Package ‘gt’

November 17, 2022

Type Package

Version 0.8.0

Title Easily Create Presentation-Ready Display Tables

Description Build display tables from tabular data with an easy-to-use set of functions. With its progressive approach, we can construct display tables with a cohesive set of table parts. Table values can be formatted using any of the included formatting functions. Footnotes and cell styles can be precisely added through a location targeting system. The way in which 'gt' handles things for you means that you don't often have to worry about the fine details.

License MIT + file LICENSE

URL https://gt.rstudio.com/, https://github.com/rstudio/gt

BugReports https://github.com/rstudio/gt/issues

Encoding UTF-8

LazyData true

ByteCompile true

RoxygenNote 7.2.2

Depends R (>= 3.2.0)

Imports base64enc (>= 0.1-3), bigD (>= 0.1), bitops (>= 1.0-7), cli (>= 3.4.1), commonmark (>= 1.8), dplyr (>= 1.0.8), fs (>= 1.5.2), ggplot2 (>= 3.3.5), glue (>= 1.6.1), htmltools (>= 0.5.2), juicyjuice (>= 0.1.0), magrittr (>= 2.0.2), rlang (>= 1.0.2), sass (>= 0.4.1), scales (>= 1.2.0), tibble (>= 3.1.6), tidyselect (>= 1.1.1)

Suggests covr, digest (>= 0.6.29), knitr, paletteer, testthat (>= 3.0.0), RColorBrewer, lubridate, rmarkdown, rvest, shiny, tidyr, webshot2, xml2

Collate 'as_data_frame.R' 'build_data.R' 'compile_scss.R' 'data_color.R' 'datasets.R' 'dt__.R' 'dt_body.R' 'dt_boxhead.R' 'dt_cols_merge.R' 'dt_data.R' 'dt_footnotes.R' 'dt_formats.R' 'dt_groups_rows.R' 'dt_has_built.R' 'dt_heading.R'

'dt_locale.R' 'dt_options.R' 'dt_row_groups.R'
'dt_source_notes.R' 'dt_spanners.R' 'dt_stub_df.R'
'dt_stubhead.R' 'dt_styles.R' 'dt_substitutions.R'
'dt_summary.R' 'dt_transforms.R' 'export.R' 'format_data.R'
'format_vec.R' 'gt-package.R' 'gt.R' 'gt_preview.R' 'helpers.R'
'image.R' 'info_tables.R' 'knitr-utils.R' 'location_methods.R'
'modify_columns.R' 'modify_rows.R' 'tab_create_modify.R'
'opts.R' 'print.R' 'reexports.R' 'render_as_html.R'
'resolver.R' 'shiny.R' 'substitution.R' 'summary_rows.R'
'tab_info.R' 'tab_remove.R' 'tab_style_body.R'
'text_transform.R' 'utils.R' 'utils_color_contrast.R'
'utils_formatters.R' 'utils_general_str_formatting.R'
'utils_pipe.R' 'utils_render_common.R' 'utils_render_html.R'
'utils_render_latex.R' 'utils_render_rtf.R'
'utils_render_xml.R' 'z_utils_render_footnotes.R' 'zzz.R'

Config/testthat/edition 3

Config/testthat/parallel true

NeedsCompilation no

Author Richard Iannone [aut, cre] (<https://orcid.org/0000-0003-3925-190X>),
Joe Cheng [aut],
Barret Schloerke [aut] (<https://orcid.org/0000-0001-9986-114X>),
Ellis Hughes [aut] (<https://orcid.org/0000-0003-0637-4436>),
JooYoung Seo [aut] (<https://orcid.org/0000-0002-4064-6012>),
RStudio [cph, fnd]

Maintainer Richard Iannone <rich@rstudio.com>

Repository CRAN

Date/Publication 2022-11-16 23:00:02 UTC

R topics documented:

`adjust_luminance` *Adjust the luminance for a palette of colors*

Description

This function can brighten or darken a palette of colors by an arbitrary number of steps, which is defined by a real number between -2.0 and 2.0. The transformation of a palette by a fixed step in this function will tend to apply greater darkening or lightening for those colors in the midrange compared to any very dark or very light colors in the input palette.

Usage

```
adjust_luminance(colors, steps)
```

Arguments

- `colors` — A vector of colors that will undergo an adjustment in luminance. Each color value provided must either be a color name (in the set of colors provided by `grDevices::colors()`) or a hexadecimal string in the form of "#RRGGBB" or "#RRGGBBAA".
- `steps` — A positive or negative factor by which the luminance will be adjusted. Must be a number between `-2.0` and `2.0`.

Details

This function can be useful when combined with the `data_color()` function's `palette` argument, which can use a vector of colors or any of the `col_*` functions from the **scales** package (all of which have a `palette` argument).

Value

A vector of color values.

Examples

Get a palette of 8 pastel colors from the **RColorBrewer** package.

```
pal <- RColorBrewer::brewer.pal(8, "Pastel2")
```

Create lighter and darker variants of the base palette (one step lower, one step higher).

```
pal_darker  <- pal %>% adjust_luminance(-1.0)
pal_lighter <- pal %>% adjust_luminance(+1.0)
```

Create a tibble and make a **gt** table from it. Color each column in order of increasingly darker palettes (with `data_color()`).

```
dplyr::tibble(a = 1:8, b = 1:8, c = 1:8) %>%
  gt() %>%
  data_color(
    columns = a,
    colors = scales::col_numeric(
      palette = pal_lighter,
      domain = c(1, 8)
    )
  ) %>%
  data_color(
    columns = b,
    colors = scales::col_numeric(
      palette = pal,
      domain = c(1, 8)
    )
  ) %>%
```

```
data_color(
  columns = c,
  colors = scales::col_numeric(
    palette = pal_darker,
    domain = c(1, 8)
  )
)
```

Function ID

7-23

See Also

Other helper functions: cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

as_latex *Output a gt object as LaTeX*

Description

Get the LaTeX content from a gt_tbl object as a knit_asis object. This object contains the LaTeX code and attributes that serve as LaTeX dependencies (i.e., the LaTeX packages required for the table). Using as.character() on the created object will result in a single-element vector containing the LaTeX code.

Usage

```
as_latex(data)
```

Arguments

data A table object that is created using the gt() function.

Details

LaTeX packages required to generate tables are: amsmath, booktabs, caption, longtable.

In the event packages are not automatically added during the render phase of the document, please create and include a style file to load them.

Inside the document's YAML metadata, please include:

```
output:
  pdf_document: # Change to appropriate LaTeX template
    includes:
      in_header: 'gt_packages.sty'
```

The `gt_packages.sty` file would then contain the listed dependencies above:

```
\usepackage{amsmath, booktabs, caption, longtable}
```

Examples

Use `gtcars` to create a **gt** table. Add a header and then export as an object with LaTeX code.

```
tab_latex <-
  gtcars %>%
  dplyr::select(mfr, model, msrp) %>%
  dplyr::slice(1:5) %>%
  gt() %>%
  tab_header(
    title = md("Data listing from **gtcars**"),
    subtitle = md("`gtcars` is an R dataset")
  ) %>%
  as_latex()
```

What's returned is a `knit_asis` object, which makes it easy to include in R Markdown documents that are knit to PDF. We can use `as.character()` to get just the LaTeX code as a single-element vector.

Function ID

13-3

See Also

Other table export functions: `as_raw_html()`, `as_rtf()`, `as_word()`, `extract_cells()`, `extract_summary()`, `gtsave()`

`as_raw_html` *Get the HTML content of a **gt** table*

Description

Get the HTML content from a `gt_tbl` object as a single-element character vector. By default, the generated HTML will have inlined styles, where CSS styles (that were previously contained in CSS rule sets external to the `<table> element`) are included as `style` attributes in the HTML table's tags. This option is preferable when using the output HTML table in an emailing context.

Usage

```
as_raw_html(data, inline_css = TRUE)
```

Arguments

data	A table object that is created using the gt() function.
inline_css	An option to supply styles to table elements as inlined CSS styles. This is useful when including the table HTML as part of an HTML email message body, since inlined styles are largely supported in email clients over using CSS in a <style> block.

Examples

Use gtcars to create a **gt** table. Add a header and then export as HTML code with inlined CSS styles.

```
tab_html <-
  gtcars %>%
  dplyr::select(mfr, model, msrp) %>%
  dplyr::slice(1:5) %>%
  gt() %>%
  tab_header(
    title = md("Data listing from **gtcars**"),
    subtitle = md("`gtcars` is an R dataset")
  ) %>%
  as_raw_html()
```

What's returned is a single-element vector containing the HTML for the table. It has only the <table>...</table> part so it's not a complete HTML document but rather an HTML fragment.

Function ID

13-2

See Also

Other table export functions: as_latex(), as_rtf(), as_word(), extract_cells(), extract_summary(), gtsave()

---

as_rtf *Output a **gt** object as RTF*

Description

Get the RTF content from a gt_tbl object as as a single-element character vector. This object can be used with writeLines() to generate a valid .rtf file that can be opened by RTF readers.

Usage

```
as_rtf(data)
```

Arguments

data	A table object that is created using the gt() function.

Examples

Use gtcars to create a **gt** table. Add a header and then export as RTF code.

```
tab_rtf <-
  gtcars %>%
  dplyr::select(mfr, model) %>%
  dplyr::slice(1:2) %>%
  gt() %>%
  tab_header(
    title = md("Data listing from **gtcars**"),
    subtitle = md("`gtcars` is an R dataset")
  ) %>%
  as_rtf()
```

Function ID

13-4

See Also

Other table export functions: as_latex(), as_raw_html(), as_word(), extract_cells(), extract_summary(), gtsave()

---

as_word *Output a* **gt** *object as Word*

Description

Get the Open Office XML table tag content from a gt_tbl object as as a single-element character vector.

Usage

```
as_word(
  data,
  align = "center",
  caption_location = c("top", "bottom", "embed"),
  caption_align = "left",
  split = FALSE,
  keep_with_next = TRUE
)
```

Arguments

data	A table object that is created using the gt() function.
align	An option for table alignment. Can either be "center" (the default), "left", or "right".
caption_location	Determines where the caption should be positioned. This can either be "top" (the default), "bottom", or "embed".
caption_align	Determines the alignment of the caption. This is either "left" (the default), "center", or "right". This option is only used when caption_location is not set as "embed".
split	A TRUE or FALSE (the default) value that indicates whether to activate the Word option Allow row to break across pages.
keep_with_next	A TRUE (the default) or FALSE value that indicates whether a table should use Word option keep rows together.

Function ID

13-5

See Also

Other table export functions: as_latex(), as_raw_html(), as_rtf(), extract_cells(), extract_summary(), gtsave()

Examples

```
# Use `gtcars` to create a gt table;
# add a header and then export as
# OOXML code for Word
tab_rtf <-
  gtcars %>%
  dplyr::select(mfr, model) %>%
  dplyr::slice(1:2) %>%
  gt() %>%
  tab_header(
    title = md("Data listing from **gtcars**"),
    subtitle = md("`gtcars` is an R dataset")
  ) %>%
  as_word()
```

cells_body *Location helper for targeting data cells in the table body*

Description

The cells_body() function is used to target the data cells in the table body. The function can be used to apply a footnote with tab_footnote(), to add custom styling with tab_style(), or the transform the targeted cells with text_transform(). The function is expressly used in each of those functions' locations argument. The 'body' location is present by default in every **gt** table.

Usage

```
cells_body(columns = everything(), rows = everything())
```

Arguments

columns	The names of the columns that are to be targeted.
rows	The names of the rows that are to be targeted.

Value

A list object with the classes cells_body and location_cells.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a locations argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- cells_title(): targets the table title or the table subtitle depending on the value given to the groups argument ("title" or "subtitle").
- cells_stubhead(): targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the tab_stubhead() function.
- cells_column_spanners(): targets the spanner column labels with the spanners argument; spanner column labels appear above the column labels.
- cells_column_labels(): targets the column labels with its columns argument.
- cells_row_groups(): targets the row group labels in any available row groups using the groups argument.
- cells_stub(): targets row labels in the table stub using the rows argument.
- cells_body(): targets data cells in the table body using intersections of columns and rows.
- cells_summary(): targets summary cells in the table body using the groups argument and intersections of columns and rows.
- cells_grand_summary(): targets cells of the table's grand summary using intersections of columns and rows

- cells_stub_summary(): targets summary row labels in the table stub using the groups and rows arguments.
- cells_stub_grand_summary(): targets grand summary row labels in the table stub using the rows argument.
- cells_footnotes(): targets all footnotes in the table footer (cannot be used with tab_footnote()).
- cells_source_notes(): targets all source notes in the table footer (cannot be used with tab_footnote()).

When using any of the location helper functions with an appropriate function that has a locations argument (e.g., tab_style()), multiple locations can be targeted by enclosing several cells_*() helper functions in a list() (e.g., list(cells_body(), cells_grand_summary())).

Examples

Use gtcars to create a **gt** table. Add a footnote that targets a single data cell with tab_footnote(), using cells_body() in locations (rows = hp == max(hp) will target a single row in the hp column).

```
gtcars %>%
  dplyr::filter(ctry_origin == "United Kingdom") %>%
  dplyr::select(mfr, model, year, hp) %>%
  gt() %>%
  tab_footnote(
    footnote = "Highest horsepower.",
    locations = cells_body(
      columns = hp,
      rows = hp == max(hp)
    )
  ) %>%
  opt_footnote_marks(marks = c("*", "+"))
```

Function ID

7-12

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

cells_column_labels *Location helper for targeting the column labels*

Description

The cells_column_labels() function is used to target the table's column labels when applying a footnote with tab_footnote() or adding custom style with tab_style(). The function is expressly used in each of those functions' locations argument. The 'column_labels' location is present by default in every **gt** table.

Usage

```
cells_column_labels(columns = everything())
```

Arguments

columns	The names of the column labels that are to be targeted.

Value

A list object with the classes cells_column_labels and location_cells.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a locations argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- cells_title(): targets the table title or the table subtitle depending on the value given to the groups argument ("title" or "subtitle").
- cells_stubhead(): targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the tab_stubhead() function.
- cells_column_spanners(): targets the spanner column labels with the spanners argument; spanner column labels appear above the column labels.
- cells_column_labels(): targets the column labels with its columns argument.
- cells_row_groups(): targets the row group labels in any available row groups using the groups argument.
- cells_stub(): targets row labels in the table stub using the rows argument.
- cells_body(): targets data cells in the table body using intersections of columns and rows.
- cells_summary(): targets summary cells in the table body using the groups argument and intersections of columns and rows.
- cells_grand_summary(): targets cells of the table's grand summary using intersections of columns and rows
- cells_stub_summary(): targets summary row labels in the table stub using the groups and rows arguments.

- `cells_stub_grand_summary()`: targets grand summary row labels in the table stub using the `rows` argument.
- `cells_footnotes()`: targets all footnotes in the table footer (cannot be used with `tab_footnote()`).
- `cells_source_notes()`: targets all source notes in the table footer (cannot be used with `tab_footnote()`).

When using any of the location helper functions with an appropriate function that has a `locations` argument (e.g., `tab_style()`), multiple locations can be targeted by enclosing several `cells_*()` helper functions in a `list()` (e.g., `list(cells_body(), cells_grand_summary())`).

Examples

Use `sza` to create a **gt** table. Add footnotes to the column labels with `tab_footnote()` and `cells_column_labels()` in `locations`.

```
sza %>%
  dplyr::filter(
    latitude == 20 & month == "jan" &
      !is.na(sza)
  ) %>%
  dplyr::select(-latitude, -month) %>%
  gt() %>%
  tab_footnote(
    footnote = "True solar time.",
    locations = cells_column_labels(
      columns = tst
    )
  ) %>%
  tab_footnote(
    footnote = "Solar zenith angle.",
    locations = cells_column_labels(
      columns = sza
    )
  )
```

Function ID

7-9

See Also

Other helper functions: `adjust_luminance()`, `cell_borders()`, `cell_fill()`, `cell_text()`, `cells_body()`, `cells_column_spanners()`, `cells_footnotes()`, `cells_grand_summary()`, `cells_row_groups()`, `cells_source_notes()`, `cells_stub_grand_summary()`, `cells_stub_summary()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`, `stub()`

cells_column_spanners *Location helper for targeting the column spanners*

Description

The `cells_column_spanners()` function is used to target the cells that contain the table column spanners. This is useful when applying a footnote with `tab_footnote()` or adding custom style with `tab_style()`. The function is expressly used in each of those functions' `locations` argument. The 'column_spanners' location is generated by one or more uses of the `tab_spanner()` function or the `tab_spanner_delim()` function.

Usage

```
cells_column_spanners(spanners = everything())
```

Arguments

`spanners`	The names of the spanners that are to be targeted.

Value

A list object with the classes `cells_column_spanners` and `location_cells`.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a `locations` argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- `cells_title()`: targets the table title or the table subtitle depending on the value given to the `groups` argument (`"title"` or `"subtitle"`).
- `cells_stubhead()`: targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the `tab_stubhead()` function.
- `cells_column_spanners()`: targets the spanner column labels with the `spanners` argument; spanner column labels appear above the column labels.
- `cells_column_labels()`: targets the column labels with its `columns` argument.
- `cells_row_groups()`: targets the row group labels in any available row groups using the `groups` argument.
- `cells_stub()`: targets row labels in the table stub using the `rows` argument.
- `cells_body()`: targets data cells in the table body using intersections of `columns` and `rows`.
- `cells_summary()`: targets summary cells in the table body using the `groups` argument and intersections of `columns` and `rows`.
- `cells_grand_summary()`: targets cells of the table's grand summary using intersections of `columns` and `rows`

- `cells_stub_summary()`: targets summary row labels in the table stub using the `groups` and `rows` arguments.
- `cells_stub_grand_summary()`: targets grand summary row labels in the table stub using the `rows` argument.
- `cells_footnotes()`: targets all footnotes in the table footer (cannot be used with `tab_footnote()`).
- `cells_source_notes()`: targets all source notes in the table footer (cannot be used with `tab_footnote()`).

When using any of the location helper functions with an appropriate function that has a `locations` argument (e.g., `tab_style()`), multiple locations can be targeted by enclosing several `cells_*()` helper functions in a `list()` (e.g., `list(cells_body(), cells_grand_summary())`).

Examples

Use `exibble` to create a **gt** table. Add a spanner column label over three column labels with `tab_spanner()` and then use `tab_style()` and `cells_column_spanners()` to make the spanner label text bold.

```
exibble %>%
  dplyr::select(-fctr, -currency, -group) %>%
  gt(rowname_col = "row") %>%
  tab_spanner(
    label = "dates and times",
    id = "dt",
    columns = c(date, time, datetime)
  ) %>%
  tab_style(
    style = cell_text(weight = "bold"),
    locations = cells_column_spanners(spanners = "dt")
  )
```

Function ID

7-8

See Also

Other helper functions: `adjust_luminance()`, `cell_borders()`, `cell_fill()`, `cell_text()`, `cells_body()`, `cells_column_labels()`, `cells_footnotes()`, `cells_grand_summary()`, `cells_row_groups()`, `cells_source_notes()`, `cells_stub_grand_summary()`, `cells_stub_summary()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`, `stub()`

cells_footnotes *Location helper for targeting the footnotes*

Description

The `cells_footnotes()` function is used to target all footnotes in the footer section of the table. This is useful for adding custom styles to the footnotes with `tab_style()` (using the `locations` argument). The 'footnotes' location is generated by one or more uses of the `tab_footnote()` function. This location helper function cannot be used for the `locations` argument of `tab_footnote()` and doing so will result in a warning (with no change made to the table).

Usage

```
cells_footnotes()
```

Value

A list object with the classes `cells_footnotes` and `location_cells`.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a `locations` argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- `cells_title()`: targets the table title or the table subtitle depending on the value given to the `groups` argument (`"title"` or `"subtitle"`).
- `cells_stubhead()`: targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the `tab_stubhead()` function.
- `cells_column_spanners()`: targets the spanner column labels with the `spanners` argument; spanner column labels appear above the column labels.
- `cells_column_labels()`: targets the column labels with its `columns` argument.
- `cells_row_groups()`: targets the row group labels in any available row groups using the `groups` argument.
- `cells_stub()`: targets row labels in the table stub using the `rows` argument.
- `cells_body()`: targets data cells in the table body using intersections of `columns` and `rows`.
- `cells_summary()`: targets summary cells in the table body using the `groups` argument and intersections of `columns` and `rows`.
- `cells_grand_summary()`: targets cells of the table's grand summary using intersections of `columns` and `rows`
- `cells_stub_summary()`: targets summary row labels in the table stub using the `groups` and `rows` arguments.
- `cells_stub_grand_summary()`: targets grand summary row labels in the table stub using the `rows` argument.

- cells_footnotes(): targets all footnotes in the table footer (cannot be used with tab_footnote()).
- cells_source_notes(): targets all source notes in the table footer (cannot be used with tab_footnote()).

When using any of the location helper functions with an appropriate function that has a locations argument (e.g., tab_style()), multiple locations can be targeted by enclosing several cells_*() helper functions in a list() (e.g., list(cells_body(), cells_grand_summary())).

Examples

Use sza to create a **gt** table. Color the sza column using the data_color() function, add a footnote and also style the footnotes section.

```
sza %>%
  dplyr::filter(
    latitude == 20 &
      month == "jan" &
      !is.na(sza)
  ) %>%
  dplyr::select(-latitude, -month) %>%
  gt() %>%
  data_color(
    columns = sza,
    colors = scales::col_numeric(
      palette = c("white", "yellow", "navyblue"),
      domain = c(0, 90)
    )
  ) %>%
  tab_footnote(
    footnote = "Color indicates height of sun.",
    locations = cells_column_labels(columns = sza)
  ) %>%
  tab_options(table.width = px(320)) %>%
  tab_style(
    style = list(
      cell_text(size = "smaller"),
      cell_fill(color = "gray90")
      ),
    locations = cells_footnotes()
  )
```

Function ID

7-17

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

cells_grand_summary *Location helper for targeting cells in a grand summary*

Description

The cells_grand_summary() function is used to target the cells in a grand summary and it is useful when applying a footnote with tab_footnote() or adding custom styles with tab_style(). The function is expressly used in each of those functions' locations argument. The 'grand_summary' location is generated by the grand_summary_rows() function.

Usage

```
cells_grand_summary(columns = everything(), rows = everything())
```

Arguments

columns	The names of the columns that are to be targeted.
rows	The names of the rows that are to be targeted.

Value

A list object with the classes cells_summary and location_cells.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a locations argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- cells_title(): targets the table title or the table subtitle depending on the value given to the groups argument ("title" or "subtitle").
- cells_stubhead(): targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the tab_stubhead() function.
- cells_column_spanners(): targets the spanner column labels with the spanners argument; spanner column labels appear above the column labels.
- cells_column_labels(): targets the column labels with its columns argument.
- cells_row_groups(): targets the row group labels in any available row groups using the groups argument.
- cells_stub(): targets row labels in the table stub using the rows argument.
- cells_body(): targets data cells in the table body using intersections of columns and rows.

- cells_summary(): targets summary cells in the table body using the groups argument and intersections of columns and rows.
- cells_grand_summary(): targets cells of the table's grand summary using intersections of columns and rows
- cells_stub_summary(): targets summary row labels in the table stub using the groups and rows arguments.
- cells_stub_grand_summary(): targets grand summary row labels in the table stub using the rows argument.
- cells_footnotes(): targets all footnotes in the table footer (cannot be used with tab_footnote()).
- cells_source_notes(): targets all source notes in the table footer (cannot be used with tab_footnote()).

When using any of the location helper functions with an appropriate function that has a locations argument (e.g., tab_style()), multiple locations can be targeted by enclosing several cells_*() helper functions in a list() (e.g., list(cells_body(), cells_grand_summary())).

Examples

Use countrypops to create a **gt** table. Add some styling to a grand summary cell with with tab_style() and cells_grand_summary().

```
countrypops %>%
  dplyr::filter(country_name == "Spain", year < 1970) %>%
  dplyr::select(-contains("country")) %>%
  gt(rowname_col = "year") %>%
  fmt_number(
    columns = population,
    decimals = 0
  ) %>%
  grand_summary_rows(
    columns = population,
    fns = list(
      change = ~max(.) - min(.)
    ),
    formatter = fmt_number,
    decimals = 0
  ) %>%
  tab_style(
    style = list(
      cell_text(style = "italic"),
      cell_fill(color = "lightblue")
    ),
    locations = cells_grand_summary(
      columns = population,
      rows = 1
    )
  )
```

Function ID

7-14

See Also

Other helper functions: `adjust_luminance()`, `cell_borders()`, `cell_fill()`, `cell_text()`, `cells_body()`, `cells_column_labels()`, `cells_column_spanners()`, `cells_footnotes()`, `cells_row_groups()`, `cells_source_notes()`, `cells_stub_grand_summary()`, `cells_stub_summary()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`, `stub()`

`cells_row_groups` *Location helper for targeting row groups*

Description

The `cells_row_groups()` function is used to target the table's row groups when applying a footnote with `tab_footnote()` or adding custom style with `tab_style()`. The function is expressly used in each of those functions' `locations` argument. The 'row_groups' location can be generated by the specifying a `groupname_col` in `gt()`, by introducing grouped data to `gt()` (by way of `dplyr::group_by()`), or, by specifying groups with the `tab_row_group()` function.

Usage

```
cells_row_groups(groups = everything())
```

Arguments

`groups`	The names of the row groups that are to be targeted.

Value

A list object with the classes `cells_row_groups` and `location_cells`.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a `locations` argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- `cells_title()`: targets the table title or the table subtitle depending on the value given to the `groups` argument (`"title"` or `"subtitle"`).
- `cells_stubhead()`: targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the `tab_stubhead()` function.
- `cells_column_spanners()`: targets the spanner column labels with the `spanners` argument; spanner column labels appear above the column labels.
- `cells_column_labels()`: targets the column labels with its `columns` argument.

- `cells_row_groups()`: targets the row group labels in any available row groups using the `groups` argument.
- `cells_stub()`: targets row labels in the table stub using the `rows` argument.
- `cells_body()`: targets data cells in the table body using intersections of `columns` and `rows`.
- `cells_summary()`: targets summary cells in the table body using the `groups` argument and intersections of `columns` and `rows`.
- `cells_grand_summary()`: targets cells of the table's grand summary using intersections of `columns` and `rows`
- `cells_stub_summary()`: targets summary row labels in the table stub using the `groups` and `rows` arguments.
- `cells_stub_grand_summary()`: targets grand summary row labels in the table stub using the `rows` argument.
- `cells_footnotes()`: targets all footnotes in the table footer (cannot be used with `tab_footnote()`).
- `cells_source_notes()`: targets all source notes in the table footer (cannot be used with `tab_footnote()`).

