks line up.
6. `engine_paginate()` — split into pages (vertical row chunks + horizontal panel chunks). The plan drives the per-page slicing of cells / styles / ASTs below.

The grid is the backend contract. Every backend (`backend_md`, future `backend_html`, etc.) consumes a `tabular_grid` — never a `tabular_spec`. New backends only need to walk `grid@pages` and `grid@metadata`; the engine pipeline is a fixed dependency they never re-implement.

No I/O. `as_grid()` writes nothing to disk and touches no global state. It is safe to call repeatedly during interactive exploration; cost is roughly that of one `emit()` without the backend write step.

Value

A `tabular_grid` *S7 object*. Two slots:

- `@pages` — a list of one entry per display page. Each entry is a named list with pagination fields (`page_index`, `panel_index`, `is_continuation`, `continuation`, `show_titles`, `repeat_headers`, `show_footnotes_here`), row + column slice indices (`row_indices`, `col_indices`, `col_names`), the sliced cell text (`cells_text` — character matrix), sliced inline ASTs (`cells_ast` — list-matrix of `inline_ast`), sliced style nodes (`cells_style` — list-matrix of `style_node`), and the column-label ASTs for the visible columns (`col_labels_ast`).
- `@metadata` — per-table information backends consume once per render: `format` (the resolved backend tag, `NA_character_` for `as_grid()` calls), `rows_per_page`, `total_pages`, `total_panels`, `nrow_data`, `ncol_data`, `col_names`, `cols` (the original `col_spec()` entries keyed by column name), `headers` (the flattened header band grid), `titles`, `footnotes`, `titles_ast`, `footnotes_ast`, `col_labels_ast`, `pagehead_ast` / `pagefoot_ast` (resolved page-band content — NULL when the active preset declares no band, otherwise list(`left`, `center`, `right`) of length-N lists of `inline_ast` where N = row count and index 1 is the body-edge row).

See Also

I/O sibling: `emit()` writes the resolved grid to a file via a registered backend; `as_grid()` is the no-I/O entry into the same pipeline.

Build verbs the pipeline feeds from: `tabular()`, `cols()` / `col_spec()`, `headers()`, `sort_rows()`, `style()`, `paginate()`, `preset()`.

Inline formatting helpers: `md()`, `html()`.

Examples

```
# ---- Example 1: Demographics – inspect the resolved grid ----
#
# Resolve the canonical safety-pop demographics pipeline into a
# `tabular_grid` and inspect what `emit()` would hand a backend.
# The first page's `cells_text` matrix is the decimal-aligned
# output as the backend would render it; the metadata carries the
# pagination plan + header / title / footnote ASTs.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

demo <- tabular(
  cdisc_saf_demo,
  titles = c(
    "Table 14.1.1",
    "Demographics and Baseline Characteristics",
    "Safety Population"
  ),
  footnotes = "Source: ADSL."
) |>
cols(
  variable = col_spec(usage = "group", label = "Characteristic"),
  stat_label = col_spec(label = "Statistic"),
  placebo = col_spec(label = "Placebo\nN={n['placebo']}", align = "decimal"),
  drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}", align = "decimal"),
  drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}", align = "decimal"),
  Total = col_spec(label = "Total\nN={n['Total']}", align = "decimal")
) |>
sort_rows(by = c("variable", "stat_label"))

demo_grid <- as_grid(demo)
demo_grid@metadata$total_pages
demo_grid@pages[[1]]$cells_text[1:3, c("stat_label", "placebo")]

# ---- Example 2: AE-by-SOC/PT paginated grid – verify the split ----
#
# Same shape as Example 1 plus pagination protecting the SOC
# grouping. With a tight font size the grid carries multiple page
# entries; concatenating each page's `row_indices` reconstructs
# the full data, and every page carries the full header band grid
# at `grid@metadata$headers` so backends can re-render the header
# on every continuation page.
ae <- cdisc_saf_aesocpt
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))
ae$n_total <- as.integer(sub(".*", "", ae$Total))

ae_spec <- tabular(
  ae,
  titles = c(
    "Table 14.3.1",
    "Adverse Events by SOC and Preferred Term",
    "Safety Population"
  ),
)
```

```

footnotes = "Subjects counted once per SOC and once per PT."
) |>
cols(
  label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
  soc = col_spec(usage = "group", visible = FALSE,
    group_display = "column_repeat"),
  row_type = col_spec(visible = FALSE),
  soc_n = col_spec(visible = FALSE),
  n_total = col_spec(visible = FALSE),
  placebo = col_spec(label = "Placebo\nN={n['placebo']}", align = "decimal"),
  drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}", align = "decimal"),
  drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}", align = "decimal"),
  Total = col_spec(label = "Total\nN={n['Total']}", align = "decimal")
) |>
sort_rows(by = c("row_type", "n_total"), descending = c(FALSE, TRUE)) |>
paginate(keep_together = "soc")

ae_grid <- as_grid(ae_spec)
length(ae_grid@pages)

# ---- Example 3: Subgroup partition - one page set per group ----
#
# When `subgroup()` is attached, `as_grid()` runs the resolve
# pipeline once per group and concatenates the pages. `cdisc_saf_subgroup`
# carries `sex` as a natural partition axis; inspect
# `@pages[[i]]$subgroup_index` and `@pages[[i]]$subgroup_line_ast`
# to confirm each page knows its group identity and banner text.
# `sex` auto-hides as the partition `by` column; no explicit
# `col_spec(visible = FALSE)` needed.
sg_spec <- tabular(cdisc_saf_subgroup) |>
cols(
  agegr = col_spec(usage = "group", label = "Age Group"),
  sex_n = col_spec(visible = FALSE),
  agegr_n = col_spec(visible = FALSE),
  paramcd = col_spec(visible = FALSE),
  param = col_spec(usage = "group", label = "Parameter"),
  stat_label = col_spec(label = "Statistic")
) |>
subgroup("sex")

sg_grid <- as_grid(sg_spec)
length(sg_grid@pages)
vapply(sg_grid@pages, function(p) p$subgroup_index %||% NA_integer_, integer(1))

# ---- Example 4: Pre-flight inspection before emit() ----
#
# Resolve a spec to its grid without writing anywhere. Useful in
# tests, for snapshotting cell text under different presets, or
# for spec-introspection inside higher-level wrappers that need
# to know how many pages a render will produce.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
demog_spec <- tabular(
  cdisc_saf_demo,

```

```

  titles = "Demographics"
) |>
  cols(
    variable = col_spec(usage = "group", label = "Characteristic"),
    stat_label = col_spec(label = "Statistic"),
    placebo = col_spec(
      label = "Placebo\nN={n['placebo']}",
      align = "decimal"
    ),
    drug_50 = col_spec(
      label = "Drug 50\nN={n['drug_50']}",
      align = "decimal"
    ),
    drug_100 = col_spec(
      label = "Drug 100\nN={n['drug_100']}",
      align = "decimal"
    ),
    Total = col_spec(
      label = "Total\nN={n['Total']}",
      align = "decimal"
    )
  )
)
grid <- as_grid(demog_spec)
length(grid@pages)
dim(grid@pages[[1]]$cells_text)

```

brdr

*Border-line specification***Description**

Build a small immutable record describing one border line — width, style, and colour. A `brdr()` value is the stroke you hand to the `preset()` rules knob (one entry per rule name, e.g. `rules = list(midrule = brdr(width = 0.75))`) or to `style()`'s border arguments (`style(border_top = brdr(...), .at = cells_table(side = "rows"))`). Successive `preset()` calls layer cleanly, so a one-off override composes onto a house-style template without disturbing the other rules.

Usage

```
brdr(width = "thin", style = "solid", color = "ink")
```

```
is_brdr(x)
```

Arguments

width *Stroke width.* <numeric(1) | character(1)>: default "thin". Either a numeric in points (≥ 0)

style *Line style.* <character(1)>: default "solid". One of "solid", "dashed", "dotted", "double", "dashdot", "none".

color	<i>Stroke colour.</i> <character(1)>: default "currentColor"*. Hex ("#RRGGBB"), CSS colour name, "currentColor" to inherit the surrounding text colour.
x	<i>Any R object</i> — tested by <code>is_brdr()</code> for membership in the <code>tabular_brdr</code> S3 class.

Details

Surface. A single `tabular_brdr` value is a length-3 named list with class "tabular_brdr": `list(style, width, color)`. The shape is identical to the bare triple `style()`'s per-side scalars accept, so the resolver in `R/borders.R` can ingest either form transparently. Construct with `brdr()`; test with `is_brdr()`.

Width keywords. `width` accepts either a numeric in points (typical clinical values: 0.25, 0.5, 1, 1.5) or one of the four named keywords:

keyword	points
"hairline"	0.25
"thin"	0.5
"medium"	1
"thick"	1.5

Keywords resolve to numeric points immediately; the constructed value carries a numeric width. Numeric inputs pass through unchanged after a non-negative check.

Style enum. `style` is one of "solid" (default), "dashed", "dotted", "double", "dashdot", "none". "none" is the explicit clear-this-rule sentinel: setting a rule to `brdr(style = "none")` (or the bare string "none") in `preset(rules = list(...))` suppresses the baseline rule that backend would otherwise draw.

Color. Hex ("#212529"), CSS colour name ("black", "slategray"), the "ink" token (default; resolves to the primary rule ink #212529, decoupled from the surrounding text colour so a re-coloured header keeps a neutral rule), or "currentColor" (inherit the surrounding text colour per backend convention — `w:color="auto"` in DOCX, the document text colour in RTF, the CSS `currentColor` keyword in HTML).

Value

A `tabular_brdr` S3 object — a length-3 named list suitable for `preset(rules = list(<rule> = .))` or `style(border_* = .)`.

See Also

Where to attach: `preset()`'s `rules` knob (one `brdr()` per rule name) and `style()`'s `border_*` arguments.

Per-cell predicates: `style()` accepts the same per-side `border_<side>_{style,width,color}` triples without going through `brdr()`.

Resolver internals: `tabular_classes` (`style_node`'s 12 border scalars).

Examples

```

# ---- Example 1: A house-style rule set ----
#
# The `rules` knob takes one brdr() value per rule name. Here a
# thick column-label divider (midrule), a hairline dotted rule
# between body rows (rowrule), and the muted spanner rule dropped.
# Unlisted rules keep their booktabs defaults.
demo_n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
tabular(
  cdisc_saf_ae,
  titles = c(
    "Table 14.3.1",
    "Overall Summary of Adverse Events",
    "Safety Population"
  ),
  footnotes = "Subjects counted once per category."
) |>
cols(
  stat_label = col_spec(usage = "group", label = "Category"),
  placebo    = col_spec(label = "Placebo\nN={demo_n['placebo']}"),
  drug_50    = col_spec(label = "Drug 50\nN={demo_n['drug_50']}"),
  drug_100   = col_spec(label = "Drug 100\nN={demo_n['drug_100']}"),
  Total      = col_spec(label = "Total\nN={demo_n['Total']}")
) |>
preset(
  rules = list(
    midrule = brdr(width = "thick"),
    rowrule = brdr(width = "hairline", style = "dotted"),
    spanrule = "none"
  )
)

# ---- Example 2: Wrap a custom style into a reusable function ----
#
# The recommended way to share a rule style across many tables is to
# wrap the `preset()` call in a small function. A later `preset()` /
# `style()` call layers a one-off override cleanly on top.
custom_style <- function(spec) {
  spec |>
  preset(
    rules = list(
      toprule   = brdr(width = "thin", color = "#212529"),
      midrule   = brdr(width = "thin", color = "#212529"),
      bottomrule = brdr(width = "thin", color = "#212529")
    )
  )
}

tabular(cdisc_saf_n) |>
  custom_style() |>
  preset(rules = list(rowrule = brdr("hairline", "dashed")))

```

```

# ---- Example 3: Width keyword vs numeric, every style enum value ----
#
# Width accepts both the four named keywords and a bare numeric
# in points; style accepts six enum values. Use `is_brdr()` to
# confirm the constructor returned a valid `tabular_brdr` rather
# than a fallback list.
for (w in c("hairline", "thin", "medium", "thick")) {
  cat(w, "=", brdr(width = w)$width, "pt\n")
}
is_brdr(brdr(width = 0.75))

lapply(
  c("solid", "dashed", "dotted", "double", "dashdot", "none"),
  function(s) brdr(style = s)
)

# ---- Example 4: A full grid via the body-edge style() path ----
#
# The `rules` knob covers the named booktabs anatomy; for the body
# outer frame and inter-column separators, hand brdr() to
# `style(.at = cells_table(side = ...))`. Here a medium outer frame
# plus hairline column separators on a demographics table.
tabular(cdisc_saf_demo, titles = "Demographics with a full grid") |>
  cols(
    variable = col_spec(usage = "group", label = "Characteristic"),
    stat_label = col_spec(label = "Statistic"),
    placebo = col_spec(label = "Placebo", align = "decimal"),
    drug_50 = col_spec(label = "Drug 50", align = "decimal"),
    drug_100 = col_spec(label = "Drug 100", align = "decimal"),
    Total = col_spec(label = "Total", align = "decimal")
  ) |>
  style(border = brdr(width = "medium"), .at = cells_table(side = "outer")) |>
  style(border_left = brdr("hairline"), .at = cells_table(side = "cols"))

```

cdisc_eff_estimates *Treatment-effect estimates by model*

Description

Four competing efficacy models with their treatment-effect point estimate, 95% confidence-interval bounds, and nominal p-value. Shaped as a numeric-cell table (one row per model) rather than the usual pre-formatted character cells, so it exercises the `col_spec(format = ...)` + `col_spec(na_text = ...)` cascade. One row (MMRM) carries NA CI bounds to demonstrate `na_text`.

Usage

```
cdisc_eff_estimates
```

Format

A data frame with 4 rows and 5 columns:

model Model name ("ANCOVA", "MMRM", "Cox PH", "Bootstrap (1000 reps)").
 estimate Numeric point estimate.
 lower_ci, upper_ci Numeric 95% CI bounds. The MMRM row has NA bounds.
 p_value Nominal p-value (numeric).

Source

Synthetic estimates following the `_archive/.../arframe-examples/tables/tte-summary.qmd` and `efficacy-bor.qmd` shapes. Not derived from any patient-level data — illustrative values only.

See Also

`col_spec()` for the formatting cascade these values exercise.

Examples

```
# Numeric-cell efficacy table - format = "%.2f" pins precision,
# na_text = "--" renders the MMRM row's NA bounds as dashes.
tabular(cdisc_eff_estimates, titles = "Treatment-effect estimates by model") |>
  cols(
    model = col_spec(usage = "group", label = "Model", valign = "top"),
    estimate = col_spec(label = "Estimate", align = "decimal",
      format = "%.2f"),
    lower_ci = col_spec(label = "Lower\n95% CI", align = "decimal",
      format = "%.2f", na_text = "--"),
    upper_ci = col_spec(label = "Upper\n95% CI", align = "decimal",
      format = "%.2f", na_text = "--"),
    p_value = col_spec(label = "p-value", align = "decimal",
      format = "%.4f")
  )
```

cdisc_eff_n

Efficacy-population BigN per arm

Description

Per-arm subject counts (BigN) for the efficacy population used by `cdisc_eff_resp/eff_resp_card` — subjects with a BOR record in `pharmaverseadam::adrs_onco`. Same two-column naming convention as `cdisc_saf_n`; the totals differ from `cdisc_saf_n` because not every safety-pop subject contributes a best-overall-response record.

Usage

`cdisc_eff_n`

Format

A data frame with 4 rows and 3 columns; same schema as `cdisc_saf_n` (arm, arm_short, n).

Source

Derived in `data-raw/bundle-demo.R` from the per-arm BOR denominator computed inside the `cdisc_eff_resp` pipeline.

See Also

`cdisc_saf_n` for the safety-population counterpart.

Examples

```
# Efficacy BigN joined into column headers.
ne <- stats::setNames(cdisc_eff_n$n, cdisc_eff_n$arm_short)
col_spec(label = "Placebo\nN={ne['placebo']}")@label
```

cdisc_eff_resp

Best Overall Response and Response Rates

Description

Pre-summarised efficacy table. Per-arm counts of best overall response (BOR) per CDISC category, plus derived ORR, CBR, and DCR rate rows each followed by an exact (Clopper-Pearson) 95% CI row. Four sections (Best Overall Response, Objective Response Rate, Clinical Benefit Rate, Disease Control Rate) are encoded via the `groupid + group_label` pair so a single usage = "group" / `group_display = "header_row"` on `group_label` synthesises one bold section band per `groupid` block; the body rows render below each band via usage = "indent" on `stat_label`.

Usage

```
cdisc_eff_resp
```

Format

A data frame with 13 rows and 7 columns:

```
stat_label Row label ("CR", "PR", "SD", "NON-CR/NON-PD", "PD", "NE", "MISSING", "ORR (CR
+ PR)", "95% CI (Clopper-Pearson)", "CBR (CR + PR + SD)", "95% CI (Clopper-Pearson)",
"DCR (CR + PR + SD + NON-CR/NON-PD)", "95% CI (Clopper-Pearson)").
```

```
row_type "category" for BOR categorical rows, "derived" for ORR / CBR / DCR rate rows,
"ci" for the paired confidence-interval rows. Hide via col_spec(visible = FALSE).
```

```
placebo, drug_50, drug_100 Per-arm cell text ("n (pct)" on rate rows, "(lower, upper)" on CI
rows).
```

groupid Integer section id (1 = Best Overall Response, 2 = Objective Response Rate, 3 = Clinical Benefit Rate, 4 = Disease Control Rate). Hide via `col_spec(visible = FALSE)`; used as the section sort / partition key.

group_label Character section label, repeating across every row of its groupid block ("Best Overall Response" x7, "Objective Response Rate" x2, ...). Drives the engine's usage = "group" header_row synthesis when paired with `group_display = "header_row"`.

Source

Derived in `data-raw/bundle-demo.R` from `pharmaverseadam::adrs_onco` filtered to `PARAMCD == "BOR"`.

See Also

[cdisc_eff_n](#) for BigN denominators.

Examples

```
# 95% efficacy pattern: four bold section bands (Best Overall
# Response / Objective Response Rate / Clinical Benefit Rate /
# Disease Control Rate), each followed by indented stat rows. The
# source already ships in the right display order, so no sort step
# is needed; `group_label` repeats across every row of its section
# so the engine's `header_row` mode emits exactly one band per
# section.
ne <- stats::setNames(cdisc_eff_n$n, cdisc_eff_n$arm_short)
tabular(
  cdisc_eff_resp,
  titles = c(
    "Table 14.2.1",
    "Best Overall Response and Response Rates",
    "Efficacy Evaluable Population"
  )
) |>
cols(
  group_label = col_spec(usage = "group", group_display = "header_row"),
  stat_label = col_spec(usage = "indent", label = "Response"),
  groupid     = col_spec(visible = FALSE),
  row_type    = col_spec(visible = FALSE),
  placebo     = col_spec(
    label = "Placebo\nN={ne['placebo']}",
    align = "decimal"
  ),
  drug_50     = col_spec(
    label = "Drug 50\nN={ne['drug_50']}",
    align = "decimal"
  ),
  drug_100    = col_spec(
    label = "Drug 100\nN={ne['drug_100']}",
    align = "decimal"
  )
)
```

cdisc_saf_ae

*Overall adverse-event summary, Safety Population***Description**

Pre-summarised wide-format AE overview. Two clinical blocks: high-level flag rows (any TEAE, any SAE, any treatment-related, any AE leading to death, any AE recovered / resolved) and maximum-severity rows (mild / moderate / severe). Severity rows are indented with two leading spaces so a single `cols(stat_label = col_spec(usage = "group"))` declaration drives both the block-header rows and the indented detail rows.

Usage

cdisc_saf_ae

Format

A data frame with 8 rows and 5 columns:

`stat_label` Row label ("Any TEAE", "Any Serious AE (SAE)", "Any AE Related to Study Drug", "Any AE Leading to Death", "Any AE Recovered / Resolved", "Maximum severity: Mild", "Maximum severity: Moderate", "Maximum severity: Severe").

`placebo` Placebo arm cell text ("n (pct)").

`drug_50` Drug 50 arm cell text.

`drug_100` Drug 100 arm cell text.

`Total` Pooled-across-arms cell text.

Source

Derived in `data-raw/bundle-demo.R` from `pharmaverseadam: :adae` filtered to `SAFFL == "Y"` and `TRTEMFL == "Y"`.

See Also

[cdisc_saf_n](#) for BigN denominators; [cdisc_saf_aesocpt](#) for the SOC / PT detail companion.

Examples

```
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
tabular(
  cdisc_saf_ae,
  titles = c(
    "Table 14.3.0",
    "Adverse Event Overview",
    "Safety Population"
  )
) |>
```

```

cols(
  stat_label = col_spec(usage = "group", label = ""),
  placebo    = col_spec(
    label = "Placebo\nN={n['placebo']}",
    align = "decimal"
  ),
),
drug_50     = col_spec(
  label = "Drug 50\nN={n['drug_50']}",
  align = "decimal"
),
drug_100    = col_spec(
  label = "Drug 100\nN={n['drug_100']}",
  align = "decimal"
),
Total       = col_spec(
  label = "Total\nN={n['Total']}",
  align = "decimal"
)
)
)

```

cdisc_saf_aesocpt

Adverse events by System Organ Class and Preferred Term

Description

Pre-summarised AE-by-SOC/PT table. Interleaved row order: overall "any TEAE" row first, then per-SOC blocks where each SOC row is followed by its preferred-term detail rows. Top 10 SOCs and top 5 PTs per SOC are kept; row_type marks the role of each row and indent_level carries the canonical depth (0 for overall and SOC, 1 for PT) so the downstream pipeline drives the SOC -> PT indent via col_spec(indent_by = "indent_level") without reconstructing it in every script. The richer SOC x PT slice exercises [paginate\(\)](#) and the engine's horizontal-panel splitter end-to-end on a realistic submission shell.

Usage

```
cdisc_saf_aesocpt
```

Format

A data frame with 61 rows and 10 columns:

soc System Organ Class label. Repeats across the SOC's PT rows; hide via col_spec(visible = FALSE) once label carries the same SOC text on SOC rows.

label The row's display label. Equal to soc on the overall and SOC-summary rows; equal to the preferred-term name on PT detail rows. Promoted to the primary display column — pair with indent_by = "indent_level" to drive the SOC -> PT indent.

row_type One of "overall", "soc", "pt". Partition marker; hide via col_spec(visible = FALSE).

`indent_level` Integer depth (0 on overall and SOC rows, 1 on PT rows). Consumed by `col_spec(indent_by = "indent_level")` on the label column; the engine auto-hides this column at resolve time.

`n_total` Integer. The row's own subject count — overall TEAE count on the overall row, the SOC's count on each SOC row, the PT's count on each PT row. Inner sort key.

`soc_n` Integer. The parent SOC's count, broadcast to every row in that SOC's cluster (SOC row + its PT children) so a descending sort on `soc_n` keeps PTs grouped under their parent. On the overall row, equal to the overall TEAE count. Outer sort key.

`placebo` Placebo arm cell text ("n (pct)").

`drug_50, drug_100` Drug arms cell text.

`Total` Pooled-across-arms cell text.

Source

Derived in `data-raw/bundle-demo.R` from `pharmaverseadam: :adae`. Filtered to the top 10 SOC's by total incidence and the top 5 PTs per SOC. Body rows are pre-sorted with the cards-style two-level rule (`arrange(desc(soc_n), soc, desc(n_total))`) so the canonical render order is already baked in; the render-time `sort_rows(by = c("soc_n", "n_total"), descending = c(TRUE, TRUE))` reproduces it via stable sort.

See Also

[cdisc_saf_aesocpt_ard](#) for the hierarchical long ARD; [cdisc_saf_n](#) for BigN denominators.

Examples

```
# 95% safety pattern: SOC/PT table where `label` carries SOC text
# on SOC rows and PT text on PT rows, indented by `indent_level`.
# `soc` / `row_type` / `n_total` / `soc_n` ride along as hidden
# partition + sort keys. `sort_rows(soc_n, n_total)` clusters PTs
# under their parent SOC and orders both levels by descending count.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

tabular(
  cdisc_saf_aesocpt,
  titles = c(
    "Table 14.3.1",
    "Adverse Events by SOC and Preferred Term",
    "Safety Population"
  )
) |>
cols(
  label = col_spec(
    label = "SOC / PT",
    indent_by = "indent_level",
    align = "left"
  ),
  soc = col_spec(visible = FALSE),
  row_type = col_spec(visible = FALSE),
  n_total = col_spec(visible = FALSE),
  soc_n = col_spec(visible = FALSE),

```

```

  placebo = col_spec(
    label = "Placebo\nN={n['placebo']}",
    align = "decimal"
  ),
  drug_50 = col_spec(
    label = "Drug 50\nN={n['drug_50']}",
    align = "decimal"
  ),
  drug_100 = col_spec(
    label = "Drug 100\nN={n['drug_100']}",
    align = "decimal"
  ),
  Total = col_spec(
    label = "Total\nN={n['Total']}",
    align = "decimal"
  )
) |>
sort_rows(
  by = c("soc_n", "n_total"),
  descending = c(TRUE, TRUE)
)

```

cdisc_saf_aesocpt_ard *Cards hierarchical ARD for AEs by SOC and PT*

Description

Long-format companion to `cdisc_saf_aesocpt`. Produced by `cards::ard_stack_hierarchical()` over (AEBODSYS, AEDECOD) with `adsl`-level denominators, sorted by descending overall incidence via `cards::sort_ard_hierarchical()`. Limited to the same top-10 SOC, top-5 PT subset as `cdisc_saf_aesocpt` so the two datasets describe the same slice of the data.

Usage

```
cdisc_saf_aesocpt_ard
```

Format

A card-classed tibble. Carries an `..ard_hierarchical_overall..` sentinel row that `pivot_across()` passes through as the table's "overall" row.

Details

This is the package's canonical **hierarchical ARD** demo (two grouping variables nested SOC -> PT). Its flat counterpart is `cdisc_saf_demo_ard`; together they cover both shapes `pivot_across()` must handle.

Source

Derived in data-raw/bundle-demo.R via `cards::ard_stack_hierarchical()` over `pharmaverseadam::adae` filtered to the top SOC / PT subset.

See Also

[pivot_across\(\)](#) for the long-to-wide bridge; [cdisc_saf_aesocpt](#) for the wide companion.