When using any of the location helper functions with an appropriate function that has a `locations` argument (e.g., `tab_style()`), multiple locations can be targeted by enclosing several `cells_*()` helper functions in a `list()` (e.g., `list(cells_body(), cells_grand_summary())`).

Examples

Use `pizzaplace` to create a **gt** table with grouped data. Add a summary with the `summary_rows()` function and then add a footnote to the `"peppr_salami"` row group label with `tab_footnote()` and with `cells_row_groups()` in `locations`.

```
pizzaplace %>%
  dplyr::filter(name %in% c("soppressata", "peppr_salami")) %>%
  dplyr::group_by(name, size) %>%
  dplyr::summarize(`Pizzas Sold` = dplyr::n(), .groups = "drop") %>%
  gt(rowname_col = "size", groupname_col = "name") %>%
  summary_rows(
    groups = TRUE,
    columns = `Pizzas Sold`,
    fns = list(TOTAL = "sum"),
    formatter = fmt_number,
    decimals = 0,
    use_seps = TRUE
  ) %>%
  tab_footnote(
    footnote = "The Pepper-Salami.",
    cells_row_groups(groups = "peppr_salami")
  )
```

Function ID

7-10

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

cells_source_notes *Location helper for targeting the source notes*

Description

The cells_source_notes() function is used to target all source notes in the footer section of the table. This is useful for adding custom styles to the source notes with tab_style() (using the locations argument). The 'source_notes' location is generated by the tab_source_note() function. This location helper function cannot be used for the locations argument of tab_footnote() and doing so will result in a warning (with no change made to the table).

Usage

```
cells_source_notes()
```

Value

A list object with the classes cells_source_notes and location_cells.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a locations argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- cells_title(): targets the table title or the table subtitle depending on the value given to the groups argument ("title" or "subtitle").
- cells_stubhead(): targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the tab_stubhead() function.
- cells_column_spanners(): targets the spanner column labels with the spanners argument; spanner column labels appear above the column labels.
- cells_column_labels(): targets the column labels with its columns argument.
- cells_row_groups(): targets the row group labels in any available row groups using the groups argument.
- cells_stub(): targets row labels in the table stub using the rows argument.
- cells_body(): targets data cells in the table body using intersections of columns and rows.
- cells_summary(): targets summary cells in the table body using the groups argument and intersections of columns and rows.

- cells_grand_summary(): targets cells of the table's grand summary using intersections of columns and rows
- cells_stub_summary(): targets summary row labels in the table stub using the groups and rows arguments.
- cells_stub_grand_summary(): targets grand summary row labels in the table stub using the rows argument.
- cells_footnotes(): targets all footnotes in the table footer (cannot be used with tab_footnote()).
- cells_source_notes(): targets all source notes in the table footer (cannot be used with tab_footnote()).

When using any of the location helper functions with an appropriate function that has a locations argument (e.g., tab_style()), multiple locations can be targeted by enclosing several cells_*() helper functions in a list() (e.g., list(cells_body(), cells_grand_summary())).

Examples

Use gtcars to create a **gt** table. Add a source note and style the source notes section.

```
gtcars %>%
  dplyr::select(mfr, model, msrp) %>%
  dplyr::slice(1:5) %>%
  gt() %>%
  tab_source_note(source_note = "From edmunds.com") %>%
  tab_style(
    style = cell_text(
      color = "#A9A9A9",
      size = "small"
    ),
    locations = cells_source_notes()
  )
```

Function ID

7-18

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

cells_stub *Location helper for targeting cells in the table stub*

Description

The `cells_stub()` function is used to target the table's stub cells and it is useful when applying a footnote with `tab_footnote()` or adding a custom style with `tab_style()`. The function is expressly used in each of those functions' `locations` argument. Here are several ways that a stub location might be available in a **gt** table: (1) through specification of a `rowname_col` in `gt()`, (2) by introducing a data frame with row names to `gt()` with `rownames_to_stub = TRUE`, or (3) by using `summary_rows()` or `grand_summary_rows()` with neither of the previous two conditions being true.

Usage

```
cells_stub(rows = everything())
```

Arguments

`rows`	The names of the rows that are to be targeted.

Value

A list object with the classes `cells_stub` and `location_cells`.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a `locations` argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- `cells_title()`: targets the table title or the table subtitle depending on the value given to the `groups` argument (`"title"` or `"subtitle"`).
- `cells_stubhead()`: targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the `tab_stubhead()` function.
- `cells_column_spanners()`: targets the spanner column labels with the `spanners` argument; spanner column labels appear above the column labels.
- `cells_column_labels()`: targets the column labels with its `columns` argument.
- `cells_row_groups()`: targets the row group labels in any available row groups using the `groups` argument.
- `cells_stub()`: targets row labels in the table stub using the `rows` argument.
- `cells_body()`: targets data cells in the table body using intersections of `columns` and `rows`.
- `cells_summary()`: targets summary cells in the table body using the `groups` argument and intersections of `columns` and `rows`.
- `cells_grand_summary()`: targets cells of the table's grand summary using intersections of `columns` and `rows`

- cells_stub_summary(): targets summary row labels in the table stub using the groups and rows arguments.
- cells_stub_grand_summary(): targets grand summary row labels in the table stub using the rows argument.
- cells_footnotes(): targets all footnotes in the table footer (cannot be used with tab_footnote()).
- cells_source_notes(): targets all source notes in the table footer (cannot be used with tab_footnote()).

When using any of the location helper functions with an appropriate function that has a locations argument (e.g., tab_style()), multiple locations can be targeted by enclosing several cells_*() helper functions in a list() (e.g., list(cells_body(), cells_grand_summary())).

Examples

Use sza to create a **gt** table. Color all of the month values in the table stub with tab_style(), using cells_stub() in locations.

```
sza %>%
  dplyr::filter(latitude == 20 & tst <= "1000") %>%
  dplyr::select(-latitude) %>%
  dplyr::filter(!is.na(sza)) %>%
  tidyr::spread(key = "tst", value = sza) %>%
  gt(rowname_col = "month") %>%
  sub_missing(missing_text = "") %>%
  tab_style(
    style = list(
      cell_fill(color = "darkblue"),
      cell_text(color = "white")
      ),
    locations = cells_stub()
  )
```

Function ID

7-11

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

cells_stubhead *Location helper for targeting the table stubhead cell*

Description

The cells_stubhead() function is used to target the table stubhead location when applying a footnote with tab_footnote() or adding custom style with tab_style(). The function is expressly used in each of those functions' locations argument. The 'stubhead' location is always present alongside the 'stub' location.

Usage

```
cells_stubhead()
```

Value

A list object with the classes cells_stubhead and location_cells.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a locations argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- cells_title(): targets the table title or the table subtitle depending on the value given to the groups argument ("title" or "subtitle").
- cells_stubhead(): targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the tab_stubhead() function.
- cells_column_spanners(): targets the spanner column labels with the spanners argument; spanner column labels appear above the column labels.
- cells_column_labels(): targets the column labels with its columns argument.
- cells_row_groups(): targets the row group labels in any available row groups using the groups argument.
- cells_stub(): targets row labels in the table stub using the rows argument.
- cells_body(): targets data cells in the table body using intersections of columns and rows.
- cells_summary(): targets summary cells in the table body using the groups argument and intersections of columns and rows.
- cells_grand_summary(): targets cells of the table's grand summary using intersections of columns and rows
- cells_stub_summary(): targets summary row labels in the table stub using the groups and rows arguments.
- cells_stub_grand_summary(): targets grand summary row labels in the table stub using the rows argument.
- cells_footnotes(): targets all footnotes in the table footer (cannot be used with tab_footnote()).

- cells_source_notes(): targets all source notes in the table footer (cannot be used with tab_footnote()).

When using any of the location helper functions with an appropriate function that has a locations argument (e.g., tab_style()), multiple locations can be targeted by enclosing several cells_*() helper functions in a list() (e.g., list(cells_body(), cells_grand_summary())).

Examples

Use pizzaplace to create a **gt** table. Add a stubhead label with tab_stubhead() and then style it with tab_style() and cells_stubhead().

```
pizzaplace %>%
  dplyr::mutate(month = as.numeric(substr(date, 6, 7))) %>%
  dplyr::group_by(month, type) %>%
  dplyr::summarize(sold = dplyr::n(), .groups = "drop") %>%
  dplyr::filter(month %in% 1:2) %>%
  gt(rowname_col = "type") %>%
  tab_stubhead(label = "type") %>%
  tab_style(
    style = cell_fill(color = "lightblue"),
    locations = cells_stubhead()
  )
```

Function ID

7-7

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

cells_stub_grand_summary

Location helper for targeting the stub cells in a grand summary

---

Description

The cells_stub_grand_summary() function is used to target the stub cells of a grand summary and it is useful when applying a footnote with tab_footnote() or adding custom styles with tab_style(). The function is expressly used in each of those functions' locations argument. The 'stub_grand_summary' location is generated by the grand_summary_rows() function.

Usage

```
cells_stub_grand_summary(rows = everything())
```

Arguments

rows	The names of the rows that are to be targeted.

Value

A list object with the classes `cells_stub_grand_summary` and `location_cells`.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a `locations` argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- `cells_title()`: targets the table title or the table subtitle depending on the value given to the `groups` argument (`"title"` or `"subtitle"`).
- `cells_stubhead()`: targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the `tab_stubhead()` function.
- `cells_column_spanners()`: targets the spanner column labels with the `spanners` argument; spanner column labels appear above the column labels.
- `cells_column_labels()`: targets the column labels with its `columns` argument.
- `cells_row_groups()`: targets the row group labels in any available row groups using the `groups` argument.
- `cells_stub()`: targets row labels in the table stub using the `rows` argument.
- `cells_body()`: targets data cells in the table body using intersections of `columns` and `rows`.
- `cells_summary()`: targets summary cells in the table body using the `groups` argument and intersections of `columns` and `rows`.
- `cells_grand_summary()`: targets cells of the table's grand summary using intersections of `columns` and `rows`
- `cells_stub_summary()`: targets summary row labels in the table stub using the `groups` and `rows` arguments.
- `cells_stub_grand_summary()`: targets grand summary row labels in the table stub using the `rows` argument.
- `cells_footnotes()`: targets all footnotes in the table footer (cannot be used with `tab_footnote()`).
- `cells_source_notes()`: targets all source notes in the table footer (cannot be used with `tab_footnote()`).

When using any of the location helper functions with an appropriate function that has a `locations` argument (e.g., `tab_style()`), multiple locations can be targeted by enclosing several `cells_*()` helper functions in a `list()` (e.g., `list(cells_body(), cells_grand_summary())`).

Examples

Use countrypops to create a **gt** table. Add some styling to a grand summary stub cell with with the tab_style() and cells_stub_grand_summary() functions.

```
countrypops %>%
  dplyr::filter(country_name == "Spain", year < 1970) %>%
  dplyr::select(-contains("country")) %>%
  gt(rowname_col = "year") %>%
  fmt_number(
    columns = population,
    decimals = 0
  ) %>%
  grand_summary_rows(
    columns = population,
    fns = list(change = ~max(.) - min(.)),
    formatter = fmt_number,
    decimals = 0
  ) %>%
  tab_style(
    style = cell_text(weight = "bold", transform = "uppercase"),
    locations = cells_stub_grand_summary(rows = "change")
  )
```

Function ID

7-16

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

cells_stub_summary *Location helper for targeting the stub cells in a summary*

Description

The cells_stub_summary() function is used to target the stub cells of summary and it is useful when applying a footnote with tab_footnote() or adding custom styles with tab_style(). The function is expressly used in each of those functions' locations argument. The 'stub_summary' location is generated by the summary_rows() function.

Usage

```
cells_stub_summary(groups = everything(), rows = everything())
```

Arguments

groups — The names of the groups that are to be targeted.

rows — The names of the rows that are to be targeted.

Value

A list object with the classes `cells_stub_summary` and `location_cells`.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a `locations` argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- `cells_title()`: targets the table title or the table subtitle depending on the value given to the `groups` argument (`"title"` or `"subtitle"`).
- `cells_stubhead()`: targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the `tab_stubhead()` function.
- `cells_column_spanners()`: targets the spanner column labels with the `spanners` argument; spanner column labels appear above the column labels.
- `cells_column_labels()`: targets the column labels with its `columns` argument.
- `cells_row_groups()`: targets the row group labels in any available row groups using the `groups` argument.
- `cells_stub()`: targets row labels in the table stub using the `rows` argument.
- `cells_body()`: targets data cells in the table body using intersections of `columns` and `rows`.
- `cells_summary()`: targets summary cells in the table body using the `groups` argument and intersections of `columns` and `rows`.
- `cells_grand_summary()`: targets cells of the table's grand summary using intersections of `columns` and `rows`
- `cells_stub_summary()`: targets summary row labels in the table stub using the `groups` and `rows` arguments.
- `cells_stub_grand_summary()`: targets grand summary row labels in the table stub using the `rows` argument.
- `cells_footnotes()`: targets all footnotes in the table footer (cannot be used with `tab_footnote()`).
- `cells_source_notes()`: targets all source notes in the table footer (cannot be used with `tab_footnote()`).

When using any of the location helper functions with an appropriate function that has a `locations` argument (e.g., `tab_style()`), multiple locations can be targeted by enclosing several `cells_*()` helper functions in a `list()` (e.g., `list(cells_body(), cells_grand_summary())`).

Examples

Use countrypops to create a **gt** table. Add some styling to the summary data stub cells with tab_style() and cells_stub_summary().

```
countrypops %>%
  dplyr::filter(country_name == "Japan", year < 1970) %>%
  dplyr::select(-contains("country")) %>%
  dplyr::mutate(decade = paste0(substr(year, 1, 3), "0s")) %>%
  gt(
    rowname_col = "year",
    groupname_col = "decade"
  ) %>%
  fmt_number(
    columns = population,
    decimals = 0
  ) %>%
  summary_rows(
    groups = "1960s",
    columns = population,
    fns = list("min", "max"),
    formatter = fmt_number,
    decimals = 0
  ) %>%
  tab_style(
    style = list(
      cell_text(
        weight = "bold",
        transform = "capitalize"
      ),
      cell_fill(
        color = "lightblue",
        alpha = 0.5
      )
    ),
    locations = cells_stub_summary(
      groups = "1960s"
    )
  )
```

Function ID

7-15

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

cells_summary *Location helper for targeting group summary cells*

Description

The cells_summary() function is used to target the cells in a group summary and it is useful when applying a footnote with tab_footnote() or adding a custom style with tab_style(). The function is expressly used in each of those functions' locations argument. The 'summary' location is generated by the summary_rows() function.

Usage

```
cells_summary(
  groups = everything(),
  columns = everything(),
  rows = everything()
)
```

Arguments

groups	The names of the groups that the summary rows reside in.
columns	The names of the columns that are to be targeted.
rows	The names of the rows that are to be targeted.

Value

A list object with the classes cells_summary and location_cells.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a locations argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- cells_title(): targets the table title or the table subtitle depending on the value given to the groups argument ("title" or "subtitle").
- cells_stubhead(): targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the tab_stubhead() function.
- cells_column_spanners(): targets the spanner column labels with the spanners argument; spanner column labels appear above the column labels.
- cells_column_labels(): targets the column labels with its columns argument.
- cells_row_groups(): targets the row group labels in any available row groups using the groups argument.

- cells_stub(): targets row labels in the table stub using the rows argument.
- cells_body(): targets data cells in the table body using intersections of columns and rows.
- cells_summary(): targets summary cells in the table body using the groups argument and intersections of columns and rows.
- cells_grand_summary(): targets cells of the table's grand summary using intersections of columns and rows
- cells_stub_summary(): targets summary row labels in the table stub using the groups and rows arguments.
- cells_stub_grand_summary(): targets grand summary row labels in the table stub using the rows argument.
- cells_footnotes(): targets all footnotes in the table footer (cannot be used with tab_footnote()).
- cells_source_notes(): targets all source notes in the table footer (cannot be used with tab_footnote()).

When using any of the location helper functions with an appropriate function that has a locations argument (e.g., tab_style()), multiple locations can be targeted by enclosing several cells_*() helper functions in a list() (e.g., list(cells_body(), cells_grand_summary())).

Examples

Use countrypops to create a **gt** table. Add some styling to the summary data cells with with tab_style(), using cells_summary() in locations.

```
countrypops %>%
  dplyr::filter(country_name == "Japan", year < 1970) %>%
  dplyr::select(-contains("country")) %>%
  dplyr::mutate(decade = paste0(substr(year, 1, 3), "0s")) %>%
  gt(
    rowname_col = "year",
    groupname_col = "decade"
  ) %>%
  fmt_number(
    columns = population,
    decimals = 0
  ) %>%
  summary_rows(
    groups = "1960s",
    columns = population,
    fns = list("min", "max"),
    formatter = fmt_number,
    decimals = 0
  ) %>%
  tab_style(
    style = list(
      cell_text(style = "italic"),
      cell_fill(color = "lightblue")
    ),
```

```
    locations = cells_summary(
      groups = "1960s",
      columns = population,
      rows = 1
    )
  ) %>%
  tab_style(
    style = list(
      cell_text(style = "italic"),
      cell_fill(color = "lightgreen")
    ),
    locations = cells_summary(
      groups = "1960s",
      columns = population,
      rows = 2
    )
  )
```

Function ID

7-13

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

cells_title — *Location helper for targeting the table title and subtitle*

Description

The `cells_title()` function is used to target the table title or subtitle when applying a footnote with `tab_footnote()` or adding custom style with `tab_style()`. The function is expressly used in each of those functions' `locations` argument. The header location where the title and optionally the subtitle reside is generated by the `tab_header()` function.

Usage

```
cells_title(groups = c("title", "subtitle"))
```

Arguments

`groups`	We can either specify `"title"`, `"subtitle"`, or both (the default) in a vector to target the title element, the subtitle element, or both elements.

Value

A list object of classes cells_title and location_cells.

Overview of Location Helper Functions

Location helper functions can be used to target cells with virtually any function that has a locations argument. Here is a listing of all of the location helper functions, with locations corresponding roughly from top to bottom of a table:

- cells_title(): targets the table title or the table subtitle depending on the value given to the groups argument ("title" or "subtitle").
- cells_stubhead(): targets the stubhead location, a cell of which is only available when there is a stub; a label in that location can be created by using the tab_stubhead() function.
- cells_column_spanners(): targets the spanner column labels with the spanners argument; spanner column labels appear above the column labels.
- cells_column_labels(): targets the column labels with its columns argument.
- cells_row_groups(): targets the row group labels in any available row groups using the groups argument.
- cells_stub(): targets row labels in the table stub using the rows argument.
- cells_body(): targets data cells in the table body using intersections of columns and rows.
- cells_summary(): targets summary cells in the table body using the groups argument and intersections of columns and rows.
- cells_grand_summary(): targets cells of the table's grand summary using intersections of columns and rows
- cells_stub_summary(): targets summary row labels in the table stub using the groups and rows arguments.
- cells_stub_grand_summary(): targets grand summary row labels in the table stub using the rows argument.
- cells_footnotes(): targets all footnotes in the table footer (cannot be used with tab_footnote()).
- cells_source_notes(): targets all source notes in the table footer (cannot be used with tab_footnote()).

When using any of the location helper functions with an appropriate function that has a locations argument (e.g., tab_style()), multiple locations can be targeted by enclosing several cells_*() helper functions in a list() (e.g., list(cells_body(), cells_grand_summary())).

Examples

Use sp500 to create a **gt** table. Add a header with a title, and then add a footnote to the title with tab_footnote() and cells_title() (in locations).

```
sp500 %>%
  dplyr::filter(date >= "2015-01-05" & date <="2015-01-10") %>%
  dplyr::select(-c(adj_close, volume, high, low)) %>%
  gt() %>%
```

```
tab_header(title = "S&P 500") %>%
tab_footnote(
  footnote = "All values in USD.",
  locations = cells_title(groups = "title")
)
```

Function ID

7-6

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

cell_borders *Helper for defining custom borders for table cells*

Description

The cell_borders() helper function is to be used with the tab_style() function, which itself allows for the setting of custom styles to one or more cells. Specifically, the call to cell_borders() should be bound to the styles argument of tab_style(). The selection argument is where we define which borders should be modified (e.g., "left", "right", etc.). With that selection, the color, style, and weight of the selected borders can then be modified.

Usage

```
cell_borders(sides = "all", color = "#000000", style = "solid", weight = px(1))
```

Arguments

sides	The border sides to be modified. Options include "left", "right", "top", and "bottom". For all borders surrounding the selected cells, we can use the "all" option.
color, style, weight	The border color, style, and weight. The color can be defined with a color name or with a hexadecimal color code. The default color value is "#000000" (black). The style can be one of either "solid" (the default), "dashed", "dotted", or "hidden". The weight of the border lines is to be given in pixel values (the px() helper function is useful for this. The default value for weight is "1px". Borders for any defined sides can be removed by supplying NULL to any of color, style, or weight.

Value

A list object of class `cell_styles`.

Examples

Add horizontal border lines for all table body rows in `exibble` using `tab_style()` and `cell_borders()`.

```
exibble %>%
  gt() %>%
  tab_options(row.striping.include_table_body = FALSE) %>%
  tab_style(
    style = cell_borders(
      sides = c("top", "bottom"),
      color = "red",
      weight = px(1.5),
      style = "solid"
    ),
    locations = cells_body(
      columns = everything(),
      rows = everything()
    )
  )
```

Incorporate different horizontal and vertical borders at several locations. This uses multiple `cell_borders()` and `cells_body()` calls within `list()`s.

```
exibble %>%
  gt() %>%
  tab_style(
    style = list(
      cell_borders(
        sides = c("top", "bottom"),
        color = "#FF0000",
        weight = px(2)
      ),
      cell_borders(
        sides = c("left", "right"),
        color = "#0000FF",
        weight = px(2)
      )
    ),
    locations = list(
      cells_body(
        columns = num,
        rows = is.na(num)
      ),
      cells_body(
        columns = currency,
```

```
      rows = is.na(currency)
    )
  )
)
```

Function ID

7-22

See Also

Other helper functions: adjust_luminance(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

cell_fill   *Helper for defining custom fills for table cells*

Description

The `cell_fill()` helper function is to be used with the `tab_style()` function, which itself allows for the setting of custom styles to one or more cells. Specifically, the call to `cell_fill()` should be bound to the `styles` argument of `tab_style()`.

Usage

```
cell_fill(color = "#D3D3D3", alpha = NULL)
```

Arguments

`color`	The fill color. If nothing is provided, then `"#D3D3D3"` (light gray) will be used as a default.
`alpha`	An optional alpha transparency value for the `color` as single value in the range of `0` (fully transparent) to `1` (fully opaque). If not provided the fill color will either be fully opaque or use alpha information from the color value if it is supplied in the #RRGGBBAA format.

Value

A list object of class `cell_styles`.

Examples

Use exibble to create a **gt** table. Add styles with tab_style() and the cell_fill() helper function.

```
exibble %>%
  dplyr::select(num, currency) %>%
  gt() %>%
  fmt_number(
    columns = c(num, currency),
    decimals = 1
  ) %>%
  tab_style(
    style = cell_fill(color = "lightblue"),
    locations = cells_body(
      columns = num,
      rows = num >= 5000
    )
  ) %>%
  tab_style(
    style = cell_fill(color = "gray85"),
    locations = cells_body(
      columns = currency,
      rows = currency < 100
    )
  )
```

Function ID

7-21

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

cell_text *Helper for defining custom text styles for table cells*

Description

This helper function is to be used with the tab_style() function, which itself allows for the setting of custom styles to one or more cells. We can also define several styles within a single call of cell_text() and tab_style() will reliably apply those styles to the targeted element.

Usage

```
cell_text(
  color = NULL,
  font = NULL,
  size = NULL,
  align = NULL,
  v_align = NULL,
  style = NULL,
  weight = NULL,
  stretch = NULL,
  decorate = NULL,
  transform = NULL,
  whitespace = NULL,
  indent = NULL
)
```

Arguments

`color`	The text color.
`font`	The font or collection of fonts (subsequent font names are) used as fallbacks.
`size`	The size of the font. Can be provided as a number that is assumed to represent px values (or could be wrapped in the `px()`) helper function. We can also use one of the following absolute size keywords: `"xx-small"`, `"x-small"`, `"small"`, `"medium"`, `"large"`, `"x-large"`, or `"xx-large"`.
`align`	The text alignment. Can be one of either `"center"`, `"left"`, `"right"`, or `"justify"`.
`v_align`	The vertical alignment of the text in the cell. Options are `"middle"`, `"top"`, or `"bottom"`.
`style`	The text style. Can be one of either `"normal"`, `"italic"`, or `"oblique"`.
`weight`	The weight of the font. Can be a text-based keyword such as `"normal"`, `"bold"`, `"lighter"`, `"bolder"`, or, a numeric value between `1` and `1000`, inclusive. Note that only variable fonts may support the numeric mapping of weight.
`stretch`	Allows for text to either be condensed or expanded. We can use one of the following text-based keywords to describe the degree of condensation/expansion: `"ultra-condensed"`, `"extra-condensed"`, `"condensed"`, `"semi-condensed"`, `"normal"`, `"semi-expanded"`, `"expanded"`, `"extra-expanded"`, or `"ultra-expanded"`. Alternatively, we can supply percentage values from `0\%` to `200\%`, inclusive. Negative percentage values are not allowed.
`decorate`	Allows for text decoration effect to be applied. Here, we can use `"overline"`, `"line-through"`, or `"underline"`.
`transform`	Allows for the transformation of text. Options are `"uppercase"`, `"lowercase"`, or `"capitalize"`.
`whitespace`	A white-space preservation option. By default, runs of white-space will be collapsed into single spaces but several options exist to govern how white-space

is collapsed and how lines might wrap at soft-wrap opportunities. The keyword options are "normal", "nowrap", "pre", "pre-wrap", "pre-line", and "break-spaces".

indent The indentation of the text. Can be provided as a number that is assumed to represent px values (or could be wrapped in the px()) helper function. Alternatively, this can be given as a percentage (easily constructed with pct()).

Value

A list object of class cell_styles.

Examples

Use exibble to create a **gt** table. Add styles with tab_style() and the cell_text() helper function.

```
exibble %>%
  dplyr::select(num, currency) %>%
  gt() %>%
  fmt_number(
    columns = c(num, currency),
    decimals = 1
  ) %>%
  tab_style(
    style = cell_text(weight = "bold"),
    locations = cells_body(
      columns = num,
      rows = num >= 5000
    )
  ) %>%
  tab_style(
    style = cell_text(style = "italic"),
    locations = cells_body(
      columns = currency,
      rows = currency < 100
    )
  )
```

Function ID

7-20

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

`cols_align` *Set the alignment of columns*

Description

The individual alignments of columns (which includes the column labels and all of their data cells) can be modified. We have the option to align text to the `left`, the `center`, and the `right`. In a less explicit manner, we can allow **gt** to automatically choose the alignment of each column based on the data type (with the `auto` option).

Usage

```
cols_align(
  data,
  align = c("auto", "left", "center", "right"),
  columns = everything()
)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`align`	The alignment type. This can be any of `"center"`, `"left"`, or `"right"` for center-, left-, or right-alignment. Alternatively, the `"auto"` option (the default), will automatically align values in columns according to the data type (see the Details section for specifics on which alignments are applied).
`columns`	The columns for which the alignment should be applied. By default this is set to `everything()` which means that the chosen alignment affects all columns.

Details

When you create a **gt** table object using `gt()`, automatic alignment of column labels and their data cells is performed. By default, left-alignment is applied to columns of class `character`, `Date`, or `POSIXct`; center-alignment is for columns of class `logical`, `factor`, or `list`; and right-alignment is used for the `numeric` and `integer` columns.

Value

An object of class `gt_tbl`.

Examples

Use `countrypops` to create a **gt** table. Align the `population` column data to the left.

```
countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
```

```
gt() %>%
cols_align(
  align = "left",
  columns = population
)
```

Function ID

4-1

See Also

Other column modification functions: cols_align_decimal(), cols_hide(), cols_label(), cols_merge_n_pct(), cols_merge_range(), cols_merge_uncert(), cols_merge(), cols_move_to_end(), cols_move_to_start(), cols_move(), cols_unhide(), cols_width()

`cols_align_decimal` *Align all numeric values in a column along the decimal mark*

Description

For numeric columns that contain values with decimal portions, it is sometimes useful to have them lined up along the decimal mark for easier readability. We can do this with `cols_align_decimal()` and provide any number of columns (the function will skip over columns that don't require this type of alignment).

Usage

```
cols_align_decimal(data, columns = everything(), dec_mark = ".", locale = NULL)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`columns`	The columns for which the alignment should be applied. By default this is set to `everything()` which means that the chosen alignment affects all columns.
`dec_mark`	The character used as a decimal mark in the numeric values to be aligned. If a locale value was used when formatting the numeric values then `locale` is better to use and it will override any value here in `dec_mark`.
`locale`	An optional locale ID that can be used to obtain the type of decimal mark used in the numeric values to be aligned. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any value provided in `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported. Any `locale` value provided here will override any global locale setting performed in `gt()`'s own `locale` argument.

Value

An object of class gt_tbl.

Examples

Let's put together a two-column table to create a **gt** table. The first column char just contains letters whereas the second column, num, has a collection of numbers and NA values. We could format the numbers with fmt_number() and elect to drop the trailing zeros past the decimal mark with drop_trailing_zeros = TRUE. This can leave formatted numbers that are hard to scan through because the decimal mark isn't fixed horizontally. We could remedy this and align the numbers by the decimal mark with cols_align_decimal().

```
dplyr::tibble(
  char = LETTERS[1:9],
  num = c(1.2, -33.52, 9023.2, -283.527, NA, 0.401, -123.1, NA, 41)
) %>%
  gt() %>%
  fmt_number(
    columns = num,
    decimals = 3,
    drop_trailing_zeros = TRUE
  ) %>%
  cols_align_decimal()
```

Function ID

4-2

See Also

Other column modification functions: cols_align(), cols_hide(), cols_label(), cols_merge_n_pct(), cols_merge_range(), cols_merge_uncert(), cols_merge(), cols_move_to_end(), cols_move_to_start(), cols_move(), cols_unhide(), cols_width()

---

cols_hide *Hide one or more columns*

Description

The cols_hide() function allows us to hide one or more columns from appearing in the final output table. While it's possible and often desirable to omit columns from the input table data before introduction to the gt() function, there can be cases where the data in certain columns is useful (as a column reference during formatting of other columns) but the final display of those columns is not necessary.

Usage

```
cols_hide(data, columns)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`columns`	The column names to hide from the output display table. Values provided that do not correspond to column names will be disregarded.

Details

The hiding of columns is internally a rendering directive, so, all columns that are 'hidden' are still accessible and useful in any expression provided to a `rows` argument. Furthermore, the `cols_hide()` function (as with many **gt** functions) can be placed anywhere in a pipeline of **gt** function calls (acting as a promise to hide columns when the timing is right). However there's perhaps greater readability when placing this call closer to the end of such a pipeline. The `cols_hide()` function quietly changes the visible state of a column (much like the `cols_unhide()` function) and doesn't yield warnings or messages when changing the state of already-invisible columns.

Value

An object of class `gt_tbl`.

Examples

Use `countrypops` to create a **gt** table. Hide the `country_code_2` and `country_code_3` columns with `cols_hide()`.

```
countrypops %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_hide(columns = c(country_code_2, country_code_3))
```

Use `countrypops` to create a **gt** table. Use the `population` column to provide the conditional placement of footnotes, then hide that column and one other. Note that the order of the `cols_hide()` and `tab_footnote()` statements has no effect.

```
countrypops %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_hide(columns = c(country_code_3, population)) %>%
  tab_footnote(
    footnote = "Population above 3,000,000.",
    locations = cells_body(
      columns = year,
      rows = population > 3000000
    )
  )
```

Function ID

4-8

See Also

`cols_unhide()` to perform the inverse operation.

Other column modification functions: `cols_align_decimal()`, `cols_align()`, `cols_label()`, `cols_merge_n_pct()`, `cols_merge_range()`, `cols_merge_uncert()`, `cols_merge()`, `cols_move_to_end()`, `cols_move_to_start()`, `cols_move()`, `cols_unhide()`, `cols_width()`

`cols_label` *Relabel one or more columns*

Description

Column labels can be modified from their default values (the names of the columns from the input table data). When you create a **gt** table object using `gt()`, column names effectively become the column labels. While this serves as a good first approximation, column names aren't often appealing as column labels in a **gt** output table. The `cols_label()` function provides the flexibility to relabel one or more columns and we even have the option to use the `md()` or `html()` helper functions for rendering column labels from Markdown or using HTML.

Usage

```
cols_label(.data, ..., .list = list2(...))
```

Arguments

`.data`	A table object that is created using the `gt()` function.
`...`	One or more named arguments of column names from the input `.data` table along with their labels for display as the column labels. We can optionally wrap the column labels with `md()` (to interpret text as Markdown) or `html()` (to interpret text as HTML).
`.list`	Allows for the use of a list as an input alternative to `...`.

Details

It's important to note that while columns can be freely relabeled, we continue to refer to columns by their original column names. Column names in a tibble or data frame must be unique whereas column labels in **gt** have no requirement for uniqueness (which is useful for labeling columns as, say, measurement units that may be repeated several times—usually under different spanner column labels). Thus, we can still easily distinguish between columns in other **gt** function calls (e.g., in all of the `fmt*()` functions) even though we may lose distinguishability in column labels once they have been relabeled.

Value

An object of class `gt_tbl`.

Examples

Use `countrypops` to create a **gt** table. Relabel all the table's columns with the `cols_label()` function to improve its presentation.

```
countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_label(
    country_name = "Name",
    year = "Year",
    population = "Population"
  )
```

#'

Using `countrypops` again to create a **gt** table, we label columns just as before but this time make the column labels bold through Markdown formatting.

```
countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_label(
    country_name = md("**Name**"),
    year = md("**Year**"),
    population = md("**Population**")
  )
```

Function ID

4-4

See Also

Other column modification functions: `cols_align_decimal()`, `cols_align()`, `cols_hide()`, `cols_merge_n_pct()`, `cols_merge_range()`, `cols_merge_uncert()`, `cols_merge()`, `cols_move_to_end()`, `cols_move_to_start()`, `cols_move()`, `cols_unhide()`, `cols_width()`

cols_merge *Merge data from two or more columns to a single column*

Description

This function takes input from two or more columns and allows the contents to be merged them into a single column, using a pattern that specifies the formatting. We can specify which columns to merge together in the `columns` argument. The string-combining pattern is given in the `pattern` argument. The first column in the `columns` series operates as the target column (i.e., will undergo mutation) whereas all following `columns` will be untouched. There is the option to hide the non-target columns (i.e., second and subsequent columns given in `columns`).

Usage

```
cols_merge(
  data,
  columns,
  hide_columns = columns[-1],
  pattern = paste0("{", seq_along(columns), "}", collapse = " ")
)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`columns`	The columns that will participate in the merging process. The first column name provided will be the target column (i.e., undergo mutation) and the other columns will serve to provide input.
`hide_columns`	Any column names provided here will have their state changed to `hidden` (via internal use of `cols_hide()` if they aren't already hidden. This is convenient if the shared purpose of these specified columns is only to provide string input to the target column. To suppress any hiding of columns, `FALSE` can be used here.
`pattern`	A formatting pattern that specifies the arrangement of the `column` values and any string literals. We need to use column numbers (corresponding to the position of columns provided in `columns`) within the pattern. These indices are to be placed in curly braces (e.g., `{1}`). All characters outside of braces are taken to be string literals.

Details

There are three other column-merging functions that offer specialized behavior that is optimized for common table tasks: `cols_merge_range()`, `cols_merge_uncert()`, and `cols_merge_n_pct()`. These functions operate similarly, where the non-target columns can be optionally hidden from the output table through the `autohide` option.

Value

An object of class `gt_tbl`.

Examples

Use sp500 to create a **gt** table. Use the `cols_merge()` function to merge the `open` & `close` columns together, and, the `low` & `high` columns (putting an em dash between both). Relabel the columns with `cols_label()`.

```
sp500 %>%
  dplyr::slice(50:55) %>%
  dplyr::select(-volume, -adj_close) %>%
  gt() %>%
  cols_merge(
    columns = c(open, close),
    pattern = "{1}&mdash;{2}"
  ) %>%
  cols_merge(
    columns = c(low, high),
    pattern = "{1}&mdash;{2}"
  ) %>%
  cols_label(
    open = "open/close",
    low = "low/high"
  )
```

Function ID

4-13

See Also

Other column modification functions: `cols_align_decimal()`, `cols_align()`, `cols_hide()`, `cols_label()`, `cols_merge_n_pct()`, `cols_merge_range()`, `cols_merge_uncert()`, `cols_move_to_end()`, `cols_move_to_start()`, `cols_move()`, `cols_unhide()`, `cols_width()`

`cols_merge_n_pct` *Merge two columns to combine counts and percentages*

Description

The `cols_merge_n_pct()` function is a specialized variant of the `cols_merge()` function. It operates by taking two columns that constitute both a count (`col_n`) and a fraction of the total population (`col_pct`) and merges them into a single column. What results is a column containing both counts and their associated percentages (e.g., `12 (23.2%)`). The column specified in `col_pct` is dropped from the output table.

Usage

```
cols_merge_n_pct(data, col_n, col_pct, autohide = TRUE)
```

Arguments

data	A table object that is created using the gt() function.
col_n	A column that contains values for the count component.
col_pct	A column that contains values for the percentage component. This column should be formatted such that percentages are displayed (e.g., with fmt_percent()).
autohide	An option to automatically hide the column specified as col_pct. Any columns with their state changed to hidden will behave the same as before, they just won't be displayed in the finalized table.

Details

This function could be somewhat replicated using cols_merge(), however, cols_merge_n_pct() employs the following specialized semantics for NA and zero-value handling:

1. NAs in col_n result in missing values for the merged column (e.g., NA + 10.2% = NA)
2. NAs in col_pct (but not col_n) result in base values only for the merged column (e.g., 13 + NA = 13)
3. NAs both col_n and col_pct result in missing values for the merged column (e.g., NA + NA = NA)
4. If a zero (0) value is in col_n then the formatted output will be "0" (i.e., no percentage will be shown)

Any resulting NA values in the col_n column following the merge operation can be easily formatted using the sub_missing() function. Separate calls of sub_missing() can be used for the col_n and col_pct columns for finer control of the replacement values. It is the responsibility of the user to ensure that values are correct in both the col_n and col_pct columns (this function neither generates nor recalculates values in either). Formatting of each column can be done independently in separate fmt_number() and fmt_percent() calls.

This function is part of a set of four column-merging functions. The other two are the general cols_merge() function and the specialized cols_merge_uncert() and cols_merge_range() functions. These functions operate similarly, where the non-target columns can be optionally hidden from the output table through the hide_columns or autohide options.

Value

An object of class gt_tbl.

Examples

Use pizzaplace to create a **gt** table that displays the counts and percentages of the top 3 pizzas sold by pizza category in 2015. The cols_merge_n_pct() function is used to merge the n and frac columns (and the frac column is formatted using fmt_percent()).

```
pizzaplace %>%
  dplyr::group_by(name, type, price) %>%
  dplyr::summarize(
    n = dplyr::n(),
```

```
    frac = n/nrow(.),
    .groups = "drop"
  ) %>%
  dplyr::arrange(type, dplyr::desc(n)) %>%
  dplyr::group_by(type) %>%
  dplyr::slice_head(n = 3) %>%
  gt(
    rowname_col = "name",
    groupname_col = "type"
  ) %>%
  fmt_currency(price) %>%
  fmt_percent(frac) %>%
  cols_merge_n_pct(
    col_n = n,
    col_pct = frac
  ) %>%
  cols_label(
    n = md("*N* (%)"),
    price = "Price"
  ) %>%
  tab_style(
    style = cell_text(font = "monospace"),
    locations = cells_stub()
  ) %>%
  tab_stubhead(md("Cat. and  \nPizza Code")) %>%
  tab_header(title = "Top 3 Pizzas Sold by Category in 2015") %>%
  tab_options(table.width = px(512))
```

Function ID

4-12

See Also

Other column modification functions: `cols_align_decimal()`, `cols_align()`, `cols_hide()`, `cols_label()`, `cols_merge_range()`, `cols_merge_uncert()`, `cols_merge()`, `cols_move_to_end()`, `cols_move_to_start()`, `cols_move()`, `cols_unhide()`, `cols_width()`

`cols_merge_range` *Merge two columns to a value range column*

Description

The `cols_merge_range()` function is a specialized variant of the `cols_merge()` function. It operates by taking a two columns that constitute a range of values (`col_begin` and `col_end`) and merges them into a single column. What results is a column containing both values separated by a long dash (e.g., `12.0  20.0`). The column specified in `col_end` is dropped from the output table.

Usage

```
cols_merge_range(data, col_begin, col_end, sep = "--", autohide = TRUE)
```

Arguments

data	A table object that is created using the gt() function.
col_begin	A column that contains values for the start of the range.
col_end	A column that contains values for the end of the range.
sep	The separator text that indicates the values are ranged. The default value of "--" indicates that an en dash will be used for the range separator. Using "---" will be taken to mean that an em dash should be used. Should you want these special symbols to be taken literally, they can be supplied within the base I() function.
autohide	An option to automatically hide the column specified as col_end. Any columns with their state changed to hidden will behave the same as before, they just won't be displayed in the finalized table.

Details

This function could be somewhat replicated using cols_merge(), however, cols_merge_range() employs the following specialized operations for NA handling:

1. NAs in col_begin (but not col_end) result in a display of only
2. NAs in col_end (but not col_begin) result in a display of only the col_begin values only for the merged column (this is the converse of the previous)
3. NAs both in col_begin and col_end result in missing values for the merged column

Any resulting NA values in the col_begin column following the merge operation can be easily formatted using the sub_missing() function. Separate calls of sub_missing() can be used for the col_begin and col_end columns for finer control of the replacement values.

This function is part of a set of four column-merging functions. The other two are the general cols_merge() function and the specialized cols_merge_uncert() and cols_merge_n_pct() functions. These functions operate similarly, where the non-target columns can be optionally hidden from the output table through the hide_columns or autohide options.

Value

An object of class gt_tbl.

Examples

Use gtcars to create a **gt** table, keeping only the model, mpg_c, and mpg_h columns. Merge the "mpg*" columns together as a single range column (which is labeled as MPG, in italics) using the cols_merge_range() function.

```
gtcars %>%
  dplyr::select(model, starts_with("mpg")) %>%
  dplyr::slice(1:8) %>%
```

```
gt() %>%
cols_merge_range(
  col_begin = mpg_c,
  col_end = mpg_h
) %>%
cols_label(mpg_c = md("*MPG*"))
```

Function ID

4-11

See Also

Other column modification functions: cols_align_decimal(), cols_align(), cols_hide(), cols_label(), cols_merge_n_pct(), cols_merge_uncert(), cols_merge(), cols_move_to_end(), cols_move_to_start(), cols_move(), cols_unhide(), cols_width()

---

cols_merge_uncert *Merge columns to a value-with-uncertainty column*

Description

The cols_merge_uncert() function is a specialized variant of the cols_merge() function. It takes as input a base value column (col_val) and either: (1) a single uncertainty column, or (2) two columns representing lower and upper uncertainty bounds. These columns will be essentially merged in a single column (that of col_val). What results is a column with values and associated uncertainties (e.g., 12.0 ± 0.1), and any columns specified in col_uncert are hidden from appearing the output table.

Usage

```
cols_merge_uncert(data, col_val, col_uncert, sep = " +/- ", autohide = TRUE)
```

Arguments

data A table object that is created using the gt() function.

col_val A single column name that contains the base values. This is the column where values will be mutated.

col_uncert Either one or two column names that contain the uncertainty values. The most common case involves supplying a single column with uncertainties; these values will be combined with those in col_val. Less commonly, lower and upper uncertainty bounds may be different. For that case two columns (representing lower and upper uncertainty values away from col_val, respectively) should be provided. Since we often don't want the uncertainty value columns in the output table, we can automatically hide any col_uncert columns through the autohide option.

`sep` The separator text that contains the uncertainty mark for a single uncertainty value. The default value of `" +/- "` indicates that an appropriate plus/minus mark will be used depending on the output context. Should you want this special symbol to be taken literally, it can be supplied within the `I()` function.

`autohide` An option to automatically hide any columns specified in `col_uncert`. Any columns with their state changed to 'hidden' will behave the same as before, they just won't be displayed in the finalized table. By default, this is set to `TRUE`.

Details

This function could be somewhat replicated using `cols_merge()` in the case where a single column is supplied for `col_uncert`, however, `cols_merge_uncert()` employs the following specialized semantics for `NA` handling:

1. `NA`s in `col_val` result in missing values for the merged column (e.g., `NA` + `0.1` = `NA`)
2. `NA`s in `col_uncert` (but not `col_val`) result in base values only for the merged column (e.g., `12.0` + `NA` = `12.0`)
3. `NA`s both `col_val` and `col_uncert` result in missing values for the merged column (e.g., `NA` + `NA` = `NA`)

Any resulting `NA` values in the `col_val` column following the merge operation can be easily formatted using the `sub_missing()` function.

This function is part of a set of four column-merging functions. The other two are the general `cols_merge()` function and the specialized `cols_merge_range()` and `cols_merge_n_pct()` functions. These functions operate similarly, where the non-target columns can be optionally hidden from the output table through the `hide_columns` or `autohide` options.

Value

An object of class `gt_tbl`.

Examples

Use `exibble` to create a **gt** table, keeping only the `currency` and `num` columns. Merge columns into one with a base value and uncertainty (after formatting the `num` column) using the `cols_merge_uncert()` function.

```
exibble %>%
  dplyr::select(currency, num) %>%
  dplyr::slice(1:7) %>%
  gt() %>%
  fmt_number(
    columns = num,
    decimals = 3,
    use_seps = FALSE
  ) %>%
  cols_merge_uncert(
    col_val = currency,
    col_uncert = num
```

```
) %>%
cols_label(currency = "value + uncert.")
```

Function ID

4-10

See Also

Other column modification functions: cols_align_decimal(), cols_align(), cols_hide(), cols_label(), cols_merge_n_pct(), cols_merge_range(), cols_merge(), cols_move_to_end(), cols_move_to_start(), cols_move(), cols_unhide(), cols_width()

`cols_move` *Move one or more columns*

Description

On those occasions where you need to move columns this way or that way, we can make use of the `cols_move()` function. While it's true that the movement of columns can be done upstream of **gt**, it is much easier and less error prone to use the function provided here. The movement procedure here takes one or more specified columns (in the `columns` argument) and places them to the right of a different column (the `after` argument). The ordering of the `columns` to be moved is preserved, as is the ordering of all other columns in the table.

Usage

```
cols_move(data, columns, after)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`columns`	The column names to move to as a group to a different position. The order of the remaining columns will be preserved.
`after`	A column name used to anchor the insertion of the moved columns. All of the moved columns will be placed to the right of this column.

Details

The columns supplied in `columns` must all exist in the table and none of them can be in the `after` argument. The `after` column must also exist and only one column should be provided here. If you need to place one or columns at the beginning of the column series, the `cols_move_to_start()` function should be used. Similarly, if those columns to move should be placed at the end of the column series then use `cols_move_to_end()`.

Value

An object of class `gt_tbl`.

Examples

Use countrypops to create a **gt** table. With the remaining columns, position population after country_name with the cols_move() function.

```
countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_move(
    columns = population,
    after = country_name
  )
```

Function ID

4-7

See Also

Other column modification functions: cols_align_decimal(), cols_align(), cols_hide(), cols_label(), cols_merge_n_pct(), cols_merge_range(), cols_merge_uncert(), cols_merge(), cols_move_to_end(), cols_move_to_start(), cols_unhide(), cols_width()

---

cols_move_to_end *Move one or more columns to the end*

Description

It's possible to move a set of columns to the end of the column series, we only need to specify which columns are to be moved. While this can be done upstream of **gt**, this function makes to process much easier and it's less error prone. The ordering of the columns that are moved to the end is preserved (same with the ordering of all other columns in the table).

Usage

```
cols_move_to_end(data, columns)
```

Arguments

data	A table object that is created using the gt() function.
columns	The column names to move to the right-most side of the table. The order in which columns are provided will be preserved (as is the case with the remaining columns).

Details

The columns supplied in columns must all exist in the table. If you need to place one or columns at the start of the column series, the cols_move_to_start() function should be used. More control is offered with the cols_move() function, where columns could be placed after a specific column.

Value

An object of class gt_tbl.

Examples

Use countrypops to create a **gt** table. With the remaining columns, move the year column to the end of the column series with the cols_move_to_end() function.

```
countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_move_to_end(columns = year)
```

Use countrypops to create a **gt** table. With the remaining columns, move year and country_name to the end of the column series.

```
countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_move_to_end(columns = c(year, country_name))
```

Function ID

4-6

See Also

Other column modification functions: cols_align_decimal(), cols_align(), cols_hide(), cols_label(), cols_merge_n_pct(), cols_merge_range(), cols_merge_uncert(), cols_merge(), cols_move_to_start(), cols_move(), cols_unhide(), cols_width()

cols_move_to_start *Move one or more columns to the start*

Description

We can easily move set of columns to the beginning of the column series and we only need to specify which columns. It's possible to do this upstream of **gt**, however, it is easier with this function and it presents less possibility for error. The ordering of the columns that are moved to the start is preserved (same with the ordering of all other columns in the table).

Usage

```
cols_move_to_start(data, columns)
```

Arguments

data	A table object that is created using the gt() function.
columns	The column names to move to the left-most side of the table. The order in which columns are provided will be preserved (as is the case with the remaining columns).

Details

The columns supplied in columns must all exist in the table. If you need to place one or columns at the end of the column series, the cols_move_to_end() function should be used. More control is offered with the cols_move() function, where columns could be placed after a specific column.

Value

An object of class gt_tbl.

Examples

Use countrypops to create a **gt** table. With the remaining columns, move the year column to the start of the column series with cols_move_to_start().

```
countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_move_to_start(columns = year)
```

Use countrypops to create a **gt** table. With the remaining columns, move year and population to the start.

```
countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_move_to_start(columns = c(year, population))
```

Function ID

4-5

See Also

Other column modification functions: `cols_align_decimal()`, `cols_align()`, `cols_hide()`, `cols_label()`, `cols_merge_n_pct()`, `cols_merge_range()`, `cols_merge_uncert()`, `cols_merge()`, `cols_move_to_end()`, `cols_move()`, `cols_unhide()`, `cols_width()`

`cols_unhide` *Unhide one or more columns*

Description

The `cols_unhide()` function allows us to take one or more hidden columns (usually made so via the `cols_hide()` function) and make them visible in the final output table. This may be important in cases where the user obtains a `gt_tbl` object with hidden columns and there is motivation to reveal one or more of those.

Usage

```
cols_unhide(data, columns)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`columns`	The column names to unhide from the output display table. Values provided that do not correspond to column names will be disregarded.

Details

The hiding and unhiding of columns is internally a rendering directive, so, all columns that are 'hidden' are still accessible and useful in any expression provided to a `rows` argument. The `cols_unhide()` function quietly changes the visible state of a column (much like the `cols_hide()` function) and doesn't yield warnings or messages when changing the state of already-visible columns.

Value

An object of class `gt_tbl`.

Examples

Use countrypops to create a **gt** table. Hide the country_code_2 and country_code_3 columns with cols_hide().

```
tab_1 <-
  countrypops %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_hide(columns = c(country_code_2, country_code_3))

tab_1
```

If the tab_1 object is provided without the code or source data to regenerate it, and, the user wants to reveal otherwise hidden columns then the cols_unhide() function becomes useful.

```
tab_1 %>% cols_unhide(columns = country_code_2)
```

Function ID

4-9

See Also

cols_hide() to perform the inverse operation.

Other column modification functions: cols_align_decimal(), cols_align(), cols_hide(), cols_label(), cols_merge_n_pct(), cols_merge_range(), cols_merge_uncert(), cols_merge(), cols_move_to_end(), cols_move_to_start(), cols_move(), cols_width()

---

cols_width *Set the widths of columns*

Description

Manual specifications of column widths can be performed using the cols_width() function. We choose which columns get specific widths. This can be in units of pixels (easily set by use of the px() helper function), or, as percentages (where the pct() helper function is useful). Width assignments are supplied in ... through two-sided formulas, where the left-hand side defines the target columns and the right-hand side is a single dimension.