Examples

```
# Hierarchical ARD pivot. pivot_across() recognises the
# ard_stack_hierarchical shape and emits soc / label / row_type.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
cdisc_saf_aesocpt_ard |>
  pivot_across(statistic = "{n} ({{p}})") |>
  tabular(
    titles = c(
      "Table 14.3.1",
      "Adverse Events by SOC and PT",
      "Safety Population"
    )
  ) |>
  cols(
    label = col_spec(label = "SOC / PT", align = "left"),
    soc = col_spec(visible = FALSE),
    row_type = col_spec(visible = FALSE),
    `Placebo` = col_spec(align = "decimal"),
    `Xanomeline Low Dose` = col_spec(align = "decimal"),
    `Xanomeline High Dose` = col_spec(align = "decimal")
  )
```

cdisc_saf_demo

Demographics summary, Safety Population

Description

Pre-summarised wide-format demographics suitable for direct passing into `tabular()`. One row per displayed statistic. Eight parameter blocks:

Usage

```
cdisc_saf_demo
```

Format

A data frame with 35 rows and 6 columns:

variable Display-block label ("Age (years)", "Age Group, n (%)", "Sex, n (%)", "Race, n (%)", "Ethnicity, n (%)"). Driven by `cols(usage = "group")` to collapse repeat values at render.

stat_label Statistic or level label ("n", "Mean (SD)", "Median", "M", "WHITE", ...).

placebo Placebo arm cell text.

drug_50 Xanomeline Low Dose (50 mg) arm cell text.

drug_100 Xanomeline High Dose (100 mg) arm cell text.

Total Pooled-across-arms cell text.

Details

- continuous: Age (years), Weight (kg), Height (cm), BMI (kg/m²) — each emitted as n, Mean (SD), Median, Q1, Q3, Min, Max
- categorical: Age Group, Sex, Race, Ethnicity, BMI Category — each level rendered as n (%)

Shaped for the display-only contract: every cell is the final string that will appear in the rendered table.

Source

Derived in data-raw/bundle-demo.R from pharmaverseadam: :adsl filtered to SAFFL == "Y" and the three CDISCPILLOT01 treatment arms. Baseline Weight / Height / BMI are joined in from pharmaverseadam: :advs.

See Also

[cdisc_saf_demo_ard](#) for the long-format ARD companion; [cdisc_saf_n](#) for the matching BigN denominators.

Examples

```
# 95% safety pattern: demographics table with BigN-embedded
# column labels and CDISC-canonical statistic order.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
tabular(
  cdisc_saf_demo,
  titles = c(
    "Table 14.1.1",
    "Demographics and Baseline Characteristics",
    "Safety Population"
  )
) |>
cols(
  variable = col_spec(usage = "group", label = "Parameter"),
  stat_label = col_spec(label = "Statistic"),
  placebo = col_spec(
    label = "Placebo\nN={n['placebo']}",
    align = "decimal"
  ),
  drug_50 = col_spec(
    label = "Drug 50\nN={n['drug_50']}",
    align = "decimal"
  )
)
```

```

),
drug_100 = col_spec(
  label = "Drug 100\nN={n['drug_100']}",
  align = "decimal"
),
Total    = col_spec(
  label = "Total\nN={n['Total']}",
  align = "decimal"
)
)
)

```

cdisc_saf_demo_ard *Cards ARD for demographics (flat ARD companion)*

Description

The same demographics summary as `cdisc_saf_demo`, but in the long Analysis Results Data (ARD) format produced by `cards::ard_stack()`. One row per (treatment arm, variable, statistic). Shipped as a teaching dataset that shows the upstream shape users typically have when they start from `cards`. Convert it to the wide form `tabular()` accepts via `pivot_across()` — `tabular` itself does **not** consume the long ARD format, since pre-summarised wide data is the package boundary.

Usage

```
cdisc_saf_demo_ard
```

Format

A card-classed tibble with columns `group1`, `group1_level`, `variable`, `variable_level`, `context`, `stat_name`, `stat_label`, `stat`. `group1 == "TRT01A"` and `group1_level` carries the original pharmaverse/adam arm labels ("Placebo", "Xanomeline Low Dose", "Xanomeline High Dose"). `cards::ard_stack(.overall = TRUE)` adds overall rows with `group1_level = NA`; `pivot_across()` renders those into a `Total` column.

Details

Continuous variables: AGE, WEIGHT, HEIGHT, BMI (each emitting N, mean, sd, median, p25, p75, min, max). Categorical variables: AGEGR1, SEX, RACE, ETHNIC, BMI_CAT (each emitting n, N, p).

This is the package's canonical **flat ARD** demo. Its hierarchical counterpart is `cdisc_saf_aesocpt_ard`; together they cover both shapes `pivot_across()` must handle.

Source

Derived in `data-raw/bundle-demo.R` via `cards::ard_stack(.by = "TRT01A", .overall = TRUE)` over `pharmaverse/adam::adsl`.

See Also

`pivot_across()` for the long-to-wide bridge; [cdisc_saf_demo](#) for the wide companion.

Examples

```
# 95% demographics pattern: cards ARD -> wide -> rendered table.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
cdisc_saf_demo_ard |>
  pivot_across(
    statistic = list(
      continuous = "{mean} ({sd})",
      categorical = "{n} ({p}%)"
    ),
    label = c(AGE = "Age (years)", SEX = "Sex", RACE = "Race")
  ) |>
  tabular(
    titles = c(
      "Table 14.1.1",
      "Demographics",
      "Safety Population"
    )
  )
)
```

cdisc_saf_n

Safety-population BigN per arm

Description

Per-arm subject counts (BigN) for the safety population, plus a Total row. Use this table to embed BigN inline in column headers with a glue-style `{expr}` template against `cols(col_spec(label = ...))`; there is no dedicated BigN field on `col_spec` because the denominator already lives here in a discoverable, joinable form.

Usage

```
cdisc_saf_n
```

Format

A data frame with 4 rows and 3 columns:

`arm` Raw pharmaverseadam arm label ("Placebo", "Xanomeline Low Dose", "Xanomeline High Dose", "Total"). Matches `group1_level` in the `_card` ARDs (so the pivot output's column names match a `setNames(cdisc_saf_n$n, cdisc_saf_n$arm)` lookup).

`arm_short` Renamed label ("placebo", "drug_50", "drug_100", "Total"). Matches the column names of `cdisc_saf_demo`, `cdisc_saf_ae`, `cdisc_saf_aesocpt`, and `cdisc_saf_vital`.

`n` Integer subject count.

Details

Two arm-naming columns are shipped side by side so the same table can serve both the `_card` ARDs (raw `pharmaverseadam` labels in `group1_level`) and the renamed wide datasets (snake-cased arm column names).

Source

Derived in `data-raw/bundle-demo.R` from `pharmaverseadam: :adsl` filtered to `SAFFL == "Y"` and the three `CDISCPILLOT01` arms.

See Also

[cdisc_eff_n](#) for the efficacy-population counterpart.

Examples

```
# Use cdisc_saf_n$arm_short when joining into the wide datasets
# (cdisc_saf_demo, cdisc_saf_ae, cdisc_saf_aesocpt, cdisc_saf_vital).
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
col_spec(label = "Placebo\nN={n['placebo']}")@label

# Use cdisc_saf_n$arm when joining into pivot_across() output
# (column names match the raw pharmaverseadam arm labels).
n_arm <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm)
col_spec(label = "Placebo\nN={n_arm['Placebo']}")@label
```

cdisc_saf_subgroup	<i>Vital-signs subgroup summary by Sex and Age Group</i>
--------------------	--

Description

Pre-summarised vital-signs stats partitioned by sex (F / M) and age group (<65 / >=65) at the End-of-Treatment visit. Two parameters (Systolic BP, Diastolic BP) emit four statistic rows each (n, Mean (SD), Median, Min, Max). Partition-constant `BigN` columns (`sex_n`, `agegr_n`) ride alongside so banners can inline the denominator via `subgroup(label = "Sex: {sex} (N = {sex_n})")` without reaching for a separate lookup.

Usage

```
cdisc_saf_subgroup
```

Format

A data frame with 32 rows and 11 columns:

```
sex Factor (F / M).
agegr Factor (<65 / >=65).
```

`sex_n` Integer BigN — number of subjects in the partition row's sex (partition-constant; rides into the banner via `{sex_n}` template tokens).

`agegr_n` Integer BigN per age group.

`paramcd` CDISC parameter code (SYSBP / DIABP).

`param` Decoded parameter name ("Systolic BP (mmHg)", "Diastolic BP (mmHg)").

`stat_label` Statistic label (n, Mean (SD), Median, Min, Max).

`placebo, drug_50, drug_100, Total` Per-arm cell text.

Details

Designed for `subgroup()` and `as_grid()` examples: the two partition axes plus the partition-constant BigN columns cover both single-variable cohort-style partitions and the multi-variable (sex × agegr) crossing.

Source

Derived in `data-raw/bundle-demo.R` from `pharmaverseadam::advs` filtered to `SAFFL == "Y"`, the three CDISCPLOT01 arms, the SYSBP / DIABP parameters, and the End-of-Treatment visit.

See Also

`cdisc_saf_n` for BigN denominators; `subgroup()` for the verb this dataset is designed for.

Examples

```
# 95% pattern: subgroup partition by sex with inline BigN.
# `sex` and `sex_n` auto-hide from the body: `sex` because it is
# the partition `by` column; `sex_n` because the banner template
# references it. No explicit `col_spec(visible = FALSE)` needed.
tabular(cdisc_saf_subgroup, titles = "Vital Signs at End of Treatment") |>
  cols(
    agegr      = col_spec(usage = "group", label = "Age Group"),
    agegr_n    = col_spec(visible = FALSE),
    paramcd    = col_spec(visible = FALSE),
    param      = col_spec(usage = "group", label = "Parameter"),
    stat_label = col_spec(label = "Statistic"),
    placebo    = col_spec(label = "Placebo", align = "decimal"),
    drug_50    = col_spec(label = "Drug 50", align = "decimal"),
    drug_100   = col_spec(label = "Drug 100", align = "decimal"),
    Total      = col_spec(label = "Total", align = "decimal")
  ) |>
  subgroup(by = "sex", label = "Sex: {sex} (N = {sex_n})")
```

cdisc_saf_vital	<i>Vital-signs summary</i>
-----------------	----------------------------

Description

Pre-summarised vital-signs stats. Four parameters (SYSBP, DIABP, PULSE, TEMP) at four visits (Baseline, Week 8, Week 16, End of Treatment), each producing four statistic rows (n, Mean (SD), Median, Min, Max). The 4 x 4 x 4 grid makes this dataset a natural fit for `paginate()` examples — 64 rows comfortably exceed a single page under typical clinical row-per-page settings.

Usage

```
cdisc_saf_vital
```

Format

A data frame with 64 rows and 7 columns:

`paramcd` CDISC parameter code (SYSBP / DIABP / PULSE / TEMP). Repeats across visit and statistic; use `col_spec(usage = "group")` to collapse.

`param` Decoded parameter name.

`visit` Analysis visit label ("Baseline" / "Week 8" / "Week 16" / "End of Treatment").

`stat_label` Statistic label.

`placebo, drug_50, drug_100` Per-arm cell text.

Source

Derived in `data-raw/bundle-demo.R` from `pharmaverseadam:adv`.

See Also

[cdisc_saf_n](#) for BigN denominators.

Examples

```
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
tabular(
  cdisc_saf_vital,
  titles = c(
    "Table 14.4.1",
    "Vital Signs Summary at Baseline and End of Treatment",
    "Safety Population"
  )
) |>
cols(
  paramcd = col_spec(visible = FALSE),
  param   = col_spec(usage = "group", label = "Parameter"),
  visit   = col_spec(usage = "group", label = "Visit"),
```

```

stat_label = col_spec(label = "Statistic"),
placebo     = col_spec(
  label = "Placebo\nN={n['placebo']}",
  align = "decimal"
),
drug_50     = col_spec(
  label = "Drug 50\nN={n['drug_50']}",
  align = "decimal"
),
drug_100    = col_spec(
  label = "Drug 100\nN={n['drug_100']}",
  align = "decimal"
)
)
)

```

cells

Cell-location constructors for style()

Description

Build a `tabular_location` value naming one region of the rendered table; pass the result to `style()`'s `at` argument. Each constructor targets one surface (body, headers, footnotes, ...); optional `i / j / where / level / labels` filters narrow the target within that surface.

Usage

```
cells_body(i = NULL, j = NULL, where = NULL)
```

```
cells_headers(level = NULL, labels = NULL, j = NULL)
```

```
cells_group_headers(j = NULL, where = NULL)
```

```
cells_title()
```

```
cells_subgroup_labels()
```

```
cells_footnotes()
```

```
cells_pagehead(slot = NULL)
```

```
cells_pagefoot(slot = NULL)
```

```
cells_table(side = NULL, i = NULL, j = NULL)
```

```
is_tabular_location(x)
```

Arguments

<code>i</code>	<i>Row index filter.</i> <integer logical character NULL>. Integer = 1-based row numbers; logical = length-nrow mask (broadcasts from scalar TRUE/FALSE); character = matches the visible row labels. NULL (default) = no filter (every row).
<code>j</code>	<i>Column index filter.</i> <integer character NULL>. Integer = 1-based column positions; character = matches column names in <code>spec@data</code> . NULL (default) = every column.
<code>where</code>	<i>Predicate.</i> An unquoted expression evaluating to a length-nrow logical vector when run against the data grid. Captured as an rlang quosure (so <code>pvalue < 0.05</code> works without needing to wrap in <code>vars()</code> or similar). Mutually exclusive with <code>i</code> .
<code>level</code>	<i>Header-band depth (for <code>cells_headers</code>).</i> <integer(1) NULL>. 1 = topmost spanner band; increasing integers walk toward the leaves. -1 = the leaf band (per-column labels built from <code>col_spec@label</code>). NULL (default) = every band at every depth.
<code>labels</code>	<i>Header-band labels (for <code>cells_headers</code>).</i> <character NULL>. Targets header_node(s) whose <code>@label</code> matches, at any depth. Mutually exclusive with <code>level</code> .
<code>slot</code>	<i>Band slot (for <code>cells_pagehead</code> / <code>cells_pagefoot</code>).</i> <character(1) NULL>. One of "left", "center", "right", or NULL for every slot.
<code>side</code>	<i>Table edge / separator (for <code>cells_table</code>).</i> <character(1) NULL>. One of "outer" (all four outer edges), "outer_top", "outer_bottom", "outer_left", "outer_right", "rows" (horizontal separator between body rows), "cols" (vertical separator between body columns), or NULL for whole-body (same as <code>cells_body()</code>).
<code>x</code>	<i>Any R object</i> — tested by <code>is_tabular_location()</code> for membership in the <code>tabular_location</code> S3 class.

Details

One surface per location. A `tabular_location` always names exactly one of: `body`, `headers`, `group_headers`, `title`, `subgroup_labels`, `footnotes`, `pagehead`, `pagefoot`, `table`. Cross-surface styling layers in via multiple chained `style()` calls (one per location).

Index vocabulary. Where supported, the `i` (rows) and `j` (columns) arguments accept integer, logical, or character vectors — matching the convention established by `flextable` (`bold(ft, i, j)`) and `tinytable` (`style_tt(i, j)`). Character vectors match against the data frame's column names (`j`) or row labels (`i`); integers are 1-based positions; logicals broadcast to `nrow` / `ncol`.

Predicate vocabulary. `cells_body(where = pvalue < 0.05)` is the canonical data-driven filter — `where` is captured as an rlang quosure and evaluated at engine time against the post-sort grid. Mutually exclusive with `i` (you target *either* by index *or* by predicate, not both).

Why `cells_headers` not `cells_column_spanners`. The verb that builds the multi-level header tree is named `headers()`. The location follows the same vocabulary: one word ("headers") covers the entire column-header section — inner spanner bands AND the leaf band of per-column labels. Pass `level` or `labels` to narrow.

Value

A tabular_location *S3 list* with slots surface, i, j, where, labels, level, slot, side (unused slots are NULL). Pass to `style()`'s at argument.

Surface filters

constructor	filters
<code>cells_body(i, j, where)</code>	row index / col index / predicate
<code>cells_headers(level, labels, j)</code>	band depth / spanner label / cols
<code>cells_group_headers(j, where)</code>	injected section rows
<code>cells_title()</code>	(no filter — whole block)
<code>cells_subgroup_labels()</code>	(no filter)
<code>cells_footnotes()</code>	(no filter)
<code>cells_pagehead(slot)</code>	"left" / "center" / "right"
<code>cells_pagefoot(slot)</code>	"left" / "center" / "right"
<code>cells_table(side, i, j)</code>	outer edge / row separator / etc.

See Also

Verb that consumes locations: `style()`.

Border value type: `brdr()`.

Reusable house style: `style_template()`.

Examples

```
# Whole body cells (the default for style())
cells_body()

# Row index 1:3, column "Total"
cells_body(i = 1:3, j = "Total")

# Data-driven subset
cells_body(where = stat_label == "Mean (SD)")

# Topmost spanner band only
cells_headers(level = 1)

# Leaf band (per-column labels)
cells_headers(level = -1)

# A specific spanner by label
cells_headers(labels = "Treatment Group")

# Section-header rows for col_spec(group_display = "header_row")
cells_group_headers()
```

```

# Title / footnotes blocks
cells_title()
cells_footnotes()

# Page-header / page-footer slots
cells_pagehead(slot = "left")
cells_pagefoot(slot = "right")

# Outer table frame
cells_table(side = "outer")

# Horizontal rules between body rows
cells_table(side = "rows")

```

check_fonts

Check font availability across backends

Description

Walks the resolved font fallback chain for each backend and reports which entries the local machine can find. Useful for answering "is the preview I'm seeing the same fonts the downstream reviewer will see?".

Usage

```
check_fonts(.spec)
```

Arguments

`.spec` A `tabular_spec` or `preset_spec`. `<tabular_spec | preset_spec>`: required. The spec whose effective preset determines which font chain to walk.

Details

The diagnostic does NOT change what `emit()` writes to the file. Tabular's backends emit font *names* (CSS strings, LaTeX `\setmainfont` commands, RTF font-table entries); the consuming application (browser, LaTeX engine, Word, Adobe Reader) on the opening machine resolves those names against its own installed fonts. `check_fonts()` is purely informational — it tells you which entries of the cross-platform fallback chain you can see on this machine, so you can predict drift.

Status markers:

- v — font is installed on this machine (via `systemfonts`).
- o — font is a CSS / LaTeX generic; always resolvable by the consuming application.
- x — font is not installed on this machine; the consuming app on a different machine may or may not have it.

Requires the `systemfonts` package (in `Suggests`); call `install.packages("systemfonts")` first if it isn't installed.

Value

Invisibly returns the resolved per-backend chains as a named list of character vectors. Side effect: prints a cli tree showing the availability marker for every entry.

See Also

Builds the spec: [tabular\(\)](#), [preset\(\)](#).

Resolves the spec: [as_grid\(\)](#), [emit\(\)](#).

Examples

```
# ---- Example 1: Inspect default font resolution ----
#
# Build a spec with the default font_family ("mono") and ask
# which entries in the cross-platform chain are findable
# locally. Useful before sharing a render with downstream
# reviewers who may be on a different OS.
spec <- tabular(
  cdisc_saf_demo,
  titles = "Demographics"
)
if (requireNamespace("systemfonts", quietly = TRUE)) {
  check_fonts(spec)
}

# ---- Example 2: Diagnose a Courier New request ----
#
# A request for "Courier New" (a specific named font) renders
# on macOS / Windows but may fall back to a serif on Linux.
# `check_fonts()` flags this so the user knows to switch to
# the "mono" generic for portable output.
spec_mono <- tabular(
  cdisc_saf_demo,
  titles = "Mono request"
) |>
  preset(font_family = "Courier New")
if (requireNamespace("systemfonts", quietly = TRUE)) {
  check_fonts(spec_mono)
}

# ---- Example 3: Explicit cross-platform stack ----
#
# A length>1 input is treated as an explicit fallback chain and
# emitted verbatim – no alias lookup, no fabrication. Use this
# when the first choice is a sponsor / brand face that needs an
# honest fallback for reviewers who don't have it installed.
spec_brand <- tabular(cdisc_saf_demo) |>
  preset(font_family = c("Inter", "Liberation Sans", "Arial", "sans"))
if (requireNamespace("systemfonts", quietly = TRUE)) {
  check_fonts(spec_brand)
}
```

```

# ---- Example 4: Compare serif vs sans fallback chains ----
#
# Side-by-side check of the two generic families. Useful when
# deciding the house-style default: the serif chain leads with
# Liberation Serif (Linux-server-first); the sans chain leads
# with Liberation Sans. Both close with the backend's native
# fallback layer (CSS generic on HTML, Latin Modern on LaTeX).
if (requireNamespace("systemfonts", quietly = TRUE)) {
  tabular(cdisc_saf_demo) |>
    preset(font_family = "serif") |>
    check_fonts()

  tabular(cdisc_saf_demo) |>
    preset(font_family = "sans") |>
    check_fonts()
}

```

check_latex

Check LaTeX-package availability for PDF output

Description

Reports, for every TeX package the LaTeX / PDF backend can emit, whether it is present in the local TeX tree, and prints the exact `tinytex::tlmgr_install()` call that installs any that are missing. Run this before `emit(spec, "out.pdf")` on a fresh machine to turn a cryptic mid-compile File 'tabularray.sty' not found into an up-front, actionable checklist.

Usage

```
check_latex(quiet = FALSE)
```

Arguments

quiet	<i>Suppress the printed cli report.</i> <logical(1)>: default FALSE. When TRUE, runs the checks and returns the result invisibly without printing. Use in scripts that branch on the return value.
-------	--

Details

The required set is a superset of every `\usepackage{}` / `\UseTblrLibrary{}` directive the backend emits, across all conditional branches (running headers / footers pull fancyhdr + lastpage; xelatex pulls fontspec; pdflatex pulls the classic font bundles). The check is informational, it does not install anything.

OS-managed TeX Live gotcha. On Linux distributions that ship TeX Live through the system package manager (RHEL / Fedora via dnf, Debian / Ubuntu via apt), tlmgr is locked against user installs and `tlmgr_install()` will fail. The fix is to install a user-space TinyTeX with

`tinytex::install_tinytex()` and let that tree own the packages. Never force a locked `tlmgr` with `--ignore-warning`: it leaves the system tree half-written.

Slow / stuck install (often Windows). The default CTAN repository `mirror.ctan.org` redirects to a random mirror on every call, and a slow or stale one makes `tinytex::tlmgr_install()` appear to hang. Pin a concrete mirror once with `tinytex::tlmgr_repo("auto")` (it follows the redirect a single time and remembers the result), then retry the install.

Status markers:

- v — package is installed in the local TeX tree.
- x — package is missing; the `tlmgr_install()` line at the bottom of the report installs every missing package at once.
- ? — availability could not be determined (no `tinytex`, or `tlmgr` not reachable); treated as missing for remediation.

Requires the `tinytex` package (in `Suggests`); call `install.packages("tinytex")` first if it isn't installed.

Value

Invisibly returns a data frame with one row per required package and columns `package` (<character>) and `installed` (<logical>, NA when undeterminable). Side effect: prints a cli report with a per-package status marker and, when anything is missing, the exact `tlmgr_install()` remedy.

See Also

Companion diagnostic: `check_fonts()`.

Consumes the result: `emit()`.

Examples

```
# ---- Example 1: Audit the PDF toolchain before emitting ----
#
# Run check_latex() on a fresh machine to confirm every LaTeX
# package the PDF backend needs is present. The call prints a
# status line per package and, if any are missing, the exact
# tinytex::tlmgr_install() command to fix them in one shot. It is
# guarded on tinytex so it is a no-op where TeX is unavailable.
if (requireNamespace("tinytex", quietly = TRUE)) {
  check_latex()
}
```

cols *Attach per-column specifications*

Description

Add `col_spec()` entries to a `tabular_spec`. Each named argument is one column: the name is the input column in `.spec@data` and the value is the `col_spec` carrying that column's display attributes (usage, label, format, alignment, width, visibility, NA text). Columns not mentioned get a default `col_spec()` (usage = display) at engine-validate time.

Usage

```
cols(.spec, ..., .default = NULL)
```

Arguments

- | | |
|-----------------------|--|
| <code>.spec</code> | <i>The tabular_spec to extend.</i> <code><tabular_spec></code> : required. Dot-prefixed so R's partial argument matching cannot accidentally bind a short user-supplied name (e.g. <code>s</code> , <code>sp</code>) in <code>...</code> to the <code>spec</code> slot. Pipe input (<code>tabular(...)</code> <code> ></code> <code>cols(...)</code>) works the normal way — the <code>spec</code> is supplied positionally. |
| <code>...</code> | <i>Named col_spec objects, one per column.</i> Each name is the input column name in <code>.spec@data</code> . Names must match an existing column — pre-compute derived columns upstream with <code>dplyr::mutate()</code> (or equivalent) before <code>tabular()</code> .
Restriction: Names must be unique within a single <code>cols()</code> call (duplicates warn; "last value wins"). Tip: To override an attribute already declared, use a second <code>cols()</code> call downstream and let the merge rule apply. |
| <code>.default</code> | <i>Fallback col_spec for unmentioned columns.</i> <code><col_spec NULL></code> : default <code>NULL</code> . When a <code>col_spec</code> , it is field-merged onto every data column that is NOT named in <code>...</code> and does not already carry a <code>spec</code> from an earlier <code>cols()</code> call. <code>NULL</code> (default) leaves unmentioned columns to the engine-time default. Use it to set one alignment / format across a variable number of arm columns in a single call.
Interaction: Explicit <code>...</code> specs always win — <code>.default</code> only fills the gaps. A column carried over from a prior <code>cols()</code> call is treated as already specified and is left untouched. |
- ```
Decimal-align every arm column without listing each by name.
tabular(cdisc_saf_demo) |>
 cols(
 variable = col_spec(usage = "group", label = "Parameter"),
 stat_label = col_spec(label = "Statistic"),
 .default = col_spec(align = "decimal")
)
```

## Details

**Sparse declaration.** Declare only the columns whose attributes differ from the default — a typical pipeline uses one `cols()` call with one entry per non-default column.

**Within-call duplicates warn.** A duplicate name inside one `cols()` call warns and "last value wins". To intentionally override an attribute, use a second `cols()` call downstream and let the merge rule below apply.

## Value

The updated `tabular_spec`. Continue chaining with [headers\(\)](#), [sort\\_rows\(\)](#), [style\(\)](#).

## Repeat-call merge semantics

When `cols()` is called more than once for the same column, the engine merges the new `col_spec` into the existing one field-by-field. A non-default value on the new spec overrides; a default-valued field leaves the existing field intact. This lets you build a column's spec in stages — declare the label-and-alignment block up front, add the width once you know it fits, then attach a sort key, all without re-stating earlier attributes. Essential when generating specs programmatically (looping over arms, layering a house-style helper).