Usage

```
cols_width(.data, ..., .list = list2(...))
```

Arguments

.data
: A table object that is created using the `gt()` function.

...
: Expressions for the assignment of column widths for the table columns in `.data`. Two-sided formulas (e.g, `<LHS> ~ <RHS>`) can be used, where the left-hand side corresponds to selections of columns and the right-hand side evaluates to single-length character values in the form `{##}px` (i.e., pixel dimensions); the `px()` helper function is best used for this purpose. Column names should be enclosed in `c()`. The column-based select helpers `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, and `everything()` can be used in the LHS. Subsequent expressions that operate on the columns assigned previously will result in overwriting column width values (both in the same `cols_width()` call and across separate calls). All other columns can be assigned a default width value by using `everything()` on the left-hand side.

.list
: Allows for the use of a list as an input alternative to `...`.

Details

Column widths can be set as absolute or relative values (with px and percentage values). Those columns not specified are treated as having variable width. The sizing behavior for column widths depends on the combination of value types, and, whether a table width has been set (which could, itself, be expressed as an absolute or relative value). Widths for the table and its container can be individually modified with the `table.width` and `container.width` arguments within `tab_options()`).

Value

An object of class `gt_tbl`.

Examples

Use `exibble` to create a **gt** table. We can specify the widths of columns with `cols_width()`. This is done with named arguments in `...`, specifying the exact widths for table columns (using `everything()` at the end will capture all remaining columns).

```
exibble %>%
  dplyr::select(
    num, char, date,
    datetime, row
  ) %>%
  gt() %>%
  cols_width(
    num ~ px(150),
    ends_with("r") ~ px(100),
    starts_with("date") ~ px(200),
    everything() ~ px(60)
  )
```

Function ID

4-3

See Also

Other column modification functions: `cols_align_decimal()`, `cols_align()`, `cols_hide()`, `cols_label()`, `cols_merge_n_pct()`, `cols_merge_range()`, `cols_merge_uncert()`, `cols_merge()`, `cols_move_to_end()`, `cols_move_to_start()`, `cols_move()`, `cols_unhide()`

`countrypops` *Yearly populations of countries from 1960 to 2017*

Description

A dataset that presents yearly, total populations of countries. Total population is based on counts of all residents regardless of legal status or citizenship. Country identifiers include the English-language country names, and the 2- and 3-letter ISO 3166-1 country codes. Each row contains a `population` value for a given `year` (from 1960 to 2017). Any `NA` values for `populations` indicate the non-existence of the country during that year.

Usage

```
countrypops
```

Format

A tibble with 12470 rows and 5 variables:

country_name Name of the country

country_code_2 The 2-letter ISO 3166-1 country code

country_code_3 The 3-letter ISO 3166-1 country code

year The year for the population estimate

population The population estimate, midway through the year

Examples

Here is a glimpse at the data available in `countrypops`.

```
dplyr::glimpse(countrypops)
#> Rows: 12,470
#> Columns: 5
#> $ country_name   <chr> "Aruba", "Aruba", "Aruba", "Aruba", "Aruba", "Aruba", "~
#> $ country_code_2 <chr> "AW", "AW", "AW", "AW", "AW", "AW", "AW", "AW", "AW", "~
#> $ country_code_3 <chr> "ABW", "ABW", "ABW", "ABW", "ABW", "ABW", "ABW", "ABW",~
#> $ year           <int> 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1~
#> $ population     <int> 54211, 55438, 56225, 56695, 57032, 57360, 57715, 58055,~
```

Function ID

11-1

Source

https://data.worldbank.org/indicator/SP.POP.TOTL

See Also

Other datasets: exibble, gtcars, pizzaplace, sp500, sza

`currency` *Supply a custom currency symbol to* `fmt_currency()`

Description

The `currency()` helper function makes it easy to specify a context-aware currency symbol to `currency` argument of `fmt_currency()`. Since **gt** can render tables to several output formats, `currency()` allows for different variations of the custom symbol based on the output context (which are `html`, `latex`, `rtf`, and `default`). The number of decimal places for the custom currency defaults to `2`, however, a value set for the `decimals` argument of `fmt_currency()` will take precedence.

Usage

```
currency(..., .list = list2(...))
```

Arguments

`...`	One or more named arguments using output contexts as the names and currency symbol text as the values.
`.list`	Allows for the use of a list as an input alternative to `...`.

Details

We can use any combination of `html`, `latex`, `rtf`, and `default` as named arguments for the currency text in each of the namesake contexts. The `default` value is used as a fallback when there doesn't exist a dedicated currency text value for a particular output context (e.g., when a table is rendered as HTML and we use `currency(latex = "LTC", default = "ltc")`, the currency symbol will be `"ltc"`. For convenience, if we provide only a single string without a name, it will be taken as the `default` (i.e., `currency("ltc")` is equivalent to `currency(default = "ltc")`). However, if we were to specify currency strings for multiple output contexts, names are required each and every context.

Value

A list object of class `gt_currency`.

Examples

Use `exibble` to create a **gt** table. Format the `currency` column to have currency values in guilder (a defunct Dutch currency).

```
exibble %>%
  gt() %>%
  fmt_currency(
    columns = currency,
    currency = currency(
      html = "&fnof;",
      default = "f"
    ),
    decimals = 2
  )
```

Function ID

7-19

See Also

Other helper functions: `adjust_luminance()`, `cell_borders()`, `cell_fill()`, `cell_text()`, `cells_body()`, `cells_column_labels()`, `cells_column_spanners()`, `cells_footnotes()`, `cells_grand_summary()`, `cells_row_groups()`, `cells_source_notes()`, `cells_stub_grand_summary()`, `cells_stub_summary()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`, `stub()`

---

data_color *Set data cell colors using a palette or a color function*

Description

It's possible to add color to data cells according to their values. The `data_color()` function colors all rows of any `columns` supplied. There are two ways to define how cells are colored: (1) through the use of a supplied color palette, and (2) through use of a color mapping function available from the **scales** package. The first method colorizes cell data according to whether values are character or numeric. The second method provides more control over how cells are colored since we provide an explicit color function and thus other requirements such as bin counts, cut points, or a numeric domain. Finally, we can choose whether to apply the cell-specific colors to either the cell background or the cell text.

Usage

```
data_color(
  data,
  columns,
  colors,
```

```
  alpha = NULL,
  apply_to = c("fill", "text"),
  autocolor_text = TRUE,
  contrast_algo = c("apca", "wcag")
)
```

Arguments

data
: A table object that is created using the gt() function.

columns
: The columns wherein changes to cell data colors should occur.

colors
: Either a color mapping function from the **scales** package or a vector of colors to use for each distinct value or level in each of the provided columns. The color mapping functions are: scales::col_quantile(), scales::col_bin(), scales::col_numeric(), and scales::col_factor(). If providing a vector of colors as a palette, each color value provided must either be a color name (in the set of colors provided by grDevices::colors()) or a hexadecimal string in the form of "#RRGGBB" or "#RRGGBBAA".

alpha
: An optional, fixed alpha transparency value that will be applied to all of the colors provided (regardless of whether a color palette was directly supplied or generated through a color mapping function).

apply_to
: Which style element should the colors be applied to? Options include the cell background (the default, given as "fill") or the cell text ("text").

autocolor_text
: An option to let **gt** modify the coloring of text within cells undergoing background coloring. This will result in better text-to-background color contrast. By default, this is set to TRUE.

contrast_algo
: The color contrast algorithm to use when autocolor_text = TRUE. By default this is "apca" (Accessible Perceptual Contrast Algorithm) and the alternative to this is "wcag" (Web Content Accessibility Guidelines).

Details

The col_*() color mapping functions from the scales package can be used in the colors argument. These functions map data values (numeric or factor/character) to colors according to the provided palette.

- scales::col_numeric(): provides a simple linear mapping from continuous numeric data to an interpolated palette.
- scales::col_bin(): provides a mapping of continuous numeric data to value-based bins. This internally uses the base::cut() function.
- scales::col_quantile(): provides a mapping of continuous numeric data to quantiles. This internally uses the stats::quantile() function.
- scales::col_factor(): provides a mapping of factors to colors. If the palette is discrete and has a different number of colors than the number of factors, interpolation is used.

By default, **gt** will choose the ideal text color (for maximal contrast) when colorizing the background of data cells. This option can be disabled by setting autocolor_text to FALSE.

Choosing the right color palette can often be difficult because it's both hard to discover suitable palettes and then obtain the vector of colors. To make this process easier we can elect to use the **paletteer** package, which makes a wide range of palettes from various R packages readily available. The `info_paletteer()` information table allows us to easily inspect all of the discrete color palettes available in **paletteer**. We only then need to specify the `package` and `palette` when calling the `paletteer::paletteer_d()` function, and, we get the palette as a vector of hexadecimal colors.

Value

An object of class `gt_tbl`.

Examples

Use `countrypops` to create a **gt** table. Apply a color scale to the `population` column with `scales::col_numeric`, four supplied colors, and a domain.

```
countrypops %>%
  dplyr::filter(country_name == "Mongolia") %>%
  dplyr::select(-contains("code")) %>%
  tail(10) %>%
  gt() %>%
  data_color(
    columns = population,
    colors = scales::col_numeric(
      palette = c("red", "orange", "green", "blue"),
      domain = c(0.2E7, 0.4E7)
    )
  )
```

Use `pizzaplace` to create a **gt** table. Apply colors from the `"ggsci::red_material"` palette (it's in the **ggsci** R package but more easily gotten from the **paletteer** package, info at `info_paletteer()`) to to `sold` and `income` columns. Setting the `domain` of `scales::col_numeric()` to `NULL` will use the bounds of the available data as the domain.

```
pizzaplace %>%
  dplyr::filter(type %in% c("chicken", "supreme")) %>%
  dplyr::group_by(type, size) %>%
  dplyr::summarize(
    sold = dplyr::n(),
    income = sum(price),
    .groups = "drop"
  ) %>%
  gt(
    rowname_col = "size",
    groupname_col = "type"
  ) %>%
  data_color(
    columns = c(sold, income),
```

```
    colors = scales::col_numeric(
      palette = paletteer::paletteer_d(
        palette = "ggsci::red_material"
      ) %>%
        as.character(),
      domain = NULL
    )
  )
```

Function ID

3-23

See Also

Other data formatting functions: fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

default_fonts — *A vector of default fonts for use with* **gt** *tables*

Description

The vector of fonts given by `default_fonts()` should be used with a **gt** table that is rendered to HTML. We can specify additional fonts to use but this default set should be placed after that to act as fallbacks. This is useful when specifying `font` values in the `cell_text()` function (itself used in the `tab_style()` function). If using `opt_table_font()` (which also has a `font` argument) we probably don't need to specify this vector of fonts since it is handled by its `add` option (which is `TRUE` by default).

Usage

```
default_fonts()
```

Value

A character vector of font names.

Examples

Use `exibble` to create a **gt** table. Attempting to modify the fonts used for the `time` column is much safer if `default_fonts()` is appended to the end of the `font` listing in the `cell_text()` call (the `"Comic Sansa"` and `"Menloa"` fonts don't exist, but, we'll get the first available font from the `default_fonts()` set).

```
exibble %>%
  dplyr::select(char, time) %>%
  gt() %>%
  tab_style(
    style = cell_text(
      font = c(
        "Comic Sansa", "Menloa",
        default_fonts()
      )
    ),
    locations = cells_body(columns = time)
  )
```

Function ID

7-28

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

escape_latex *Perform LaTeX escaping*

Description

Text may contain several characters with special meanings in LaTeX. This function will transform a character vector so that it is safe to use within LaTeX tables.

Usage

```
escape_latex(text)
```

Arguments

text	A character vector containing the text that is to be LaTeX-escaped.

Value

A character vector.

Function ID

7-25

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

exibble — *A toy example tibble for testing with gt: exibble*

Description

This tibble contains data of a few different classes, which makes it well-suited for quick experimentation with the functions in this package. It contains only eight rows with numeric, character, and factor columns. The last 4 rows contain `NA` values in the majority of this tibble's columns (1 missing value per column). The `date`, `time`, and `datetime` columns are character-based dates/times in the familiar ISO 8601 format. The `row` and `group` columns provide for unique rownames and two groups (`grp_a` and `grp_b`) for experimenting with the `gt()` function's `rowname_col` and `groupname_col` arguments.

Usage

```
exibble
```

Format

A tibble with 8 rows and 9 variables:

num a numeric column ordered with increasingly larger values

char a character column composed of names of fruits from `a` to `h`

fctr a factor column with numbers from 1 to 8, written out

date, time, datetime character columns with dates, times, and datetimes

currency a numeric column that is useful for testing currency-based formatting

row a character column in the format `row_X` which can be useful for testing with row captions in a table stub

group a character column with four `grp_a` values and four `grp_b` values which can be useful for testing tables that contain row groups

Examples

Here is the `exibble`.

```
exibble
#> # A tibble: 8 x 9
#>          num char       fctr  date       time  datetime    currency row   group
#>        <dbl> <chr>      <fct> <chr>      <chr> <chr>          <dbl> <chr> <chr>
#> 1      0.111 apricot    one   2015-01-15 13:35 2018-01-01~    50.0  row_1 grp_a
#> 2      2.22  banana     two   2015-02-15 14:40 2018-02-02~    18.0  row_2 grp_a
#> 3     33.3   coconut    three 2015-03-15 15:45 2018-03-03~     1.39 row_3 grp_a
#> 4    444.    durian     four  2015-04-15 16:50 2018-04-04~ 65100    row_4 grp_a
#> 5   5550     <NA>       five  2015-05-15 17:55 2018-05-05~  1326.   row_5 grp_b
#> 6     NA     fig        six   2015-06-15 <NA>  2018-06-06~    13.3  row_6 grp_b
#> 7 777000     grapefruit seven <NA>       19:10 2018-07-07~    NA    row_7 grp_b
#> 8 8880000    honeydew   eight 2015-08-15 20:20 <NA>            0.44 row_8 grp_b
```

Function ID

11-6

See Also

Other datasets: `countrypops`, `gtcars`, `pizzaplace`, `sp500`, `sza`

`extract_cells` *Extract a vector of formatted cells from a* **gt** *object*

Description

Get a vector of cell data from a `gt_tbl` object. The output vector will have cell data formatted in the same way as the table.

Usage

```
extract_cells(
  data,
  columns,
  rows = everything(),
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`data` A table object that is created using the `gt()` function.

`columns` The columns containing the cells to extract. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.

`rows`	Optional rows to limit the extraction of cells. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2]`
`output`	The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A vector of cell data extracted from a **gt** table.

Examples

Let's create a **gt** table with the `exibble` dataset to use in the next few examples:

```
gt_tbl <- gt(exibble, rowname_col = "row", groupname_col = "group")
```

We can extract a cell from the table with the `extract_cells()` function. This is done by providing a column and a row intersection:

```
extract_cells(gt_tbl, columns = num, row = 1)
```

```
#> [1] "1.111e-01"
```

Multiple cells can be extracted. Let's get the first four cells from the `char` column.

```
extract_cells(gt_tbl, columns = char, rows = 1:4)
```

```
#> [1] "apricot" "banana" "coconut" "durian"
```

We can format cells and expect that the formatting is fully retained after extraction.

```
gt_tbl %>%
  fmt_number(columns = num, decimals = 2) %>%
  extract_cells(columns = num, rows = 1)
```

```
#> [1] "0.11"
```

Function ID

13-7

See Also

Other table export functions: `as_latex()`, `as_raw_html()`, `as_rtf()`, `as_word()`, `extract_summary()`, `gtsave()`

`extract_summary` *Extract a summary list from a* **gt** *object*

Description

Get a list of summary row data frames from a `gt_tbl` object where summary rows were added via the `summary_rows()` function. The output data frames contain the `group_id` and `rowname` columns, whereby `rowname` contains descriptive stub labels for the summary rows.

Usage

```
extract_summary(data)
```

Arguments

`data`	A table object that is created using the `gt()` function.

Value

A list of data frames containing summary data.

Examples

Use `sp500` to create a **gt** table with row groups. Create summary rows labeled as `min`, `max`, and `avg` for every row group with `summary_rows()`. Then, extract the summary rows as a list object.

```
summary_extracted <-
  sp500 %>%
  dplyr::filter(date >= "2015-01-05" & date <="2015-01-30") %>%
  dplyr::arrange(date) %>%
  dplyr::mutate(week = paste0("W", strftime(date, format = "%V"))) %>%
  dplyr::select(-adj_close, -volume) %>%
  gt(
    rowname_col = "date",
    groupname_col = "week"
  ) %>%
  summary_rows(
    groups = TRUE,
    columns = c(open, high, low, close),
    fns = list(
      min = ~min(.),
      max = ~max(.),
      avg = ~mean(.)
    ),
    formatter = fmt_number,
    use_seps = FALSE
  ) %>%
```

```
  extract_summary()

summary_extracted
#> $summary_df_data_list
#> $summary_df_data_list$W02
#> # A tibble: 3 x 8
#>   group_id rowname  date  open  high   low close  week
#>   <chr>    <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 W02      min        NA 2006. 2030. 1992. 2003.    NA
#> 2 W02      max        NA 2063. 2064. 2038. 2062.    NA
#> 3 W02      avg        NA 2035. 2049. 2017. 2031.    NA
#>
#> $summary_df_data_list$W03
#> # A tibble: 3 x 8
#>   group_id rowname  date  open  high   low close  week
#>   <chr>    <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 W03      min        NA 1992. 2018. 1988. 1993.    NA
#> 2 W03      max        NA 2046. 2057. 2023. 2028.    NA
#> 3 W03      avg        NA 2020. 2033. 2000. 2015.    NA
#>
#> $summary_df_data_list$W04
#> # A tibble: 3 x 8
#>   group_id rowname  date  open  high   low close  week
#>   <chr>    <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 W04      min        NA 2020. 2029. 2004. 2023.    NA
#> 2 W04      max        NA 2063. 2065. 2051. 2063.    NA
#> 3 W04      avg        NA 2035. 2049. 2023. 2042.    NA
#>
#> $summary_df_data_list$W05
#> # A tibble: 3 x 8
#>   group_id rowname  date  open  high   low close  week
#>   <chr>    <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 W05      min        NA 2002. 2023. 1989. 1995.    NA
#> 2 W05      max        NA 2050. 2058. 2041. 2057.    NA
#> 3 W05      avg        NA 2030. 2039. 2009. 2021.    NA
```

Use the summary list to make a new **gt** table. The key thing is to use `dplyr::bind_rows()` and then pass the tibble to `gt()`.

```
summary_extracted %>%
  unlist(recursive = FALSE) %>%
  dplyr::bind_rows() %>%
  gt(groupname_col = "group_id")
```

Function ID

13-6

See Also

Other table export functions: as_latex(), as_raw_html(), as_rtf(), as_word(), extract_cells(), gtsave()

---

fmt *Set a column format with a formatter function*

Description

The fmt() function provides a way to execute custom formatting functionality with raw data values in a way that can consider all output contexts.

Along with the columns and rows arguments that provide some precision in targeting data cells, the fns argument allows you to define one or more functions for manipulating the raw data.

If providing a single function to fns, the recommended format is in the form: fns = function(x) .... This single function will format the targeted data cells the same way regardless of the output format (e.g., HTML, LaTeX, RTF).

If you require formatting of x that depends on the output format, a list of functions can be provided for the html, latex, rtf, and default contexts. This can be in the form of fns = list(html = function(x) ..., latex = function(x) ..., default = function(x) ...). In this multiple-function case, we recommended including the default function as a fallback if all contexts aren't provided.

Usage

```
fmt(data, columns = everything(), rows = everything(), fns)
```

Arguments

data	A table object that is created using the gt() function.
columns	The columns to format. Can either be a series of column names provided in c(), a vector of column indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything().
rows	Optional rows to format. Providing everything() (the default) results in all rows in columns being formatted. Alternatively, we can supply a vector of row captions within c(), a vector of row indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 5
fns	Either a single formatting function or a named list of functions.

Value

An object of class gt_tbl.

Targeting the values to be formatted

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the *Arguments* section for more information on this.

Examples

Use `exibble` to create a **gt** table. Format the numeric values in the `num` column with a function supplied to the `fns` argument.

```
exibble %>%
  dplyr::select(-row, -group) %>%
  gt() %>%
  fmt(
    columns = num,
    fns = function(x) {
      paste0("'", x * 1000, "'")
    }
  )
```

Function ID

3-17

See Also

Other data formatting functions: `data_color()`, `fmt_bytes()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_duration()`, `fmt_engineering()`, `fmt_fraction()`, `fmt_integer()`, `fmt_markdown()`, `fmt_number()`, `fmt_partsper()`, `fmt_passthrough()`, `fmt_percent()`, `fmt_roman()`, `fmt_scientific()`, `fmt_time()`, `sub_large_vals()`, `sub_missing()`, `sub_small_vals()`, `sub_values()`, `sub_zero()`, `text_transform()`

`fmt_bytes` *Format values as bytes*

Description

With numeric values in a **gt** table, we can transform those to values of bytes with human readable units. The `fmt_bytes()` function allows for the formatting of byte sizes to either of two common representations: (1) with decimal units (powers of 1000, examples being `"kB"` and `"MB"`), and (2) with binary units (powers of 1024, examples being `"KiB"` and `"MiB"`).

It is assumed the input numeric values represent the number of bytes and automatic truncation of values will occur. The numeric values will be scaled to be in the range of 1 to <1000 and then decorated with the correct unit symbol according to the standard chosen. For more control over the formatting of byte sizes, we can use the following options:

- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol
- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
fmt_bytes(
  data,
  columns,
  rows = everything(),
  standard = c("decimal", "binary"),
  decimals = 1,
  n_sigfig = NULL,
  drop_trailing_zeros = TRUE,
  drop_trailing_dec_mark = TRUE,
  use_seps = TRUE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  incl_space = TRUE,
  locale = NULL
)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`columns`	The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.
`rows`	Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2] < 5`
`standard`	The way to express large byte sizes.
`decimals`	An option to specify the exact number of decimal places to use. The default number of decimal places is `1`.
`n_sigfig`	A option to format numbers to *n* significant figures. By default, this is `NULL` and thus number values will be formatted according to the number of decimal places

set via `decimals`. If opting to format according to the rules of significant figures, `n_sigfig` must be a number greater than or equal to `1`. Any values passed to the `decimals` and `drop_trailing_zeros` arguments will be ignored.

`drop_trailing_zeros`
: A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).

`drop_trailing_dec_mark`
: A logical value that determines whether decimal marks should always appear even if there are no decimal digits to display after formatting (e.g, `23` becomes `23.`). The default for this is `TRUE`, which means that trailing decimal marks are not shown.

`use_seps`
: An option to use digit group separators. The type of digit group separator is set by `sep_mark` and overridden if a locale ID is provided to `locale`. This setting is `TRUE` by default.

`pattern`
: A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

`sep_mark`
: The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

`dec_mark`
: The character to use as a decimal mark (e.g., using `dec_mark = ","` with `0.152` would result in a formatted value of `0,152`).

`force_sign`
: Should the positive sign be shown for positive numbers (effectively showing a sign for all numbers except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign.

`incl_space`
: An option for whether to include a space between the value and the units. The default of `TRUE` uses a space character for separation.

`locale`
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported. Any `locale` value provided here will override any global locale setting performed in `gt()`'s own `locale` argument.

Value

An object of class `gt_tbl`.

Targeting the values to be formatted

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the *Arguments* section for more information on this.

Examples

Use `exibble` to create a **gt** table. Format the `num` column to have byte sizes in the decimal standard.

```
exibble %>%
  dplyr::select(num) %>%
  gt() %>%
  fmt_bytes(columns = num)
```

Create a similar table with the `fmt_bytes()` function, this time showing byte sizes as binary values.

```
exibble %>%
  dplyr::select(num) %>%
  gt() %>%
  fmt_bytes(
    columns = num,
    standard = "binary"
  )
```

Function ID

3-10

See Also

Other data formatting functions: `data_color()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_duration()`, `fmt_engineering()`, `fmt_fraction()`, `fmt_integer()`, `fmt_markdown()`, `fmt_number()`, `fmt_partsper()`, `fmt_passthrough()`, `fmt_percent()`, `fmt_roman()`, `fmt_scientific()`, `fmt_time()`, `fmt()`, `sub_large_vals()`, `sub_missing()`, `sub_small_vals()`, `sub_values()`, `sub_zero()`, `text_transform()`

`fmt_currency` *Format values as currencies*

Description

With numeric values in a **gt** table, we can perform currency-based formatting. This function supports both automatic formatting with a three-letter or numeric currency code. We can also specify a custom currency that is formatted according to the output context with the `currency()` helper function. Numeric formatting facilitated through the use of a locale ID. We have fine control over the conversion from numeric values to currency values, where we could take advantage of the following options:

- the currency: providing a currency code or common currency name will procure the correct currency symbol and number of currency subunits; we could also use the `currency()` helper function to specify a custom currency
- currency symbol placement: the currency symbol can be placed before or after the values
- decimals/subunits: choice of the number of decimal places, and a choice of the decimal symbol, and an option on whether to include or exclude the currency subunits (decimal portion)
- negative values: choice of a negative sign or parentheses for values less than zero

- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- scaling: we can choose to scale targeted values by a multiplier value
- large-number suffixing: larger figures (thousands, millions, etc.) can be autoscaled and decorated with the appropriate suffixes
- pattern: option to use a text pattern for decoration of the formatted currency values
- locale-based formatting: providing a locale ID will result in currency formatting specific to the chosen locale

We can use the `info_currencies()` function for a useful reference on all of the possible inputs to the `currency` argument.

Usage

```
fmt_currency(
  data,
  columns,
  rows = everything(),
  currency = "USD",
  use_subunits = TRUE,
  decimals = NULL,
  drop_trailing_dec_mark = TRUE,
  use_seps = TRUE,
  accounting = FALSE,
  scale_by = 1,
  suffixing = FALSE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  placement = "left",
  incl_space = FALSE,
  system = c("intl", "ind"),
  locale = NULL
)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`columns`	The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.
`rows`	Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`,

contains(), matches(), one_of(), num_range(), and everything(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 5

currency
: The currency to use for the numeric value. This input can be supplied as a 3-letter currency code (e.g., "USD" for U.S. Dollars, "EUR" for the Euro currency). Use info_currencies() to get an information table with all of the valid currency codes and examples of each. Alternatively, we can provide a common currency name (e.g., "dollar", "pound", "yen", etc.) to simplify the process. Use info_currencies() with the type == "symbol" option to view an information table with all of the supported currency symbol names along with examples.

  We can also use the currency() helper function to specify a custom currency, where the string could vary across output contexts. For example, using currency(html = "ƒ", default = "f") would give us a suitable glyph for the Dutch guilder in an HTML output table, and it would simply be the letter "f" in all other output contexts). Please note that decimals will default to 2 when using the currency() helper function.

  If nothing is provided to currency then "USD" (U.S. dollars) will be used.

use_subunits
: An option for whether the subunits portion of a currency value should be displayed. By default, this is TRUE.

decimals
: An option to specify the exact number of decimal places to use. The default number of decimal places is 2.

drop_trailing_dec_mark
: A logical value that determines whether decimal marks should always appear even if there are no decimal digits to display after formatting (e.g, 23 becomes 23.). The default for this is TRUE, which means that trailing decimal marks are not shown.

use_seps
: An option to use digit group separators. The type of digit group separator is set by sep_mark and overridden if a locale ID is provided to locale. This setting is TRUE by default.

accounting
: An option to use accounting style for values. With FALSE (the default), negative values will be shown with a minus sign. Using accounting = TRUE will put negative values in parentheses.

scale_by
: A value to scale the input. The default is 1.0. All numeric values will be multiplied by this value first before undergoing formatting. This value will be ignored if using any of the suffixing options (i.e., where suffixing is not set to FALSE).

suffixing
: An option to scale and apply suffixes to larger numbers (e.g., 1924000 can be transformed to 1.92M). This option can accept a logical value, where FALSE (the default) will not perform this transformation and TRUE will apply thousands (K), millions (M), billions (B), and trillions (T) suffixes after automatic value scaling. We can also specify which symbols to use for each of the value ranges by using a character vector of the preferred symbols to replace the defaults (e.g., c("k", "Ml", "Bn", "Tr")).

  Including NA values in the vector will ensure that the particular range will either not be included in the transformation (e.g, c(NA, "M", "B", "T") won't modify numbers in the thousands range) or the range will inherit a previous suffix (e.g., with `c("K", "M", NA, "T")`, all numbers in the range of millions and billions will be in terms of millions).

Any use of `suffixing` (where it is not set expressly as `FALSE`) means that any value provided to `scale_by` will be ignored.

If using `system = "ind"` then the default suffix set provided by `suffixing = TRUE` will be `c(NA, "L", "Cr")`. This doesn't apply suffixes to the thousands range, but does express values in lakhs and crores.

`pattern` A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

`sep_mark` The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

`dec_mark` The character to use as a decimal mark (e.g., using `dec_mark = ","` with `0.152` would result in a formatted value of `0,152`).

`force_sign` Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign. This option is disregarded when using accounting notation with `accounting = TRUE`.

`placement` The placement of the currency symbol. This can be either be `left` (the default) or `right`.

`incl_space` An option for whether to include a space between the value and the currency symbol. The default is to not introduce a space character.

`system` The numbering system to use. By default, this is the international numbering system (`"intl"`) whereby grouping separators (i.e., `sep_mark`) are separated by three digits. The alternative system, the Indian numbering system (`"ind"`) uses grouping separators that correspond to thousand, lakh, crore, and higher quantities.

`locale` An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported. Any `locale` value provided here will override any global locale setting performed in `gt()`'s own `locale` argument.

Value

An object of class `gt_tbl`.

Targeting the values to be formatted

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the *Arguments* section for more information on this.

Examples

Use exibble to create a **gt** table. Format the currency column to have currency values in euros ("EUR").

```
exibble %>%
  gt() %>%
  fmt_currency(
    columns = currency,
    currency = "EUR"
  )
```

Use exibble to create a **gt** table. Keep only the num and currency, columns, then, format those columns using the "CNY" and "GBP" currencies.

```
exibble %>%
  dplyr::select(num, currency) %>%
  gt() %>%
  fmt_currency(
    columns = num,
    currency = "CNY"
  ) %>%
  fmt_currency(
    columns = currency,
    currency = "GBP"
  )
```

Function ID

3-8

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

fmt_date *Format values as dates*

Description

Format input values to time values using one of 41 preset date styles. Input can be in the form of POSIXt (i.e., datetimes), the Date type, or character (must be in the ISO 8601 form of YYYY-MM-DD HH:MM:SS or YYYY-MM-DD).

Usage

```
fmt_date(
  data,
  columns,
  rows = everything(),
  date_style = "iso",
  pattern = "{x}",
  locale = NULL
)
```

Arguments

data
: A table object that is created using the `gt()` function.

columns
: The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.

rows
: Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2] < 5`

date_style
: The date style to use. By default this is `"iso"` which corresponds to ISO 8601 date formatting. The other date styles can be viewed using `info_date_style()`.

pattern
: A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

locale
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported. Any `locale` value provided here will override any global locale setting performed in `gt()`'s own `locale` argument.

Value

An object of class `gt_tbl`.

Targeting the values to be formatted

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the *Arguments* section for more information on this.

Formatting with the `date_style` argument

We need to supply a preset date style to the `date_style` argument. The date styles are numerous and can handle localization to any supported locale. A large segment of date styles are termed flexible date formats and this means that their output will adapt to any `locale` provided. That feature makes the flexible date formats a better option for locales other than `"en"` (the default locale).

The following table provides a listing of all date styles and their output values (corresponding to an input date of `2000-02-29`).

	Date Style	Output	Notes
1	"iso"	"2000-02-29"	ISO 8601
2	"wday_month_day_year"	"Tuesday, February 29, 2000"	
3	"wd_m_day_year"	"Tue, Feb 29, 2000"	
4	"wday_day_month_year"	"Tuesday 29 February 2000"	
5	"month_day_year"	"February 29, 2000"	
6	"m_day_year"	"Feb 29, 2000"	
7	"day_m_year"	"29 Feb 2000"	
8	"day_month_year"	"29 February 2000"	
9	"day_month"	"29 February"	
10	"day_m"	"29 Feb"	
11	"year"	"2000"	
12	"month"	"February"	
13	"day"	"29"	
14	"year.mn.day"	"2000/02/29"	
15	"y.mn.day"	"00/02/29"	
16	"year_week"	"2000-W09"	
17	"year_quarter"	"2000-Q1"	
18	"yMd"	"2/29/2000"	flexible
19	"yMEd"	"Tue, 2/29/2000"	flexible
20	"yMMM"	"Feb 2000"	flexible
21	"yMMMM"	"February 2000"	flexible
22	"yMMMd"	"Feb 29, 2000"	flexible
23	"yMMMEd"	"Tue, Feb 29, 2000"	flexible
24	"GyMd"	"2/29/2000 A"	flexible
25	"GyMMMd"	"Feb 29, 2000 AD"	flexible
26	"GyMMMEd"	"Tue, Feb 29, 2000 AD"	flexible
27	"yM"	"2/2000"	flexible
28	"Md"	"2/29"	flexible
29	"MEd"	"Tue, 2/29"	flexible
30	"MMMd"	"Feb 29"	flexible
31	"MMMEd"	"Tue, Feb 29"	flexible
32	"MMMMd"	"February 29"	flexible
33	"GyMMM"	"Feb 2000 AD"	flexible
34	"yQQQ"	"Q1 2000"	flexible
35	"yQQQQ"	"1st quarter 2000"	flexible
36	"Gy"	"2000 AD"	flexible
37	"y"	"2000"	flexible
38	"M"	"2"	flexible
39	"MMM"	"Feb"	flexible

40	"d"	"29"	flexible
41	"Ed"	"29 Tue"	flexible

We can use the info_date_style() within the console to view a similar table of date styles with example output.

Examples

Use exibble to create a **gt** table. Keep only the date and time columns. Format the date column to have dates formatted with the "month_day_year" date style.

```
exibble %>%
  dplyr::select(date, time) %>%
  gt() %>%
  fmt_date(
    columns = date,
    date_style = "month_day_year"
  )
```

Use exibble to create a **gt** table. Keep only the date and time columns. Format the date column to have mixed date formats (dates after April will be different than the others because of the expressions used in the rows argument).

```
exibble %>%
  dplyr::select(date, time) %>%
  gt() %>%
  fmt_date(
    columns = date,
    rows = as.Date(date) > as.Date("2015-04-01"),
    date_style = "m_day_year"
  ) %>%
  fmt_date(
    columns = date,
    rows = as.Date(date) <= as.Date("2015-04-01"),
    date_style = "day_m_year"
  )
```

Use exibble to create another **gt** table, this time only with the date column. Format the date column to use the "yMMMEd" date style (which is one of the 'flexible' styles). Also, set the locale to "nl" to get the dates in Dutch.

```
exibble %>%
  dplyr::select(date) %>%
  gt() %>%
  fmt_date(
    columns = date,
    date_style = "yMMMEd",
    locale = "nl"
  )
```

Function ID

3-11

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

fmt_datetime *Format values as datetimes*

Description

Format input values to datetime values using either presets for the date and time components or a formatting directive (this can either use a *CLDR* datetime pattern or strptime formatting). Input can be in the form of POSIXt (i.e., datetimes), the Date type, or character (must be in the ISO 8601 form of YYYY-MM-DD HH:MM:SS or YYYY-MM-DD).

Usage

```
fmt_datetime(
  data,
  columns,
  rows = everything(),
  date_style = "iso",
  time_style = "iso",
  sep = " ",
  format = NULL,
  tz = NULL,
  pattern = "{x}",
  locale = NULL
)
```

Arguments

data	A table object that is created using the gt() function.
columns	The columns to format. Can either be a series of column names provided in c(), a vector of column indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything().
rows	Optional rows to format. Providing everything() (the default) results in all rows in columns being formatted. Alternatively, we can supply a vector of row captions within c(), a vector of row indices, or a helper function focused on

selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 5

date_style
: The date style to use. By default this is "iso" which corresponds to ISO 8601 date formatting. The other date styles can be viewed using info_date_style().

time_style
: The time style to use. By default this is "iso" which corresponds to how times are formatted within ISO 8601 datetime values. The other time styles can be viewed using info_time_style().

sep
: The separator string to use between the date and time components. By default, this is a single space character (" "). Only used when not specifying a format code.

format
: An optional formatting string used for generating custom dates/times. If used then the arguments governing preset styles (date_style and time_style) will be ignored in favor of formatting via the format string.

tz
: The time zone for printing dates/times (i.e., the output). The default of NULL will preserve the time zone of the input data in the output. If providing a time zone, it must be one that is recognized by the user's operating system (a vector of all valid tz values can be produced with OlsonNames()).

pattern
: A formatting pattern that allows for decoration of the formatted value. The value itself is represented by {x} and all other characters are taken to be string literals.

locale
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include "en" for English (United States) and "fr" for French (France). The use of a valid locale ID will override any values provided in sep_mark and dec_mark. We can use the info_locales() function as a useful reference for all of the locales that are supported. Any locale value provided here will override any global locale setting performed in gt()'s own locale argument.

Value

An object of class gt_tbl.

Targeting the values to be formatted

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the rows argument. See the *Arguments* section for more information on this.

Formatting with the date_style argument

We can supply a preset date style to the date_style argument to separately handle the date portion of the output. The date styles are numerous and can handle localization to any supported locale. A large segment of date styles are termed flexible date formats and this means that their output will adapt to any locale provided. That feature makes the flexible date formats a better option for locales other than "en" (the default locale).

The following table provides a listing of all date styles and their output values (corresponding to an input date of 2000-02-29).

	Date Style	Output	Notes
1	"iso"	"2000-02-29"	ISO 8601
2	"wday_month_day_year"	"Tuesday, February 29, 2000"	
3	"wd_m_day_year"	"Tue, Feb 29, 2000"	
4	"wday_day_month_year"	"Tuesday 29 February 2000"	
5	"month_day_year"	"February 29, 2000"	
6	"m_day_year"	"Feb 29, 2000"	
7	"day_m_year"	"29 Feb 2000"	
8	"day_month_year"	"29 February 2000"	
9	"day_month"	"29 February"	
10	"day_m"	"29 Feb"	
11	"year"	"2000"	
12	"month"	"February"	
13	"day"	"29"	
14	"year.mn.day"	"2000/02/29"	
15	"y.mn.day"	"00/02/29"	
16	"year_week"	"2000-W09"	
17	"year_quarter"	"2000-Q1"	
18	"yMd"	"2/29/2000"	flexible
19	"yMEd"	"Tue, 2/29/2000"	flexible
20	"yMMM"	"Feb 2000"	flexible
21	"yMMMM"	"February 2000"	flexible
22	"yMMMd"	"Feb 29, 2000"	flexible
23	"yMMMEd"	"Tue, Feb 29, 2000"	flexible
24	"GyMd"	"2/29/2000 A"	flexible
25	"GyMMMd"	"Feb 29, 2000 AD"	flexible
26	"GyMMMEd"	"Tue, Feb 29, 2000 AD"	flexible
27	"yM"	"2/2000"	flexible
28	"Md"	"2/29"	flexible
29	"MEd"	"Tue, 2/29"	flexible
30	"MMMd"	"Feb 29"	flexible
31	"MMMEd"	"Tue, Feb 29"	flexible
32	"MMMMd"	"February 29"	flexible
33	"GyMMM"	"Feb 2000 AD"	flexible
34	"yQQQ"	"Q1 2000"	flexible
35	"yQQQQ"	"1st quarter 2000"	flexible
36	"Gy"	"2000 AD"	flexible
37	"y"	"2000"	flexible
38	"M"	"2"	flexible
39	"MMM"	"Feb"	flexible
40	"d"	"29"	flexible
41	"Ed"	"29 Tue"	flexible

We can use the `info_date_style()` within the console to view a similar table of date styles with example output.

Formatting with the `time_style` argument

We can supply a preset time style to the `time_style` argument to separately handle the time portion of the output. There are many time styles and all of them can handle localization to any supported locale. Many of the time styles are termed flexible time formats and this means that their output will adapt to any `locale` provided. That feature makes the flexible time formats a better option for locales other than `"en"` (the default locale).

The following table provides a listing of all time styles and their output values (corresponding to an input time of `14:35:00`). It is noted which of these represent 12- or 24-hour time. Some of the flexible formats (those that begin with `"E"`) include the the day of the week. Keep this in mind when pairing such `time_style` values with a `date_style` so as to avoid redundant or repeating information.

	Time Style	Output	Notes
1	`"iso"`	`"14:35:00"`	ISO 8601, 24h
2	`"iso-short"`	`"14:35"`	ISO 8601, 24h
3	`"h_m_s_p"`	`"2:35:00 PM"`	12h
4	`"h_m_p"`	`"2:35 PM"`	12h
5	`"h_p"`	`"2 PM"`	12h
6	`"Hms"`	`"14:35:00"`	flexible, 24h
7	`"Hm"`	`"14:35"`	flexible, 24h
8	`"H"`	`"14"`	flexible, 24h
9	`"EHm"`	`"Thu 14:35"`	flexible, 24h
10	`"EHms"`	`"Thu 14:35:00"`	flexible, 24h
11	`"Hmsv"`	`"14:35:00 GMT+00:00"`	flexible, 24h
12	`"Hmv"`	`"14:35 GMT+00:00"`	flexible, 24h
13	`"hms"`	`"2:35:00 PM"`	flexible, 12h
14	`"hm"`	`"2:35 PM"`	flexible, 12h
15	`"h"`	`"2 PM"`	flexible, 12h
16	`"Ehm"`	`"Thu 2:35 PM"`	flexible, 12h
17	`"Ehms"`	`"Thu 2:35:00 PM"`	flexible, 12h
18	`"EBhms"`	`"Thu 2:35:00 in the afternoon"`	flexible, 12h
19	`"Bhms"`	`"2:35:00 in the afternoon"`	flexible, 12h
20	`"EBhm"`	`"Thu 2:35 in the afternoon"`	flexible, 12h
21	`"Bhm"`	`"2:35 in the afternoon"`	flexible, 12h
22	`"Bh"`	`"2 in the afternoon"`	flexible, 12h
23	`"hmsv"`	`"2:35:00 PM GMT+00:00"`	flexible, 12h
24	`"hmv"`	`"2:35 PM GMT+00:00"`	flexible, 12h
25	`"ms"`	`"35:00"`	flexible

We can use the `info_time_style()` within the console to view a similar table of time styles with example output.

Formatting with a *CLDR* datetime pattern

We can use a *CLDR* datetime pattern with the `format` argument to create a highly customized and locale-aware output. This is a character string that consists of two types of elements:

- Pattern fields, which repeat a specific pattern character one or more times. These fields are replaced with date and time data when formatting. The character sets of `A-Z` and `a-z` are reserved for use as pattern characters.
- Literal text, which is output verbatim when formatting. This can include:
  - Any characters outside the reserved character sets, including spaces and punctuation.
  - Any text between single vertical quotes (e.g., `'text'`).
  - Two adjacent single vertical quotes (''), which represent a literal single quote, either inside or outside quoted text.

The number of pattern fields is quite sizable so let's first look at how some *CLDR* datetime patterns work. We'll use the datetime string `"2018-07-04T22:05:09.2358(America/Vancouver)"` for all of the examples that follow.

- `"mm/dd/y"` -> `"05/04/2018"`
- `"EEEE, MMMM d, y"` -> `"Wednesday, July 4, 2018"`
- `"MMM d E"` -> `"Jul 4 Wed"`
- `"HH:mm"` -> `"22:05"`
- `"h:mm a"` -> `"10:05 PM"`
- `"EEEE, MMMM d, y 'at' h:mm a"` -> `"Wednesday, July 4, 2018 at 10:05 PM"`

Here are the individual pattern fields:

Year:

Calendar Year:
This yields the calendar year, which is always numeric. In most cases the length of the `"y"` field specifies the minimum number of digits to display, zero-padded as necessary. More digits will be displayed if needed to show the full year. There is an exception: `"yy"` gives use just the two low-order digits of the year, zero-padded as necessary. For most use cases, `"y"` or `"yy"` should be good enough.

Field Patterns	Output
`"y"`	`"2018"`
`"yy"`	`"18"`
`"yyy"` to `"yyyyyyyyy"`	`"2018"` to `"000002018"`

Year in the Week in Year Calendar:
This is the year in 'Week of Year' based calendars in which the year transition occurs on a week boundary. This may differ from calendar year `"y"` near a year transition. This numeric year designation is used in conjunction with pattern character `"w"` in the ISO year-week calendar as defined by ISO 8601.

Field Patterns	Output
`"Y"`	`"2018"`
`"YY"`	`"18"`
`"YYY"` to `"YYYYYYYYY"`	`"2018"` to `"000002018"`

Quarter:

Quarter of the Year: formatting and standalone versions:
The quarter names are identified numerically, starting at 1 and ending at 4. Quarter names may vary along two axes: the width and the context. The context is either 'formatting' (taken as a default), which the form used within a complete date format string, or, 'standalone', the form for date elements used independently (such as in calendar headers). The standalone form may be used in any other date format that shares the same form of the name. Here, the formatting form for quarters of the year consists of some run of "Q" values whereas the standalone form uses "q".

Field Patterns	Output	Notes
"Q"/"q"	"3"	Numeric, one digit
"QQ"/"qq"	"03"	Numeric, two digits (zero padded)
"QQQ"/"qqq"	"Q3"	Abbreviated
"QQQQ"/"qqqq"	"3rd quarter"	Wide
"QQQQQ"/"qqqqq"	"3"	Narrow

Month:

Month: formatting and standalone versions:
The month names are identified numerically, starting at 1 and ending at 12. Month names may vary along two axes: the width and the context. The context is either 'formatting' (taken as a default), which the form used within a complete date format string, or, 'standalone', the form for date elements used independently (such as in calendar headers). The standalone form may be used in any other date format that shares the same form of the name. Here, the formatting form for months consists of some run of "M" values whereas the standalone form uses "L".

Field Patterns	Output	Notes
"M"/"L"	"7"	Numeric, minimum digits
"MM"/"LL"	"07"	Numeric, two digits (zero padded)
"MMM"/"LLL"	"Jul"	Abbreviated
"MMMM"/"LLLL"	"July"	Wide
"MMMMM"/"LLLLL"	"J"	Narrow

Week:

Week of Year:
Values calculated for the week of year range from 1 to 53. Week 1 for a year is the first week that contains at least the specified minimum number of days from that year. Weeks between week 1 of one year and week 1 of the following year are numbered sequentially from 2 to 52 or 53 (if needed).
There are two available field lengths. Both will display the week of year value but the "ww" width will always show two digits (where weeks 1 to 9 are zero padded).

Field Patterns	Output	Notes
"w"	"27"	Minimum digits
"ww"	"27"	Two digits (zero padded)

Week of Month:
The week of a month can range from 1 to 5. The first day of every month always begins at week 1 and with every transition into the beginning of a week, the week of month value is incremented by 1.

Field Pattern	Output
"W"	"1"

Day:

Day of Month:
The day of month value is always numeric and there are two available field length choices in its formatting. Both will display the day of month value but the "dd" formatting will always show two digits (where days 1 to 9 are zero padded).

Field Patterns	Output	Notes
"d"	"4"	Minimum digits
"dd"	"04"	Two digits, zero padded

Day of Year:
The day of year value ranges from 1 (January 1) to either 365 or 366 (December 31), where the higher value of the range indicates that the year is a leap year (29 days in February, instead of 28). The field length specifies the minimum number of digits, with zero-padding as necessary.

Field Patterns	Output	Notes
"D"	"185"	
"DD"	"185"	Zero padded to minimum width of 2
"DDD"	"185"	Zero padded to minimum width of 3

Day of Week in Month:
The day of week in month returns a numerical value indicating the number of times a given weekday had occurred in the month (e.g., '2nd Monday in March'). This conveniently resolves to predicable case structure where ranges of day of the month values return predictable day of week in month values:

- days 1 - 7 -> 1
- days 8 - 14 -> 2
- days 15 - 21 -> 3
- days 22 - 28 -> 4
- days 29 - 31 -> 5

Field Pattern	Output
"F"	"1"

Modified Julian Date:
The modified version of the Julian date is obtained by subtracting 2,400,000.5 days from the

Julian date (the number of days since January 1, 4713 BC). This essentially results in the number of days since midnight November 17, 1858. There is a half day offset (unlike the Julian date, the modified Julian date is referenced to midnight instead of noon).

Field Patterns	Output
`"g"` to `"ggggggggg"`	`"58303"` -> `"000058303"`

Weekday:

Day of Week Name:
The name of the day of week is offered in four different widths.

Field Patterns	Output	Notes
`"E"`, `"EE"`, or `"EEE"`	`"Wed"`	Abbreviated
`"EEEE"`	`"Wednesday"`	Wide
`"EEEEE"`	`"W"`	Narrow
`"EEEEEE"`	`"We"`	Short

Periods:

AM/PM Period of Day:
This denotes before noon and after noon time periods. May be upper or lowercase depending on the locale and other options. The wide form may be the same as the short form if the 'real' long form (e.g. 'ante meridiem') is not customarily used. The narrow form must be unique, unlike some other fields.

Field Patterns	Output	Notes
`"a"`, `"aa"`, or `"aaa"`	`"PM"`	Abbreviated
`"aaaa"`	`"PM"`	Wide
`"aaaaa"`	`"p"`	Narrow

AM/PM Period of Day Plus Noon and Midnight:
Provide AM and PM as well as phrases for exactly noon and midnight. May be upper or lowercase depending on the locale and other options. If the locale doesn't have the notion of a unique 'noon' (i.e., 12:00), then the PM form may be substituted. A similar behavior can occur for 'midnight' (00:00) and the AM form. The narrow form must be unique, unlike some other fields.
(a) `input_midnight`: `"2020-05-05T00:00:00"` (b) `input_noon`: `"2020-05-05T12:00:00"`

Field Patterns	Output	Notes
`"b"`, `"bb"`, or `"bbb"`	(a) `"midnight"` (b) `"noon"`	Abbreviated
`"bbbb"`	(a) `"midnight"` (b) `"noon"`	Wide
`"bbbbb"`	(a) `"mi"` (b) `"n"`	Narrow

Flexible Day Periods:
Flexible day periods denotes things like 'in the afternoon', 'in the evening', etc., and the flexibility comes from a locale's language and script. Each locale has an associated rule set that specifies when the day periods start and end for that locale.
(a) `input_morning`: `"2020-05-05T00:08:30"` (b) `input_afternoon`: `"2020-05-05T14:00:00"`

Field Patterns	Output	Notes
`"B"`, `"BB"`, or `"BBB"`	(a) `"in the morning"`	Abbreviated
	(b) `"in the afternoon"`	
`"BBBB"`	(a) `"in the morning"`	Wide
	(b) `"in the afternoon"`	
`"BBBBB"`	(a) `"in the morning"`	Narrow
	(b) `"in the afternoon"`	

Hours, Minutes, and Seconds:

Hour 0-23:
Hours from `0` to `23` are for a standard 24-hour clock cycle (midnight plus 1 minute is `00:01`) when using `"HH"` (which is the more common width that indicates zero-padding to 2 digits).
Using `"2015-08-01T08:35:09"`:

Field Patterns	Output	Notes
`"H"`	`"8"`	Numeric, minimum digits
`"HH"`	`"08"`	Numeric, 2 digits (zero padded)

Hour 1-12:
Hours from `1` to `12` are for a standard 12-hour clock cycle (midnight plus 1 minute is `12:01`) when using `"hh"` (which is the more common width that indicates zero-padding to 2 digits).
Using `"2015-08-01T08:35:09"`:

Field Patterns	Output	Notes
`"h"`	`"8"`	Numeric, minimum digits
`"hh"`	`"08"`	Numeric, 2 digits (zero padded)

Hour 1-24:
Using hours from `1` to `24` is a less common way to express a 24-hour clock cycle (midnight plus 1 minute is `24:01`) when using `"kk"` (which is the more common width that indicates zero-padding to 2 digits).
Using `"2015-08-01T08:35:09"`:

Field Patterns	Output	Notes
`"k"`	`"9"`	Numeric, minimum digits
`"kk"`	`"09"`	Numeric, 2 digits (zero padded)

Hour 0-11:
Using hours from `0` to `11` is a less common way to express a 12-hour clock cycle (midnight

plus 1 minute is `00:01`) when using `"KK"` (which is the more common width that indicates zero-padding to 2 digits).
Using `"2015-08-01T08:35:09"`:

Field Patterns	Output	Notes
`"K"`	`"7"`	Numeric, minimum digits
`"KK"`	`"07"`	Numeric, 2 digits (zero padded)

Minute:
The minute of the hour which can be any number from `0` to `59`. Use `"m"` to show the minimum number of digits, or `"mm"` to always show two digits (zero-padding, if necessary).