Default values that do NOT override the existing field:

| field   | default that does not override |
|---------|--------------------------------|
| usage   | NA_character_                  |
| label   | NA_character_                  |
| format  | NULL                           |
| visible | TRUE                           |
| width   | NA_real_                       |
| align   | NA_character_                  |
| na_text | NA_character_ (inherit preset) |

```
Three-stage build: label/usage first, alignment second, width
third. Each stage leaves earlier fields intact.
tabular(cdisc_saf_demo) |>
 cols(variable = col_spec(usage = "group", label = "Parameter")) |>
 cols(variable = col_spec(align = "left")) |>
 cols(variable = col_spec(width = 2.0))
Result: variable has usage="group", label="Parameter",
align="left", width=2.0 – all four fields set.
```

## See Also

**Companion constructor:** [col\\_spec\(\)](#) builds the per-column DSL object that `cols()` attaches.

**Sibling build verbs:** [headers\(\)](#), [sort\\_rows\(\)](#), [style\(\)](#), [paginate\(\)](#), [preset\(\)](#).

**Entry / terminal verbs:** [tabular\(\)](#), [emit\(\)](#), [as\\_grid\(\)](#).

## Examples

```

---- Example 1: Demographics with arm BigN inline in headers ----
#
Demographics table where the row-label columns sit on the left
and the four treatment-arm columns embed BigN in the header
label (drawn inline from the bundled `cdisc_saf_n` data frame). Every
arm column is decimal-aligned so mixed-format cells like
"5 (3.2%)" line up on the decimal mark.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

tabular(
 cdisc_saf_demo,
 titles = c(
 "Table 14.1.1",
 "Demographics and Baseline Characteristics",
 "Safety Population"
),
 footnotes = "Percentages based on N per treatment group."
) |>
cols(
 variable = col_spec(usage = "group", label = "Parameter"),
 stat_label = col_spec(label = "Statistic"),
 placebo = col_spec(label = "Placebo\nN={n['placebo']}", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}", align = "decimal"),
 Total = col_spec(label = "Total\nN={n['Total']}", align = "decimal")
) |>
sort_rows(by = c("variable", "stat_label"))

---- Example 2: BOR table with CDISC factor ordering and hidden helper ----
#
Best Overall Response table where `stat_label` carries the
canonical CDISC factor levels (driving the sort) and `row_type`
is hidden – present in the data for the sort, absent from the
rendered output via `col_spec(visible = FALSE)`.
bor_levels <- c(
 "CR", "PR", "SD", "NON-CR/NON-PD", "PD", "NE", "MISSING",
 "ORR (CR + PR)", "CBR (CR + PR + SD)",
 "DCR (CR + PR + SD + NON-CR/NON-PD)", "95% CI (Clopper-Pearson)"
)
eff <- cdisc_eff_resp
eff$stat_label <- factor(eff$stat_label, levels = bor_levels)
ne <- stats::setNames(cdisc_eff_n$n, cdisc_eff_n$arm_short)

tabular(
 eff,
 titles = c(
 "Table 14.2.1",
 "Best Overall Response and Response Rates",
 "Efficacy Evaluable Population"
),
 footnotes = "Response per RECIST 1.1, investigator assessment."
)

```

```

) |>
 cols(
 stat_label = col_spec(label = "Response"),
 row_type = col_spec(visible = FALSE),
 groupid = col_spec(visible = FALSE),
 group_label = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={ne['placebo']}", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50\nN={ne['drug_50']}", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100\nN={ne['drug_100']}", align = "decimal")
) |>
 sort_rows(by = c("groupid", "stat_label"))

---- Example 3: AE-by-SOC/PT with indented label + repeat-call merge ----
#
`label` carries SOC text on SOC rows and PT text on PT rows;
`indent_by = "indent_level"` indents the PT rows one level under
their SOC. `soc`, `row_type`, and `n_total` ride along as hidden
sort keys. A second `cols()` call later in the chain adds widths
once the user knows the page geometry; the repeat-call merge
preserves prior attributes (label, indent_by, align, visible)
without restating them.
ae <- cdisc_saf_aesocpt
ae$n_total <- as.integer(sub(".*", "", ae$Total))
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))

tabular(
 ae,
 titles = c("Table 14.3.1", "Adverse Events by SOC and Preferred Term")
) |>
 cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal"),
 Total = col_spec(label = "Total", align = "decimal")
) |>
 sort_rows(by = c("row_type", "n_total"), descending = c(FALSE, TRUE)) |>
 # Second `cols()` call: add widths after the rest of the spec
 # is built. Repeat-call merge preserves prior attributes.
 cols(
 label = col_spec(width = "2.5in"),
 placebo = col_spec(width = "0.9in"),
 drug_50 = col_spec(width = "0.9in"),
 drug_100 = col_spec(width = "0.9in"),
 Total = col_spec(width = "0.9in")
)

---- Example 4: Compact AE-overall with pre-derived Active column ----
#

```

```

Drop the per-arm columns and surface only the Total. Pre-compute
the pooled "Active" column upstream (here `paste0(drug_50, " / ",
drug_100)` before piping into `tabular()`; `cols()` then just
declares each column's display role. The same pattern handles
any post-pivot derivation (`pivot_across() |> mutate(...) |>
tabular()`).
ae <- cdisc_saf_ae
ae$active <- paste0(ae$drug_50, " / ", ae$drug_100)

tabular(
 ae,
 titles = c("Table 14.3.0", "Adverse Event Overview"),
 footnotes = "Active = pooled Drug 50 + Drug 100 columns."
) |>
cols(
 stat_label = col_spec(usage = "group", label = ""),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 active = col_spec(label = "Active arms"),
 drug_50 = col_spec(visible = FALSE),
 drug_100 = col_spec(visible = FALSE),
 Total = col_spec(label = "Total", align = "decimal")
)

```

cols\_apply

*Apply one column spec to many columns*

## Description

Field-merge a single `col_spec()` onto every column matched by name or by a predicate. The vectorized companion to `cols()` for the common case of a variable number of treatment-arm columns that all share the same display rule (decimal alignment, a numeric format), so you avoid `do.call()` / `!!!` splicing one named argument per arm.

## Usage

```
cols_apply(.spec, .cols, .col_spec)
```

## Arguments

`.spec` *The tabular\_spec to extend.* `<tabular_spec>`: required. Dot-prefixed so partial matching cannot bind a user name in another slot.

`.cols` *Columns to match.* `<character | function>`: required. Either a character vector of input column names in `.spec@data`, or a predicate function(`names`)  $\rightarrow$  logical evaluated against `names(.spec@data)` (one logical per column, same length).

**Restriction:** Named columns must exist in `.spec@data`. A predicate must return a logical vector the length of `names(.spec@data)`. **Tip:** No `tidyselect` helpers ship; pass a base vector (`grep("^ARM", names(df), value = TRUE)`) or a predicate (`\(nm) startsWith(nm, "ARM")`).

`.col_spec`      *The spec to field-merge onto every match.* `<col_spec>`: required. Built with `col_spec()`.

## Details

**Field-merge, not replace.** `cols_apply()` reuses the same field-by-field merge as repeated `cols()` calls: a non-default field on `.col_spec` overrides; a default-valued field leaves any prior attribute on the matched column intact. Set the shared rule across arms first, then refine an individual arm with a later `cols()` call (or the reverse).

**Per-column label token.** A label that references `{.name}` (or its alias `{.col}`) inside a `{expr}` is resolved *per matched column*, with `.name` and `.col` both bound to that column's name. This makes a variable-N arm header a single declarative call instead of a hand-written loop. The rest of the `{expr}` evaluates in the calling environment, so a per-arm BigN looked up from a named vector works directly:

```
n <- c(placebo = 86, drug_50 = 84, drug_100 = 84)
cols_apply(
 spec, c("placebo", "drug_50", "drug_100"),
 col_spec(label = "{.name}\n(N={n[.name]})", align = "decimal")
)
placebo -> "placebo\n(N=86)" ; drug_50 -> "drug_50\n(N=84)" ; ...
```

The token is a plain-string feature; a label wrapped in `md()` / `html()` is parsed eagerly and does not interpolate. A failing token expression aborts naming the offending column.

**width merge.** `width`'s default sentinel for the merge is "auto": a later `cols()` / `cols_apply()` call carrying the default `width = "auto"` leaves a previously pinned width intact (only an explicit non-"auto" width overrides). Apply a shared width last to broadcast it across arms.

## Value

*The updated* `tabular_spec`. Continue chaining with `headers()`, `sort_rows()`, `style()`.

## See Also

**Companion verbs:** `cols()` attaches per-column specs by name; `col_spec()` builds the spec.

**Sibling build verbs:** `headers()`, `sort_rows()`, `style()`, `paginate()`, `preset()`.

## Examples

```
---- Example 1: Decimal-align every arm column by name vector ----
#
Demographics table whose treatment-arm columns are selected by a
name vector (`grep()` against the data) and given one shared
decimal-alignment spec, while the two row-label columns keep
their own roles set with `cols()`.
arm_cols <- grep("^placebo$|^drug_|^Total$", names(cdisc_saf_demo), value = TRUE)

tabular(
 cdisc_saf_demo,
```

```

titles = c(
 "Table 14.1.1",
 "Demographics and Baseline Characteristics",
 "Safety Population"
)
) |>
cols(
 variable = col_spec(usage = "group", label = "Parameter"),
 stat_label = col_spec(label = "Statistic")
) |>
cols_apply(arm_cols, col_spec(align = "decimal")) |>
sort_rows(by = c("variable", "stat_label"))

---- Example 2: Select arm columns with a predicate ----
#
Best Overall Response table. The arm columns are matched with a
predicate over the column names; the hidden sort helpers and the
response label are declared with `cols()`. The predicate scales
to any number of arms without editing the call.
bor_levels <- c(
 "CR", "PR", "SD", "NON-CR/NON-PD", "PD", "NE", "MISSING",
 "ORR (CR + PR)", "CBR (CR + PR + SD)",
 "DCR (CR + PR + SD + NON-CR/NON-PD)", "95% CI (Clopper-Pearson)"
)
)
eff <- cdisc_eff_resp
eff$stat_label <- factor(eff$stat_label, levels = bor_levels)

tabular(
 eff,
 titles = c(
 "Table 14.2.1",
 "Best Overall Response and Response Rates",
 "Efficacy Evaluable Population"
)
) |>
cols(
 stat_label = col_spec(label = "Response"),
 row_type = col_spec(visible = FALSE),
 groupid = col_spec(visible = FALSE),
 group_label = col_spec(visible = FALSE)
) |>
cols_apply(
 \(nm) nm %in% c("placebo", "drug_50", "drug_100"),
 col_spec(align = "decimal")
) |>
sort_rows(by = c("groupid", "stat_label"))

```

## Description

Build a single column's display attributes — usage, label, format, visibility, width, alignment, NA text. The result feeds `cols()`, which stamps the input column name onto the spec from its named-argument position and attaches it to the parent `tabular_spec`.

## Usage

```
col_spec(
 usage = NULL,
 label = NA_character_,
 format = NULL,
 visible = TRUE,
 width = "auto",
 group_display = "header_row",
 group_skip = NA,
 align = NULL,
 valign = NULL,
 na_text = NA_character_,
 indent_by = NA_character_
)
```

## Arguments

`usage` *Engine role.* <character(1) | NULL>: default NULL. One of:

- "display" (default in `cols()`) — pass-through.
- "group" — row-label with repeat-suppression and continuation-page repeat keys. Use for `variable`, `soc`, `stat_label`.
- "indent" — prefix every body cell of this column with one indent level (preset@indent\_size space-widths). Composes additively with `indent_by` (a column with both gets `depth_by + 1` indent levels per row). Backends with native padding-left semantics (HTML / LaTeX / RTF / DOCX / PDF) emit this as cell padding so wrapped continuation lines align with the indented baseline; Markdown carries the literal space-prefix. Synthesised group-header rows (under `group_display = "header_row"`) are NEVER indented — they're the parent at depth 0.
- "id" — a row-identifier column. Renders like "display" (one value per row, never collapses) but joins the *stub*: it repeats on every horizontal panel (`paginate(panels = N)`) and shows once on the left when a continuous backend (HTML / Markdown) collapses the panels into one table. The PROC REPORT ID role, orthogonal to grouping. Use for a per-row statistic label ("n", "Mean", "SD") that must stay legible on every panel of a wide demographics or efficacy table.
- NULL — inferred as "display" in `cols()`.

# Two row-label columns and four arm columns.

```
cols(
 variable = col_spec(usage = "group"),
 stat_label = col_spec(usage = "group"),
```

```

 placebo = col_spec(),
 drug_50 = col_spec()
)

Section-band table: the `group_label` column drives section
headers; `stat_label` body rows auto-indent under each header
without an explicit depth column.
cols(
 group_label = col_spec(usage = "group", group_display = "header_row"),
 stat_label = col_spec(usage = "indent", label = "Response"),
 placebo = col_spec(align = "decimal")
)

End-to-end ARD → wide → tabular pipeline. The cards ARD
`cdisc_saf_demo_ard` is the long upstream input; `pivot_across()`
widens to one column per arm and stamps an internal marker
so [sort_rows()] can reject sort keys on those arm columns.
`cols()` then attaches per-column display rules.
wide <- pivot_across(
 cdisc_saf_demo_ard,
 statistic = list(
 continuous = c(N = "{N}", "Mean (SD)" = "{mean} ({sd})"),
 categorical = "{n} ({p}%)"
)
)
tabular(wide, titles = "Demographics") |>
 cols(
 variable = col_spec(
 usage = "group", label = "Characteristic"
),
 stat_label = col_spec(
 usage = "group", label = "Statistic"
),
 Placebo = col_spec(align = "decimal"),
 `Xanomeline High Dose` = col_spec(
 label = "High Dose", align = "decimal"
),
 `Xanomeline Low Dose` = col_spec(
 label = "Low Dose", align = "decimal"
),
 Total = col_spec(align = "decimal")
)

```

label *Display label for the column header.* <character(1)>: default NA\_character\_. Embed \n for multi-line headers (arm name on row 1, BigN denominator on row 2 is the clinical convention). NA\_character\_ means use the input column name verbatim.

**Restriction:** Empty string and whitespace-only labels are accepted here, unlike `headers()` band labels which are strict.

Supports glue-style `{expr}` interpolation: braces are evaluated as R code in the calling environment at build time, so a `BigN` value folds inline, `label = "Placebo (N={n['placebo']})"`. Double a brace (`{{` or `}}`) for a literal one. An `md()` / `html()` label is passed through without interpolation.

**Per-column token.** `{.name}` (alias `{.col}`) inside a `{expr}` is *deferred* and resolved to the matched column's name when the spec is stamped by `cols()` / `cols_apply()`, so one spec can carry a variable-`N` arm header. See `cols_apply()` for the loop-free idiom.

```
Two-line header with arm name and BigN from cdisc_saf_n.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
col_spec(
 label = "Placebo\nN={n['placebo']}",
 align = "decimal"
)
```

**format** *Post-cell formatter.* `<character(1) | function | NULL>`: default `NULL`. A `sprintf` template applied per cell, OR a unary function(`x`) `->` `character` of the same length, OR `NULL` for backend default.

**Restriction:** Character templates are probed with `sprintf(format, 0)` at construction; malformed templates fail fast. **Tip:** Use a function for non-`sprintf` formatting (locale-aware numbers, thousand separators, conditional symbols).

```
sprintf template vs. function form.
col_spec(format = "%.1f")
col_spec(format = function(x) formatC(x, format = "f", digits = 1, big.mark = ","))
```

**visible** *Whether the column renders.* `<logical(1)>`: default `TRUE`. `FALSE` hides the column from output but keeps it in `spec@data` so `sort_rows()` and `style()` predicates can still reference it.

**Interaction:** Hidden columns are the standard pattern for sort-key helpers (`row_type`, `n_total`) and for the numeric counts behind formatted-text percentage cells.

**Auto-hide.** The depth column named by `indent_by` and every column named by `subgroup(by = ...)` or referenced via a `{col}` placeholder in the subgroup banner template are flipped to `visible = FALSE` automatically at engine time — restating it here is redundant.

**Break-only group column.** A hidden `usage = "group"` column emits no header rows and no in-column text; it contributes only its `group_skip` transitions, so `group_display` is ignored while hidden. This is the canonical "spacer" that drops a blank line wherever a marker value changes (e.g. continuous stats vs. categorical groups inside one characteristic): `col_spec(usage = "group", group_skip = TRUE, visible = FALSE)`.

**width** *Column width — auto-sized, pinned, or proportional.* `<character(1) | numeric(1)>`: default `"auto"`

- `"auto"` (*default*) — engine measures the widest cell (header + body) using bundled Adobe AFM Core 13 glyph metrics and distributes against the available content width. The **header** is sized to its widest *word*, so a multi-word header (e.g. "n, median") wraps at spaces; a non-breaking space () keeps a run whole. The **body** is sized to its widest *line* and never wraps, so numeric values stay intact. Pin a numeric width to wrap the body too.

- `<number>` — pinned in inches. Backends wrap content inside the pinned width (tabulararray `Q[wd=...]`, HTML `style="width:..."`, RTF / DOCX after twips conversion).
- `"2.5in" / "60mm" / "4cm" / "30pt" / "5pc"` — pinned dimension with an explicit TeX unit. Same behaviour as a bare numeric.
- `"30%"` — proportional width, percent of available content width. Resolved at engine time against the printable area.

**Tip:** Mix freely. Pinned and percent widths take priority; "auto" columns distribute whatever space remains. If pinned widths together exceed the available content width, the engine warns and leaves "auto" columns at their natural fit (layout may overflow).

**Restriction:** Must be positive. Percent values must fall in  $[0, 100]$ . Font-relative units (em, ex, rem) are rejected (no font-size context at parse time).

**Cross-format semantics (gt convention).** The width value is the user's source-of-truth. HTML emits it verbatim into `<col style="width:...">` (CSS accepts every unit: %, in, px, pt, cm, mm). Paper backends (LaTeX / RTF / PDF / DOCX) convert to their native unit via the AFM / distribute-widths pipeline. HTML is unconditionally responsive: when `width = "auto"` (default), the browser auto-sizes the column and cells wrap when the viewport narrows.

**Note:** NA and NULL are rejected. In pre-v0.1.0 tabular NA deferred to backend auto-fit; that path was inconsistent across backends and is replaced by the "auto" default, which produces identical widths across RTF / LaTeX / HTML.

**Merge sentinel.** For the field-merge across repeated `cols()` / `cols_apply()` calls, "auto" is treated as the default: a later call carrying `width = "auto"` leaves a previously pinned width intact, and only an explicit non-"auto" width overrides.

`group_display` *How usage = "group" values render in the body.* `<character(1)>`: default "header\_row". Active only when `usage = "group"`; ignored otherwise.

- `"header_row"` (*default*) — each unique value emits as a section header row above its block of data rows. The source column is hidden from the visible body. Matches the canonical submission Appendix I shape used by clinical TFL house templates (Disposition, Demographics, Statistical Report sections).
- `"column"` — column stays visible; repeated values are suppressed (only the first row of each value shows the label). PROC REPORT's default for grouping variables.
- `"column_repeat"` — column stays visible; every row repeats the value (no suppression). The shape `R's print.data.frame` produces.

**Composition under multiple group columns.** When more than one `usage = "group"` column is declared, the FIRST one encountered in `cols()` order is the outer group; subsequent group columns nest inside it. Each column's `group_display` choice is independent — a common clinical pattern is the outer variable as "header\_row" plus the inner `stat_label` as "column" (visible row labels under each section header).

```
Demographics layout: variable as section header, stat_label
as visible suppressed column.
```

```
cols(
 variable = col_spec(usage = "group", group_display = "header_row"),
 stat_label = col_spec(usage = "group", group_display = "column"),
 placebo = col_spec(label = "Placebo", align = "decimal")
)
```

group\_skip

*Insert a blank row between consecutive groups.* <logical(1)>: default NA. Active only when usage = "group"; ignored otherwise. Three values:

- TRUE — engine injects one blank row immediately before each value transition on this column (PROC REPORT's BREAK AFTER var / SKIP semantics, lifted to per-column control). Never trails the final group.
- FALSE — never insert a blank row for this column.
- NA (*default*) — follow group\_display: TRUE when group\_display = "header\_row", FALSE when "column" or "column\_repeat". Picks the canonical Appendix-I shape without an extra knob to set.

**Interaction:** When two or more columns have an effective group\_skip = TRUE and their value transitions coincide on the same row, the engine emits ONE blank row at that boundary, not one per column. Transition row indices are unioned across all contributing group columns.

```
Default: header_row mode auto-injects blanks between sections.
col_spec(usage = "group", group_display = "header_row")
```

```
Override: keep the column visible (suppressed-value mode) but
still insert blank-row separators between value changes.
col_spec(usage = "group", group_display = "column", group_skip = TRUE)
```

```
Override: section headers without the blank-row separator
(denser layout, used when vertical space is tight).
col_spec(usage = "group", group_display = "header_row", group_skip = FALSE)
```

```
Break-only "spacer": pairs with visible = FALSE to drop a blank
line wherever a hidden marker changes, without rendering the
column or any header row. group_display is ignored when hidden.
col_spec(usage = "group", group_skip = TRUE, visible = FALSE)
```

align

*Horizontal alignment within the column.* <character(1) | NULL>: default NULL.

One of:

- "left" — character columns; row labels.
- "center" — column-header band; rarely on data cells.
- "right" — numeric content without decimals.
- "decimal" — numeric or mixed-format cells aligned on the decimal mark. Use for "5 (3.2%)" next to "54 (32.1%)".
- NULL (*default*) — falls through to preset (alignment = list(body\_halign = ...)) and then to the baked default "left".

**Tip:** "decimal" pads numerics with non-breaking spaces so the decimal mark falls on a single column-wide anchor. The active preset's decimal\_metrics

knob is reserved for future em-aware padding refinement (see `preset()`); the current engine pads by character count.

**Default behaviour.** When `align` is unset (NULL / NA), every column emits with body left-aligned and header centred, regardless of the column's R data type. `tabular`'s canonical input is pre-summarised wide data frames where numeric content is already formatted as character strings (e.g. "52 (60.5)"), so `is.numeric()`-based auto-detection would mis-classify those columns as text and align them left — the opposite of intent. Use explicit `align = "decimal"` for NBSP-padded numeric columns (centred header over the padded centroid) or `align = "right"` for plain right-aligned numeric columns. The default cascade is `body → preset(alignment = list( body_halign = ... )) → CSS text-align: left; header → preset(alignment = list(header_halign = ... )) → CSS text-align: center.`

`valign` *Vertical alignment within the cell.* <character(1) | NULL>: default NULL. One of "top", "middle", "bottom". NULL falls through to `preset(alignment = list(body_valign = ...))` (baked default "top"). Per-cell overrides via `style(valign = ...)` still win over the column setting.

**Tip:** Set "middle" on the row-label column of a banded- row table so the label stays centred against the multi-line stat-block in the adjacent cell.

`na_text` *Text substituted for NA cells.* <character(1) | NA>: default NA. Substituted BEFORE the format step, so format does not need to anticipate NA. NA (default) inherits the preset's table-wide `na_text`; any string overrides it for this column, including "" to force blank cells even when the preset uses a non-empty token.

**Tip:** Use a sentinel ("-", "NR", ". ") when blank cells would be ambiguous, e.g. when "not applicable" and "not reported" both render blank.

`indent_by` *Name of a column in spec@data whose per-row integer / logical values drive indent depth on this column.* <character(1)>: default NA\_character\_. When set, the engine reads `spec@data[[indent_by]]` and prefixes this column's text

- AST in each row with `strrep(" ", preset@indent_size * depth)`. The referenced depth column is auto-hidden — no need to set `visible = FALSE` on it.

Typical SOC / PT pattern (the bundled `cdisc_saf_aesocpt` ships with the canonical depth column already attached, so no upstream construction is needed):

```
cols(
 label = col_spec(label = "Category", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE)
)
```

Multi-depth nesting works the same way — values `c(0L, 1L, 2L, ...)` produce 0, 1, 2, ... indent levels of `preset@indent_size` space-widths each. Negative values clamp to 0 (warn); fractional numerics floor (warn); NA → 0 (silent).

Composes orthogonally with `group_display = "header_row"`: synthetic group headers (depth 0) stay flush as parents; data rows under them carry their column's declared depth. Works in flat listings too — `indent_by` does not require any usage = "group" columns.

## Details

**Constructor-only.** `col_spec()` does not know which input column it belongs to until `cols()` stamps the name. Build reusable specs as ordinary R objects (e.g. `arm_col <- col_spec(align = "decimal")`) and apply them to multiple inputs without restating the name.

**Merge semantics across repeated `cols()` calls.** When `cols()` is called twice for the same column, the engine merges field-by-field: a non-default value on the new spec overrides; a default-valued field (NA / NULL / "" / TRUE) leaves the existing field intact. Build a column's spec in stages without re-stating earlier attributes.

**Validation timing.** Argument shapes are validated eagerly — a malformed `sprintf` template is probed at construction (`sprintf(format, 0)`) and fails fast at write time, not at render time.

## Value

A `col_spec` *S7 object*. Pass it to `cols()` keyed by the input column name; the constructor itself does not stamp a name.