Field Patterns	Output	Notes
`"m"`	`"5"`	Numeric, minimum digits
`"mm"`	`"06"`	Numeric, 2 digits (zero padded)

Seconds:
The second of the minute which can be any number from `0` to `59`. Use `"s"` to show the minimum number of digits, or `"ss"` to always show two digits (zero-padding, if necessary).

Field Patterns	Output	Notes
`"s"`	`"9"`	Numeric, minimum digits
`"ss"`	`"09"`	Numeric, 2 digits (zero padded)

Fractional Second:
The fractional second truncates (like other time fields) to the width requested (i.e., count of letters). So using pattern `"SSSS"` will display four digits past the decimal (which, incidentally, needs to be added manually to the pattern).

Field Patterns	Output
`"S"` to `"SSSSSSSSS"`	`"2"` -> `"235000000"`

Milliseconds Elapsed in Day:
There are 86,400,000 milliseconds in a day and the `"A"` pattern will provide the whole number. The width can go up to nine digits with `"AAAAAAAAA"` and these higher field widths will result in zero padding if necessary.
Using `"2011-07-27T00:07:19.7223"`:

Field Patterns	Output
`"A"` to `"AAAAAAAAA"`	`"439722"` -> `"000439722"`

Era:

The Era Designator:
This provides the era name for the given date. The Gregorian calendar has two eras: AD and

BC. In the AD year numbering system, AD 1 is immediately preceded by 1 BC, with nothing in between them (there was no year zero).

Field Patterns	Output	Notes
`"G"`, `"GG"`, or `"GGG"`	`"AD"`	Abbreviated
`"GGGG"`	`"Anno Domini"`	Wide
`"GGGGG"`	`"A"`	Narrow

Time Zones:

TZ // Short and Long Specific non-Location Format:
The short and long specific non-location formats for time zones are suggested for displaying a time with a user friendly time zone name. Where the short specific format is unavailable, it will fall back to the short localized GMT format (`"O"`). Where the long specific format is unavailable, it will fall back to the long localized GMT format (`"OOOO"`).

Field Patterns	Output	Notes
`"z"`, `"zz"`, or `"zzz"`	`"PDT"`	Short Specific
`"zzzz"`	`"Pacific Daylight Time"`	Long Specific

TZ // Common UTC Offset Formats:
The ISO8601 basic format with hours, minutes and optional seconds fields is represented by `"Z"`, `"ZZ"`, or `"ZZZ"`. The format is equivalent to RFC 822 zone format (when the optional seconds field is absent). This is equivalent to the `"xxxx"` specifier. The field pattern `"ZZZZ"` represents the long localized GMT format. This is equivalent to the `"OOOO"` specifier. Finally, `"ZZZZZ"` pattern yields the ISO8601 extended format with hours, minutes and optional seconds fields. The ISO8601 UTC indicator `Z` is used when local time offset is `0`. This is equivalent to the `"XXXXX"` specifier.

Field Patterns	Output	Notes
`"Z"`, `"ZZ"`, or `"ZZZ"`	`"-0700"`	ISO 8601 basic format
`"ZZZZ"`	`"GMT-7:00"`	Long localized GMT format
`"ZZZZZ"`	`"-07:00"`	ISO 8601 extended format

TZ // Short and Long Localized GMT Formats:
The localized GMT formats come in two widths `"O"` (which removes the minutes field if it's `0`) and `"OOOO"` (which always contains the minutes field). The use of the `GMT` indicator changes according to the locale.

Field Patterns	Output	Notes
`"O"`	`"GMT-7"`	Short localized GMT format
`"OOOO"`	`"GMT-07:00"`	Long localized GMT format

TZ // Short and Long Generic non-Location Formats:
The generic non-location formats are useful for displaying a recurring wall time (e.g., events, meetings) or anywhere people do not want to be overly specific. Where either of these is un-

available, there is a fallback to the generic location format ("VVVV"), then the short localized GMT format as the final fallback.

Field Patterns	Output	Notes
"v"	"PT"	Short generic non-location format
"vvvv"	"Pacific Time"	Long generic non-location format

TZ // Short Time Zone IDs and Exemplar City Formats:
These formats provide variations of the time zone ID and often include the exemplar city. The widest of these formats, "VVVV", is useful for populating a choice list for time zones, because it supports 1-to-1 name/zone ID mapping and is more uniform than other text formats.

Field Patterns	Output	Notes
"V"	"cavan"	Short time zone ID
"VV"	"America/Vancouver"	Long time zone ID
"VVV"	"Vancouver"	The tz exemplar city
"VVVV"	"Vancouver Time"	Generic location format

TZ // ISO 8601 Formats with Z for +0000:
The "X"-"XXX" field patterns represent valid ISO 8601 patterns for time zone offsets in datetimes. The final two widths, "XXXX" and "XXXXX" allow for optional seconds fields. The seconds field is *not* supported by the ISO 8601 specification. For all of these, the ISO 8601 UTC indicator Z is used when the local time offset is 0.

Field Patterns	Output	Notes
"X"	"-07"	ISO 8601 basic format (h, optional m)
"XX"	"-0700"	ISO 8601 basic format (h & m)
"XXX"	"-07:00"	ISO 8601 extended format (h & m)
"XXXX"	"-0700"	ISO 8601 basic format (h & m, optional s)
"XXXXX"	"-07:00"	ISO 8601 extended format (h & m, optional s)

TZ // ISO 8601 Formats (no use of Z for +0000):
The "x"-"xxxxx" field patterns represent valid ISO 8601 patterns for time zone offsets in datetimes. They are similar to the "X"-"XXXXX" field patterns except that the ISO 8601 UTC indicator Z *will not* be used when the local time offset is 0.

Field Patterns	Output	Notes
"x"	"-07"	ISO 8601 basic format (h, optional m)
"xx"	"-0700"	ISO 8601 basic format (h & m)
"xxx"	"-07:00"	ISO 8601 extended format (h & m)
"xxxx"	"-0700"	ISO 8601 basic format (h & m, optional s)
"xxxxx"	"-07:00"	ISO 8601 extended format (h & m, optional s)

Formatting with a `strptime` format code

Performing custom date/time formatting with the `format` argument can also occur with a `strptime` format code. This works by constructing a string of individual format codes representing formatted date and time elements. These are all indicated with a leading `%`, literal characters are interpreted as any characters not starting with a `%` character.

First off, let's look at a few format code combinations that work well together as a `strptime` format. This will give us an intuition on how these generally work. We'll use the datetime `"2015-06-08 23:05:37.48"` for all of the examples that follow.

- `"%m/%d/%Y"` -> `"06/08/2015"`
- `"%A, %B %e, %Y"` -> `"Monday, June 8, 2015"`
- `"%b %e %a"` -> `"Jun 8 Mon"`
- `"%H:%M"` -> `"23:05"`
- `"%I:%M %p"` -> `"11:05 pm"`
- `"%A, %B %e, %Y at %I:%M %p"` -> `"Monday, June 8, 2015 at 11:05 pm"`

Here are the individual format codes for the date components:

- `"%a"` -> `"Mon"` (abbreviated day of week name)
- `"%A"` -> `"Monday"` (full day of week name)
- `"%w"` -> `"1"` (day of week number in `0..6`; Sunday is `0`)
- `"%u"` -> `"1"` (day of week number in `1..7`; Monday is `1`, Sunday `7`)
- `"%y"` -> `"15"` (abbreviated year, using the final two digits)
- `"%Y"` -> `"2015"` (full year)
- `"%b"` -> `"Jun"` (abbreviated month name)
- `"%B"` -> `"June"` (full month name)
- `"%m"` -> `"06"` (month number)
- `"%d"` -> `"08"` (day number, zero-padded)
- `"%e"` -> `"8"` (day number without zero padding)
- `"%j"` -> `"159"` (day of the year, always zero-padded)
- `"%W"` -> `"23"` (week number for the year, always zero-padded)
- `"%V"` -> `"24"` (week number for the year, following the ISO 8601 standard)
- `"%C"` -> `"20"` (the century number)

Here are the individual format codes for the time components:

- `"%H"` -> `"23"` (24h hour)
- `"%I"` -> `"11"` (12h hour)
- `"%M"` -> `"05"` (minute)
- `"%S"` -> `"37"` (second)
- `"%OS3"` -> `"37.480"` (seconds with decimals; `3` decimal places here)
- `%p` -> `"pm"` (AM or PM indicator)

Here are some extra formats that you may find useful:

- `"%z"` -> `"+0000"` (signed time zone offset, here using UTC)
- `"%F"` -> `"2015-06-08"` (the date in the ISO 8601 date format)
- `"%%"` -> `"%"` (the literal "`%`" character, in case you need it)

Examples

Use `exibble` to create a **gt** table. Keep only the `datetime` column. Format the column to have dates formatted with the `"month_day_year"` style and times with the `"h_m_s_p"` 12-hour time style.

```
exibble %>%
  dplyr::select(datetime) %>%
  gt() %>%
  fmt_datetime(
    columns = datetime,
    date_style = "month_day_year",
    time_style = "h_m_s_p"
  )
```

Using the same input table, we can use flexible date and time styles. Two that work well together are `"MMMEd"` and `"Hms"`. These will mutate depending on the locale. Let's use the default locale for the first 3 rows and the Danish locale (`"da"`) for the remaining rows.

```
exibble %>%
  dplyr::select(datetime) %>%
  gt() %>%
  fmt_datetime(
    columns = datetime,
    date_style = "MMMEd",
    time_style = "Hms",
    locale = "da"
  ) %>%
  fmt_datetime(
    columns = datetime,
    rows = 1:3,
    date_style = "MMMEd",
    time_style = "Hms"
  )
```

It's possible to use the `format` argument and write our own formatting specification. Using the CLDR datetime pattern `"EEEE, MMMM d, y 'at' h:mm a (zzzz)"` gives us datetime outputs with time zone formatting. Let's provide a time zone ID (`"America/Vancouver"`) to the `tz` argument.

```
exibble %>%
  dplyr::select(datetime) %>%
  gt() %>%
  fmt_datetime(
```

```
    columns = datetime,
    format = "EEEE, MMMM d, y 'at' h:mm a (zzzz)",
    tz = "America/Vancouver"
  )
```

Function ID

3-13

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

fmt_duration *Format numeric or duration values as styled time duration strings*

Description

Format input values to time duration values whether those input values are numbers or of the difftime class. We can specify which time units any numeric input values have (as weeks, days, hours, minutes, or seconds) and the output can be customized with a duration style (corresponding to narrow, wide, colon-separated, and ISO forms) and a choice of output units ranging from weeks to seconds.

Usage

```
fmt_duration(
  data,
  columns,
  rows = everything(),
  input_units = NULL,
  output_units = NULL,
  duration_style = c("narrow", "wide", "colon-sep", "iso"),
  trim_zero_units = TRUE,
  max_output_units = NULL,
  pattern = "{x}",
  use_seps = TRUE,
  sep_mark = ",",
  force_sign = FALSE,
  system = c("intl", "ind"),
  locale = NULL
)
```

Arguments

`data` A table object that is created using the `gt()` function.

`columns` The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.

`rows` Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2] < 5`

`input_units` If one or more selected columns contains numeric values, a keyword must be provided for `input_units` for **gt** to determine how those values are to be interpreted in terms of duration. The accepted units are: `"seconds"`, `"minutes"`, `"hours"`, `"days"`, and `"weeks"`.

`output_units` Controls the output time units. The default, `NULL`, means that **gt** will automatically choose time units based on the input duration value. To control which time units are to be considered for output (before trimming with `trim_zero_units`) we can specify a vector of one or more of the following keywords: `"weeks"`, `"days"`, `"hours"`, `"minutes"`, or `"seconds"`.

`duration_style` A choice of four formatting styles for the output duration values. With `"narrow"` (the default style), duration values will be formatted with single letter time-part units (e.g., 1.35 days will be styled as `"1d 8h 24m`). With `"wide"`, this example value will be expanded to `"1 day 8 hours 24 minutes"` after formatting. The `"colon-sep"` style will put days, hours, minutes, and seconds in the `"([D]/)[HH]:[MM]:[SS]"` format. The `"iso"` style will produce a value that conforms to the ISO 8601 rules for duration values (e.g., 1.35 days will become `"P1DT8H24M"`).

`trim_zero_units`

Provides methods to remove output time units that have zero values. By default this is `TRUE` and duration values that might otherwise be formatted as `"0w 1d 0h 4m 19s"` with `trim_zero_units = FALSE` are instead displayed as `"1d 4m 19s"`. Aside from using `TRUE`/`FALSE` we could provide a vector of keywords for more precise control. These keywords are: (1) `"leading"`, to omit all leading zero-value time units (e.g., `"0w 1d"` -> `"1d"`), (2) `"trailing"`, to omit all trailing zero-value time units (e.g., `"3d 5h 0s"` -> `"3d 5h"`), and `"internal"`, which removes all internal zero-value time units (e.g., `"5d 0h 33m"` -> `"5d 33m"`).

`max_output_units`

If `output_units` is `NULL`, where the output time units are unspecified and left to **gt** to handle, a numeric value provided for `max_output_units` will be taken as the maximum number of time units to display in all output time duration values. By default, this is `NULL` and all possible time units will be displayed. This option has no effect when `duration_style = "colon-sep"` (only `output_units` can be used to customize that type of duration output).

`pattern` A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

use_seps	An option to use digit group separators. The type of digit group separator is set by sep_mark and overridden if a locale ID is provided to locale. This setting is TRUE by default.
sep_mark	The mark to use as a separator between groups of digits (e.g., using sep_mark = "," with 1000 would result in a formatted value of 1,000).
force_sign	Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use TRUE for this option. The default is FALSE, where only negative value will display a minus sign.
system	The numbering system to use. By default, this is the international numbering system ("intl") whereby grouping separators (i.e., sep_mark) are separated by three digits. The alternative system, the Indian numbering system ("ind") uses grouping separators that correspond to thousand, lakh, crore, and higher quantities.
locale	An optional locale ID that can be used for formatting the value according the locale's rules. Examples include "en" for English (United States) and "fr" for French (France). The use of a valid locale ID will override any values provided in sep_mark and dec_mark. We can use the info_locales() function as a useful reference for all of the locales that are supported. Any locale value provided here will override any global locale setting performed in gt()'s own locale argument.

Value

An object of class gt_tbl.

Targeting the values to be formatted

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the rows argument. See the *Arguments* section for more information on this.

Output units for the colon-separated duration style

The colon-separated duration style (enabled when duration_style = "colon-sep") is essentially a clock-based output format which uses the display logic of chronograph watch functionality. It will, by default, display duration values in the (D/)HH:MM:SS format. Any duration values greater than or equal to 24 hours will have the number of days prepended with an adjoining slash mark. While this output format is versatile, it can be changed somewhat with the output_units option. The following combinations of output units are permitted:

- c("minutes", "seconds") -> MM:SS
- c("hours", "minutes") -> HH:MM
- c("hours", "minutes", "seconds") -> HH:MM:SS
- c("days", "hours", "minutes") -> (D/)HH:MM

Any other specialized combinations will result in the default set being used, which is c("days", "hours", "minutes", "seconds")

Examples

Use part of the sp500 table to create a **gt** table. Create a difftime-based column and format the duration values to be displayed as the number of days since March 30, 2020.

```
sp500 %>%
  dplyr::slice_head(n = 10) %>%
  dplyr::mutate(
    time_point = lubridate::ymd("2020-03-30"),
    time_passed = difftime(time_point, date)
  ) %>%
  dplyr::select(time_passed, open, close) %>%
  gt(rowname_col = "month") %>%
  fmt_duration(
    columns = time_passed,
    output_units = "days",
    duration_style = "wide"
  ) %>%
  fmt_currency(columns = c(open, close))
```

Function ID

3-14

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

fmt_engineering *Format values to engineering notation*

Description

With numeric values in a **gt** table, we can perform formatting so that the targeted values are rendered in engineering notation.

With this function, there is fine control over the formatted values with the following options:

- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol
- digit grouping separators: choice of separator symbol
- scaling: we can choose to scale targeted values by a multiplier value
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in formatting specific to the chosen locale

Usage

```
fmt_engineering(
  data,
  columns,
  rows = everything(),
  decimals = 2,
  drop_trailing_zeros = FALSE,
  scale_by = 1,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  locale = NULL
)
```

Arguments

`data` A table object that is created using the `gt()` function.

`columns` The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.

`rows` Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2] < 5`

`decimals` An option to specify the exact number of decimal places to use. The default number of decimal places is `2`.

`drop_trailing_zeros`
A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).

`scale_by` A value to scale the input. The default is `1.0`. All numeric values will be multiplied by this value first before undergoing formatting.

`pattern` A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

`sep_mark` The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

`dec_mark` The character to use as a decimal mark (e.g., using `dec_mark = ","` with `0.152` would result in a formatted value of `0,152`).

`force_sign` Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign.

`locale` An optional locale ID that can be used for formatting the value according the locale's rules. Examples include "en" for English (United States) and "fr" for French (France). The use of a valid locale ID will override any values provided in sep_mark and dec_mark. We can use the info_locales() function as a useful reference for all of the locales that are supported. Any locale value provided here will override any global locale setting performed in gt()'s own locale argument.

Value

An object of class gt_tbl.

Targeting the values to be formatted

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the rows argument. See the *Arguments* section for more information on this.

Examples

Use exibble to create a **gt** table. Format the num column in engineering notation.

```
exibble %>%
  gt() %>%
  fmt_engineering(columns = num)
```

Function ID

3-4

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

fmt_fraction *Format values as a mixed fractions*

Description

With numeric values in a **gt** table, we can perform mixed-fraction-based formatting. There are several options for setting the accuracy of the fractions. Furthermore, there is an option for choosing a layout (i.e., typesetting style) for the mixed-fraction output.

The following options are available for controlling this type of formatting:

- accuracy: how to express the fractional part of the mixed fractions; there are three keyword options for this and an allowance for arbitrary denominator settings
- simplification: an option to simplify fractions whenever possible
- layout: We can choose to output values with diagonal or inline fractions
- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol for the whole number portion
- pattern: option to use a text pattern for decoration of the formatted mixed fractions
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
fmt_fraction(
  data,
  columns,
  rows = everything(),
  accuracy = NULL,
  simplify = TRUE,
  layout = c("inline", "diagonal"),
  use_seps = TRUE,
  pattern = "{x}",
  sep_mark = ",",
  system = c("intl", "ind"),
  locale = NULL
)
```

Arguments

`data` A table object that is created using the `gt()` function.

`columns` The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.

`rows` Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2] < 5`

`accuracy` The type of fractions to generate. This can either be one of the keywords `"low"`, `"med"`, or `"high"` (to generate fractions with denominators of up to 1, 2, or 3 digits, respectively) or an integer value greater than zero to obtain fractions with a fixed denominator (`2` yields halves, `3` is for thirds, `4` is quarters, etc.). For the latter option, using `simplify = TRUE` will simplify fractions where possible (e.g., `2/4` will be simplified as `1/2`). By default, the `"low"` option is used.

simplify — If choosing to provide a numeric value for accuracy, the option to simplify the fraction (where possible) can be taken with TRUE (the default). With FALSE, denominators in fractions will be fixed to the value provided in accuracy.

layout — For HTML output, the "inline" layout is the default. This layout places the numerals of the fraction on the baseline and uses a standard slash character. The "diagonal" layout will generate fractions that are typeset with raised/lowered numerals and a virgule.

use_seps — An option to use digit group separators. The type of digit group separator is set by sep_mark and overridden if a locale ID is provided to locale. This setting is TRUE by default.

pattern — A formatting pattern that allows for decoration of the formatted value. The value itself is represented by {x} and all other characters are taken to be string literals.

sep_mark — The mark to use as a separator between groups of digits (e.g., using sep_mark = "," with 1000 would result in a formatted value of 1,000).

system — The numbering system to use. By default, this is the international numbering system ("intl") whereby grouping separators (i.e., sep_mark) are separated by three digits. The alternative system, the Indian numbering system ("ind") uses grouping separators that correspond to thousand, lakh, crore, and higher quantities.

locale — An optional locale ID that can be used for formatting the value according the locale's rules. Examples include "en" for English (United States) and "fr" for French (France). The use of a valid locale ID will override any values provided in sep_mark and dec_mark. We can use the info_locales() function as a useful reference for all of the locales that are supported. Any locale value provided here will override any global locale setting performed in gt()'s own locale argument.

Value

An object of class gt_tbl.

Targeting the values to be formatted

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the rows argument. See the *Arguments* section for more information on this.

Examples

Use pizzaplace to create a **gt** table. Format the f_sold and f_income columns to display fractions.

```
pizzaplace %>%
  dplyr::group_by(type, size) %>%
  dplyr::summarize(
    sold = dplyr::n(),
    income = sum(price),
    .groups = "drop_last"
```

```
) %>%
dplyr::group_by(type) %>%
dplyr::mutate(
  f_sold = sold / sum(sold),
  f_income = income / sum(income),
) %>%
dplyr::arrange(type, dplyr::desc(income)) %>%
gt(rowname_col = "size") %>%
tab_header(
  title = "Pizzas Sold in 2015",
  subtitle = "Fraction of Sell Count and Revenue by Size per Type"
) %>%
fmt_integer(columns = sold) %>%
fmt_currency(columns = income) %>%
fmt_fraction(
  columns = starts_with("f_"),
  accuracy = 10,
  simplify = FALSE,
  layout = "diagonal"
) %>%
sub_missing(missing_text = "") %>%
tab_spanner(
  label = "Sold",
  columns = contains("sold")
) %>%
tab_spanner(
  label = "Revenue",
  columns = contains("income")
) %>%
text_transform(
  locations = cells_body(),
  fn = function(x) {
    dplyr::case_when(
      x == 0 ~ "<em>nil</em>",
      x != 0 ~ x
    )
  }
) %>%
cols_label(
  sold = "Amount",
  income = "Amount",
  f_sold = md("_f_"),
  f_income = md("_f_")
) %>%
cols_align(align = "center", columns = starts_with("f")) %>%
tab_options(
  table.width = px(400),
  row_group.as_column = TRUE
```

```
)
```

Function ID

3-7

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

fmt_integer *Format values as integers*

Description

With numeric values in a **gt** table, we can perform number-based formatting so that the targeted values are always rendered as integer values. We can have fine control over integer formatting with the following options:

- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- scaling: we can choose to scale targeted values by a multiplier value
- large-number suffixing: larger figures (thousands, millions, etc.) can be autoscaled and decorated with the appropriate suffixes
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
fmt_integer(
  data,
  columns,
  rows = everything(),
  use_seps = TRUE,
  accounting = FALSE,
  scale_by = 1,
  suffixing = FALSE,
  pattern = "{x}",
  sep_mark = ",",
  force_sign = FALSE,
  system = c("intl", "ind"),
  locale = NULL
)
```

Arguments

`data` A table object that is created using the `gt()` function.

`columns` The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.

`rows` Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2] < 5`

`use_seps` An option to use digit group separators. The type of digit group separator is set by `sep_mark` and overridden if a locale ID is provided to `locale`. This setting is `TRUE` by default.

`accounting` An option to use accounting style for values. With `FALSE` (the default), negative values will be shown with a minus sign. Using `accounting = TRUE` will put negative values in parentheses.

`scale_by` A value to scale the input. The default is `1.0`. All numeric values will be multiplied by this value first before undergoing formatting. This value will be ignored if using any of the `suffixing` options (i.e., where `suffixing` is not set to `FALSE`).

`suffixing` An option to scale and apply suffixes to larger numbers (e.g., `1924000` can be transformed to `2M`). This option can accept a logical value, where `FALSE` (the default) will not perform this transformation and `TRUE` will apply thousands (`K`), millions (`M`), billions (`B`), and trillions (`T`) suffixes after automatic value scaling. We can also specify which symbols to use for each of the value ranges by using a character vector of the preferred symbols to replace the defaults (e.g., `c("k", "Ml", "Bn", "Tr")`).

Including `NA` values in the vector will ensure that the particular range will either not be included in the transformation (e.g, `c(NA, "M", "B", "T")` won't modify numbers in the thousands range) or the range will inherit a previous suffix (e.g., with `c("K", "M", NA, "T")`, all numbers in the range of millions and billions will be in terms of millions).

Any use of `suffixing` (where it is not set expressly as `FALSE`) means that any value provided to `scale_by` will be ignored.

`pattern` A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

`sep_mark` The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

`force_sign` Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign. This option is disregarded when using accounting notation with `accounting = TRUE`.

system
: The numbering system to use. By default, this is the international numbering system ("intl") whereby grouping separators (i.e., sep_mark) are separated by three digits. The alternative system, the Indian numbering system ("ind") uses grouping separators that correspond to thousand, lakh, crore, and higher quantities.

locale
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include "en" for English (United States) and "fr" for French (France). The use of a valid locale ID will override any values provided in sep_mark and dec_mark. We can use the info_locales() function as a useful reference for all of the locales that are supported. Any locale value provided here will override any global locale setting performed in gt()'s own locale argument.

Value

An object of class gt_tbl.

Targeting the values to be formatted

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the rows argument. See the *Arguments* section for more information on this.

Examples

Use exibble to create a **gt** table. format the num column as integer values having no digit separators (with the use_seps = FALSE option).

```
exibble %>%
  dplyr::select(num, char) %>%
  gt() %>%
  fmt_integer(
    columns = num,
    use_seps = FALSE
  )
```

Function ID

3-2

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

fmt_markdown *Format Markdown text*

Description

Any Markdown-formatted text in the incoming cells will be transformed to the appropriate output type during render when using `fmt_markdown()`.

Usage

```
fmt_markdown(data, columns, rows = everything())
```

Arguments

data	A table object that is created using the `gt()` function.
columns	The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.
rows	Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2] < 5`

Value

An object of class `gt_tbl`.

Targeting the values to be formatted

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the *Arguments* section for more information on this.