## See Also

**Companion verb:** `cols()` attaches `col_spec` entries to a `tabular_spec` keyed by input column name.

**Sibling build verbs:** `headers()`, `sort_rows()`, `style()`, `paginate()`, `preset()`.

**Entry / terminal verbs:** `tabular()`, `emit()`, `as_grid()`.

**Inline label formatting:** `md()`, `html()`.

## Examples

```
---- Example 1: Demographics with every col_spec field exercised ----
#
Demographics table where every `col_spec` field is in play:
the row-label columns are pinned to a fixed width and aligned
left, the four arm columns embed BigN inline in the header,
decimal-align numeric content, and render `NA` cells as "-".
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

tabular(
 cdisc_saf_demo,
 titles = c(
 "Table 14.1.1",
 "Demographics and Baseline Characteristics",
 "Safety Population"
),
 footnotes = "Percentages based on N per treatment group."
) |>
cols(
 variable = col_spec(
 usage = "group", label = "Parameter",
 width = 2.0, align = "left"
),
 stat_label = col_spec(label = "Statistic", align = "left"),
```

```

 placebo = col_spec(
 label = "Placebo\nN={n['placebo']}",
 align = "decimal", na_text = "-"
),
 drug_50 = col_spec(
 label = "Drug 50\nN={n['drug_50']}",
 align = "decimal", na_text = "-"
),
 drug_100 = col_spec(
 label = "Drug 100\nN={n['drug_100']}",
 align = "decimal", na_text = "-"
),
 Total = col_spec(
 label = "Total\nN={n['Total']}",
 align = "decimal", na_text = "-"
)
) |>
 sort_rows(by = c("variable", "stat_label"))

---- Example 2: AE table with indented label + hidden helpers ----
#
AE-by-SOC/PT table where `label` carries SOC and PT text under
one column, indented by `indent_level`. Hidden helpers
(`row_type`, `n_total`) drive the sort while staying off the
rendered page. Demonstrates `indent_by` plus `visible = FALSE`
for sort-only columns, fixed width on the wide label column, and
decimal alignment on all four arm columns.
ae <- cdisc_saf_aesocpt
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))
ae$n_total <- as.integer(sub(".*", "", ae$Total))

tabular(
 ae,
 titles = c(
 "Table 14.3.1",
 "Adverse Events by SOC and Preferred Term",
 "Safety Population"
)
) |>
 cols(
 label = col_spec(label = "SOC / Preferred Term",
 indent_by = "indent_level", width = 2.5),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={n['placebo']}", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}", align = "decimal"),
 Total = col_spec(label = "Total\nN={n['Total']}", align = "decimal")
) |>
 sort_rows(by = c("row_type", "n_total"), descending = c(FALSE, TRUE))

```

```

---- Example 3: Format string + na_text for clean numeric display ----
#
`cdisc_eff_estimates` ships four competing efficacy models with
pre-computed numeric estimates, 95% CI bounds (NA on the MMRM
row), and a nominal p-value. `format` pins the printed
precision; `na_text` renders the missing CI bounds as a dash
rather than a literal "NA". `valign = "top"` keeps the multi-
line cell text aligned to the top.
tabular(cdisc_eff_estimates, titles = "Treatment-effect estimates by model") |>
 cols(
 model = col_spec(usage = "group", label = "Model", valign = "top"),
 estimate = col_spec(label = "Estimate", align = "decimal", format = "%.2f"),
 lower_ci = col_spec(
 label = "Lower\n95% CI",
 align = "decimal",
 format = "%.2f",
 na_text = "--"
),
 upper_ci = col_spec(
 label = "Upper\n95% CI",
 align = "decimal",
 format = "%.2f",
 na_text = "--"
),
 p_value = col_spec(
 label = "p-value",
 align = "decimal",
 format = "%.4f"
)
)

---- Example 4: Per-column width + halign override for vitals ----
#
`width` accepts a numeric (inches), a CSS-style string ("1.5in",
"20%"), or "auto". Centering the visit column under a wider
group-column setup demonstrates the alignment cascade -
col_spec@align beats the engine default but yields to a more
specific style() rule downstream.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
tabular(
 cdisc_saf_vital,
 titles = "Vital Signs at Baseline and End of Treatment"
) |>
 cols(
 paramcd = col_spec(visible = FALSE),
 param = col_spec(usage = "group", label = "Parameter",
 width = "1.6in"),
 visit = col_spec(usage = "group", label = "Visit",
 width = "1.2in", align = "center"),
 stat_label = col_spec(label = "Statistic", width = "1.0in"),
 placebo = col_spec(
 label = "Placebo\nN={n['placebo']}",
 align = "decimal", width = "0.9in"
)
)

```

```

),
 drug_50 = col_spec(
 label = "Drug 50\\nN={n['drug_50']}",
 align = "decimal", width = "0.9in"
),
 drug_100 = col_spec(
 label = "Drug 100\\nN={n['drug_100']}",
 align = "decimal", width = "0.9in"
)
)
)

---- Example 5: Non-collapsing `id` stub for a panelled table ----
#
`usage = "id"` marks `stat_label` ("n", "Mean", "SD", ...) as a
row identifier: like `display` it shows on every row, but it also
joins the stub, so it repeats on each horizontal panel created by
`paginate(panels = 2)`. On HTML / Markdown (no page width) the
panels collapse into one scrollable table with a "Panel 1 / Panel
2" header note; on RTF / Word each panel is its own page with the
`variable` + `stat_label` stub repeated.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
tabular(
 cdisc_saf_demo,
 titles = c("Table 14.1.1", "Demographics", "Safety Population")
) |>
 cols(
 variable = col_spec(usage = "group", group_display = "column",
 label = "Parameter"),
 stat_label = col_spec(usage = "id", label = "Statistic"),
 placebo = col_spec(label = "Placebo\\nN={n['placebo']}", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50\\nN={n['drug_50']}", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100\\nN={n['drug_100']}", align = "decimal"),
 Total = col_spec(label = "Total\\nN={n['Total']}", align = "decimal")
) |>
 paginate(panels = 2)

```

---

emit

*Render a tabular\_spec to a file*


---

## Description

Resolve spec through the engine pipeline, dispatch to the backend registered for the chosen format, and (optionally) write a QC data file and a CDISC ARS audit manifest alongside the rendered artefact. `emit()` is the package's terminal verb — it returns `file` invisibly so the call can sit at the bottom of a pipe without losing the path.

## Usage

```
emit(
```

```

 .spec,
 file,
 format = NULL,
 data_file = NULL,
 manifest = FALSE,
 create_dir = FALSE
)

```

## Arguments

|            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .spec      | <i>The tabular_spec to render.</i> <tabular_spec>: required. The full verb chain ( <code>tabular()</code> -> <code>cols()</code> -> <code>headers()</code> -> <code>sort_rows()</code> -> <code>style()</code> -> <code>paginate()</code> -> <code>preset()</code> ) feeds into <code>emit()</code> 's first argument by pipe.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| file       | <i>Destination path for the rendered artefact.</i> <character(1)>: required. Extension drives the backend (see the dispatch table in the Details section). The parent directory must already exist; <code>emit()</code> does not auto-create directories.<br><b>Tip:</b> Use <code>tempfile(fileext = ".md")</code> inside vignettes and examples so the example runs in R CMD check without polluting the package directory.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| format     | <i>Explicit backend override.</i> <character(1)   NULL>: default NULL. When set, wins over the file extension. Useful for writing .txt files that should contain RTF, for round-trip testing, or when the user has a custom backend registered under a non-standard name.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| data_file  | <i>QC artefact writer.</i> <character(1)   function(file) -> character(1)   NULL>: default NULL. When set, writes the resolved wide data frame alongside the render. A character path writes there directly; a lambda receives the render path and returns the data file path (typical for sponsor-flexible naming).<br><b>Restriction:</b> Returned-path extension must be .csv, .tsv / .txt, or .rds. <b>Tip:</b> The data frame the lambda governs is pre-backend — the same CSV is emitted regardless of whether file is RTF, PDF, or DOCX.<br><br># Three canonical sponsor patterns for the lambda.<br>data_file = \(\f) paste0(tools::file_path_sans_ext(f), "_qc.csv")<br>data_file = \(\f) file.path(<br>"validation",<br>paste0("val_", basename(tools::file_path_sans_ext(f)), ".csv")<br>)<br>data_file = \(\f) file.path(<br>"rd",<br>paste0("rd_", basename(tools::file_path_sans_ext(f)), ".rds")<br>) |
| manifest   | <i>Emit the CDISC ARS audit manifest sidecar.</i> <logical(1)>: default FALSE. TRUE writes <file>.audit.yml with verbatim CDISC ARS LDM v1.0 Output keys; see the manifest = TRUE invariant in the Details section for what the file contains and the determinism contract it satisfies.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| create_dir | <i>Create the destination directory if it is missing.</i> <logical(1)>: default FALSE. When TRUE, the parent directory of file (and any missing ancestors) is created recursively before rendering, instead of aborting. The default FALSE keeps the safe behaviour of erroring on a missing parent.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

## Details

**Validation before I/O.** Every argument is validated and the backend is resolved BEFORE the engine runs. An unsupported extension, a malformed `data_file` path, or a missing backend raises `tabular_error_input` without writing any file. A spec that resolves cleanly but whose backend errors mid-write may leave a partial file behind; this is the only failure mode that touches disk.

**Backend dispatch.** The effective backend is resolved from the file extension via the table below; the format argument always wins when both are supplied. Each backend lives in its own `R/backend_<fmt>.R` file and self-registers at package load time.

| extension(s)                              | format             | backend                              |
|-------------------------------------------|--------------------|--------------------------------------|
| <code>.md</code> , <code>.markdown</code> | <code>md</code>    | GFM pipe table (Step 15; shipped)    |
| <code>.html</code> , <code>.htm</code>    | <code>html</code>  | self-contained Bootstrap 5 (planned) |
| <code>.tex</code> , <code>.latex</code>   | <code>latex</code> | tabularray (planned)                 |
| <code>.pdf</code>                         | <code>pdf</code>   | tinytex compile of LaTeX (planned)   |
| <code>.rtf</code>                         | <code>rtf</code>   | RTF 1.9.1, native (shipped)          |
| <code>.docx</code>                        | <code>docx</code>  | OOXML native, no JVM (planned)       |

Unknown extensions, missing extensions, and formats with no registered backend all raise `tabular_error_input`. The error message lists the currently registered formats so the failure is actionable.

`data_file` is **sponsor-neutral**. Pass an explicit path ("`out/qc.csv`") for a fixed location, or a lambda (`function(file) -> path`) for sponsor-flexible naming. The lambda receives the resolved render path so it can derive the QC file from it (suffix, sibling folder, separate sponsor-styled name). Recognised extensions on the returned path are `.csv`, `.tsv` (alias: `.txt`), and `.rds`; anything else raises `tabular_error_input`. The written data frame is the post- `sort_rows()` / `post-engine_decimal()` wide grid — exactly the cell text the backend wrote.

`manifest = TRUE` **writes a sidecar**. The audit manifest is written to `<file>.audit.yml` next to the render (e.g. `out.md -> out.audit.yml`). Keys are CDISC ARS LDM v1.0 Output verbatim: `id`, `name`, `programmingCode` (best-effort `git + R + platform`

- `timestamp`), `fileSpecifications` (sha256 of every emitted artefact including `data_file`), `displays/displaySections` (Title / Header / Body / Footnote), `referencedAnalyses` (empty in v0.1; reserved for the mintverse handoff), `x-tabular` (rendering geometry, pagination, style trace, input provenance). Determinism contract: two consecutive `emit()` calls are byte-identical except for the `rendered_at` parameter timestamp; the YAML round-trips through `yaml::read_yaml() + yaml::write_yaml()`.

**Pure dispatcher.** `emit()` does not do any rendering itself; it composes `as_grid()` with a backend writer. To inspect the resolved grid without writing a file (during development, or to build a custom downstream consumer), call `as_grid()` directly.

## Value

*The file path, invisibly.* Use this when chaining `emit()` into a downstream consumer that needs the resolved path (e.g. printing the link in a Quarto chunk, copying the sidecar manifest into an archive, attaching the render to a submission folder builder).

## See Also

**No-I/O sibling:** `as_grid()` returns the resolved grid without writing a file — use during development to inspect what `emit()` would hand a backend.

**Build verbs the pipeline feeds from:** `tabular()`, `cols()` / `col_spec()`, `headers()`, `sort_rows()`, `style()`, `paginate()`, `preset()`.

**Inline formatting helpers:** `md()`, `html()` (titles, footnotes, labels, cell text).

## Examples

```
---- Example 1: Render demographics to Markdown ----
#
Smallest possible emit: spec in, .md out. The backend is chosen
from the file extension; the engine pipeline runs internally,
then the registered md backend writes a GFM pipe table you can
preview in any Markdown renderer. tempfile() keeps the example
clean for `R CMD check`.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

demo <- tabular(
 cdisc_saf_demo,
 titles = c(
 "Table 14.1.1",
 "Demographics and Baseline Characteristics",
 "Safety Population"
),
 footnotes = "Source: ADSL."
) |>
cols(
 variable = col_spec(usage = "group", label = "Characteristic"),
 stat_label = col_spec(label = "Statistic"),
 placebo = col_spec(label = "Placebo\nN={n['placebo']}", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}", align = "decimal"),
 Total = col_spec(label = "Total\nN={n['Total']}", align = "decimal")
) |>
sort_rows(by = c("variable", "stat_label"))

demo_md <- tempfile(fileext = ".md")
emit(demo, demo_md)

---- Example 2: Render + QC data + CDISC audit manifest ----
#
The clinical double-programming pattern: render the table,
write a QC CSV alongside it for an independent programmer to
verify cell-for-cell, and emit the CDISC ARS audit manifest
for submission packaging. The lambda derives the QC path from
the render path so the sponsor's naming convention lives in one
place.
ae <- cdisc_saf_aesocpt
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))
ae$n_total <- as.integer(sub(".*", "", ae$Total))
```

```

ae_spec <- tabular(
 ae,
 titles = c(
 "Table 14.3.1",
 "Adverse Events by SOC and Preferred Term",
 "Safety Population"
),
 footnotes = "Subjects counted once per SOC and once per PT."
) |>
cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={n['placebo']}", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}", align = "decimal"),
 Total = col_spec(label = "Total\nN={n['Total']}", align = "decimal")
) |>
sort_rows(by = c("row_type", "n_total"), descending = c(FALSE, TRUE))

ae_md <- tempfile(fileext = ".md")
emit(
 ae_spec,
 ae_md,
 data_file = \f paste0(tools::file_path_sans_ext(f), "_qc.csv"),
 manifest = TRUE
)

---- Example 3: Same spec, four backends – one-loop fan-out ----
#
`emit()` dispatches by file extension, so the same spec can
render to every backend in one loop. Useful for visual diffs
across formats during development and for shipping a build
artefact set (RTF for submission, HTML for review, PDF for the
CSR appendix).
eff_spec <- tabular(cdisc_eff_resp, titles = "Best Overall Response") |>
cols(
 stat_label = col_spec(usage = "group", label = "Response"),
 row_type = col_spec(visible = FALSE),
 groupid = col_spec(visible = FALSE),
 group_label = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal")
)

out_dir <- tempfile()
dir.create(out_dir)
for (ext in c(".html", ".rtf", ".tex", ".docx", ".md")) {
 emit(eff_spec, file.path(out_dir, paste0("eff", ext)))
}

```

```

}
list.files(out_dir)

---- Example 4: QC artefact via data_file alongside the render ----
#
`emit(data_file = ...)` writes the resolved post-engine wide
data frame alongside the rendered table. The sponsor's QC
programmer picks up the side-car .csv (or .rds) and validates
cell values without parsing the rendered RTF.
rtf_out <- tempfile(fileext = ".rtf")
data_out <- tempfile(fileext = ".csv")
emit(eff_spec, rtf_out, data_file = data_out)
file.exists(rtf_out)
file.exists(data_out)

---- Example 5: Render into a not-yet-existing output folder ----
#
`create_dir = TRUE` builds the destination directory tree on the
fly, so a submission-folder layout can be written in one pass
without a separate `dir.create()` step.
nested <- file.path(tempfile(), "tables", "safety", "eff.md")
emit(eff_spec, nested, create_dir = TRUE)
file.exists(nested)

```

---

footnote

---

*Attach an auto-numbered footnote to a table location*


---

## Description

Anchor a footnote to a cell, column header, title line, or any other `cells_*()` location. The engine assigns the marker, places a superscript at every matching anchor, and emits the marked-footnote line at the foot of the table. Markers are assigned **once**, in reading order, deduped by id, and are byte-identical across every backend (RTF / LaTeX / PDF / HTML / DOCX) and every page, so the marker at the anchor can never desynchronise from its note.

## Usage

```
footnote(.spec, text, .at = cells_body(), id = NULL, symbol = NULL)
```

## Arguments

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>.spec</code> | <i>The tabular_spec to annotate.</i> <code>&lt;tabular_spec&gt;</code> : required.                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>text</code>  | <i>The footnote text.</i> <code>&lt;character(1)&gt;</code>   <code>md()</code>   <code>html()</code> . Wrap in <code>md()</code> / <code>html()</code> for inline markup; plain strings are shown verbatim. A plain string supports glue-style <code>{expr}</code> interpolation, evaluated as R code in the calling environment at build time (double a brace for a literal one); an <code>md()</code> / <code>html()</code> value is passed through without interpolation. |

|        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .at    | <p>Where the marker is placed. &lt;tabular_location&gt;: default [cells_body()]. Any cells_*() location: a body-cell predicate (cells_body(where = ...)), a column header (cells_headers()), a title line (cells_title()), and so on.</p> <pre># data-driven body anchor: mark every high-frequency preferred term footnote(spec, "Includes events of any severity.",          .at = cells_body(where = n_total &gt;= 50, j = "label"))  # column-header anchor: mark the analysis-population denominator footnote(spec, "Safety population.",          .at = cells_headers(j = "Total"))</pre> <p><b>Note:</b> the styling argument is .at, never at.</p> |
| id     | <p>Stable identifier for sharing one marker across anchors. &lt;character(1)&gt;   NULL. Two footnote() calls with the same id share a single marker and a single note line. NULL (default) makes each call its own note.</p>                                                                                                                                                                                                                                                                                                                                                                                                                              |
| symbol | <p>Pin an explicit marker glyph. &lt;character(1)&gt;   NULL. Overrides the auto-allocated marker for this note (e.g. "*"). A pinned symbol is reserved and skipped by the auto-allocator, so it never collides. NULL (default) auto-allocates from the preset scheme.</p>                                                                                                                                                                                                                                                                                                                                                                                 |

## Details

**Engine-assigned, never hand-typed.** Unlike a literal ^a^ typed into both a cell and the footnotes argument, a footnote() marker is allocated by the resolve engine after decimal alignment, so it never disturbs column alignment and never drifts out of sync. The scheme (letters / numbers / symbols) and the block-line format come from the active preset (footnote\_markers, footnote\_label).

**Dedup by id.** Give two anchors the same id to share one marker and one note line. Without an id, each footnote() call is its own note.

**Coexists with** footnotes. Manual footnotes lines render first; the auto-numbered block follows. The two systems do not cross-dedup, so do not mix a hand-typed marker with an engine one for the same note.

## Value

A tabular\_spec. Pipe it onward to more verbs or to `emit()`.

## See Also

**Manual footnote lines:** the footnotes argument to `tabular()`.

**Location helpers:** `cells_body()`, `cells_headers()`, `cells_title()`.

**Inline markup:** `md()`, `html()`.

## Examples

```
---- Example 1: a denominator note on a column header ----
#
AE-by-SOC/PT table whose Total column header carries the analysis-
```

```

population note. The engine drops a superscript "a" on the header
and prints "a <text>" beneath the table.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
tabular(cdisc_saf_aesocpt) |>
 cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={n['placebo']}"),
 drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}"),
 Total = col_spec(label = "Total\nN={n['Total']}")
) |>
 footnote(
 "Safety population: all randomised subjects who took study drug.",
 .at = cells_headers(j = "Total")
)
)

---- Example 2: a data-driven note shared across cells ----
#
A single note marks every high-frequency preferred term (n >= 50 in
the Total column) in the SOC/PT stub. Sharing one `id` keeps it to
one marker and one line; the marker lands on each matching cell.
tabular(cdisc_saf_aesocpt) |>
 cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo"),
 drug_100 = col_spec(label = "Drug 100"),
 Total = col_spec(label = "Total")
) |>
 footnote(
 md("Includes events of *any* severity."),
 .at = cells_body(where = n_total >= 50, j = "label"),
 id = "anysev"
)
)

```

---

get\_preset

*Get the active session-default preset*


---

## Description

Return the `preset_spec` last attached via `set_preset()`, or `NULL` when no session default has been set. The cascade resolver calls this internally; users call it for diagnostics ("what is my session inheriting?") or to copy the active default into a per-spec override via `preset()`.

**Usage**

```
get_preset()
```

**Value**

A `preset_spec`, or `NULL` when no session default is active.

**See Also**

**Session-scope setter:** [set\\_preset\(\)](#).

**Per-spec partner:** [preset\(\)](#).

**Entry / terminal verbs:** [tabular\(\)](#), [emit\(\)](#), [as\\_grid\(\)](#).

**Examples**

```
---- Example 1: Inspect after setting a session default ----
#
`get_preset()` returns NULL before any session default has been
attached, then returns the `preset_spec` after `set_preset()`.
get_preset() # NULL

set_preset(font_size = 8, orientation = "landscape")

active <- get_preset()
is_preset_spec(active) # TRUE
active@font_size # 8
active@orientation # "landscape"

---- Example 2: Copy the session default into a per-spec override ----
#
Read the session preset, tweak one knob for a single table, and
attach as a per-spec override without disturbing the session.
set_preset(font_size = 9, paper_size = "letter")

Read-tweak-attach without mutating the session default.
base_knobs <- get_preset()
tabular(cdisc_saf_n) |>
 preset(
 font_size = base_knobs@font_size,
 paper_size = base_knobs@paper_size,
 orientation = "landscape"
)

Reset the session default so subsequent examples / R sessions
are not affected.
set_preset(.reset = TRUE)
```

---

headers

*Attach multi-level column headers*


---

## Description

Build the column-header band(s) above the rendered table. Each named argument is one band; the value is either a character vector of column names (leaf band) or a named list of further bands (inner band). Nesting depth is arbitrary — the engine renders one band row per depth level, with each cell spanning the columns of its leaves.

## Usage

```
headers(.spec, ...)
```

## Arguments

`.spec` *The tabular\_spec to attach the header tree to.* <tabular\_spec>: required. Dot-prefixed so R's partial argument matching cannot accidentally bind a short user-supplied band label in `...` to the spec slot.

`...` *Named header bands.* Each name is the band label (must be non-blank); each value is either:

- a **character vector** of data-column names — leaf band, or
- a **named list** whose entries follow the same recursive pattern — inner band.

Inside a nested-list value, an unnamed character-vector entry declares a passthrough leaf (see the Passthrough section above).

**Restriction:** Every column referenced must exist in `.spec@data`. A column may appear under at most one leaf. Names must be unique within one `headers()` call. **Tip:** Pass `headers()` with no arguments to clear the tree. **Interaction:** Band labels support glue-style `{expr}` interpolation, evaluated as R code in the calling environment at build time (double a brace for a literal one). The non-blank and uniqueness checks apply to the raw author-typed name, before interpolation.

## Details

**Replace, not stack.** A second `headers()` call REPLACES the prior tree — header structure is a single spec, not a stackable list. Call with no arguments to clear the tree.

**Strict label rule.** Every declared band label must carry visible text — empty strings, NA, and whitespace-only labels are rejected at every nesting level. This is stricter than `col_spec()`, which DOES accept empty labels (a row-label column with no header text is a legitimate clinical case). A silently-blank band would be a layout artefact.

**Uncovered columns render naked.** Columns not referenced under any band render with their `col_spec.label` only — no extra band row above them. This is the canonical pattern for row-label columns (`variable`, `soc`, `stat_label`).

**Multi-line band labels.** Embed `\n` in a band label for a two-line band cell (arm name on row 1, BigN on row 2).

**Spanner underline trim (backend limitation).** Each spanner's underline is trimmed at both ends, `booktabs \cmidrule(lr)` style, so adjacent spanners are separated by a visible gap rather than merging into one continuous line. PDF / LaTeX (`tabulararray leftpos/rightpos`) and HTML (an inset rule) render the trim natively. RTF and DOCX cannot inset a cell border horizontally, so there the spanner underline spans the full band width (adjacent spanner rules abut). This is a known, documented limitation of the OOXML / RTF cell-border model, not a bug.

## Value

*The updated* `tabular_spec`. Continue chaining with `sort_rows()`, `style()`.

## Passthrough leaves inside a nested band

Inside a nested-list value, a child entry may be **unnamed** — the entry is then a character vector of column names that sit directly under the parent with no intermediate band at this depth. Use this when one column under a band has no sub-grouping while its siblings do. The strict-label rule still applies to every declared band; an unnamed passthrough is NOT a band with a missing label — it is "no band declared at this depth for this column."

## See Also

**Companion verb:** `cols()` / `col_spec()` sets per-column labels — the leaf-row header text that sits below the band rows this verb builds.

**Sibling build verbs:** `sort_rows()`, `style()`, `paginate()`, `preset()`.

**Entry / terminal verbs:** `tabular()`, `emit()`, `as_grid()`.

**Inline label formatting:** `md()`, `html()`.

## Examples

```
---- Example 1: Single "Treatment Group" band over four arms ----
#
AE-by-SOC/PT table with one flat band labelled "Treatment Group"
spanning the four arm columns and the Total column. The
row-label column (`soc`) sits to the left of the band with no
header covering it - the canonical clinical layout.
ae <- cdisc_saf_aesocpt
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))
ae$n_total <- as.integer(sub(".*", "", ae$Total))
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

tabular(
 ae,
 titles = c(
 "Table 14.3.1",
 "Adverse Events by System Organ Class and Preferred Term",
 "Safety Population"
),
 footnotes = "Subjects are counted once per SOC and once per PT."
```

```

) |>
 cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={n['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}"),
 Total = col_spec(label = "Total\nN={n['Total']}")
) |>
 headers(
 "Treatment Group" = c("placebo", "drug_50", "drug_100", "Total")
) |>
 sort_rows(by = c("row_type", "n_total"), descending = c(FALSE, TRUE))

---- Example 2: Two-level nested band - Control vs Active arms ----
#
Efficacy BOR table where the active arms are grouped under an
"Active" sub-band and the placebo arm under a "Control"
sub-band, both under a single "Treatment Group" parent.
Demonstrates the named-list value form for arbitrary-depth
nesting.
bor_levels <- c(
 "CR", "PR", "SD", "NON-CR/NON-PD", "PD", "NE", "MISSING",
 "ORR (CR + PR)", "CBR (CR + PR + SD)",
 "DCR (CR + PR + SD + NON-CR/NON-PD)", "95% CI (Clopper-Pearson)"
)
eff <- cdisc_eff_resp
eff$stat_label <- factor(eff$stat_label, levels = bor_levels)
ne <- stats::setNames(cdisc_eff_n$n, cdisc_eff_n$arm_short)

tabular(
 eff,
 titles = c(
 "Table 14.2.1",
 "Best Overall Response and Response Rates",
 "Efficacy Evaluable Population"
),
 footnotes = "Response per RECIST 1.1, investigator assessment."
) |>
 cols(
 stat_label = col_spec(label = "Response"),
 row_type = col_spec(visible = FALSE),
 groupid = col_spec(visible = FALSE),
 group_label = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={ne['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={ne['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={ne['drug_100']}")
) |>
 headers(
 "Treatment Group" = list(

```

```

 "Control" = "placebo",
 "Active" = c("drug_50", "drug_100")
)
) |>
 sort_rows(by = c("groupid", "stat_label"))

---- Example 3: Multiple peer bands side by side ----
#
Vital-signs summary where the parameter columns (param,
paramcd, visit, stat_label) sit on the left under a "Variable"
band, and the arm columns sit on the right under "Treatment
Group". Demonstrates multiple top-level bands in one call --
bands render side by side in the order declared.
vit <- cdisc_saf_vital
tabular(vit, titles = c("Table 14.4.1", "Vital Signs Summary")) |>
 cols(
 param = col_spec(usage = "group", label = "Parameter"),
 paramcd = col_spec(visible = FALSE),
 visit = col_spec(usage = "group", label = "Visit"),
 stat_label = col_spec(label = "Statistic"),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal")
) |>
 headers(
 "Variable" = c("param", "paramcd", "visit", "stat_label"),
 "Treatment Group" = c("placebo", "drug_50", "drug_100")
)

---- Example 4: Three-tier band over efficacy arms + Total ----
#
Demographics-style three-tier nesting: top band labels the
whole arm strip, middle band splits Active vs Placebo, leaf
bands carry the per-arm column labels. Each child within a
`list(...)` may itself be a `list(...)` - bands nest to
arbitrary depth using nested list literals.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
tabular(cdisc_saf_demo, titles = "Demographics, hierarchical headers") |>
 cols(
 variable = col_spec(usage = "group", label = "Characteristic"),
 stat_label = col_spec(label = "Statistic"),
 placebo = col_spec(label = "N={n['placebo']}"),
 drug_50 = col_spec(label = "N={n['drug_50']}"),
 drug_100 = col_spec(label = "N={n['drug_100']}"),
 Total = col_spec(label = "N={n['Total']}")
) |>
 headers(
 "Treatment Group" = list(
 "Control" = "placebo",
 "Active" = list(
 "Drug 50" = "drug_50",
 "Drug 100" = "drug_100"
),
),
)

```

```

 "Pooled" = "Total"
)
)
```

---

html

*Mark a string as HTML for inline formatting*


---

## Description

Wrap a length-1 character vector so `tabular()`, `col_spec()`, and similar string slots interpret it as a constrained HTML subset at render time. Use when CommonMark cannot express the formatting (custom CSS via `<span style="...">`, raw destination codes via `<span data-rtf="...">`).

## Usage

```
html(text)
```

## Arguments

`text` *The HTML fragment.* `<character(1)>`: required. Length-1 character vector. NA is rejected.

## Details

**Recognised tag whitelist.** `<p>`, `<br>` / `<br/>`, `<strong>`, `<b>`, `<em>`, `<i>`, `<sup>`, `<sub>`, `<code>`, `<a href>`, `<span style>`. Tags outside this set drop their wrapper and keep their text content (no arbitrary HTML attack surface).

**Span styles.** `<span style="color: red; font-weight: bold">x</span>` parses the style attribute into a named character vector (`c(color = "red", "font-weight" = "bold")`). Backends translate CSS keys to destination-specific markup (RTF `\cf`, LaTeX `\textcolor`, DOCX `<w:color>`, HTML inline style).

**Backend-specific raw codes.** A span with `data-rtf`, `data-latex`, `data-html`, or `data-docx` attributes carries per-backend raw markup. The matching backend emits its data value verbatim and ignores the others; non-matching backends render the span's text content as plain. Use for cases the AST cannot express portably.

## Value

A length-1 character vector classed `c("from_html", "character")`. Pass it directly into any string-bearing slot (`tabular()` titles / footnotes, `col_spec()` label, `style()` pretext / posttext); the resolve engine calls `parse_inline()` internally and backends walk the resulting `inline_ast`.

## See Also

**Sibling helper:** `md()` — Markdown wrapper for the common case.

**String slots that consume the wrapper:** `tabular()` (titles, footnotes), `col_spec()` (label), `style()` (pretext, posttext).

**Entry / terminal verbs:** `tabular()`, `emit()`, `as_grid()`.

## Examples

```
---- Example 1: Colour-styled span in a title ----
#
Demographics table title with the population subset shaded
red. The HTML wrapper carries an inline CSS style; backends
translate (RTF: \cf, LaTeX: \textcolor, HTML: inline style).
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

tabular(
 cdisc_saf_demo,
 titles = c(
 "Table 14.1.1",
 "Demographics",
 html(sprintf("Safety Pop (N=%d)", n["Total"]))
)
)

---- Example 2: HTML link plus superscript footnote marker ----
#
AE table footnote with an HTML link and a superscript marker.
`html()` lets the user write tags directly when CommonMark
would be awkward (e.g. attributes that Markdown does not
surface).
tabular(
 cdisc_saf_ae,
 titles = c("Table 14.3.0", "Overall Adverse Event Summary"),
 footnotes = c(
 html('See MedDRA coding¹.')
)
) |>
 cols(stat_label = col_spec(usage = "group", label = "Category"))
```

---

 md

*Mark a string as Markdown for inline formatting*

---

## Description

Wrap a length-1 character vector so `tabular()`, `col_spec()`, `style()` pretext / posttext, and similar string slots interpret it as CommonMark Markdown at render time. Supports the GitHub-flavoured plus Pandoc-style superscript (`^sup^`) and subscript (`~sub~`) extensions; raw HTML inside Markdown passes through to the constrained tag set documented under `html()`.

## Usage

```
md(text)
```

## Arguments

`text` *The Markdown string.* `<character(1)>`: required. Length-1 character vector. NA is rejected; the empty string "" renders as no content.

## Details

**Convention adopted from gt.** Marking strings with `md()` and `html()` mirrors the well-tested `gt` convention. Plain (unwrapped) strings render as plain text — a stray `**` will NOT silently bold the surrounding span. Wrap explicitly to opt in.

**Recognised Markdown.** `**bold**`, `*italic*`, ``code``, `[link text](url)`, hard line break (two trailing spaces + `\n` or `\\ + \n`), Pandoc `^sup^` and `~sub~`. Single embedded `\n` (a "soft break" in CommonMark) renders as a space in HTML; `tabular` preserves it as a line break for clinical-table use where multi-line cells / titles are routine.

**HTML pass-through.** Raw HTML in Markdown (e.g. `md("Drug A <span style='color:red'>warning</span>")`) is parsed as HTML using the same tag whitelist as `html()`. Tags outside the whitelist drop their wrapper and keep their text content.

**Composition with plain strings.** `md()` and `html()` wrap the input with an internal control-character prefix that survives `c()` concatenation, so you can freely mix plain and marked strings in a single character vector: `c("Table 14.3.1", md("**Drug A**"), "third")`. Backends strip the marker before rendering; users never see it.

## Value

*A length-1 character vector classed `c("from_markdown", "character")`.* Pass it directly into any string-bearing slot (`tabular()` titles / footnotes, `col_spec()` label, `style()` pretext / posttext); the resolve engine calls `parse_inline()` internally and backends walk the resulting `inline_ast`.

## See Also

**Sibling helper:** `html()` — same wrapper pattern for raw HTML when Markdown cannot express the formatting.

**String slots that consume the wrapper:** `tabular()` (titles, footnotes), `col_spec()` (label), `style()` (pretext, posttext).

**Entry / terminal verbs:** `tabular()`, `emit()`, `as_grid()`.

## Examples

```
---- Example 1: Italic title qualifier with Pandoc footnote marker ----
#
AE-by-SOC/PT table. Title lines are bold by default, so the third
line italicises "Safety Population" via `md("**...*")` for a visible
contrast; the first footnote carries a Pandoc-style superscript
marker `^a^` that the backends render as a true superscript.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

tabular(
 cdisc_saf_aesocpt,
 titles = c(
```

```

 "Table 14.3.1",
 "Adverse Events by System Organ Class and Preferred Term",
 md("*Safety Population*")
),
 footnotes = c(
 md("^^ Subjects counted once per SOC and once per PT.")
)
) |>
 cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={n['placebo']}", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}", align = "decimal"),
 Total = col_spec(label = "Total\nN={n['Total']}", align = "decimal")
)

---- Example 2: Markdown link in a footnote ----
#
Efficacy BOR table that footnotes the response criteria with
a Markdown link. HTML / PDF / DOCX render as clickable; RTF /
LaTeX render the link text with the URL inline (backend
decides).
ne <- stats::setNames(cdisc_eff_n$n, cdisc_eff_n$arm_short)

tabular(
 cdisc_eff_resp,
 titles = c(
 "Table 14.2.1",
 "Best Overall Response",
 "Efficacy Evaluable Population"
),
 footnotes = c(
 md("Response per [RECIST 1.1](https://recist.eortc.org/), investigator assessment.")
)
) |>
 cols(
 stat_label = col_spec(usage = "group", label = "Response"),
 row_type = col_spec(visible = FALSE),
 groupid = col_spec(visible = FALSE),
 group_label = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={ne['placebo']}", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50\nN={ne['drug_50']}", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100\nN={ne['drug_100']}", align = "decimal")
)

```

## Description

Attach a `pagination_spec` to a `tabular_spec`. The engine uses the spec at render time to decide where page breaks fall, how wide tables split into horizontal panels, and what continuation marker (if any) prints on continued pages. The row budget per page is computed by the engine from the active preset (paper, orientation, margins, font size) and the chrome rows consumed by titles, column headers, and footnotes — you do not set rows-per-page directly.

## Usage

```
paginate(
 .spec,
 keep_together = character(),
 panels = 1,
 orphan_floor = 3,
 widow_floor = 2,
 repeat_content = c("titles", "headers", "footnotes"),
 continuation = NULL
)
```

## Arguments

|                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>.spec</code>          | <i>The tabular_spec to attach pagination to.</i> <code>&lt;tabular_spec&gt;</code> : required.                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <code>keep_together</code>  | <i>Group columns whose runs of identical values must not be split across a page break.</i> <code>&lt;character&gt;</code> : default <code>character()</code> . Every entry must be a usage = "group" column declared in <code>cols()</code> .<br><b>Interaction:</b> A run too tall to fit in the computed row budget less <code>orphan_floor</code> is split anyway; pagination is best-effort, not a hard contract.<br><br># Protect the SOC-level grouping in an AE-by-SOC/PT table.<br><code>paginate(keep_together = "soc")</code> |
| <code>panels</code>         | <i>Number of horizontal panels for wide tables.</i> <code>&lt;integer(1)   "auto"&gt;</code> : default 1. With 1, every column is on every page (single vertical scroll). With $N > 1$ , the engine splits non-group columns into $N$ chunks and repeats every group column on every panel.<br><b>Note:</b> "auto" is accepted but treated as 1 until preset-aware column-width metrics land; once they do, "auto" will split when the total table width exceeds the printable area.                                                    |
| <code>orphan_floor</code>   | <i>Minimum rows on a continued-from page.</i> <code>&lt;integer(1)&gt;</code> : default 3. When <code>keep_together</code> would move a page break back so far that fewer than <code>orphan_floor</code> rows would ride on the current page, the engine splits the protected run anyway. Acts as the escape valve for groups too tall to fit.                                                                                                                                                                                          |
| <code>widow_floor</code>    | <i>Minimum rows on the final page.</i> <code>&lt;integer(1)&gt;</code> : default 2. If the last page would carry fewer than <code>widow_floor</code> rows, the engine merges those rows back onto the previous page (page overflow accepted). Avoids the "one-row-orphaned-on-page-N" look without complicating the primary split rule.                                                                                                                                                                                                 |
| <code>repeat_content</code> | <i>Which page chrome repeats on every page.</i> <code>&lt;character&gt;</code> : default <code>c("titles", "headers", "footno</code><br>A subset of those three values; each is governed independently:                                                                                                                                                                                                                                                                                                                                 |

- "titles" — title block on every page (else page 1 only).
- "headers" — column-header band on every page (else page 1 only).
- "footnotes" — footnote block on every page (else last page only).

The default repeats all three so each page is self-contained per the submission layout contract. Pass a subset to drop one (e.g. `c("headers", "footnotes")` keeps the title on page 1 only), or `character()` to repeat nothing.

**Note:** Footnotes are always anchored to the page foot when present; membership only chooses every-page vs last-page-only, never table-body placement.

**HTML / MD:** ignored. HTML renders one continuous `<table>` and browsers natively repeat `<thead>` on print; MD has no print model. Effective only for the page-oriented backends (RTF, PDF, LaTeX, DOCX).

continuation *Marker text appended after a continuing table's title block.* `<character(1) | NULL>`: default NULL. NULL (the default) renders no marker — pick the wording your submission style guide expects (e.g. "(continued)", "(Cont'd)", "Page %d of %d") and pass it explicitly.

**Backend support is uneven** — verify against your render target:

- **PDF / LaTeX** — full: the marker prints on every continuation page (both vertical page overflow and horizontal panels).
- **RTF** — horizontal continuation *panels* only (`paginate(panels = N)`); the marker does NOT appear on vertical page-overflow continuations.
- **DOCX** — not marked. DOCX paginates natively but emits no continuation marker.
- **HTML / MD** — ignored. With one continuous document on screen there is no continuing-page boundary to mark.

## Details

**Replace, not stack.** A second `paginate()` call REPLACES the prior spec — pagination is a single configuration block, not a stackable list. Call with all defaults to clear back to the engine's auto behaviour.

**Rows per page are computed, not configured.** The engine takes the paper height for the active orientation (letter, a4) and subtracts the top + bottom margins, the title block height (number of title lines + a blank separator), the column-header band height (max embedded `\n` line count across visible column labels, plus any spanning header levels), and the footnote block height (number of footnote lines + a blank separator). The remainder, divided by the row height for the active font size, gives the body-row budget per page. Landscape pages naturally carry fewer rows than portrait at the same paper size; smaller fonts carry more.

`keep_together` **protects group runs.** When a page break would fall in the middle of a contiguous run of identical values in a `usage = "group"` column listed in `keep_together`, the engine moves the break BACK to the start of the run so the whole run rides on the next page. Single rule of escape: if moving the break back would leave fewer than `orphan_floor` rows on the current page, the engine splits the run anyway (a single group too tall to fit on one page cannot be kept together).

`panels` **and group stickiness.** With `panels > 1`, the engine splits the NON-group columns into approximately equal slices and repeats every `usage = "group"` column on every panel for row context. `panels = "auto"` defers the decision to preset-aware column-width metrics; until those metrics land in a future release the engine treats "auto" as 1.

**Value**

The updated `tabular_spec`. Continue chaining with `style()`, `preset()`, then render via `emit()` (or resolve without I/O via `as_grid()`).

**See Also**

**Render-geometry partner:** `preset()` / `set_preset()` — the preset's paper, orientation, margins, and font size feed the per-page row budget this verb depends on.

**Sibling build verbs:** `cols()` / `col_spec()`, `headers()`, `sort_rows()`, `style()`.

**Entry / terminal verbs:** `tabular()`, `emit()`, `as_grid()`.

**Examples**

```
---- Example 1: AE table paginated by SOC ----
#
AE-by-SOC/PT table that may run several pages. The SOC column is
protected by `keep_together` so a page break never lands in the
middle of one SOC's PT rows. The engine derives the row budget
from the preset's orientation + font_size + paper size and from
the title / footnote / header line counts on the spec - no
manual rows-per-page knob to keep in sync.
ae <- cdisc_saf_aesocpt
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))
ae$n_total <- as.integer(sub(".*", "", ae$Total))
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

tabular(
 ae,
 titles = c(
 "Table 14.3.1",
 "Adverse Events by System Organ Class and Preferred Term",
 "Safety Population"
),
 footnotes = "Subjects are counted once per SOC and once per PT."
) |>
cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(usage = "group", visible = FALSE,
 group_display = "column_repeat"),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={n['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}"),
 Total = col_spec(label = "Total\nN={n['Total']}")
) |>
headers("Treatment Group" = c("placebo", "drug_50", "drug_100", "Total")) |>
sort_rows(by = c("row_type", "n_total"), descending = c(FALSE, TRUE)) |>
paginate(
 keep_together = "soc",
```

```

 repeat_content = c("titles", "headers", "footnotes"),
 continuation = "(continued)"
)

---- Example 2: Wide ACROSS-style efficacy table split across 2 panels ----
#
BOR table where the four-arm column block is too wide for portrait
paper. Split into 2 horizontal panels; the group column
(`stat_label`) repeats on every panel for row context. Vertical
pagination still applies, so on a tall table you would see panel A
pages 1-2, then panel B pages 1-2.
bor_levels <- c(
 "CR", "PR", "SD", "NON-CR/NON-PD", "PD", "NE", "MISSING",
 "ORR (CR + PR)", "CBR (CR + PR + SD)",
 "DCR (CR + PR + SD + NON-CR/NON-PD)", "95% CI (Clopper-Pearson)"
)
eff <- cdisc_eff_resp
eff$stat_label <- factor(eff$stat_label, levels = bor_levels)
ne <- stats::setNames(cdisc_eff_n$n, cdisc_eff_n$arm_short)

tabular(
 eff,
 titles = c(
 "Table 14.2.1",
 "Best Overall Response and Response Rates",
 "Efficacy Evaluatable Population"
),
 footnotes = "Response per RECIST 1.1, investigator assessment."
) |>
cols(
 stat_label = col_spec(usage = "id", label = "Response"),
 row_type = col_spec(visible = FALSE),
 groupid = col_spec(visible = FALSE),
 group_label = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={ne['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={ne['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={ne['drug_100']}")
) |>
sort_rows(by = c("groupid", "stat_label")) |>
paginate(panels = 2, repeat_content = c("titles", "headers", "footnotes"))

---- Example 3: Orphan / widow floors + continuation marker ----
#
Long vital-signs table with two safeguards: orphan_floor = 4
prevents fewer than 4 rows of a group landing alone at the
bottom of a page; widow_floor = 2 prevents fewer than 2 rows of
a group landing alone at the top of the next page; the
continuation marker prints on every page after the first.
tabular(
 cdisc_saf_vital,
 titles = c("Table 14.4.1", "Vital Signs Summary at Each Visit")
) |>
cols(

```

```

 param = col_spec(usage = "group", label = "Parameter"),
 paramcd = col_spec(visible = FALSE),
 visit = col_spec(usage = "group", label = "Visit"),
 stat_label = col_spec(label = "Statistic"),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal")
) |>
 paginate(
 keep_together = "param",
 orphan_floor = 4L,
 widow_floor = 2L,
 continuation = "(continued)"
)

---- Example 4: Many-arm horizontal pagination via column-fit ----
#
Wide AE-by-SOC/PT table where the column strip itself does not
fit on a single page. The engine slices columns into groups
(each group keeping the `usage = "group"` columns repeated on
every horizontal page) so the SOC / PT label band re-appears
alongside whichever arm columns land on each panel.
tabular(
 cdisc_saf_aesocpt,
 titles = c("Table 14.3.1", "AEs by SOC and PT (wide-page split)")
) |>
 cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level",
 width = "2.5in"),
 soc = col_spec(usage = "group", visible = FALSE,
 group_display = "column_repeat"),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo", align = "decimal",
 width = "2.0in"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal",
 width = "2.0in"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal",
 width = "2.0in"),
 Total = col_spec(label = "Total", align = "decimal",
 width = "2.0in")
) |>
 paginate(keep_together = "soc")

```

## Description

`pivot_across()` is tabular's input-side helper: it consumes a long Analysis Results Data (ARD) data frame (typically produced by `cards::ard_stack()` or `cards::ard_stack_hierarchical()`) and returns a wide display data.frame ready to pass to `tabular()`.

## Usage

```

pivot_across(
 data,
 statistic = list(continuous = "{mean} ({sd})", categorical = "{n} ({p}%)"),
 column = NULL,
 row_group = NULL,
 label = NULL,
 overall = "Total",
 decimals = NULL,
 fmt = NULL
)

```

## Arguments

**data** *Long ARD input data.* <data.frame>: required. At minimum needs `stat_name` and `stat`. Cards-style group columns (`group1`, `group1_level`, ...) and `variable / variable_level` are auto-detected. Tibbles / card objects / arrow tables are coerced via `as.data.frame()`.

**statistic** *Format spec for cell composition.* <character(1) | named list>: required. Combines one or more ARD stats into one display cell. Three accepted forms — each illustrated below. Inside a format string, `{stat_name}` substitutes that stat's value from the ARD (for example, `"{n} ({p}%)"` interpolates the `n` and `p` stats into a "53 (62%)" cell). The lookup order when a value is needed for a variable is: per-variable -> per-context -> default -> the literal "{n}".

### Form 1: single string:

One format string applied to every variable regardless of context. Use when your ARD is homogeneous (e.g. all categorical).

# Every variable rendered as "n (p%)" – categorical-only slice.

```

cat_only <- cdisc_saf_demo_ard[cdisc_saf_demo_ard$context == "categorical",]
pivot_across(
 cat_only,
 statistic = "{n} ({p}%)"
)

```

### Form 2: named list by context:

Different formats per context. This is the typical clinical-table form because demographics mix continuous and categorical variables.

**The list names must match the values in the ARD's context column verbatim.** Which strings appear there depends on how the ARD was built:

- `cards::ard_continuous() / ard_categorical()` emit "continuous" / "categorical".
- `cards::ard_summary() / ard_tabulate()` emit "summary" / "tabulate".

So an ARD assembled with `ard_stack(ard_summary(...), ard_tabulate(...))` is keyed summary / tabulate, not continuous / categorical. Inspect `unique(ard$context)` when unsure.

```
AGE (continuous) -> "75.2 (8.59)"; SEX (categorical) -> "53 (62%)"
pivot_across(
 cdisc_saf_demo_ard,
 statistic = list(
 continuous = "{mean} ({sd})",
 categorical = "{n} ({p}%)"
)
)
```

### Form 3: named list by variable:

Override on a per-variable basis; fall back to default or context. Use when one variable needs a custom format.

```
AGE shows just the mean; SEX / RACE keep the categorical default.
pivot_across(
 cdisc_saf_demo_ard,
 statistic = list(
 AGE = "{mean}",
 categorical = "{n} ({p}%)",
 default = "{mean} ({sd})"
)
)
```

### Multi-row continuous spec:

Any single entry can itself be a **named character vector** — each element becomes one display row, with the name as the row label. Use for N / Mean (SD) / Median / Min, Max-style blocks.

```
pivot_across(
 cdisc_saf_demo_ard,
 statistic = list(
 continuous = c(
 N = "{N}",
 "Mean (SD)" = "{mean} ({sd})",
 Median = "{median}",
 "Min, Max" = "{min}, {max}"
),
 categorical = "{n} ({p}%)"
)
)
```

|           |                                                                                                                                                                                                                                                                                           |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| column    | <i>Grouping column whose unique values become arms.</i> <character(1)   NULL>: default NULL. NULL auto-detects from the ARD's <code>group1</code> value or — for renamed input — picks the single non-standard column. Pass a string when multiple group columns exist.                   |
| row_group | <i>Second, non-column grouping dimension.</i> <character(1)   NULL>: default NULL. Names the non-arm group variable of a two-variable <code>.by</code> (e.g. <code>SEX</code> in <code>ard_stack(.by = c(ARM, SEX))</code> ). It widens into a leading row column (not a pivoted arm col- |

umn), so the result composes with `subgroup(by = ...)` or `col_spec(usage = "group")` downstream.

**Why it is required.** `cards` encodes a crossing factor and a SOC/PT hierarchy identically (the second group variable appears in `variable` on its by-marginal rows), so the two cannot be told apart automatically. Naming `row_group` declares "this is a crossing factor": the by-marginal rows are dropped and the flat path is used. Leave it `NULL` for a genuine hierarchy.

**Restriction:** Must name a second grouping variable present in the ARD and must differ from `column`.

|          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| label    | <i>Variable-name to display-label map.</i> <character>   <code>NULL</code> : default <code>NULL</code> .<br>Named character vector mapping variable names to display labels (e.g. <code>c(AGE = "Age (years)", SEX = "Sex")</code> ). Applies to <code>variable</code> , <code>soc</code> , and <code>label</code> columns of the output. <code>NULL</code> leaves the upstream variable names verbatim.                                                                                            |
| overall  | <i>Column name for NA-arm (overall / total) rows.</i> <character(1)   <code>NULL</code> >: default "Total".<br>Pass <code>NULL</code> to drop overall rows entirely (per-arm only output).                                                                                                                                                                                                                                                                                                          |
| decimals | <i>Per-stat decimal precision.</i> <named integer   named list>: default <code>c()</code> .<br>Accepts two forms: <ul style="list-style-type: none"> <li>• <b>named integer vector</b> — global per-stat overrides (<code>c(mean = 1, sd = 2, p = 0)</code>).</li> <li>• <b>named list</b> — per-variable plus <code>.default</code> (<code>list(AGE = c(mean = 2), .default = c(p = 1))</code>).</li> </ul> Built-in defaults apply when neither sets a stat.                                      |
| fmt      | <i>Per-stat custom formatter functions.</i> <named list of function>: default <code>list()</code> .<br>Each function takes a numeric value and returns a character string; overrides built-ins and <code>decimals</code> for that stat. Useful for p-value styling and other domain-specific formatting.<br><br># p-value formatter: render below-threshold values as "<0.001".<br><pre>fmt = list(   p.value = function(x) {     ifelse(x &lt; 0.001, "&lt;0.001", sprintf("%.3f", x))   } )</pre> |

## Details

`tabular`'s package boundary is **display-only**: pre-summarised data in, rendered file out. `pivot_across()` is the canonical bridge between the `cards` aggregation backend and that boundary. It does not aggregate — it pivots arms to columns, interpolates per-cell display strings from the stat values, and applies decimal precision. Filtering, weighting, and aggregation happen upstream in `cards` or your own data-prep step.

### Key statistic by the ARD context:

`statistic` (and `fmt`) are matched against the ARD's context column verbatim, and that value differs per generating function. Keying by the wrong name silently drops the format. Inspect `unique(ard$context)` first and key to match (or pass a single format string / `default =` to cover

everything). When an explicitly-supplied statistic matches no context at all, `pivot_across()` warns rather than silently emitting `{n}`.

| Generating function                          | context to key on       |
|----------------------------------------------|-------------------------|
| <code>cards::ard_summary()</code>            | summary                 |
| <code>cards::ard_tabulate()</code>           | tabulate                |
| <code>cards::ard_continuous()</code>         | continuous              |
| <code>cards::ard_categorical()</code>        | categorical             |
| <code>cards::ard_stack_hierarchical()</code> | tabulate + hierarchical |
| <code>cardx::ard_categorical_ci()</code>     | proportion_ci           |
| <code>cardx::ard_continuous_ci()</code>      | continuous_ci           |

#### Indentation of `stat_label`:

Categorical levels and the multi-row continuous stat labels come back already indented with two leading spaces, ready to render as a plain display column. Do **not** also set `col_spec(usage = "indent")` on `stat_label` — that stacks the engine indent on top of the string indent (a double indent). Use one or the other.

#### Zero-suppression (always-on default):

A row whose `n` value equals zero renders the whole cell as the bare `n` value instead of fully interpolating the format string. For a categorical level with `n = 0`, the cell shows `"0"`, not `"0 (0.0%)"`. This is clinical convention — empty cells should read as a single zero, not advertise a meaningless rate.

**How the default fires (chain of events).** During cell assembly, before format-string interpolation, the engine checks the row's `n` stat. If it is zero, the engine short-circuits and returns the formatted `n` value (`"0"`) as the entire cell — `{p}` is never substituted, so the `(0.0%)` half of the format string is dropped.