Examples

Create a few Markdown-based text snippets.

```
text_1a <- "
### This is Markdown.

Markdown's syntax is comprised entirely of
punctuation characters, which punctuation
characters have been carefully chosen so as
to look like what they mean... assuming
```

```
you've ever used email.
"

text_1b <- "
Info on Markdown syntax can be found
[here](https://daringfireball.net/projects/markdown/).
"

text_2a <- "
The **gt** package has these datasets:

 - `countrypops`
 - `sza`
 - `gtcars`
 - `sp500`
 - `pizzaplace`
 - `exibble`
"

text_2b <- "
There's a quick reference [here](https://commonmark.org/help/).
"
```

Arrange the text snippets as a tibble using the `dplyr::tribble()` function. then, create a **gt** table and format all columns with `fmt_markdown()`.

```
dplyr::tribble(
  ~Markdown, ~md,
  text_1a,   text_2a,
  text_1b,   text_2b,
) %>%
  gt() %>%
  fmt_markdown(columns = everything()) %>%
  tab_options(table.width = px(400))
```

Function ID

3-15

See Also

Other data formatting functions: `data_color()`, `fmt_bytes()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_duration()`, `fmt_engineering()`, `fmt_fraction()`, `fmt_integer()`, `fmt_number()`, `fmt_partsper()`, `fmt_passthrough()`, `fmt_percent()`, `fmt_roman()`, `fmt_scientific()`, `fmt_time()`, `fmt()`, `sub_large_vals()`, `sub_missing()`, `sub_small_vals()`, `sub_values()`, `sub_zero()`, `text_transform()`

fmt_number *Format numeric values*

Description

With numeric values in a **gt** table, we can perform number-based formatting so that the targeted values are rendered with a higher consideration for tabular presentation. Furthermore, there is finer control over numeric formatting with the following options:

- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol
- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- scaling: we can choose to scale targeted values by a multiplier value
- large-number suffixing: larger figures (thousands, millions, etc.) can be autoscaled and decorated with the appropriate suffixes
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
fmt_number(
  data,
  columns,
  rows = everything(),
  decimals = 2,
  n_sigfig = NULL,
  drop_trailing_zeros = FALSE,
  drop_trailing_dec_mark = TRUE,
  use_seps = TRUE,
  accounting = FALSE,
  scale_by = 1,
  suffixing = FALSE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  system = c("intl", "ind"),
  locale = NULL
)
```

Arguments

`data`	A table object that is created using the `gt()` function.

columns
: The columns to format. Can either be a series of column names provided in c(), a vector of column indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything().

rows
: Optional rows to format. Providing everything() (the default) results in all rows in columns being formatted. Alternatively, we can supply a vector of row captions within c(), a vector of row indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 5

decimals
: An option to specify the exact number of decimal places to use. The default number of decimal places is 2.

n_sigfig
: A option to format numbers to *n* significant figures. By default, this is NULL and thus number values will be formatted according to the number of decimal places set via decimals. If opting to format according to the rules of significant figures, n_sigfig must be a number greater than or equal to 1. Any values passed to the decimals and drop_trailing_zeros arguments will be ignored.

drop_trailing_zeros
: A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).

drop_trailing_dec_mark
: A logical value that determines whether decimal marks should always appear even if there are no decimal digits to display after formatting (e.g, 23 becomes 23.). The default for this is TRUE, which means that trailing decimal marks are not shown.

use_seps
: An option to use digit group separators. The type of digit group separator is set by sep_mark and overridden if a locale ID is provided to locale. This setting is TRUE by default.

accounting
: An option to use accounting style for values. With FALSE (the default), negative values will be shown with a minus sign. Using accounting = TRUE will put negative values in parentheses.

scale_by
: A value to scale the input. The default is 1.0. All numeric values will be multiplied by this value first before undergoing formatting. This value will be ignored if using any of the suffixing options (i.e., where suffixing is not set to FALSE).

suffixing
: An option to scale and apply suffixes to larger numbers (e.g., 1924000 can be transformed to 1.92M). This option can accept a logical value, where FALSE (the default) will not perform this transformation and TRUE will apply thousands (K), millions (M), billions (B), and trillions (T) suffixes after automatic value scaling. We can also specify which symbols to use for each of the value ranges by using a character vector of the preferred symbols to replace the defaults (e.g., c("k", "Ml", "Bn", "Tr")).

  Including NA values in the vector will ensure that the particular range will either not be included in the transformation (e.g, c(NA, "M", "B", "T") won't modify numbers in the thousands range) or the range will inherit a previous suffix (e.g., with `c("K", "M", NA, "T")`, all numbers in the range of millions and billions will be in terms of millions).

Any use of `suffixing` (where it is not set expressly as `FALSE`) means that any value provided to `scale_by` will be ignored.

If using `system = "ind"` then the default suffix set provided by `suffixing = TRUE` will be `c(NA, "L", "Cr")`. This doesn't apply suffixes to the thousands range, but does express values in lakhs and crores.

`pattern` A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

`sep_mark` The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

`dec_mark` The character to use as a decimal mark (e.g., using `dec_mark = ","` with `0.152` would result in a formatted value of `0,152`).

`force_sign` Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign. This option is disregarded when using accounting notation with `accounting = TRUE`.

`system` The numbering system to use. By default, this is the international numbering system (`"intl"`) whereby grouping separators (i.e., `sep_mark`) are separated by three digits. The alternative system, the Indian numbering system (`"ind"`) uses grouping separators that correspond to thousand, lakh, crore, and higher quantities.

`locale` An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported. Any `locale` value provided here will override any global locale setting performed in `gt()`'s own `locale` argument.

Value

An object of class `gt_tbl`.

Targeting the values to be formatted

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the *Arguments* section for more information on this.

Examples

Use `exibble` to create a **gt** table. Format the `num` column as numeric with three decimal places and with no use of digit separators.

```
exibble %>%
  gt() %>%
```

```
  fmt_number(
    columns = num,
    decimals = 3,
    use_seps = FALSE
  )
```

Use countrypops to create a **gt** table. Format all numeric columns to use large-number suffixing with the suffixing = TRUE option.

```
countrypops %>%
  dplyr::select(country_code_3, year, population) %>%
  dplyr::filter(country_code_3 %in% c("CHN", "IND", "USA", "PAK", "IDN")) %>%
  dplyr::filter(year > 1975 & year %% 5 == 0) %>%
  tidyr::spread(year, population) %>%
  dplyr::arrange(desc(`2015`)) %>%
  gt(rowname_col = "country_code_3") %>%
  fmt_number(
    columns = 2:9,
    decimals = 2,
    suffixing = TRUE
  )
```

Function ID

3-1

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

fmt_partsper   *Format values as parts-per quantities*

Description

With numeric values in a **gt** table we can format the values so that they are rendered as *per mille*, *ppm*, *ppb*, etc., quantities. The following list of keywords (with associated naming and scaling factors) is available to use within fmt_partsper():

- "per-mille": Per mille, (1 part in 1,000)
- "per-myriad": Per myriad, (1 part in 10,000)
- "pcm": Per cent mille (1 part in 100,000)

- "ppm": Parts per million, (1 part in 1,000,000)
- "ppb": Parts per billion, (1 part in 1,000,000,000)
- "ppt": Parts per trillion, (1 part in 1,000,000,000,000)
- "ppq": Parts per quadrillion, (1 part in 1,000,000,000,000,000)

The function provides a lot of formatting control and we can use the following options:

- custom symbol/units: we can override the automatic symbol or units display with our own choice as the situation warrants
- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol
- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- value scaling toggle: choose to disable automatic value scaling in the situation that values are already scaled coming in (and just require the appropriate symbol or unit display)
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
fmt_partsper(
  data,
  columns,
  rows = everything(),
  to_units = c("per-mille", "per-myriad", "pcm", "ppm", "ppb", "ppt", "ppq"),
  symbol = "auto",
  decimals = 2,
  drop_trailing_zeros = FALSE,
  drop_trailing_dec_mark = TRUE,
  scale_values = TRUE,
  use_seps = TRUE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  incl_space = "auto",
  system = c("intl", "ind"),
  locale = NULL
)
```

Arguments

data	A table object that is created using the gt() function.
columns	The columns to format. Can either be a series of column names provided in c(), a vector of column indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything().

rows
: Optional rows to format. Providing everything() (the default) results in all rows in columns being formatted. Alternatively, we can supply a vector of row captions within c(), a vector of row indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 5

to_units
: A keyword that signifies the desired output quantity. This can be any from the following set: "per-mille", "per-myriad", "pcm", "ppm", "ppb", "ppt", or "ppq".

symbol
: The symbol/units to use for the quantity. By default, this is set to "auto" and **gt** will choose the appropriate symbol based on the to_units keyword and the output context. However, this can be changed by supplying a string (e.g, using symbol = "ppbV" when to_units = "ppb").

decimals
: An option to specify the exact number of decimal places to use. The default number of decimal places is 2.

drop_trailing_zeros
: A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).

drop_trailing_dec_mark
: A logical value that determines whether decimal marks should always appear even if there are no decimal digits to display after formatting (e.g, 23 becomes 23.). The default for this is TRUE, which means that trailing decimal marks are not shown.

scale_values
: Should the values be scaled through multiplication according to the keyword set in to_units? By default this is TRUE since the expectation is that normally values are proportions. Setting to FALSE signifies that the values are already scaled and require only the appropriate symbol/units when formatted.

use_seps
: An option to use digit group separators. The type of digit group separator is set by sep_mark and overridden if a locale ID is provided to locale. This setting is TRUE by default.

pattern
: A formatting pattern that allows for decoration of the formatted value. The value itself is represented by {x} and all other characters are taken to be string literals.

sep_mark
: The mark to use as a separator between groups of digits (e.g., using sep_mark = "," with 1000 would result in a formatted value of 1,000).

dec_mark
: The character to use as a decimal mark (e.g., using dec_mark = "," with 0.152 would result in a formatted value of 0,152).

force_sign
: Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use TRUE for this option. The default is FALSE, where only negative numbers will display a minus sign. This option is disregarded when using accounting notation with accounting = TRUE.

incl_space
: An option for whether to include a space between the value and the symbol/units. The default is "auto" which provides spacing dependent on the mark itself. This can be directly controlled by using either TRUE or FALSE.

system
: The numbering system to use. By default, this is the international numbering system ("intl") whereby grouping separators (i.e., sep_mark) are separated

by three digits. The alternative system, the Indian numbering system ("ind") uses grouping separators that correspond to thousand, lakh, crore, and higher quantities.

locale
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include "en" for English (United States) and "fr" for French (France). The use of a valid locale ID will override any values provided in sep_mark and dec_mark. We can use the info_locales() function as a useful reference for all of the locales that are supported. Any locale value provided here will override any global locale setting performed in gt()'s own locale argument.

Value

An object of class gt_tbl.

Targeting the values to be formatted

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the rows argument. See the *Arguments* section for more information on this.

Examples

Create a tibble of small numeric values and generate a **gt** table. Format the a column to appear in scientific notation with fmt_scientific() and format the b column as *per mille* values with fmt_partsper().

```
dplyr::tibble(x = 0:-5, a = 10^(0:-5), b = a) %>%
  gt(rowname_col = "x") %>%
  fmt_scientific(a, decimals = 0) %>%
  fmt_partsper(
    columns = b,
    to_units = "per-mille"
  )
```

Function ID

3-6

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

fmt_passthrough *Format by simply passing data through*

Description

Format by passing data through no other transformation other than: (1) coercing to character (as all the fmt_*() functions do), and (2) applying text via the pattern argument (the default is to apply nothing). All of this is useful when don't want to modify the input data other than to decorate it within a pattern. Also, this function is useful when used as the formatter function in the summary_rows() function, where the output may be text or useful as is.

Usage

```
fmt_passthrough(
  data,
  columns,
  rows = everything(),
  escape = TRUE,
  pattern = "{x}"
)
```

Arguments

data	A table object that is created using the gt() function.
columns	The columns to format. Can either be a series of column names provided in c(), a vector of column indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything().
rows	Optional rows to format. Providing everything() (the default) results in all rows in columns being formatted. Alternatively, we can supply a vector of row captions within c(), a vector of row indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 5
escape	An option to escape text according to the final output format of the table. For example, if a LaTeX table is to be generated then LaTeX escaping would be performed during rendering. By default this is set to TRUE and setting to FALSE would be useful in the case where text is crafted for a specific output format in mind.
pattern	A formatting pattern that allows for decoration of the formatted value. The value itself is represented by {x} and all other characters are taken to be string literals.

Value

An object of class gt_tbl.

Targeting the values to be formatted

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the *Arguments* section for more information on this.

Examples

Use `exibble` to create a **gt** table. Keep only the `char` column. Pass the data in that column through but apply a simple pattern that adds an `"s"` to the non-`NA` values.

```
exibble %>%
  dplyr::select(char) %>%
  gt() %>%
  fmt_passthrough(
    columns = char,
    rows = !is.na(char),
    pattern = "{x}s"
  )
```

Function ID

3-16

See Also

Other data formatting functions: `data_color()`, `fmt_bytes()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_duration()`, `fmt_engineering()`, `fmt_fraction()`, `fmt_integer()`, `fmt_markdown()`, `fmt_number()`, `fmt_partsper()`, `fmt_percent()`, `fmt_roman()`, `fmt_scientific()`, `fmt_time()`, `fmt()`, `sub_large_vals()`, `sub_missing()`, `sub_small_vals()`, `sub_values()`, `sub_zero()`, `text_transform()`

`fmt_percent` *Format values as a percentage*

Description

With numeric values in a **gt** table, we can perform percentage-based formatting. It is assumed the input numeric values are proportional values and, in this case, the values will be automatically multiplied by `100` before decorating with a percent sign (the other case is accommodated though setting the `scale_values` to `FALSE`). For more control over percentage formatting, we can use the following options:

- percent sign placement: the percent sign can be placed after or before the values and a space can be inserted between the symbol and the value.
- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol

- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- value scaling toggle: choose to disable automatic value scaling in the situation that values are already scaled coming in (and just require the percent symbol)
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
fmt_percent(
  data,
  columns,
  rows = everything(),
  decimals = 2,
  drop_trailing_zeros = FALSE,
  drop_trailing_dec_mark = TRUE,
  scale_values = TRUE,
  use_seps = TRUE,
  accounting = FALSE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  incl_space = FALSE,
  placement = "right",
  system = c("intl", "ind"),
  locale = NULL
)
```

Arguments

`data` A table object that is created using the `gt()` function.

`columns` The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.

`rows` Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2] < 5`

`decimals` An option to specify the exact number of decimal places to use. The default number of decimal places is 2.

`drop_trailing_zeros`
A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).

drop_trailing_dec_mark
: A logical value that determines whether decimal marks should always appear even if there are no decimal digits to display after formatting (e.g, `23` becomes `23.`). The default for this is `TRUE`, which means that trailing decimal marks are not shown.

scale_values
: Should the values be scaled through multiplication by 100? By default this is `TRUE` since the expectation is that normally values are proportions. Setting to `FALSE` signifies that the values are already scaled and require only the percent sign when formatted.

use_seps
: An option to use digit group separators. The type of digit group separator is set by `sep_mark` and overridden if a locale ID is provided to `locale`. This setting is `TRUE` by default.

accounting
: An option to use accounting style for values. With `FALSE` (the default), negative values will be shown with a minus sign. Using `accounting = TRUE` will put negative values in parentheses.

pattern
: A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

sep_mark
: The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

dec_mark
: The character to use as a decimal mark (e.g., using `dec_mark = ","` with `0.152` would result in a formatted value of `0,152`).

force_sign
: Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign. This option is disregarded when using accounting notation with `accounting = TRUE`.

incl_space
: An option for whether to include a space between the value and the percent sign. The default is to not introduce a space character.

placement
: The placement of the percent sign. This can be either be `right` (the default) or `left`.

system
: The numbering system to use. By default, this is the international numbering system (`"intl"`) whereby grouping separators (i.e., `sep_mark`) are separated by three digits. The alternative system, the Indian numbering system (`"ind"`) uses grouping separators that correspond to thousand, lakh, crore, and higher quantities.

locale
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported. Any `locale` value provided here will override any global locale setting performed in `gt()`'s own `locale` argument.

Value

An object of class `gt_tbl`.

Targeting the values to be formatted

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the *Arguments* section for more information on this.

Examples

Use `pizzaplace` to create a **gt** table. Format the `frac_of_quota` column to display values as percentages.

```
pizzaplace %>%
  dplyr::mutate(month = as.numeric(substr(date, 6, 7))) %>%
  dplyr::group_by(month) %>%
  dplyr::summarize(pizzas_sold = dplyr::n()) %>%
  dplyr::ungroup() %>%
  dplyr::mutate(frac_of_quota = pizzas_sold / 4000) %>%
  gt(rowname_col = "month") %>%
  fmt_percent(
    columns = frac_of_quota,
    decimals = 1
  )
```

Function ID

3-5

See Also

Other data formatting functions: `data_color()`, `fmt_bytes()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_duration()`, `fmt_engineering()`, `fmt_fraction()`, `fmt_integer()`, `fmt_markdown()`, `fmt_number()`, `fmt_partsper()`, `fmt_passthrough()`, `fmt_roman()`, `fmt_scientific()`, `fmt_time()`, `fmt()`, `sub_large_vals()`, `sub_missing()`, `sub_small_vals()`, `sub_values()`, `sub_zero()`, `text_transform()`

---

fmt_roman — *Format values as Roman numerals*

Description

With numeric values in a **gt** table we can transform those to Roman numerals, rounding values as necessary.

Usage

```
fmt_roman(
  data,
  columns,
  rows = everything(),
  case = c("upper", "lower"),
  pattern = "{x}"
)
```

Arguments

data	A table object that is created using the gt() function.
columns	The columns to format. Can either be a series of column names provided in c(), a vector of column indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything().
rows	Optional rows to format. Providing everything() (the default) results in all rows in columns being formatted. Alternatively, we can supply a vector of row captions within c(), a vector of row indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 5
case	Should Roman numerals should be rendered as uppercase ("upper") or lowercase ("lower") letters? By default, this is set to "upper".
pattern	A formatting pattern that allows for decoration of the formatted value. The value itself is represented by {x} and all other characters are taken to be string literals.

Value

An object of class gt_tbl.

Targeting the values to be formatted

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the rows argument. See the *Arguments* section for more information on this.

Examples

Create a tibble of small numeric values and generate a **gt** table. Format the roman column to appear as Roman numerals with fmt_roman().

```
dplyr::tibble(arabic = c(1, 8, 24, 85), roman = arabic) %>%
  gt(rowname_col = "arabic") %>%
  fmt_roman(columns = roman)
```

Function ID

3-9

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

fmt_scientific *Format values to scientific notation*

Description

With numeric values in a **gt** table, we can perform formatting so that the targeted values are rendered in scientific notation. Furthermore, there is fine control with the following options:

- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol
- scaling: we can choose to scale targeted values by a multiplier value
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in formatting specific to the chosen locale

Usage

```
fmt_scientific(
  data,
  columns,
  rows = everything(),
  decimals = 2,
  drop_trailing_zeros = FALSE,
  scale_by = 1,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  locale = NULL
)
```

Arguments

data A table object that is created using the `gt()` function.

columns The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.

rows Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2] < 5`

decimals An option to specify the exact number of decimal places to use. The default number of decimal places is `2`.

drop_trailing_zeros
A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).

scale_by A value to scale the input. The default is `1.0`. All numeric values will be multiplied by this value first before undergoing formatting.

pattern A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

sep_mark The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

dec_mark The character to use as a decimal mark (e.g., using `dec_mark = ","` with `0.152` would result in a formatted value of `0,152`).

force_sign Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign.

locale An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported. Any `locale` value provided here will override any global locale setting performed in `gt()`'s own `locale` argument.

Value

An object of class `gt_tbl`.

Targeting the values to be formatted

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the *Arguments* section for more information on this.

Examples

Use exibble to create a **gt** table. Format the num column as partially numeric and partially in scientific notation (using the num > 500 and num <= 500 expressions in the respective rows arguments).

```
exibble %>%
  gt() %>%
  fmt_number(
    columns = num,
    rows = num > 500,
    decimals = 1,
    scale_by = 1/1000,
    pattern = "{x}K"
  ) %>%
  fmt_scientific(
    columns = num,
    rows = num <= 500,
    decimals = 1
  )
```

Function ID

3-3

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

fmt_time — *Format values as times*

Description

Format input values to time values using one of 25 preset time styles. Input can be in the form of POSIXt (i.e., datetimes), character (must be in the ISO 8601 forms of HH:MM:SS or YYYY-MM-DD HH:MM:SS), or Date (which always results in the formatting of 00:00:00).

Usage

```
fmt_time(
  data,
  columns,
  rows = everything(),
  time_style = "iso",
```

```
    pattern = "{x}",
    locale = NULL
  )
```

Arguments

- `data` A table object that is created using the `gt()` function.
- `columns` The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.
- `rows` Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2] < 5`
- `time_style` The time style to use. By default this is `"iso"` which corresponds to how times are formatted within ISO 8601 datetime values. The other time styles can be viewed using `info_time_style()`.
- `pattern` A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.
- `locale` An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported. Any `locale` value provided here will override any global locale setting performed in `gt()`'s own `locale` argument.

Value

An object of class `gt_tbl`.

Targeting the values to be formatted

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the *Arguments* section for more information on this.

Formatting with the `time_style` argument

We need to supply a preset time style to the `time_style` argument. There are many time styles and all of them can handle localization to any supported locale. Many of the time styles are termed flexible time formats and this means that their output will adapt to any `locale` provided. That feature makes the flexible time formats a better option for locales other than `"en"` (the default locale).

The following table provides a listing of all time styles and their output values (corresponding to an input time of `14:35:00`). It is noted which of these represent 12- or 24-hour time.

	Time Style	Output	Notes
1	"iso"	"14:35:00"	ISO 8601, 24h
2	"iso-short"	"14:35"	ISO 8601, 24h
3	"h_m_s_p"	"2:35:00 PM"	12h
4	"h_m_p"	"2:35 PM"	12h
5	"h_p"	"2 PM"	12h
6	"Hms"	"14:35:00"	flexible, 24h
7	"Hm"	"14:35"	flexible, 24h
8	"H"	"14"	flexible, 24h
9	"EHm"	"Thu 14:35"	flexible, 24h
10	"EHms"	"Thu 14:35:00"	flexible, 24h
11	"Hmsv"	"14:35:00 GMT+00:00"	flexible, 24h
12	"Hmv"	"14:35 GMT+00:00"	flexible, 24h
13	"hms"	"2:35:00 PM"	flexible, 12h
14	"hm"	"2:35 PM"	flexible, 12h
15	"h"	"2 PM"	flexible, 12h
16	"Ehm"	"Thu 2:35 PM"	flexible, 12h
17	"Ehms"	"Thu 2:35:00 PM"	flexible, 12h
18	"EBhms"	"Thu 2:35:00 in the afternoon"	flexible, 12h
19	"Bhms"	"2:35:00 in the afternoon"	flexible, 12h
20	"EBhm"	"Thu 2:35 in the afternoon"	flexible, 12h
21	"Bhm"	"2:35 in the afternoon"	flexible, 12h
22	"Bh"	"2 in the afternoon"	flexible, 12h
23	"hmsv"	"2:35:00 PM GMT+00:00"	flexible, 12h
24	"hmv"	"2:35 PM GMT+00:00"	flexible, 12h
25	"ms"	"35:00"	flexible

We can use the `info_time_style()` within the console to view a similar table of time styles with example output.

Examples

Use `exibble` to create a **gt** table. Keep only the `date` and `time` columns. Format the `time` column to have times formatted as `hms_p` (time style `3`).

```
exibble %>%
  dplyr::select(date, time) %>%
  gt() %>%
  fmt_time(
    columns = time,
    time_style = "h_m_s_p"
  )
```

Use `exibble` to create a **gt** table. Keep only the `date` and `time` columns. Format the `time` column to have mixed time formats (times after 16:00 will be different than the others because of the expressions used in the rows argument).

```
exibble %>%
  dplyr::select(date, time) %>%
  gt() %>%
  fmt_time(
    columns = time,
    rows = time > "16:00",
    time_style = "h_m_s_p"
  ) %>%
  fmt_time(
    columns = time,
    rows = time <= "16:00",
    time_style = "h_m_p"
  )
```

Use exibble to create another **gt** table, this time only with the time column. Format the time column to use the "EBhms" time style (which is one of the 'flexible' styles). Also, set the locale to "sv" to get the dates in Swedish.

```
exibble %>%
  dplyr::select(time) %>%
  gt() %>%
  fmt_time(
    columns = time,
    time_style = "EBhms",
    locale = "sv"
  )
```

Function ID

3-12

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

ggplot_image *Helper function for adding a ggplot*

Description

We can add a **ggplot2** plot inside of a table with the help of the ggplot_image() function. The function provides a convenient way to generate an HTML fragment with a ggplot object. Because this function is currently HTML-based, it is only useful for HTML table output. To use this function inside of data cells, it is recommended that the text_transform() function is used. With that function, we can specify which data cells to target and then include a call to ggplot_image() within the required user-defined function (for the fn argument). If we want to include a plot in other places (e.g., in the header, within footnote text, etc.) we need to use ggplot_image() within the html() helper function.

Usage

```
ggplot_image(plot_object, height = 100, aspect_ratio = 1)
```

Arguments

plot_object	A ggplot plot object.
height	The absolute height (px) of the image in the table cell.
aspect_ratio	The plot's final aspect ratio. Where the height of the plot is fixed using the height argument, the aspect_ratio will either compress (aspect_ratio < 1.0) or expand (aspect_ratio > 1.0) the plot horizontally. The default value of 1.0 will neither compress nor expand the plot.

Details

By itself, the function creates an HTML image tag with an image URI embedded within (a 100 dpi PNG). We can easily experiment with any ggplot2 plot object, and using it within ggplot_image(plot_object = <plot ob
evaluates to:
<img src=<data URI> style=\"height:100px;\">

where a height of 100px is a default height chosen to work well within the heights of most table rows. There is the option to modify the aspect ratio of the plot (the default aspect_ratio is 1.0) and this is useful for elongating any given plot to fit better within the table construct.

Value

A character object with an HTML fragment that can be placed inside of a cell.

Examples

Create a **ggplot** plot.

```
library(ggplot2)

plot_object <-
  ggplot(
    data = gtcars,
    aes(x = hp, y = trq, size = msrp)
  ) +
```

```
  geom_point(color = "blue") +
  theme(legend.position = "none")
```

Create a tibble that contains two cells (where one is a placeholder for an image), then, create a **gt** table. Use the `text_transform()` function to insert the plot using by calling `ggplot_object()` within the user- defined function.