**How to opt out: supply a custom `fmt$n`.** Setting any function under `fmt$n` is the engine's signal that the user owns the `n` rendering. The short-circuit is disabled for the whole table; for every row the full format string interpolates, so `{n}` becomes your formatter's output and `{p}` becomes the standard percentage. For `n = 0`, that's `"0 (0.0%)"`.

```
Force "0 (0.0%)" for n = 0 rows by attaching a custom n formatter.
The body of fmt$n can be the default integer rendering - its
presence alone is what disables the zero-suppression branch.
pivot_across(
 cdisc_saf_demo_ard,
 statistic = list(
 continuous = "{mean} ({sd})",
 categorical = "{n} ({p}%"
),
 fmt = list(n = function(x) sprintf("%d", as.integer(x)))
)
```

#### Pharma rounding (always-on default):

A percentage that would otherwise round to `0` (when the value is positive but smaller than the chosen precision) renders as `<0.1`; one that would round to `100` (positive but smaller than `100`) renders as `>99.9`. The threshold is precision-aware: `decimals = c(p = 2)` produces `<0.01 / >99.99`. This

matches the pharma convention of never claiming exactly 0% or 100% when at least one subject contributed.

Override per-stat via `fmt`:

```
Show exact rounded percentages even at the extremes
pivot_across(
 data,
 statistic = "{n} ({p}%)",
 decimals = c(p = 1),
 fmt = list(p = function(x) sprintf("%.1f", x * 100))
)
```

Your `fmt$p` receives the raw stat value (a proportion between 0 and 1) and returns the displayed string. The `pharma-threshold` branch only fires inside the built-in `p` formatter and the `decimals-` driven path, so any custom `fmt$p` bypasses it.

## Value

A wide data frame ready for `tabular()`. Schema:

- `variable` — variable name (or label after `label = ...`).
- `stat_label` — display-row label.
- One column per arm level (named after the `group1_level` values or the renamed arm column).
- Total (or whatever overall is set to) when applicable.
- A leading column named after `row_group` when set (the second grouping dimension).
- Hierarchical ARD adds `soc`, `label`, `row_type` instead of `variable`.

Pass the result straight into `tabular()` to start the render pipeline.

## See Also

**Pipeline entry consumer:** `tabular()` — wraps the wide data frame this helper returns.

**Downstream spec-build verbs:** `cols()` / `col_spec()`, `headers()`, `sort_rows()`, `style()`, `paginate()`, `preset()`.

**Terminal verbs:** `emit()`, `as_grid()`.

## Examples

```
---- Example 1: Demographics - long ARD to rendered spec ----
#
Full pipeline from a `cards::ard_stack()`-style long ARD to a
sorted `tabular_spec`. The multi-row continuous block (N /
Mean (SD) / Median / Min, Max) sits above each categorical
block; decimals are set per-stat (mean 1, sd 2, p 1) to match
the CDISC convention.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

cdisc_saf_demo_ard |>
 pivot_across(
```

```

statistic = list(
 continuous = c(
 N = "{N}",
 "Mean (SD)" = "{mean} ({sd})",
 Median = "{median}",
 "Min, Max" = "{min}, {max}"
),
 categorical = "{n} ({p}%)"
),
decimals = c(mean = 1, sd = 2, p = 1, median = 1, min = 0, max = 0),
label = c(AGE = "Age (years)", SEX = "Sex", RACE = "Race")
) |>
tabular(
 titles = c(
 "Table 14.1.1",
 "Demographics and Baseline Characteristics",
 "Safety Population"
),
 footnotes = "Percentages based on N per treatment group."
) |>
cols(
 variable = col_spec(usage = "group", label = "Parameter"),
 stat_label = col_spec(label = "Statistic"),
 Placebo = col_spec(
 label = "Placebo\nN={n['placebo']}",
 align = "decimal"
),
 `Xanomeline Low Dose` = col_spec(
 label = "Drug 50\nN={n['drug_50']}",
 align = "decimal"
),
 `Xanomeline High Dose` = col_spec(
 label = "Drug 100\nN={n['drug_100']}",
 align = "decimal"
),
 Total = col_spec(
 label = "Total\nN={n['Total']}",
 align = "decimal"
)
)

---- Example 2: Hierarchical SOC/PT AE table ----
#
Hierarchical `cards::ard_stack_hierarchical()` output threaded
through `pivot_across()`. The hierarchical ARD emits a
(soc, label, row_type) triple plus one stat row per (arm, SOC, PT);
`pivot_across()` folds the arm dimension to columns and preserves
the hierarchy markers. Derive `indent_level` from `row_type` so
`col_spec(indent_by = "indent_level")` drives the SOC -> PT
indent on the `label` column.
wide <- cdisc_saf_aesocpt_ard |>
 pivot_across(statistic = "{n} ({p}%)"
wide$indent_level <- as.integer(wide$row_type == "pt")

```

```

tabular(
 wide,
 titles = c(
 "Table 14.3.1",
 "Adverse Events by System Organ Class and Preferred Term",
 "Safety Population"
),
 footnotes = c(
 "Subjects are counted once per SOC and once per PT.",
 "Percentages based on N per treatment group."
)
) |>
cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 Placebo = col_spec(
 label = "Placebo\nN={n['placebo']}",
 align = "decimal"
),
 `Xanomeline Low Dose` = col_spec(
 label = "Drug 50\nN={n['drug_50']}",
 align = "decimal"
),
 `Xanomeline High Dose` = col_spec(
 label = "Drug 100\nN={n['drug_100']}",
 align = "decimal"
)
)

---- Example 3: Hierarchical ARD (SOC / PT) ----
#
`cdisc_saf_aesocpt_ard` carries an `ard_stack_hierarchical` shape with
two grouping variables (AEBODSYS / AEDECOD). `pivot_across()`
recognises the hierarchical structure and emits dedicated `soc`,
`label`, and `row_type` columns so the SOC -> PT nesting survives
the pivot. The result is ready for `tabular()` plus `sort_rows()`.
head(cdisc_saf_aesocpt_ard, 3)

wide <- cdisc_saf_aesocpt_ard |>
 pivot_across(statistic = "{n} ({p}%)")
head(wide, 3)

---- Example 4: Multi-row continuous spec + label re-labelling ----
#
`statistic = c(<label> = <template>, ...)` produces one display
row per named entry – the canonical "N / Mean (SD) / Median /
Min, Max" block for continuous variables. `label = c(...)`
renames the variable headings emitted into the wide output.
cdisc_saf_demo_ard |>
 pivot_across(
 statistic = list(

```

```

 continuous = c(
 N = "{N}",
 "Mean (SD)" = "{mean} ({sd})",
 Median = "{median}",
 "Q1, Q3" = "{p25}, {p75}",
 "Min, Max" = "{min}, {max}"
),
 categorical = "{n} ({p}%)"
),
 label = c(
 AGE = "Age (years)",
 WEIGHT = "Weight (kg)",
 HEIGHT = "Height (cm)",
 BMI = "BMI (kg/m^2)"
)
)

---- Example 5: ARD keyed by summary / tabulate contexts ----
#
The `statistic` list names must match the ARD's `context` column
verbatim. `cards::ard_summary()` / `ard_tabulate()` emit `summary` /
`tabulate` (not the `continuous` / `categorical` of
`ard_continuous()` / `ard_categorical()`), so a list keyed
`continuous`/`categorical` would silently match nothing. Always check
`unique(ard$context)` first. Here the bundled `cdisc_saf_demo_ard` is
relabelled to mimic `ard_summary()` + `ard_tabulate()` output; the
by-variable's own row drops automatically and both the summary and
the tabulate variables survive.
card_st <- cdisc_saf_demo_ard
card_st$context[card_st$context == "continuous"] <- "summary"
card_st$context[card_st$context == "categorical"] <- "tabulate"
pivot_across(
 card_st,
 statistic = list(
 summary = "{mean} ({sd})",
 tabulate = "{n} ({p}%)"
)
)

```

---

preset

*Override the render preset on a spec*

---

## Description

Attach a `preset_spec` to a `tabular_spec`, carrying page-geometry knobs (paper, orientation, margins, body font\_size + family, h-rule policy, decimal metric, typography defaults). The engine consults the per-spec preset first when computing the per-page row budget, decimal-aligned column widths, and the chrome that the backend renders around the body grid.

**Usage**

```
preset(.spec, ..., .template = NULL, .style = NULL, .reset = FALSE)
```

**Arguments**

`.spec` *The tabular\_spec to attach the preset to.* <tabular\_spec>: required. Dot-prefixed so R's partial argument matching cannot accidentally bind a knob name in ... to the spec slot.

... *Named preset knobs.* Any subset of the preset knobs the preset\_spec class carries. Knob values are validated against the class's enum / length / type rules; bad values raise tabular\_error\_input. Unknown knob names raise tabular\_error\_input with the recognised set listed.

Recognised knobs:

- `font_size` — body point size. <numeric(1)>.
- `font_family` — body font family. <character | character(1)>. Default "mono". Three accepted shapes:
  1. **Generic family** — "mono" (default), "serif", "sans" (CSS aliases "monospace" / "sans-serif" also recognised). The resolver expands to a per-backend chain that leads with the Linux-installed **Liberation** face (Posit Workbench / Domino / Citrix / RStudio Server), then the Microsoft Office face (Courier New / Times New Roman / Arial) for desktop Win / Mac consumers, then TeX Gyre for LaTeX compile, then the CSS generic for HTML. Liberation Mono / Serif / Sans are metric-compatible with Courier New / TNR / Arial, so layout, line breaks, and decimal alignment hold across every render context. The mono default matches the dominant submission-TFL convention where deterministic glyph widths drive n (%) cell alignment.
  2. **Named alias** — "Times", "Times New Roman", "Arial", "Helvetica", "Courier", "Courier New". These PostScript-era names alias to the appropriate generic family (Times -> serif, Arial / Helvetica -> sans, Courier -> mono) and emit the same expanded chain. Honours the user's intent ("I want Times-like rendering") on every OS instead of hard-erroring on a Linux server with no TNR installed.
  3. **Named font** — "Inter", "JetBrains Mono", "Source Serif Pro", sponsor-specific face, etc. Emitted verbatim with no fallback fabricated. The consuming app (browser, xelatex, Word, LibreOffice) resolves the name against its own font matcher. RTF and DOCX fall back to the consuming app's substitution table when the name is missing; xelatex hard-errors at compile time; HTML browsers fall through to the browser's default font (not necessarily class-matched).
  4. **Explicit stack** — c("Inter", "Helvetica", "sans"). User owns the chain. Returned verbatim — alias lookup is **bypassed**, so c("Times", "Times") honours the exact name with no chain expansion (escape hatch for users who genuinely want exact-name semantics).

**Note:** Adobe Source Pro is no longer the default lead. Source Pro is not pre-installed on production Linux servers, so leading with it walks through

2-3 missing names before resolving. Users who installed Source Pro can opt in via the explicit-stack form (c("Source Serif Pro", "serif")).

**What you see in Word's font dropdown vs. what renders.** When you open a tabular-generated .rtf in Word on macOS or Windows, the font dropdown displays the file's *requested* face — "Liberation Mono" by default (the Linux-server-installed face). The rendered text on screen is whatever Word's `\\*\falt` substitution resolved to — typically Courier New on macOS / Windows. This is correct: Liberation Mono and Courier New are metric-compatible by design, so the rendered layout (line breaks, decimal alignment, page breaks) is identical regardless of which face Word actually used to render. The same `\\*\falt` substitution model applies to serif (Liberation Serif -> Times New Roman) and sans (Liberation Sans -> Arial).

**How to force Office names as the primary.** If reviewers will be confused by seeing "Liberation Mono" in the Word font dropdown (cosmetic concern; doesn't affect rendering), pass an explicit `length>1` stack with the Office name first. The resolver returns the vector verbatim — no alias lookup, no chain expansion — so the RTF file then names the Office face as primary and your chosen alternate as `\\*\falt`:

```
preset(font_family = c("Courier New", "Courier", "Liberation Mono"))
```

This is the canonical escape hatch for authors who know their consumer audience is Mac / Windows Word users and want the dropdown to show the Office face directly.

- orientation — page orientation. `<character(1)>`. One of "landscape" (default), "portrait".
- paper\_size — paper key. `<character(1)>`. One of "letter" (default), "a4".
- margins — page margins in inches. `<numeric(1) | numeric(4)>`. Length 1 = all four sides; length 4 = top, right, bottom, left.
- pagehead, pagefoot — per-page header / footer band content. `<list>`. Each band is a named list with slots from left / center / right; every other slot name is rejected. Each slot accepts NULL (omit), a character scalar, a character vector (multi-row content), or an `inline_ast`. Empty `list()` (the default) -> no band emitted.

**Single-row form** (scalar slots):

```
pagehead = list(
 left = "Protocol: ABC-123",
 center = "Draft",
 right = "Page {page} of {npages}"
)
```

**Multi-row form** (vector slots, index-aligned):

```
pagehead = list(
 left = c("Protocol: ABC-123", "Analysis Set: Safety"),
 right = "Page {page} of {npages}" # scalar -> body-edge row
)
```

**Growth direction.** Vector index 1 = body edge; index N = far from body. pagehead rows stack **upward** away from the table (the row closest to the

table is index 1). pagefoot rows stack **downward** away from the table (the row closest to the table is index 1). Shorter slots pad with "" at the FAR end (high index), so a scalar slot naturally lands on the body-edge row.

**Token vocabulary** — substituted into slot text:

| Token          | Phase   | Expansion                           |
|----------------|---------|-------------------------------------|
| {page}         | backend | current page number (field code)    |
| {npages}       | backend | total page count (field code)       |
| {program}      | engine  | calling script's base name          |
| {program_path} | engine  | calling script's full path          |
| {datetime}     | engine  | DDMMYYYY HH:MM:SS UTC (render time) |

{program}, {program\_path}, and {datetime} resolve once per render (at `as_grid()` / `emit()`); {page} and {npages} resolve per page (filled in by Word / xelatex / the browser's print engine at view time). The program tokens walk a 5-mode detection chain — RStudio API, source() frame, Rscript / R CMD BATCH commandArgs (covers Domino + Linux batch + CI), knitr current\_input, fallback "<interactive>".

- **rules** — the single border vocabulary (replaces the old borders knob). String sugar "booktabs" (default, the clinical baseline), "grid", "frame", "none"; a single `brdr()` broadcast to every active rule; or a named list keyed by the nine rule names (toprule, midrule, bottomrule, spanrule, rowrule, footnoterule, leftrule, rightrule, colrule) — unlisted rules keep their default, and the bare string "none" drops one. `rules = list(rowrule = brdr())` reproduces the old `hlines = "all"`. **bottomrule vs footnoterule**. These are mutually exclusive: exactly one rule sits at the data -> footnote boundary. The default is bottomrule (the table's bottom edge); footnoterule (a table-width rule opening the footnote section) is OFF by default. As a distinct footnote-section rule, footnoterule is drawn only by the paginated backends — **RTF, LaTeX / PDF, and DOCX**. The **HTML** backend is continuous (non-paginated) and has no separate footnote section, so it folds both into one rule: whichever of bottomrule / footnoterule is active becomes the table's bottom edge (bottomrule wins when both are set). Setting footnoterule therefore still draws a closing rule on HTML — rendered as the bottomrule.
- **spacing** — region-keyed blank-line control. A named list keyed by title / body / subgroup / footnote, each a named numeric c(above = , below = ) (footnote: above only). Default is the Appendix-I one blank line above and below the title block. Two adjoining region-sides that target the same physical gap resolve to the MAX (never the sum), so a gap is never accidentally doubled.
- **stripe** — zebra body-row fills. A single colour (applied to even rows) or a named c(odd = , even = ); NULL (default) is off.
- **indent\_size** — row-label indent width, in monospace- space units. `<integer(1)>`. Default 2L. Each indent level adds this many space-widths of left padding to the cell. 0L disables the indent prefix entirely. Backends with native padding-left semantics (HTML / LaTeX / RTF / DOCX / PDF) emit this

as cell padding so wrapped continuation lines align with the indented baseline; Markdown carries the literal space-prefix. Block alignment for the title / footnote / header / subgroup / body surfaces is set via the alignment named-list knob (`alignment = list(title_halign = "left", ...)`), not a scalar knob; blank-line spacing is set via `spacing` (above).

- `na_text` — global NA fallback. `<character(1)>`.
- `decimal_metrics` — decimal-padding metric. `<character(1)>`. Only "chars" (default); the engine pads decimal columns by character count.
- `decimal_markers` — missing-value tokens recognised by `col_spec(aligned = "decimal")`. `<character>`. Default `c("NR", "NE", "NC", "ND", "BLQ")`. A cell whose trimmed value is one of these is treated as a non-numeric *marker*: it is shown and right-aligned in the column rather than parsed as a number, and a marker appearing inside a compound (e.g. the upper bound of 14.3 (11.2, NR)) is preserved and slot-aligned. Excludes "-", "NA", and "INF"/"-INF" by default: "-" collides with range separators, "NA" is handled by `na_text`, and infinities are real values. Set to `character(0)` to disable marker handling.
- `width_mode` — table-level column-sizing policy. Mirrors Word's Table Layout menu. `<character(1)>`. One of:
  - "content" (*default*) — Each column auto-sized to `max(body, header)`. The table doesn't fill the page. Word's "Auto-fit Contents".
  - "window" — Auto-sized columns expand to share the residual page width equally. Pinned and percent columns keep their pins. Word's "Auto-fit Window".
  - "fixed" — Only explicit per-column widths drive the layout. Auto-sized columns collapse to a minimum sliver. Word's "Fixed Column Width".

**Interaction:** Pair with `col_spec(width = ...)` pins to drive the layout under "window" / "fixed". Under "content", pins still take priority over auto columns.

**HTML backend.** `width_mode` drives paper backends (LaTeX / RTF / PDF / DOCX) only. HTML is unconditionally responsive — the table always fills its parent and columns wrap when the viewport narrows, regardless of `width_mode`. Per-column widths (`col_spec(width)`) emit verbatim into the HTML colgroup per the `gt` convention.

- `whitespace` — how significant ASCII spaces in labels and cells render. `<character(1)>`. One of:
  - "preserve" (*default*) — leading, trailing, and interior runs of 2+ spaces become the backend non-breaking token (`&nbsp;`; `/~/\~;`; DOCX preserves via `xml:space`), so a hand-built indent like `col_spec(label = "Placebo")` renders verbatim across every backend. A single interior space stays breakable, so cells still wrap.
  - "collapse" — leave the backend's native run-folding in place (HTML / md / LaTeX collapse runs to one space).

**Note:** never affects `col_spec(aligned = "decimal")` padding, which uses `U+00A0` and is preserved unconditionally.

- `footnote_markers` — the glyph scheme for `footnote()` markers, which the engine allocates once in reading order. `<character(1)>`. One of:
  - “letters” (*default*) — a, b, ..., z, aa, ab, ... (bijective base-26).
  - “numbers” — 1, 2, 3, ...
  - “symbols” — L<sup>a</sup>mpo<sup>r</sup>t’s sequence \*, †, ‡, §, ¶, ||, then doubled (\*\*, ††, ...) once it spills past the sixth.

**Interaction:** a note’s *anchor* is fixed by `footnote()`; its *scheme* (this knob) and *label* (`footnote_label`) are resolved from the active preset at render, so flipping either re-letters every marker at once.
- `footnote_label` — block-line template for a `footnote()` marker. `<character(1)>`. Default “{m}”; the {m} token is replaced by the allocated marker, so “[{m}]” prints [a] ahead of the note text on the footnote line.
- `cell_padding` — cell padding in points, CSS shorthand of length l / 2 / 4 (all | vertical horizontal | top right bottom left), parsed by the same length rule as margins. `<numeric>`: default c(0, 5.4) (vertical 0, horizontal 5.4pt). The single source of truth for both auto column-width measurement (left + right) and every backend’s cell margin, so measured and rendered widths agree.
 

**Interaction:** A body per-side padding override (`preset(padding = list(body = ...))` or `style(.at = cells_body(), padding = ...)`) takes precedence at both measurement and render.

**Note:** DOCX and LaTeX render left / right exactly; RTF (`\trgaph` is one symmetric gap) renders the average, so the total width still matches but the two sides look equal.

# Landscape A4, 8pt body, slim margins for one wide table.

```

preset(
 orientation = "landscape",
 paper_size = "a4",
 font_size = 8,
 margins = c(0.75, 0.5, 0.75, 0.5)
)

```

- `.template` *A preset\_spec to bulk-apply before ...* `<preset_spec | NULL>`: default NULL. When supplied, every knob the template has set away from its factory default feeds in as the base layer; user-supplied ... knobs then merge on top. List-valued knobs (rules, fonts, colors, padding, alignment) shallow-merge per key; scalars replace. Use this to layer a house-style preset\_spec onto a chain without restating its knobs.
- `.style` *A style\_template() to layer onto the cascade.* `<style_template | NULL>`: default NULL. When supplied, every layer the template has accumulated via `style()` is replayed in order at engine time, after the per-spec `style()` layers on .spec. Use this to attach a sponsor’s reusable house style to a chain without restating every per-region rule.
- `.reset` *Discard the spec’s existing preset before applying ...* `<logical(1)>`: default FALSE. When TRUE, the spec’s prior preset\_spec (if any) is dropped and ... knobs are merged onto fresh preset\_spec() defaults. With no knobs, the per-spec preset is cleared back to NULL (the spec falls through to `set_preset()` or `preset_spec()` defaults).

## Details

**Per-spec, chained.** `preset()` is the per-spec override — a verb that returns a modified spec, composable on the pipe alongside `cols()` / `headers()` / `paginate()`. Use it when a single table needs a one-off geometry (e.g. landscape A4 for one wide efficacy summary inside a portfolio of portrait letter tables).

**Merge, not replace.** A second `preset()` call merges its scalar knobs onto the spec's existing preset; unspecified knobs keep their prior value. The five named-list knobs (`alignment` / `rules` / `fonts` / `colors` / `padding`) lower to `style_layer` records on `preset@style` via `.preset_args_to_layers()` (internal) and append in call order; layer order is precedence within the engine cascade, so a later `preset()` call's lowered attribute wins over an earlier one at the cell. Pass `.reset = TRUE` to discard the existing knobs and start from `preset_spec()` defaults. `preset(.spec, .reset = TRUE)` with no knobs clears the per-spec override entirely (the spec then falls through to `set_preset()` or `preset_spec()` defaults at render time).

**Direct `preset_spec()` calls bypass lowering.** The five named-list knobs are no longer slots on the `preset_spec S7` class — they exist only as `preset()` / `set_preset()` arguments that lower into `@style`. `preset_spec(rules = list(...))` (and analogous direct calls) raise "unused argument". Wrap such calls in `tabular(...)|>preset(...)` so the lowering helper fires and the layers land on `@style`.

**Cascade with `set_preset()`.** The engine resolves the active preset in this order: (1) the spec's per-call preset (this verb), (2) the session default attached via `set_preset()`, (3) `preset_spec()` factory defaults. The first non-NULL layer wins; layers are not field-merged across the cascade.

## Value

*The updated* `tabular_spec`. Continue chaining with `paginate()`, `style()`, then render via `emit()` (or resolve without I/O via `as_grid()`).

## See Also

**Session-scope partners:** `set_preset()`, `get_preset()`.

**Render-geometry consumer:** `paginate()` derives the per-page row budget from the active preset's paper, orientation, margins, and font size.

**Sibling build verbs:** `cols()` / `col_spec()`, `headers()`, `sort_rows()`, `style()`.

**Entry / terminal verbs:** `tabular()`, `emit()`, `as_grid()`.

## Examples

```
---- Example 1: Landscape A4 for a wide efficacy table ----
#
BOR table where the four-arm column block fits portrait letter
with a smaller body font, but the sponsor wants A4 landscape at
8pt for visual breathing room. `preset()` attaches the geometry;
`paginate()` reads it later to size the per-page row budget.
bor_levels <- c(
 "CR", "PR", "SD", "NON-CR/NON-PD", "PD", "NE", "MISSING",
 "ORR (CR + PR)", "CBR (CR + PR + SD)",
 "DCR (CR + PR + SD + NON-CR/NON-PD)", "95% CI (Clopper-Pearson)"
)
```

```

eff <- cdisc_eff_resp
eff$stat_label <- factor(eff$stat_label, levels = bor_levels)
ne <- stats::setNames(cdisc_eff_n$n, cdisc_eff_n$arm_short)

tabular(
 eff,
 titles = c(
 "Table 14.2.1",
 "Best Overall Response and Response Rates",
 "Efficacy Evaluable Population"
),
 footnotes = "Response per RECIST 1.1, investigator assessment."
) |>
cols(
 stat_label = col_spec(label = "Response"),
 row_type = col_spec(visible = FALSE),
 groupid = col_spec(visible = FALSE),
 group_label = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={ne['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={ne['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={ne['drug_100']}")
) |>
sort_rows(by = c("groupid", "stat_label")) |>
preset(
 orientation = "landscape",
 paper_size = "a4",
 font_size = 8
) |>
paginate()

---- Example 2: Per-spec override with per-page chrome ----
#
The submission session sets a portrait letter 9pt default (typical
safety-table geometry). One particular AE table needs landscape
for a long PT label band; the per-spec `preset()` overrides only
orientation. The same per-spec call wires the canonical
per-page header band (protocol on the left, page X of Y on the
right) and a footer band that auto-resolves the calling
script's name and the current render timestamp via the
`{program}` and `{datetime}` tokens.
set_preset(font_size = 9, paper_size = "letter")

ae <- cdisc_saf_aesocpt
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))
ae$n_total <- as.integer(sub(".*", "", ae$Total))
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

tabular(
 ae,
 titles = c(
 "Table 14.3.1",
 "Adverse Events by System Organ Class and Preferred Term",
 "Safety Population"
)
)

```

```

),
 footnotes = "Subjects are counted once per SOC and once per PT."
) |>
cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(usage = "group", visible = FALSE,
 group_display = "column_repeat"),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={n['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}"),
 Total = col_spec(label = "Total\nN={n['Total']}")
) |>
headers("Treatment Group" = c("placebo", "drug_50", "drug_100", "Total")) |>
sort_rows(by = c("row_type", "n_total"), descending = c(FALSE, TRUE)) |>
preset(
 orientation = "landscape",
 pagehead = list(
 left = "Protocol: ABC-123",
 right = "Page {page} of {npages}"
),
 pagefoot = list(
 left = "{program}",
 right = "{datetime}"
)
) |>
paginate(keep_together = "soc")

Reset the session default so subsequent examples / R sessions
are not affected.
set_preset(.reset = TRUE)

```

---

```

preset_minimal

```

```

Minimal theme: one header rule, normal weight throughout

```

---

## Description

Apply the stripped-down table look in one verb. The column-label divider (`midrule`) becomes the only rule drawn, and every bold-by-default surface renders in normal weight: the title block, the column-header band, the subgroup banner, and the section-header rows synthesized for `usage = "group"` columns. The analogue of `ggplot2`'s `theme_minimal()`, composable on the pipe between the build verbs and the terminal `emit()` / `as_grid()`.

## Usage

```

preset_minimal(.spec, ...)

```

## Arguments

- `.spec`      *The tabular\_spec to apply the minimal theme to.* `<tabular_spec>`: required. Dot-prefixed so partial matching cannot bind a `...` knob to the spec slot.
- `...`      *Named preset knobs.* Forwarded verbatim to `preset()` (e.g. `font_size`, `font_family`, `orientation`, `paper_size`, `margins`), so a single call sets both the minimal look and the page geometry.
- Restriction:** the `rules` (and legacy borders) knob is owned by this helper and may not be passed here; call `preset()` directly for a custom rule set.

## Details

**What it sets**, both at theme (lowest) precedence so an explicit later `style()` wins:

1. **Rules.** Drops the booktabs `toprule` and `bottomrule` (the outer frame), keeping the `midrule` under the column labels and the muted column-spanner `spanrule`. Equivalent to `preset(rules = list(toprule = "none", bottomrule = "none"))`.
2. **Weight.** Sets `bold = FALSE` on the title, column-header, subgroup-label, and group-header surfaces. The HTML backend overrides its `font-weight: 600` class default with an inline `font-weight: normal`; the paginated backends (RTF / LaTeX / PDF / DOCX) suppress the surface's bold run.

**Last verb wins.** Because the weight layers ride the theme tier, a later explicit `style(bold = TRUE, .at = cells_title())` (or any surface) re-bolds it. Treat `preset_minimal()` as the theme baseline and override individual surfaces afterwards.

**Markdown.** GFM cannot represent colour / background / font on a surface; rendering a styled surface to `.md` emits a one-time `tabular_warning_fidelity` and degrades gracefully. Weight (bold) and italic carry through.

## Value

*The updated* `tabular_spec`. Continue chaining with `paginate()` / `style()`, then render via `emit()` (or resolve without I/O via `as_grid()`).

## See Also

**Underlying verbs:** `preset()` (the rule presets "booktabs" / "grid" / "frame" / "none" live there as rules string sugar), `style()`.

**Target the surfaces it touches:** `cells_title()`, `cells_headers()`, `cells_subgroup_labels()`, `cells_group_headers()`.

**Entry / terminal verbs:** `tabular()`, `emit()`, `as_grid()`.

## Examples

```
---- Example 1: Minimal AE overall summary ----
#
The overall adverse-event summary with a single rule under the
column labels and no bold anywhere. `preset_minimal()` is the theme
baseline; the page stays at the session default geometry.
```

```

demo_n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

tabular(
 cdisc_saf_ae,
 titles = c(
 "Table 14.3.1",
 "Overall Summary of Adverse Events",
 "Safety Population"
),
 footnotes = "Subjects counted once per category."
) |>
cols(
 stat_label = col_spec(usage = "group", label = "Category"),
 placebo = col_spec(label = "Placebo\nN={demo_n['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={demo_n['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={demo_n['drug_100']}"),
 Total = col_spec(label = "Total\nN={demo_n['Total']}")
) |>
preset_minimal()

---- Example 2: Section headers normal, then re-bold the title ----
#
AE by SOC / PT with the SOC as a section-header row. Under
`preset_minimal()` the SOC section labels render in normal weight
(not the default bold); a trailing `style()` re-bolds only the
title (last verb wins), and `font_size` forwards through `...`.
ae <- cdisc_saf_aesocpt
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))

tabular(
 ae,
 titles = c("Table 14.3.2", "Adverse Events by SOC and Preferred Term"),
 footnotes = "Subjects counted once per SOC and once per PT."
) |>
cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(usage = "group", group_display = "header_row"),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={demo_n['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={demo_n['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={demo_n['drug_100']}"),
 Total = col_spec(label = "Total\nN={demo_n['Total']}")
) |>
preset_minimal(font_size = 8) |>
style(bold = TRUE, .at = cells_title())

```

## Description

Renders a `tabular_spec` interactively. The default behaviour mirrors `gt::gt()`: convert the spec to an `htmltools` tag list and let `htmltools` dispatch — RStudio + Positron viewer panes, Quarto / Rmd notebook inline, Databricks `displayHTML`, and plain-console `cat()` are all handled without any IDE- specific branching.

## Arguments

|                     |                                                                                                                                                                                                                                                                                                                                               |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>x</code>      | <i>The tabular_spec to render.</i> <code>&lt;tabular_spec&gt;</code> : required. The same object you'd hand to <code>emit()</code> .                                                                                                                                                                                                          |
| <code>...</code>    | <i>Forwarded to</i> <code>htmltools::print / as.tags()</code> . Use this to pass id, style, class overrides to the wrapping <code>&lt;div&gt;</code> .                                                                                                                                                                                        |
| <code>view</code>   | <i>Open the viewer?</i> <code>&lt;logical(1)&gt;</code> : default <code>interactive()</code> . Same role as <code>gt::gt</code> 's 'view' argument: passes through to <code>htmltools</code> as 'browse = view'. Set 'view = FALSE' to suppress the viewer for one call (e.g. to capture the HTML string without launching a window).         |
| <code>output</code> | <i>Force a specific preview format.</i> <code>&lt;character(1)   NULL&gt;</code> : default <code>NULL</code> (auto). See the output <b>argument</b> section above for the full list. The session default can be set via <code>options(tabular_print_output = "cli")</code> for users who prefer the structural summary over the HTML preview. |

## Details

**Dispatch.** `print()` delegates to `as.tags.tabular_spec()` which returns an `htmltools::tagList`. That tag list is handed to `htmltools`'s own `print` method with `browse = view`: `htmltools` opens the IDE viewer when one is registered, inlines under a Quarto / Rmd chunk when running inside one, or `cat()`s the HTML when neither applies. No `is_rstudio()` / `is_positron()` / `is_notebook()` heuristics — `htmltools` already knows.

**view argument.** Defaults to `interactive()`, the same universal off-switch `gt::gt()` uses. Non-interactive contexts (Rscript, R CMD check, CI, `devtools::test`) bypass the viewer automatically. Pass `view = FALSE` explicitly at an interactive prompt to suppress the viewer for a single call.

**output argument.** Forces a specific preview format instead of the default HTML-via-`htmltools` path. One of:

- "html" — same as the default, but explicit.
- "md" / "markdown" — `cat()` the markdown source to the console (round-trips through `backend_md`).
- "latex" — `cat()` the markdown source as a temporary placeholder (real LaTeX preview lands with `backend_latex`).
- "rtf" / "docx" / "pdf" — render an HTML preview and emit a cli note pointing at `emit()` for the real artefact. The viewer pane cannot render RTF / OOXML, and we deliberately do *not* compile through `tinytex` on every autoprnt.
- "cli" — print the structural cli-tree summary (props, headers, sort, pagination, preset). Useful for debugging spec composition without paying the HTML render cost.

**Robustness.** The HTML render is wrapped in `tryCatch`; if rendering fails for any reason the printer falls back to the cli-tree summary and a `cli::cli_warn()` describing the failure, so a broken spec never crashes the REPL.

**Tempdir.** Preview HTML files live under `getOption("tabular_preview_dir", default = tempdir())`. Override the option to keep them in a stable location (handy on Linux distros where browsers don't have read access to `/tmp/`).

### Value

*Invisibly returns* x. Side effect: opens the viewer, inlines under a chunk, or `cat()`'s output.

### See Also

**Tag conversion:** `as.tags.tabular_spec()` — the `htmltools` tag list that `print()` delegates to. Call it directly to embed the table in a custom `htmltools::tagList` or Shiny UI.

**Terminal verb:** `emit()` writes the resolved artefact to disk; `print()` is for in-session preview only.

**Pipeline shape:** `as_grid()` resolves the engine pipeline to a `tabular_grid` without I/O.

### Examples

```
---- Example 1: Build + autoprint (HTML preview) ----
#
Build a spec and let autoprint render it. Inside RStudio /
Positron the HTML lands in the viewer pane; inside a
Quarto / Rmd chunk it inlines under the chunk; at a plain
console the HTML source is `cat()`-ed.
tabular(
 cdisc_saf_demo,
 titles = c("Table 14.1.1", "Demographics"),
 footnotes = "Safety Population."
)

---- Example 2: Force the cli-tree structural view ----
#
The cli-tree summary shows props at a glance. Useful for
debugging spec composition without paying the HTML render
cost.
spec <- tabular(cdisc_saf_demo, titles = "Demographics") |>
 cols(variable = col_spec(usage = "group", label = "Characteristic"))

print(spec, output = "cli")
```

---

set\_preset

*Set or clear the session default preset*

---

### Description

Stash a `preset_spec` in the package-internal session environment. Every subsequent `tabular()` chain that does not attach its own `preset()` inherits these knobs at render time. Mirrors `ggplot2::theme_set()`: one call up front, many tables downstream.

**Usage**

```
set_preset(new = NULL, ..., .template = NULL, .style = NULL, .reset = FALSE)
```

**Arguments**

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>new</code>       | <i>A preset_spec to install wholesale.</i> <preset_spec   NULL>: default NULL. When non-NULL, replaces the session preset in one call without touching knobs. The primary use is the save/restore round-trip ( <code>old &lt;- set_preset(...)</code> ; <code>set_preset(old)</code> ) — <code>new</code> accepts any <code>preset_spec</code> previously returned by <code>set_preset()</code> or <code>get_preset()</code> . Mutually exclusive with <code>...</code> , <code>.template</code> , <code>.style</code> , <code>.reset</code> : passing any of those alongside a non-NULL <code>new</code> raises <code>tabular_error_input</code> . |
| <code>...</code>       | <i>Named preset knobs.</i> Same shape as <code>preset()</code> ; see that verb for the full list of 13 recognised knobs. Unknown names raise <code>tabular_error_input</code> . Mutually exclusive with a non-NULL <code>new</code> .                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>.template</code> | <i>A preset_spec to bulk-apply before ...</i> <preset_spec   NULL>: default NULL. Same semantics as <code>preset()</code> 's <code>.template</code> : every knob set away from its factory default feeds in as the base layer; user-supplied <code>...</code> knobs then merge on top with shallow-merge per list-valued knob.                                                                                                                                                                                                                                                                                                                      |
| <code>.style</code>    | <i>A style_template() to layer into the session default.</i> <style_template   NULL>: default NULL. Same semantics as <code>preset()</code> 's <code>.style</code> : the template's accumulated layers feed in as session-default style, layered before any per-spec <code>style()</code> calls.                                                                                                                                                                                                                                                                                                                                                    |
| <code>.reset</code>    | <i>Discard the existing session preset before applying ...</i> <logical(1)>: default FALSE. With no knobs, clears the session default back to NULL.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |

**Details**

**Persistence.** The session preset lives in a package-internal environment populated when `tabular` is loaded and emptied when the namespace unloads. There is no on-disk persistence; set the default at the top of each analysis script (or in a project-level `.Rprofile`) when a sticky house style is needed.

**Merge, not replace.** A second `set_preset()` call merges its knobs onto the existing session preset; unspecified knobs keep their prior value. Pass `.reset = TRUE` to discard the existing session preset and start from `preset_spec()` defaults. `set_preset(.reset = TRUE)` with no knobs clears the session default back to NULL.

**Save and restore.** Every call returns the *previous* session preset invisibly, the same primitive `ggplot2`'s `ggplot2::theme_set()` ships. Capture it once, render, and restore by passing the saved value back as the positional `new` argument:

```
old <- set_preset(font_size = 10, paper_size = "a4")
... one renegade render at 10pt A4 ...
set_preset(old) # restore
```

When the prior was NULL (no session preset ever attached), the restore is `set_preset(.reset = TRUE)` instead — `set_preset(NULL)` is the same shape as `set_preset()` and falls through to factory defaults rather than clearing the session.

**Cascade with `preset()`.** A per-spec `preset()` always wins over the session default. The session default fills in only when the spec carries no preset of its own.

**Value**

The previous session `preset_spec` (*invisible*). Returns NULL when no session preset was attached prior to the call. Capture it to round-trip a temporary override: `old <- set_preset(...); set_preset(old)`. Mirrors `ggplot2::theme_set()` and `base::options()` — the canonical tidyverse save/restore primitive.

**See Also**

**Per-spec partner:** `preset()` — overrides the session default on one chain.

**Inspect:** `get_preset()`.

**Entry / terminal verbs:** `tabular()`, `emit()`, `as_grid()`.

**Examples**

```
---- Example 1: Sticky session default for an analysis script ----
#
The submission's safety tables all use portrait letter, 9pt
Times New Roman with 1-inch margins. Set once at the top of the
analysis script and every `tabular()` chain inherits it - no
per-table `preset()` call needed unless one table deviates.
set_preset(
 font_size = 9,
 font_family = "Times New Roman",
 orientation = "portrait",
 paper_size = "letter",
 margins = 1
)

Subsequent tabular() chains pick up the session preset at render.
demo_n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
tabular(
 cdisc_saf_ae,
 titles = c(
 "Table 14.3.1",
 "Overall Summary of Adverse Events",
 "Safety Population"
),
 footnotes = "Subjects counted once per category."
) |>
cols(
 stat_label = col_spec(usage = "group", label = "Category"),
 placebo = col_spec(label = "Placebo\nN={demo_n['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={demo_n['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={demo_n['drug_100']}"),
 Total = col_spec(label = "Total\nN={demo_n['Total']}")
)

---- Example 2: Reset the session default mid-script ----
#
The first half of the script produces safety tables at 9pt; the
second half produces efficacy tables at 10pt on landscape A4. A
```

```

single `set_preset(.reset = TRUE, ...)` resets the cascade before
the second batch starts.
set_preset(font_size = 9, paper_size = "letter")
get_preset()@font_size # 9

set_preset(
 .reset = TRUE,
 font_size = 10,
 orientation = "landscape",
 paper_size = "a4"
)
get_preset()@orientation # "landscape"

Reset the session default so subsequent examples / R sessions
are not affected.
set_preset(.reset = TRUE)

---- Example 3: Save and restore around a renegade table ----
#
Most of the submission renders portrait letter at 9pt. One
renegade efficacy table needs landscape A4 at 10pt. Capture
the prior session preset, render the renegade, then restore.
set_preset(font_size = 9, paper_size = "letter")

old <- set_preset(
 font_size = 10,
 paper_size = "a4",
 orientation = "landscape"
)
... one renegade render ...
if (is.null(old)) {
 set_preset(.reset = TRUE) # was no prior - clear
} else {
 set_preset(old) # round-trip via the positional `new` arg
}
get_preset()@paper_size # "letter" - restored

---- Example 4: Snapshot current preset, mutate, restore ----
#
Capture whatever the session preset is right now (may be NULL),
let a downstream helper mutate it, then put it back when done.
set_preset(font_size = 9, paper_size = "letter")
snapshot <- get_preset()

Simulate downstream code mutating session state.
set_preset(font_size = 11, orientation = "landscape")

Restore. The wholesale-install path of `set_preset(new)`
accepts any `preset_spec` returned by `get_preset()` /
`set_preset()`.
if (is.null(snapshot)) {
 set_preset(.reset = TRUE)
} else {

```

```

 set_preset(snapshot)
}
get_preset()@font_size # 9 - restored

Reset for subsequent examples / R sessions.
set_preset(.reset = TRUE)

```

---

 sort\_rows

*Sort the display rows*


---

## Description

Attach a `sort_spec` to a `tabular_spec`. The engine applies the sort before pagination, so by may reference any column in `spec@data` whether or not the column is declared in `cols()`.

## Usage

```
sort_rows(.spec, by = character(), descending = FALSE)
```

## Arguments

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>.spec</code>      | <i>The tabular_spec to attach the sort to.</i> <tabular_spec>: required.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>by</code>         | <i>Ordered column names to sort by, in precedence order.</i> <character>: default <code>character()</code> . Length 0 is accepted (no-op sort). May reference columns not declared in <code>cols()</code> — sort-only helper columns ride along through the engine.<br><b>Restriction:</b> Every entry must be a column in <code>spec@data</code> . Cannot reference arm columns produced by <code>pivot_across()</code> ; pivot upstream of the sort instead. Arm cells hold rendered stat strings (e.g. "75.2 (8.3)") that do not order meaningfully.<br><br># Two-key clinical sort: row_type ascending, n_total descending.<br><code>sort_rows(by = c("row_type", "n_total"), descending = c(FALSE, TRUE))</code> |
| <code>descending</code> | <i>Per-key sort direction.</i> <logical(1)   logical(length(by))>: default FALSE. TRUE sorts the corresponding key descending; length 1 recycles to every key.<br><b>Restriction:</b> No NAs. Length must be 1 or <code>length(by)</code> . <b>Tip:</b> For mixed-direction multi-key sorts, pass <code>length(by)</code> values; the engine inverts the <code>xtfrm</code> rank of each descending key and calls <code>order()</code> once on all keys.                                                                                                                                                                                                                                                              |

## Details

**Replace, not stack.** A second `sort_rows()` call REPLACES the prior sort — sort is a single spec, not a stackable list. Call with no arguments to clear.

**NA last, regardless of direction.** NA values in a sort key are placed at the end whether the key is ascending or descending (matching `order(..., na.last = TRUE)`).

**Factor levels drive the order.** Factor columns sort by factor levels, not by the character label. The CDISC BOR ordering (CR < PR < SD < NON-CR/NON-PD < PD < NE < MISSING) survives a tabular pipeline without an explicit `mutate()` — coerce `stat_label` to a factor with the levels in clinical order upstream, then `sort_rows(by = "stat_label")` does the rest.

**Value**

The updated `tabular_spec`. Continue chaining with `style()`, `paginate()`, `preset()`, then render via `emit()` (or resolve without I/O via `as_grid()`).

**See Also**

**Sibling build verbs:** `cols()` / `col_spec()`, `headers()`, `style()`, `paginate()`, `preset()`.

**Entry / terminal verbs:** `tabular()`, `emit()`, `as_grid()`.

**Examples**

```
---- Example 1: AE table sorted by SOC, then by descending subject count ----
#
AE-by-SOC/PT table where the SOCs and PTs appear in descending
order of subject count within the row-type hierarchy (overall
first, then SOCs, then PTs). `cdisc_saf_aesocpt$Total` cells are
formatted text ("171 (67.3)"), so a lexical sort on `Total`
would be wrong ("14" < "171" < "29") - attach a numeric rank
column upstream and sort on (row_type, n_total).
ae <- cdisc_saf_aesocpt
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))
ae$n_total <- as.integer(sub(".*", "", ae$Total))
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

tabular(
 ae,
 titles = c(
 "Table 14.3.1",
 "Adverse Events by System Organ Class and Preferred Term",
 "Safety Population"
),
 footnotes = "Subjects are counted once per SOC and once per PT."
) |>
cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={n['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}"),
 Total = col_spec(label = "Total\nN={n['Total']}")
) |>
sort_rows(by = c("row_type", "n_total"), descending = c(FALSE, TRUE))

---- Example 2: BOR table in CDISC factor order ----
#
Efficacy BOR table that must appear in CDISC clinical order
(CR < PR < SD < NON-CR/NON-PD < PD < NE < MISSING), then the
derived ORR / CBR / DCR rate rows ordered by `groupid`,
```

```

not alphabetical. `cdisc_eff_resp$stat_label` arrives as character, so
coerce to a factor with the canonical levels upstream and
`sort_rows()` uses those levels directly.
bor_levels <- c(
 "CR", "PR", "SD", "NON-CR/NON-PD", "PD", "NE", "MISSING",
 "ORR (CR + PR)", "CBR (CR + PR + SD)",
 "DCR (CR + PR + SD + NON-CR/NON-PD)", "95% CI (Clopper-Pearson)"
)
eff <- cdisc_eff_resp
eff$stat_label <- factor(eff$stat_label, levels = bor_levels)
ne <- stats::setNames(cdisc_eff_n$n, cdisc_eff_n$arm_short)

tabular(
 eff,
 titles = c(
 "Table 14.2.1",
 "Best Overall Response and Response Rates",
 "Efficacy Evaluable Population"
),
 footnotes = "Response per RECIST 1.1, investigator assessment."
) |>
cols(
 stat_label = col_spec(label = "Response"),
 row_type = col_spec(visible = FALSE),
 groupid = col_spec(visible = FALSE),
 group_label = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={ne['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={ne['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={ne['drug_100']}")
) |>
sort_rows(by = c("groupid", "stat_label"))

---- Example 3: Mixed-direction multi-key sort with hidden helper ----
#
Demographics-style table sorted by `variable` ascending and a
hidden numeric key descending. The `descending` argument takes
one value per `by` entry so each key can flip direction
independently. The helper column rides in `spec@data` for the
sort but never renders (visible = FALSE on its col_spec).
demo <- cdisc_saf_demo
demo$display_order <- match(demo$variable, unique(demo$variable))

tabular(demo, titles = "Demographics, ranked within section") |>
cols(
 variable = col_spec(usage = "group", label = "Characteristic"),
 stat_label = col_spec(label = "Statistic"),
 display_order = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal"),
 Total = col_spec(label = "Total", align = "decimal")
) |>
sort_rows(

```

```

 by = c("display_order", "stat_label"),
 descending = c(FALSE, TRUE)
)

---- Example 4: Hierarchical SOC -> PT sort with factor outer key ----
#
A factor outer key locks the SOC display order to the canonical
interleaved sequence (`overall` first, then `soc` blocks, then
`pt` detail rows inside each SOC) regardless of input order. The
numeric inner key sorts PTs within each SOC by descending total
subject count.
ae <- cdisc_saf_aesocpt
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))
ae$n_total <- as.integer(sub(".*", "", ae$Total))
tabular(ae, titles = "AE by SOC and PT, ranked within SOC") |>
 cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal"),
 Total = col_spec(label = "Total", align = "decimal")
) |>
 sort_rows(
 by = c("row_type", "soc", "n_total"),
 descending = c(FALSE, FALSE, TRUE)
)

```

---

style

*Attach a style layer to a tabular\_spec or style\_template*


---

## Description

One verb, one cascade. Each `style()` call appends a single `style_layer` (location + style attributes) to the `spec` or `template`. Layers accumulate in declaration order; the engine merges them at render time so later layers win per attribute, NA-valued fields leave the prior layer intact.

## Usage

```
style(.spec, ..., .at = cells_body())
```

## Arguments

`.spec` *A tabular\_spec OR a tabular\_style\_template.* `<tabular_spec | tabular_style_template>`: `re`  
 Dot-prefixed so R's partial argument matching cannot accidentally bind a short attribute name in `...` to the `spec` slot. When piping through `style_template()` `|> style(...)` layers accumulate onto the template instead of a `spec`.

... *Named style attributes.* At least one required. See the vocabulary list above for recognised names.

.at *Location object selecting which surface the layer targets.* <tabular\_location>: default cells\_body()  
Build with one of the cells\_\*() constructors; see [cells\\_body\(\)](#) and siblings.  
Dot-prefixed (tidyverse convention) because it comes AFTER ... — that way a user-passed style attribute can never collide with this arg's name.

## Details

**Locations.** The at argument selects which surface the layer targets. Every region of the rendered page has a cells\_\*() constructor:

- cells\_body() — body cells (default)
- cells\_headers() — column header band
- cells\_group\_headers() — synthetic group-header rows
- cells\_title() — title block
- cells\_subgroup\_labels() — subgroup banner row
- cells\_footnotes() — footnote block
- cells\_pagehead() — page-header band
- cells\_pagefoot() — page-footer band
- cells\_table() — table-wide regions (outer borders, body-row separators)

Body filters live on cells\_body():  $i = 1:3$  for integer-index rows,  $j = \text{"Total"}$  for column-name targeting, where  $= <expr>$  for a quosure-captured predicate evaluated against spec@data.

**Attribute vocabulary.** Each layer carries a style\_node built from .... Recognised attribute names:

- Text — bold, italic, underline, color, background, font\_family, font\_size
- Alignment — halign("left" / "center" / "right"), valign("top" / "middle" / "bottom")
- Borders — border (umbrella), border\_top, border\_bottom, border\_left, border\_right (each takes a brdr() value or the literal "none"); per-side scalars border\_<side>\_{style,width,color} for finer control
- Padding — padding (a scalar applies to all four sides; a named vector c(top = , right = , bottom = , left = ) sets each side); or the per-side scalars padding\_<side> directly
- Spacing — blank\_above, blank\_below (integer blank lines above / below the block — for cells\_title() / cells\_footnotes() / cells\_subgroup\_labels())
- Inline — pretext, posttext (literal text prepended / appended around the cell value)

Unknown attribute names emit a cli::cli\_warn and drop from the constructed node; the engine never sees a foreign property.

## Value

The updated tabular\_spec (or tabular\_style\_template, when called against one).

**See Also**

**Companion verbs:** `cols()`, `headers()`, `preset()`, `set_preset()`.

**Location constructors:** `cells_body()`, `cells_headers()`, `cells_group_headers()`, `cells_title()`, `cells_subgroup_labels()`, `cells_footnotes()`, `cells_pagehead()`, `cells_pagefoot()`, `cells_table()`.

**Style values:** `brdr()`, `style_template()`.

**Examples**

```
---- AE table by SOC and PT with per-row indent + styled hierarchy ----
`cdisc_saf_aesocpt` ships with `indent_level` (0 on overall/SOC rows,
1 on PT rows); `col_spec(indent_by = "indent_level")` drives the
PT indent on the `label` column.
tabular(cdisc_saf_aesocpt, titles = "Adverse Events by SOC / PT",
 footnotes = "") |>
 cols(
 label = col_spec(label = "Category", align = "left",
 indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal"),
 Total = col_spec(label = "Total", align = "decimal")
) |>
SOC summary rows bolded (depth 0 - flush)
style(bold = TRUE,
 .at = cells_body(where = row_type == "soc")) |>
Overall row gets a light background
style(background = "#f0f0f0",
 .at = cells_body(where = row_type == "overall"))

---- Chrome styling ----
Each layer changes the surface VISIBLY from its default: a coloured
rule under the header band, a dark-blue header text, a left-aligned
title (default is centred), and a blank line above + below the title.
tabular(cdisc_saf_demo) |>
 style(color = "#1a5276", .at = cells_headers()) |>
 style(border_bottom = brdr("thick", "double", "#1a5276"),
 .at = cells_headers()) |>
 style(halign = "left", .at = cells_title()) |>
 style(blank_above = 1, blank_below = 1,
 .at = cells_title())

---- Table-wide borders ----
tabular(cdisc_saf_demo) |>
 style(border = brdr("medium"),
 .at = cells_table(side = "outer")) |>
 style(border_top = brdr("hairline", "dotted"),
 .at = cells_table(side = "rows"))
```

```
---- House style via style_template() ----
house <- style_template() |>
 style(bold = TRUE, .at = cells_headers()) |>
 style(border_top = brdr("thick"), .at = cells_headers()) |>
 style(border_bottom = brdr("thick"), .at = cells_headers()) |>
 style(border_bottom = brdr("medium"),
 .at = cells_table(side = "outer_bottom"))
Attach once via set_preset(); every tabular() chain then inherits it.
set_preset(.style = house, font_size = 9)
set_preset(.reset = TRUE) # restore the default for later examples
```

---

style\_template

*Reusable style template (for house-style presets)*


---

## Description

Build a reusable, composable style template by chaining `style()` calls against a `tabular_style_template`. The template carries an ordered list of `style_layer` records and can be attached to `preset()` / `set_preset()` as a `style =` argument — every downstream `tabular()` chain then inherits the template's layers via the engine cascade.

## Usage

```
style_template()
```

```
is_style_template(x)
```

## Arguments

`x` *Any R object.* The predicate inspects the class via `inherits()`; no other introspection is performed.

## Details

**One verb, two surfaces.** The same `style(.spec_or_template, ..., .at = ...)` call that attaches a layer to a per-table spec also accumulates layers onto a template. Symmetric API — no need to learn a second function for the multi-table use case.

**Submission workflow.** A submission typically renders 100–200 tables with one visual identity. Build the template once at the top of the submission script, pass it to `set_preset(style = template)`, and every subsequent `tabular()` produces output that inherits the same column-header rules, group-header bolding, title spacing, and outer-frame borders without a single per-table `style()` call.

**Cascade order.** Engines apply layers low-to-high priority: backend defaults → session preset's `@style` → spec preset's `@style` → per-spec `style()` layers. Later layers override prior ones per attribute; NA fields leave the prior layer's value in place.

**Value**

A `tabular_style_template` — a small S3 list with a `layers` slot. Pipe through `style()` to add layers.

**See Also**

**Style verb:** `style()` — the same verb chains onto a spec or a template.

**Locations:** `cells_body` — locations that name the *where* half of every layer.

**Examples**

```
---- Sponsor "house style" composed once ----
#
The result becomes the default look for every table rendered
against this preset. No per-table style() boilerplate.
house <- style_template() |>
 style(bold = TRUE, .at = cells_headers(level = -1)) |>
 style(bold = TRUE, .at = cells_group_headers()) |>
 style(
 border_top = brdr("thick", "double"),
 border_bottom = brdr("thick", "double"),
 .at = cells_headers()
) |>
 style(blank_above = 1, blank_below = 1, .at = cells_title())

length(house$layers)

---- Verify class ----
is_style_template(house)
```

---

subgroup

*Partition the report by a variable*


---

**Description**

Attach a `subgroup_spec` to a `tabular_spec`. At render time the engine partitions `spec@data` by the unique values of `by`, runs the full resolve pipeline per group, and concatenates the results. **A hard page break is inserted between groups** — every subgroup value starts on its own page. A centred banner line appears above the column-header rule on every page of the group (including continuation pages), matching the canonical submission page-layout convention.

**Usage**

```
subgroup(.spec, by, label = NULL, big_n = NULL, big_n_fmt = "\\n(N={n})")
```

**Arguments**

`.spec` *The tabular\_spec to partition.* `<tabular_spec>`: required.

`by` *Column name(s) to partition by.* `<character>`: required. Must reference a column in `spec@data`. Length-0 (or `character(0)`) clears the partition. Matches the `by = arg` convention of `sort_rows()`.  
**Multi-variable.** Pass `c("var1", "var2")` to cross on every combination present in the data. Multi-var partitions require an explicit label template (the single-var auto-default does not generalise).

`label` *Banner template.* `<character(1) | NULL>`: default `NULL`. Glue-style template with `{column_name}` placeholders. `NULL` derives a default from the partition variable's `attr(data[[by]], "label")` (falling back to the column name).  
**Tip:** reference auxiliary columns to inline the `BigN` or any qualifier that is constant within group — e.g. `"Cohort: {cohort} (N = {n})"`.  
**Restriction:** Every `{col}` reference must be a column in `spec@data`. Unknown columns raise `tabular_error_subgroup_template_unknown_col`.

`big_n` *Per-page BigN denominators.* `<data.frame> | NULL`: default `NULL`. A table giving the (N=x) denominator each arm's header should show on each subgroup page. Each arm is named as it appears in the header — either a data column (the N rides that column's leaf label) **or** a `headers()` band label (the N rides that spanner band). Ns are non-negative whole numbers; provide one per by combination present in the data. Accepts **either** shape:

- **Wide** — the by column(s) plus one numeric column per arm (cells are the Ns).
- **Long** — the by column(s) plus one arm-name column and one numeric N column, i.e. `dplyr::count() / summarise()` output used directly with no reshaping.

```
Wide: one column per arm.
wide <- data.frame(
 sex = factor(c("F", "M")),
 placebo = c(24L, 18L), drug_50 = c(9L, 15L), Total = c(42L, 47L)
)
Long: count()-style, pivoted internally. Equivalent to `wide`.
long <- data.frame(
 sex = factor(rep(c("F", "M"), each = 3)),
 arm = rep(c("placebo", "drug_50", "Total"), 2),
 n = c(24L, 9L, 42L, 18L, 15L, 47L)
)
spec |> subgroup(by = "sex", big_n = long)
```

**Requirement:** band keying needs `headers()` **before** `subgroup()` in the pipeline; each arm name must resolve to exactly one leaf XOR one band. Every missing per-page N is a call-time error, never a silently wrong denominator.

**Note:** the per-arm N renders in every backend. The paged backends (RTF, PDF / LaTeX, DOCX) carry it on the column header that repeats on every page of the subgroup. HTML and Markdown are continuous (one stacked table, one header), so they instead emit a per-arm N row directly under each subgroup banner, the (N=x) aligned beneath its arm column.

`big_n_fmt` *Per-page BigN template.* `<character(1)>`: default `"\n(N={n})"`. Appended to each arm's header label, with `{n}` substituted by that page/column's integer `N`. Only the `{n}` token is allowed; the default puts the `N` on its own line under the arm name.

## Details

**Label is a glue-style template.** When `label` carries `{col}` placeholders, the engine substitutes each placeholder against the **FIRST ROW** of the group's filtered data — so any column whose value is constant within group (BigN, cohort descriptor, qualifier text) can ride into the banner. Columns that vary within group also resolve, but always to the first row's value; pre-compute aggregates upstream.

**Default label** (when `label = NULL`, single var): the engine generates `"<attr(data[[by]], 'label') %||% by>: {<by>}"`, so `subgroup(by = "cohort")` renders banners like "Cohort: A" and "Cohort: B" without further configuration.

**Replace, not stack.** A second `subgroup()` call REPLACES the prior partition — `subgroup` is a single spec, not a stackable list. Passing `by = character(0)` clears the slot, though typical clinical pipelines set the partition once up front.

**Display-side only.** `subgroup()` partitions a pre-summarised wide data frame; it does not aggregate, filter, or weight. The user supplies one summary row per displayed row per group; `tabular`'s job is solely to lay them out with the per-group banner and page break.

**Multi-variable crossing.** `by = c("SEX", "AGEGR1")` partitions on every combination present in the data (first variable varies slowest, matching `expand.grid()` convention). An explicit `label` template is required for multi-var partitions since the single-var default `"<var>: {<var>}"` does not generalise; raise `tabular_error_subgroup_label_required` otherwise.

**Auto-hide of partition + template columns.** Every column named in `by`, plus every column referenced via a `{col}` placeholder in `label`, automatically flips to `visible = FALSE` at engine time. Users do not restate `col_spec(visible = FALSE)` inside `cols()` for these columns — mirroring the `col_spec(indent_by = ...)` auto-hide ergonomic.

## Value

*The updated* `tabular_spec`. Continue chaining or resolve via `as_grid()` / `emit()`.

## See Also

**Pipeline siblings:** `sort_rows()`, `paginate()`.

**Resolve / render:** `as_grid()`, `emit()`.

## Examples

```
---- Example 1: TEAEs by treatment arm - one set of pages per arm ----
#
Partition the AE-by-SOC/PT pipeline by treatment arm. Each arm
value gets its own page set with a centred `Treatment Arm: <value>`
banner above the column-header rule on every page, separated by
hard page breaks. The default label uses the variable's `label`
attribute when present, falling back to the column name.
```

```

n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
ae <- cdisc_saf_aesocpt
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))
ae$n_total <- as.integer(sub(".*", "", ae$Total))
attr(ae$row_type, "label") <- "Row Type"

tabular(
 ae,
 titles = c(
 "Table 14.3.1",
 "Adverse Events by SOC and Preferred Term",
 "Safety Population"
),
 footnotes = "Subjects counted once per SOC and once per PT."
) |>
cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal"),
 Total = col_spec(label = "Total", align = "decimal")
) |>
sort_rows(by = c("row_type", "n_total"), descending = c(FALSE, TRUE)) |>
subgroup(by = "row_type")

---- Example 2: Partition by Sex with inline BigN via template ----
#
`label` is a glue-style template; any column whose value is
constant within group can ride into the banner. `cdisc_saf_subgroup`
ships partition-constant `sex_n` / `agegr_n` BigN columns
alongside the value cells, so each banner reads
`Sex: F (N = 106)`, etc. `sex` and `sex_n` auto-hide from the
body (partition `by` and template-referenced columns).
tabular(cdisc_saf_subgroup, titles = "Vital Signs at End of Treatment") |>
cols(
 agegr = col_spec(usage = "group", label = "Age Group"),
 agegr_n = col_spec(visible = FALSE),
 paramcd = col_spec(visible = FALSE),
 param = col_spec(usage = "group", label = "Parameter"),
 stat_label = col_spec(label = "Statistic"),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal"),
 Total = col_spec(label = "Total", align = "decimal")
) |>
subgroup(by = "sex", label = "Sex: {sex} (N = {sex_n})")

---- Example 3: Multi-variable crossing (Sex x Age group) ----
#

```

```

Pass two columns to partition on every combination present in
the data. The label template MUST reference each variable
explicitly because the single-var auto-default does not
generalise. expand.grid order: first var (sex) varies slowest,
second (agegr) fastest, giving banner sequence F/<65, F/>=65,
M/<65, M/>=65.
tabular(cdisc_saf_subgroup, titles = "Vital Signs by Sex and Age Group") |>
 cols(
 sex_n = col_spec(visible = FALSE),
 agegr_n = col_spec(visible = FALSE),
 paramcd = col_spec(visible = FALSE),
 param = col_spec(usage = "group", label = "Parameter"),
 stat_label = col_spec(label = "Statistic"),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal"),
 Total = col_spec(label = "Total", align = "decimal")
) |>
 subgroup(
 by = c("sex", "agegr"),
 label = "Sex: {sex} / Age: {agegr}"
)

---- Example 4: Per-page BigN - different (N=) per sex page ----
#
Each sex page has a different per-arm population, so the `(N=x)`
in the arm headers must vary by page. `big_n` is a wide table:
the `by` column plus one column per arm (named as the data
column), cells are the page-specific Ns. Each arm header then
reads e.g. `Placebo` over `(N=24)` on the Female page and
`(N=18)` on the Male page. RTF / PDF / DOCX carry the N on the
repeating header; HTML and Markdown add a per-arm N row under each
banner.
big_n <- data.frame(
 sex = factor(c("F", "M"), levels = c("F", "M")),
 placebo = c(24L, 18L),
 drug_50 = c(9L, 15L),
 drug_100 = c(9L, 14L),
 Total = c(42L, 47L)
)
tabular(cdisc_saf_subgroup, titles = "Vital Signs by Sex") |>
 cols(
 sex_n = col_spec(visible = FALSE),
 agegr = col_spec(usage = "group", label = "Age Group"),
 agegr_n = col_spec(visible = FALSE),
 paramcd = col_spec(visible = FALSE),
 param = col_spec(usage = "group", label = "Parameter"),
 stat_label = col_spec(label = "Statistic"),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal"),
 Total = col_spec(label = "Total", align = "decimal")
) |>

```

```

subgroup(by = "sex", label = "Sex: {sex}", big_n = big_n)

---- Example 5: Clear a partition with subgroup(character()) ----
#
`subgroup(by = character())` (or `subgroup(by = NULL)`)
explicitly clears any prior partition. Useful in
programmatically-built pipelines where a downstream branch
decides not to paginate by group after all – the call resets
the spec back to a single-page-set render.
tabular(cdisc_saf_subgroup, titles = "Pooled (no sex partition)") |>
 cols(
 sex_n = col_spec(visible = FALSE),
 agegr = col_spec(visible = FALSE),
 agegr_n = col_spec(visible = FALSE),
 paramcd = col_spec(visible = FALSE),
 param = col_spec(usage = "group", label = "Parameter"),
 stat_label = col_spec(label = "Statistic"),
 placebo = col_spec(label = "Placebo", align = "decimal"),
 drug_50 = col_spec(label = "Drug 50", align = "decimal"),
 drug_100 = col_spec(label = "Drug 100", align = "decimal"),
 Total = col_spec(label = "Total", align = "decimal")
) |>
 subgroup("sex", label = "Sex: {sex}") |>
 # Decide later that the sex split was the wrong default –
 # clear it before rendering.
 subgroup(character())

```

---

tabular

*Start a tabular display*


---

## Description

Wrap a pre-summarised data frame into a `tabular_spec` ready for the verb chain. `tabular()` is the entry verb — it owns the data, titles, and footnotes slots; every downstream verb (`cols()`, `headers()`, `sort_rows()`, `style()`, `paginate()`, `preset()`) returns an updated spec for further chaining, terminating in `emit()` (write to file) or `as_grid()` (resolve without writing).

## Usage

```
tabular(data, titles = NULL, footnotes = NULL)
```

## Arguments

`data` *The display rows.* `<data.frame>`: required. Pre-summarised wide-format data; tibbles, data.tables, and arrow tables are coerced via `as.data.frame()`. Factor columns are preserved (their levels drive `sort_rows()`).

**Restriction:** At least one column; column names must be unique. Zero rows is accepted (engine renders a "No data" stub). **Interaction:** The cards-format

|           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|           | counterparts (cdisc_saf_demo_ard, cdisc_saf_aesocpt_ard) are NOT accepted directly; pipe through <code>pivot_across()</code> first.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| titles    | <p><i>Page-title block, one element per row.</i> &lt;character&gt;   NULL: default NULL. Each element renders on its own centred line; embedded \n wraps within that row. The backend collapses unused rows so the column-header band sits flush against the lowest used title.</p> <p><b>Restriction:</b> No NAs.</p> <p>Each element supports glue-style {expr} interpolation: braces are evaluated as R code in the calling environment at build time, e.g. "N total = {sum(n)}". Double a brace ({{ or }}) for a literal one. An <code>md()</code> / <code>html()</code> element is passed through without interpolation.</p> <pre># Canonical 3-line title block with BigN-qualified population. n &lt;- stats::setNames(cdisc_saf_n\$n, cdisc_saf_n\$arm_short) titles = c(   "Table 14.3.1",   "Adverse Events by System Organ Class and Preferred Term",   "Safety Population" )</pre> |
| footnotes | <p><i>Page-footnote block, one element per row.</i> &lt;character&gt;   NULL: default NULL. User-supplied prose rows only; the backend appends its own program-path / program-name / timestamp band below them at render time.</p> <p><b>Restriction:</b> No NAs.</p> <p>Each element supports glue-style {expr} interpolation (see titles).</p> <pre># Canonical 3-line footnote block. footnotes = c(   "Subjects are counted once per SOC and once per PT.",   "Percentages based on N per treatment group.",   "TEAE = treatment-emergent adverse event." )</pre>                                                                                                                                                                                                                                                                                                                          |

## Details

**Pre-summarised input contract.** `data` is one row per displayed row of the final table. `tabular()` does not aggregate, filter, weight, or generate subtotal rows — those happen upstream in `cards`, `dplyr`, or `SAS`. If the upstream is a long `cards::ard_stack()` ARD, pipe through `pivot_across()` first to land in the wide shape `tabular()` accepts.

**Multi-line titles and footnotes by contract.** Clinical tables routinely carry 2-4 title rows and 1-4 user footnote rows. Pass each row as one element of the character vector; the backend renders each element on its own line, collapsing unused rows so the column-header band sits flush against the lowest used title.

## Value

A `tabular_spec` S7 object. Pipe it into `cols()`, `headers()`, `sort_rows()`, `style()`, `paginate()`, and `preset()` to build the display, then into `emit()` to render or `as_grid()` to resolve without writing.

**See Also**

**Downstream build verbs:** `cols()` / `col_spec()`, `headers()`, `sort_rows()`, `style()`, `paginate()`, `preset()`.

**Terminal verbs:** `emit()` (write), `as_grid()` (resolve without I/O).

**Input helper:** `pivot_across()` (cards ARD -> wide).

**Demo data:** `cdisc_saf_demo`, `cdisc_saf_aesocpt`, `cdisc_eff_resp`, `cdisc_saf_n`, `cdisc_eff_n`.

**Examples**

```
---- Example 1: Adverse-event table by SOC and Preferred Term ----
#
The regulatory work-horse layout: AE-by-SOC/PT with the
canonical 3-line title block (table number, description,
population qualifier with BigN drawn inline from `cdisc_saf_n`) and a
two-line footnote block explaining the denominator. The
downstream pipeline hides the hierarchy markers (`row_type`,
`soc_n`, `n_total`) but keeps them in the data so `sort_rows()`
can arrange SOCs and PTs in descending order of subject count.
The dataset already ships `n_total` and `soc_n`; here we slice to
the overall row plus the two highest-incidence SOCs to keep the
preview compact.
ae <- cdisc_saf_aesocpt
keep_soc <- head(unique(ae$soc[ae$row_type == "soc"]), 2L)
ae <- ae[ae$row_type == "overall" | ae$soc %in% keep_soc,]
ae$row_type <- factor(ae$row_type, levels = c("overall", "soc", "pt"))
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)

tabular(
 ae,
 titles = c(
 "Table 14.3.1",
 "Adverse Events by System Organ Class and Preferred Term",
 "Safety Population"
),
 footnotes = c(
 "Subjects are counted once per SOC and once per PT.",
 "Percentages based on N per treatment group."
)
) |>
cols(
 label = col_spec(label = "SOC / PT", indent_by = "indent_level"),
 soc = col_spec(visible = FALSE),
 soc_n = col_spec(visible = FALSE),
 row_type = col_spec(visible = FALSE),
 n_total = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={n['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={n['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={n['drug_100']}"),
 Total = col_spec(label = "Total\nN={n['Total']}")
) |>
sort_rows(by = c("row_type", "n_total"), descending = c(FALSE, TRUE))
```

```

---- Example 2: Best overall response with CDISC factor ordering ----
#
Efficacy table where response categories must appear in CDISC
clinical order (CR < PR < SD < NON-CR/NON-PD < PD < NE <
MISSING), then the derived ORR / CBR / DCR rate rows, not
alphabetical. `groupid` keeps the four sections ordered while the
`stat_label` factor orders the response block; `sort_rows()` does
both in one pass. `groupid` / `group_label` ride along hidden.
bor_levels <- c(
 "CR", "PR", "SD", "NON-CR/NON-PD", "PD", "NE", "MISSING",
 "ORR (CR + PR)", "CBR (CR + PR + SD)",
 "DCR (CR + PR + SD + NON-CR/NON-PD)", "95% CI (Clopper-Pearson)"
)
eff <- cdisc_eff_resp
eff$stat_label <- factor(eff$stat_label, levels = bor_levels)
ne <- stats::setNames(cdisc_eff_n$n, cdisc_eff_n$arm_short)

tabular(
 eff,
 titles = c(
 "Table 14.2.1",
 "Best Overall Response and Response Rates",
 "Efficacy Evaluable Population"
),
 footnotes = "Response per RECIST 1.1, investigator assessment."
) |>
cols(
 stat_label = col_spec(label = "Response"),
 row_type = col_spec(visible = FALSE),
 groupid = col_spec(visible = FALSE),
 group_label = col_spec(visible = FALSE),
 placebo = col_spec(label = "Placebo\nN={ne['placebo']}"),
 drug_50 = col_spec(label = "Drug 50\nN={ne['drug_50']}"),
 drug_100 = col_spec(label = "Drug 100\nN={ne['drug_100']}")
) |>
sort_rows(by = c("groupid", "stat_label"))

---- Example 3: Minimal three-line BigN table from cdisc_saf_n ----
#
The smallest viable `tabular()` call: the bundled `cdisc_saf_n` 4-row
BigN table, a single-line title, no footnotes. The default
`col_spec` per column kicks in, giving sensible labels (the
data frame's column names) and left-aligned text. Useful when
teaching the core API shape without the clinical-context
surface noise.
tabular(cdisc_saf_n, titles = "Safety-population BigN per arm")

---- Example 4: Vital-signs panel with hidden code column ----
#
Show the canonical vitals shape: one parameter across four visits
x four statistics. The CDISC `paramcd` is kept in the data frame
as the natural sort key but hidden at render via

```

```

`col_spec(visible = FALSE)`, while `param` (the display label)
drives the group block. Slice to a single `paramcd` for a compact
preview; the full 4-parameter frame renders identically.
n <- stats::setNames(cdisc_saf_n$n, cdisc_saf_n$arm_short)
vs <- cdisc_saf_vital[cdisc_saf_vital$paramcd == cdisc_saf_vital$paramcd[1L],]
tabular(
 vs,
 titles = c(
 "Table 14.4.1",
 "Summary of Vital Signs",
 "Safety Population"
),
 footnotes = "Statistics computed on observed cases."
) |>
cols(
 paramcd = col_spec(visible = FALSE),
 param = col_spec(usage = "group", label = "Parameter"),
 visit = col_spec(usage = "group", label = "Visit"),
 stat_label = col_spec(label = "Statistic"),
 placebo = col_spec(
 label = "Placebo\nN={n['placebo']}",
 align = "decimal"
),
),
drug_50 = col_spec(
 label = "Drug 50\nN={n['drug_50']}",
 align = "decimal"
),
drug_100 = col_spec(
 label = "Drug 100\nN={n['drug_100']}",
 align = "decimal"
)
)
)

```

# Index

## \* datasets

- cdisc\_eff\_estimates, 11
  - cdisc\_eff\_n, 12
  - cdisc\_eff\_resp, 13
  - cdisc\_saf\_ae, 15
  - cdisc\_saf\_aesocpt, 16
  - cdisc\_saf\_aesocpt\_ard, 18
  - cdisc\_saf\_demo, 19
  - cdisc\_saf\_demo\_ard, 21
  - cdisc\_saf\_n, 22
  - cdisc\_saf\_subgroup, 23
  - cdisc\_saf\_vital, 25
- as.tags.tabular\_spec, 3
- as.tags.tabular\_spec(), 89
- as\_grid, 4
- as\_grid(), 24, 30, 34, 46, 51, 52, 57, 59, 62, 64, 68, 75, 81, 84, 86, 87, 90, 92, 95, 103, 106–108
- brdr, 8
- brdr(), 9, 28, 81, 99
- cdisc\_eff\_estimates, 11
- cdisc\_eff\_n, 12, 14, 23
- cdisc\_eff\_resp, 13
- cdisc\_saf\_ae, 15
- cdisc\_saf\_aesocpt, 15, 16, 19
- cdisc\_saf\_aesocpt\_ard, 17, 18, 21
- cdisc\_saf\_demo, 19, 22
- cdisc\_saf\_demo\_ard, 18, 20, 21
- cdisc\_saf\_n, 13, 15, 17, 20, 22, 24, 25
- cdisc\_saf\_subgroup, 23
- cdisc\_saf\_vital, 25
- cells, 26
- cells\_body, 101
- cells\_body (cells), 26
- cells\_body(), 55, 98, 99
- cells\_footnotes (cells), 26
- cells\_footnotes(), 99
- cells\_group\_headers (cells), 26
- cells\_group\_headers(), 87, 99
- cells\_headers (cells), 26
- cells\_headers(), 55, 87, 99
- cells\_pagefoot (cells), 26
- cells\_pagefoot(), 99
- cells\_pagehead (cells), 26
- cells\_pagehead(), 99
- cells\_subgroup\_labels (cells), 26
- cells\_subgroup\_labels(), 87, 99
- cells\_table (cells), 26
- cells\_table(), 99
- cells\_title (cells), 26
- cells\_title(), 55, 87, 99
- check\_fonts, 29
- check\_fonts(), 32
- check\_latex, 31
- col\_spec, 39
- col\_spec(), 5, 12, 33, 34, 37, 38, 52, 58, 59, 62–64, 68, 75, 84, 95, 108
- cols, 33
- cols(), 4, 5, 37, 38, 40, 42, 43, 46, 50, 52, 59, 66, 68, 75, 84, 94, 95, 99, 103, 106–108
- cols\_apply, 37
- cols\_apply(), 42, 43
- emit, 49
- emit(), 3–5, 30, 32, 34, 46, 55, 57, 59, 62, 64, 68, 75, 81, 84, 86, 87, 89, 90, 92, 95, 103, 106–108
- footnote, 54
- footnote(), 83
- get\_preset, 56
- get\_preset(), 84, 91, 92
- headers, 58
- headers(), 4, 5, 27, 34, 38, 41, 46, 50, 52, 68, 75, 84, 95, 99, 102, 106–108

html, 62  
 html(), 5, 38, 46, 52, 54, 55, 59, 63, 64  
  
 inherits(), 100  
 inline\_ast, 5, 80  
 is\_brdr (brdr), 8  
 is\_brdr(), 9  
 is\_style\_template (style\_template), 100  
 is\_tabular\_location (cells), 26  
  
 md, 63  
 md(), 5, 38, 46, 52, 54, 55, 59, 62  
  
 paginate, 65  
 paginate(), 4, 5, 16, 25, 34, 38, 46, 50, 52,  
     59, 75, 84, 87, 95, 103, 106–108  
 pivot\_across, 70  
 pivot\_across(), 18, 19, 21, 22, 94, 107, 108  
 preset, 78  
 preset(), 4, 5, 8, 9, 30, 34, 38, 45, 46, 50, 52,  
     56, 57, 59, 68, 75, 87, 90–92, 95, 99,  
     100, 106–108  
 preset\_minimal, 86  
 print.tabular\_spec, 3, 88  
  
 set\_preset, 90  
 set\_preset(), 56, 57, 68, 83, 84, 99, 100  
 sort\_rows, 94  
 sort\_rows(), 4, 5, 34, 38, 42, 46, 50–52, 59,  
     68, 75, 84, 102, 103, 106–108  
 style, 97  
 style(), 4, 5, 8, 9, 26–28, 34, 38, 42, 46, 50,  
     52, 59, 62–64, 68, 75, 83, 84, 87, 91,  
     95, 100, 101, 106–108  
 style\_layer, 100  
 style\_template, 100  
 style\_template(), 28, 99  
 subgroup, 101  
 subgroup(), 24  
  
 tabular, 106  
 tabular(), 4, 5, 19, 30, 33, 34, 46, 50, 52, 55,  
     57, 59, 62–64, 68, 75, 84, 87, 92, 95,  
     100  
 tabular\_classes, 9  
 tinytex::install\_tinytex(), 32  
 tinytex::tlmgr\_install(), 31, 32  
 tinytex::tlmgr\_repo(), 32