```
dplyr::tibble(
  text = "Here is a ggplot:",
  ggplot = NA
) %>%
  gt() %>%
  text_transform(
    locations = cells_body(columns = ggplot),
    fn = function(x) {
      plot_object %>%
        ggplot_image(height = px(200))
    }
  )
```

Function ID

8-3

See Also

Other image addition functions: `local_image()`, `test_image()`, `web_image()`

`google_font` *Helper function for specifying a font from the* Google Fonts *service*

Description

The `google_font()` helper function can be used wherever a font name should be specified. There are two instances where this helper can be used: the `name` argument in `opt_table_font()` (for setting a table font) and in that of `cell_text()` (used with `tab_style()`). To get a helpful listing of fonts that work well in tables, use the `info_google_fonts()` function.

Usage

```
google_font(name)
```

Arguments

`name`	The complete name of a font available in *Google Fonts*.

Value

An object of class font_css.

Examples

Use exibble to create a **gt** table of eight rows, replace missing values with em dashes. For text in the time column, we use the Google font "IBM Plex Mono" and set up the default_fonts() as fallbacks (just in case the webfont is not accessible).

```
exibble %>%
  dplyr::select(char, time) %>%
  gt() %>%
  sub_missing() %>%
  tab_style(
    style = cell_text(
      font = c(
        google_font(name = "IBM Plex Mono"),
        default_fonts()
      )
    ),
    locations = cells_body(columns = time)
  )
```

Use sp500 to create a small **gt** table, using fmt_currency() to provide a dollar sign for the first row of monetary values. Then, set a larger font size for the table and use the "Merriweather" font using the google_font() function (with two font fallbacks: "Cochin" and the catchall "Serif" group).

```
sp500 %>%
  dplyr::slice(1:10) %>%
  dplyr::select(-volume, -adj_close) %>%
  gt() %>%
  fmt_currency(
    columns = 2:5,
    rows = 1,
    currency = "USD",
    use_seps = FALSE
  ) %>%
  tab_options(table.font.size = px(20)) %>%
  opt_table_font(
    font = list(
      google_font(name = "Merriweather"),
      "Cochin", "Serif"
    )
  )
```

Function ID

7-27

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id(), stub()

---

grand_summary_rows *Add grand summary rows using aggregation functions*

Description

Add grand summary rows to the **gt** table by using applying aggregation functions to the table data. The summary rows incorporate all of the available data, regardless of whether some of the data are part of row groups. You choose how to format the values in the resulting summary cells by use of a formatter function (e.g, fmt_number) and any relevant options.

Usage

```
grand_summary_rows(
  data,
  columns = everything(),
  fns,
  missing_text = "---",
  formatter = fmt_number,
  ...
)
```

Arguments

data	A table object that is created using the gt() function.
columns	The columns for which the summaries should be calculated.
fns	Functions used for aggregations. This can include base functions like mean, min, max, median, sd, or sum or any other user-defined aggregation function. The function(s) should be supplied within a list(). Within that list, we can specify the functions by use of function names in quotes (e.g., "sum"), as bare functions (e.g., sum), or as one-sided R formulas using a leading ~. In the formula representation, a . serves as the data to be summarized (e.g., sum(., na.rm = TRUE)). The use of named arguments is recommended as the names will serve as summary row labels for the corresponding summary rows data (the labels can derived from the function names but only when not providing bare function names).
missing_text	The text to be used in place of NA values in summary cells with no data outputs.

formatter
: A formatter function name. These can be any of the `fmt_*()` functions available in the package (e.g., `fmt_number()`, `fmt_percent()`, etc.), or a custom function using `fmt()`. The default function is `fmt_number()` and its options can be accessed through `...`.

...
: Values passed to the `formatter` function, where the provided values are to be in the form of named vectors. For example, when using the default `formatter` function, `fmt_number()`, options such as `decimals`, `use_seps`, and `locale` can be used.

Details

Should we need to obtain the summary data for external purposes, the `extract_summary()` function can be used with a `gt_tbl` object where grand summary rows were added via `grand_summary_rows()`.

Value

An object of class `gt_tbl`.

Examples

Use `sp500` to create a **gt** table with row groups. Create the grand summary rows `min`, `max`, and `avg` for the table with the `grand_summary_rows()` function.

```
sp500 %>%
  dplyr::filter(date >= "2015-01-05" & date <= "2015-01-16") %>%
  dplyr::arrange(date) %>%
  dplyr::mutate(week = paste0("W", strftime(date, format = "%V"))) %>%
  dplyr::select(-adj_close, -volume) %>%
  gt(
    rowname_col = "date",
    groupname_col = "week"
  ) %>%
  grand_summary_rows(
    columns = c(open, high, low, close),
    fns = list(
      min = ~min(.),
      max = ~max(.),
      avg = ~mean(.)),
    formatter = fmt_number,
    use_seps = FALSE
  )
```

Function ID

5-2

See Also

Other row addition/modification functions: `row_group_order()`, `summary_rows()`

gt *Create a **gt** table object*

Description

The `gt()` function creates a **gt** table object when provided with table data. Using this function is the first step in a typical **gt** workflow. Once we have the **gt** table object, we can perform styling transformations before rendering to a display table of various formats.

Usage

```
gt(
  data,
  rowname_col = "rowname",
  groupname_col = dplyr::group_vars(data),
  process_md = FALSE,
  caption = NULL,
  rownames_to_stub = FALSE,
  auto_align = TRUE,
  id = NULL,
  locale = NULL,
  row_group.sep = getOption("gt.row_group.sep", " - ")
)
```

Arguments

`data` A `data.frame` object or a tibble.

`rowname_col` The column name in the input `data` table to use as row captions to be placed in the display table stub. If the `rownames_to_stub` option is `TRUE` then any column name provided to `rowname_col` will be ignored.

`groupname_col` The column name in the input `data` table to use as group labels for generation of stub row groups. If the input `data` table has the `grouped_df` class (through use of the `dplyr::group_by()` function or associated `group_by*()` functions) then any input here is ignored.

`process_md` Should the contents of the `rowname_col` and `groupname_col` be interpreted as Markdown? By default this is `FALSE`.

`caption` An optional table caption to use for cross-referencing in R Markdown, Quarto, or **bookdown**.

`rownames_to_stub` An option to take rownames from the input `data` table as row captions in the display table stub.

`auto_align` Optionally have column data be aligned depending on the content contained in each column of the input `data`. Internally, this calls `cols_align(align = "auto")` for all columns.

id	The table ID. By default, with NULL, this will be a random, ten-letter ID as generated by using the random_id() function. A custom table ID can be used with any single-length character vector.
locale	An optional locale ID that can be set as the default locale for all functions that take a locale argument. Examples of valid locales include "en_US" for English (United States) and "fr_FR" for French (France). Refer to the information provided by the info_locales() to determine which locales are supported.
row_group.sep	The separator to use between consecutive group names (a possibility when providing data as a grouped_df with multiple groups) in the displayed stub row group label.

Details

There are a few data ingest options we can consider at this stage. We can choose to create a table stub with rowname captions using the rowname_col argument. Further to this, stub row groups can be created with the groupname_col. Both arguments take the name of a column in the input table data. Typically, the data in the groupname_col will consist of categories of data in a table and the data in the rowname_col are unique labels (perhaps unique across the entire table or unique within groups).

Row groups can also be created by passing a grouped_df to gt() by using the dplyr::group_by() function on the table data. In this way, two or more columns of categorical data can be used to make row groups. The row_group.sep argument allows for control in how the row group label will appear in the display table.

Value

An object of class gt_tbl.

Examples

Create a **gt** table object using the exibble dataset. Use the row and group columns to add a stub and row groups via the rowname_col and groupname_col arguments.

```
tab_1 <-
  exibble %>%
  gt(
    rowname_col = "row",
    groupname_col = "group"
  )

tab_1
```

The resulting **gt** table object can be used in transformations with a variety of tab_*(), fmt_*(), cols_*(), and even more functions available in the package.

```
tab_1 %>%
  tab_header(
    title = "Table Title",
```

```
    subtitle = "Subtitle"
  ) %>%
  fmt_number(
    columns = num,
    decimals = 2
  ) %>%
  cols_label(num = "number")
```

Function ID

1-1

See Also

Other table creation functions: `gt_preview()`

`gtcars` *Deluxe automobiles from the 2014-2017 period*

Description

Expensive and fast cars. Not your father's `mtcars`. Each row describes a car of a certain make, model, year, and trim. Basic specifications such as horsepower, torque, EPA MPG ratings, type of drivetrain, and transmission characteristics are provided. The country of origin for the car manufacturer is also given.

Usage

```
gtcars
```

Format

A tibble with 47 rows and 15 variables:

mfr The name of the car manufacturer

model The car's model name

year The car's model year

trim A short description of the car model's trim

bdy_style An identifier of the car's body style, which is either `coupe`, `convertible`, `sedan`, or `hatchback`

hp, hp_rpm The car's horsepower and the associated RPM level

trq, trq_rpm The car's torque and the associated RPM level

mpg_c, mpg_h The miles per gallon fuel efficiency rating for city and highway driving

drivetrain The car's drivetrain which, for this dataset is either `rwd` (Rear Wheel Drive) or `awd` (All Wheel Drive)

trsmn The codified transmission type, where the number part is the number of gears; the car could have automatic transmission (a), manual transmission (m), an option to switch between both types (am), or, direct drive (dd)

ctry_origin The country name for where the vehicle manufacturer is headquartered

msrp Manufacturer's suggested retail price in U.S. dollars (USD)

Details

All of the gtcars have something else in common (aside from the high asking prices): they are all grand tourer vehicles. These are proper GT cars that blend pure driving thrills with a level of comfort that is more expected from a fine limousine (e.g., a Rolls-Royce Phantom EWB). You'll find that, with these cars, comfort is emphasized over all-out performance. Nevertheless, the driving experience should also mean motoring at speed, doing so in style and safety.

Examples

Here is a glimpse at the data available in gtcars.

```
dplyr::glimpse(gtcars)
#> Rows: 47
#> Columns: 15
#> $ mfr         <chr> "Ford", "Ferrari", "Ferrari", "Ferrari", "Ferrari", "Ferra~
#> $ model       <chr> "GT", "458 Speciale", "458 Spider", "458 Italia", "488 GTB~
#> $ year        <dbl> 2017, 2015, 2015, 2014, 2016, 2015, 2017, 2015, 2015, 2015~
#> $ trim        <chr> "Base Coupe", "Base Coupe", "Base", "Base Coupe", "Base Co~
#> $ bdy_style   <chr> "coupe", "coupe", "convertible", "coupe", "coupe", "conver~
#> $ hp          <dbl> 647, 597, 562, 562, 661, 553, 680, 652, 731, 949, 573, 545~
#> $ hp_rpm      <dbl> 6250, 9000, 9000, 9000, 8000, 7500, 8250, 8000, 8250, 9000~
#> $ trq         <dbl> 550, 398, 398, 398, 561, 557, 514, 504, 509, 664, 476, 436~
#> $ trq_rpm     <dbl> 5900, 6000, 6000, 6000, 3000, 4750, 5750, 6000, 6000, 6750~
#> $ mpg_c       <dbl> 11, 13, 13, 13, 15, 16, 12, 11, 11, 12, 21, 16, 11, 16, 12~
#> $ mpg_h       <dbl> 18, 17, 17, 17, 22, 23, 17, 16, 16, 16, 22, 22, 18, 20, 20~
#> $ drivetrain  <chr> "rwd", "rwd", "rwd", "rwd", "rwd", "rwd", "awd", "awd", "r~
#> $ trsmn       <chr> "7a", "7a", "7a", "7a", "7a", "7a", "7a", "7a", "7a", "7a"~
#> $ ctry_origin <chr> "United States", "Italy", "Italy", "Italy", "Italy", "Ital~
#> $ msrp        <dbl> 447000, 291744, 263553, 233509, 245400, 198973, 298000, 29~
```

Function ID

11-3

See Also

Other datasets: countrypops, exibble, pizzaplace, sp500, sza gtsave *Save a **gt** table as a file*

Description

The `gtsave()` function makes it easy to save a **gt** table to a file. The function guesses the file type by the extension provided in the output filename, producing either an HTML, PDF, PNG, LaTeX, or RTF file.

Usage

```
gtsave(data, filename, path = NULL, ...)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`filename`	The file name to create on disk. Ensure that an extension compatible with the output types is provided (`.html`, `.tex`, `.ltx`, `.rtf`, `.docx`). If a custom save function is provided then the file extension is disregarded.
`path`	An optional path to which the file should be saved (combined with filename).
`...`	All other options passed to the appropriate internal saving function.

Details

Output filenames with either the `.html` or `.htm` extensions will produce an HTML document. In this case, we can pass a `TRUE` or `FALSE` value to the `inline_css` option to obtain an HTML document with inlined CSS styles (the default is `FALSE`). More details on CSS inlining are available at `as_raw_html()`. We can pass values to arguments in `htmltools::save_html()` through the `...`. Those arguments are either `background` or `libdir`, please refer to the **htmltools** documentation for more details on the use of these arguments.

If the output filename is expressed with the `.rtf` extension then an RTF file will be generated. In this case, there is an option that can be passed through `...`: `page_numbering`. This controls RTF document page numbering and, by default, page numbering is not enabled (i.e., `page_numbering = "none"`).

We can create an image file based on the HTML version of the `gt` table. With the filename extension `.png`, we get a PNG image file. A PDF document can be generated by using the `.pdf` extension. This process is facilitated by the **webshot2** package, so, this package needs to be installed before attempting to save any table as an image file. There is the option of passing values to the underlying `webshot2::webshot()` function though `...`. Some of the more useful arguments for PNG saving are `zoom` (defaults to a scale level of `2`) and `expand` (adds whitespace pixels around the cropped table image, and has a default value of `5`), and `selector` (the default value is `"table"`). There are several more options available so have a look at the **webshot2** documentation for further details.

If the output filename extension is either of `.tex`, `.ltx`, or `.rnw`, a LaTeX document is produced. An output filename of `.rtf` will generate an RTF document. The LaTeX and RTF saving functions don't have any options to pass to `...`.

If the output filename extension is `.docx`, a Word document file is produced. This process is facilitated by the **rmarkdown** package, so this package needs to be installed before attempting to save any table as a `.docx` document.

Value

Invisibly returns `TRUE` if the export process is successful.

Examples

Use `gtcars` to create a **gt** table. Add a stubhead label with the `tab_stubhead()` function to describe what is in the stub.

```
tab_1 <-
  gtcars %>%
  dplyr::select(model, year, hp, trq) %>%
  dplyr::slice(1:5) %>%
  gt(rowname_col = "model") %>%
  tab_stubhead(label = "car")
```

Export the **gt** table to an HTML file with inlined CSS (which is necessary for including the table as part of an HTML email) using `gtsave()` and the `inline_css = TRUE` option.

```
tab_1 %>% gtsave(filename = "tab_1.html", inline_css = TRUE)
```

By leaving out the `inline_css` option, we get a more conventional HTML file with embedded CSS styles.

```
tab_1 %>% gtsave(filename = "tab_1.html")
```

Saving as a PNG file results in a cropped image of an HTML table. The amount of whitespace can be set with the `expand` option.

```
tab_1 %>% gtsave("tab_1.png", expand = 10)
```

Any use of the `.tex`, `.ltx`, or `.rnw` will result in the output of a LaTeX document.

```
tab_1 %>% gtsave("tab_1.tex")
```

With the `.rtf` extension, we'll get an RTF document.

```
tab_1 %>% gtsave("tab_1.rtf")
```

With the `.docx` extension, we'll get a word/docx document.

```
tab_1 %>% gtsave("tab_1.docx")
```

Function ID

13-1

See Also

Other table export functions: as_latex(), as_raw_html(), as_rtf(), as_word(), extract_cells(), extract_summary()

`gt_latex_dependencies` *Get the LaTeX dependencies required for a* ***gt*** *table*

Description

When working with Rnw (Sweave) files or otherwise writing LaTeX code, including a **gt** table can be problematic if we don't have knowledge of the LaTeX dependencies. For the most part, these dependencies are the LaTeX packages that are required for rendering a **gt** table. The `gt_latex_dependencies()` function provides an object that can be used to provide the LaTeX in an Rnw file, allowing **gt** tables to work and not yield errors due to missing packages.

Usage

```
gt_latex_dependencies()
```

Details

Here is an example Rnw document that shows how the `gt_latex_dependencies()` can be used in conjunction with a **gt** table:

```
%!sweave=knitr

\documentclass{article}

<<echo=FALSE>>=
library(gt)
 @

<<results='asis', echo=FALSE>>=
gt_latex_dependencies()
 @

\begin{document}

<<results='asis', echo=FALSE>>=
exibble
 @

\end{document}
```

Value

An object of class `knit_asis`.

Function ID

7-26

See Also

Other helper functions: `adjust_luminance()`, `cell_borders()`, `cell_fill()`, `cell_text()`, `cells_body()`, `cells_column_labels()`, `cells_column_spanners()`, `cells_footnotes()`, `cells_grand_summary()`, `cells_row_groups()`, `cells_source_notes()`, `cells_stub_grand_summary()`, `cells_stub_summary()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`, `stub()`

---

gt_output — *Create a **gt** display table output element for Shiny*

Description

Using `gt_output()` we can render a reactive **gt** table, a process initiated by using the `render_gt()` function in the `server` component of a Shiny app. The `gt_output()` call is to be used in the Shiny `ui` component, the position and context wherein this call is made determines the where the **gt** table is rendered on the app page. It's important to note that the ID given during the `render_gt()` call is needed as the `outputId` in `gt_output()` (e.g., **server**: `output$<id> <- render_gt(...)`; **ui**: `gt_output(outputId = "<id>")`).

Usage

```
gt_output(outputId)
```

Arguments

`outputId`	An output variable from which to read the table.

Details

We need to ensure that we have the **shiny** package installed first. This is easily by using `install.packages("shiny")`. More information on creating Shiny apps can be found at the Shiny Site.

Examples

Here is a Shiny app (contained within a single file) that (1) prepares a **gt** table, (2) sets up the `ui` with `gt_output()`, and (3) sets up the `server` with a `render_gt()` that uses the `gt_tbl` object as the input expression.

```
library(shiny)

gt_tbl <-
  gtcars %>%
  gt() %>%
  cols_hide(contains("_"))

ui <- fluidPage(

  gt_output(outputId = "table")
)

server <- function(input,
                   output,
                   session) {

  output$table <-
    render_gt(
      expr = gt_tbl,
      height = px(600),
      width = px(600)
    )
}
```

Function ID

12-2

See Also

Other Shiny functions: `render_gt()`

`gt_preview` *Preview a* **gt** *table object*

Description

Sometimes you may want to see just a small portion of your input data. We can use `gt_preview()` in place of `gt()` to get the first x rows of data and the last y rows of data (which can be set by the `top_n` and `bottom_n` arguments). It's not advised to use additional **gt** functions to further modify the output of `gt_preview()`. Furthermore, you cannot pass a **gt** object to `gt_preview()`.

Usage

```
gt_preview(data, top_n = 5, bottom_n = 1, incl_rownums = TRUE)
```

Arguments

`data`	A `data.frame` object or a tibble.
`top_n`	This value will be used as the number of rows from the top of the table to display. The default, `5`, will show the first five rows of the table.
`bottom_n`	The value will be used as the number of rows from the bottom of the table to display. The default, `1`, will show the final row of the table.
`incl_rownums`	An option to include the row numbers for `data` in the table stub. By default, this is `TRUE`.

Details

Any grouped data or magic columns such as `rowname` and `groupname` will be ignored by `gt_preview()` and, as such, one cannot add a stub or group rows in the output table. By default, the output table will include row numbers in a stub (including a range of row numbers for the omitted rows). This row numbering option can be deactivated by setting `incl_rownums` to `FALSE`.

Value

An object of class `gt_tbl`.

Examples

Use `gtcars` to create a **gt** table preview (with only a few of its columns). You'll see the first five rows and the last row.

```
gtcars %>%
  dplyr::select(mfr, model, year) %>%
  gt_preview()
```

Function ID

1-2

See Also

Other table creation functions: `gt()`

`html` *Interpret input text as HTML-formatted text*

Description

For certain pieces of text (like in column labels or table headings) we may want to express them as raw HTML. In fact, with HTML, anything goes so it can be much more than just text. The `html()` function will guard the input HTML against escaping, so, your HTML tags will come through as HTML when rendered... to HTML.

Usage

```
html(text, ...)
```

Arguments

`text, ...`	The text that is understood to be HTML text, which is to be preserved.

Value

A character object of class `html`. It's tagged as an HTML fragment that is not to be sanitized.

Examples

Use `exibble` to create a **gt** table. When adding a title, use the `html()` helper to use HTML formatting.

```
exibble %>%
  dplyr::select(currency, char) %>%
  gt() %>%
  tab_header(title = html("<em>HTML</em>"))
```

Function ID

7-2

See Also

Other helper functions: `adjust_luminance()`, `cell_borders()`, `cell_fill()`, `cell_text()`, `cells_body()`, `cells_column_labels()`, `cells_column_spanners()`, `cells_footnotes()`, `cells_grand_summary()`, `cells_row_groups()`, `cells_source_notes()`, `cells_stub_grand_summary()`, `cells_stub_summary()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `md()`, `pct()`, `px()`, `random_id()`, `stub()`

`info_currencies` *View a table with info on supported currencies*

Description

The `fmt_currency()` function lets us format numeric values as currencies. The table generated by the `info_currencies()` function provides a quick reference to all the available currencies. The currency identifiers are provided (name, 3-letter currency code, and 3-digit currency code) along with the each currency's exponent value (number of digits of the currency subunits). A formatted example is provided (based on the value of `49.95`) to demonstrate the default formatting of each currency.

Usage

```
info_currencies(type = c("code", "symbol"), begins_with = NULL)
```

Arguments

- `type` — The type of currency information provided. Can either be `code` where currency information corresponding to 3-letter currency codes is provided, or `symbol` where currency info for common currency names (e.g., dollar, pound, yen, etc.) is returned.
- `begins_with` — Providing a single letter will filter currencies to only those that begin with that letter in their currency code. The default (`NULL`) will produce a table with all currencies displayed. This option only constrains the information table where `type == "code"`.

Details

There are 172 currencies, which can lead to a verbose display table. To make this presentation more focused on retrieval, we can provide an initial letter corresponding to the 3-letter currency code to `begins_with`. This will filter currencies in the info table to just the set beginning with the supplied letter.

Value

An object of class `gt_tbl`.

Examples

Get a table of info on all of the currencies where the three-letter code begins with an `"h"`.

```
info_currencies(begins_with = "h")
```

Get a table of info on all of the common currency name/symbols that can be used with `fmt_currency()`.

```
info_currencies(type = "symbol")
```

Function ID

10-3

See Also

Other information functions: `info_date_style()`, `info_google_fonts()`, `info_locales()`, `info_paletteer()`, `info_time_style()`

info_date_style — *View a table with info on date styles*

Description

The fmt_date() function lets us format date-based values in a convenient manner using preset styles. The table generated by the info_date_style() function provides a quick reference to all styles, with associated format names and example outputs using a fixed date (2000-02-29).

Usage

```
info_date_style()
```

Value

An object of class gt_tbl.

Examples

Get a table of info on the different date-formatting styles (which are used by supplying a number code to the fmt_date() function).

```
info_date_style()
```

Function ID

10-1

See Also

Other information functions: info_currencies(), info_google_fonts(), info_locales(), info_paletteer(), info_time_style()

info_google_fonts — *View a table on recommended Google Fonts*

Description

The google_font() helper function can be used wherever a font name should be specified. There are two instances where this helper can be used: the name argument in opt_table_font() (for setting a table font) and in that of cell_text() (used with tab_style()). Because there is an overwhelming number of fonts available in the *Google Fonts* catalog, the info_google_fonts() provides a table with a set of helpful font recommendations. These fonts look great in the different parts of a **gt** table. Why? For the most part they are suitable for body text, having large counters, large x-height, reasonably low contrast, and open apertures. These font features all make for high legibility at smaller sizes.

Usage

```
info_google_fonts()
```

Value

An object of class `gt_tbl`.

Examples

Get a table of info on some of the recommended *Google Fonts* for tables.

```
info_google_fonts()
```

Function ID

10-6

See Also

Other information functions: `info_currencies()`, `info_date_style()`, `info_locales()`, `info_paletteer()`, `info_time_style()`

`info_locales` *View a table with info on supported locales*

Description

Many of the `fmt_*()` functions have a `locale` argument that makes locale-based formatting easier. The table generated by the `info_locales()` function provides a quick reference to all the available locales. The locale identifiers are provided (base locale ID, common display name) along with the each locale's group and decimal separator marks. A formatted numeric example is provided (based on the value of `11027`) to demonstrate the default formatting of each locale.

Usage

```
info_locales(begins_with = NULL)
```

Arguments

`begins_with`	Providing a single letter will filter locales to only those that begin with that letter in their base locale ID. The default (`NULL`) will produce a table with all locales displayed.

Details

There are 712 locales, which means that a very long display table is provided by default. To trim down the output table size, we can provide an initial letter corresponding to the base locale ID to `begins_with`. This will filter locales in the info table to just the set that begins with the supplied letter.

Value

An object of class gt_tbl.

Examples

Get a table of info on all of the locales where the base locale ID begins with a "v".

```
info_locales(begins_with = "v")
```

Function ID

10-4

See Also

Other information functions: info_currencies(), info_date_style(), info_google_fonts(), info_paletteer(), info_time_style()

---

info_paletteer *View a table with info on color palettes*

Description

While the data_color() function allows us to flexibly color data cells in our **gt** table, the harder part of this process is discovering and choosing color palettes that are suitable for the table output. We can make this process much easier in two ways: (1) by using the **paletteer** package, which makes a wide range of palettes from various R packages readily available, and (2) calling the info_paletteer() function to give us an information table that serves as a quick reference for all of the discrete color palettes available in **paletteer**.

Usage

```
info_paletteer(color_pkgs = NULL)
```

Arguments

color_pkgs A vector of color packages that determines which sets of palettes should be displayed in the information table. If this is NULL (the default) then all of the discrete palettes from all of the color packages represented in **paletteer** will be displayed.

Details

The palettes displayed are organized by package and by palette name. These values are required when obtaining a palette (as a vector of hexadecimal colors), from the the `paletteer::paletteer_d()` function. Once we are familiar with the names of the color palette packages (e.g., **RColorBrewer**, **ggthemes**, **wesanderson**), we can narrow down the content of this information table by supplying a vector of such package names to `color_pkgs`.

Colors from the following color packages (all supported by **paletteer**) are shown by default with `info_paletteer()`:

- **awtools**, 5 palettes
- **dichromat**, 17 palettes
- **dutchmasters**, 6 palettes
- **ggpomological**, 2 palettes
- **ggsci**, 42 palettes
- **ggthemes**, 31 palettes
- **ghibli**, 27 palettes
- **grDevices**, 1 palette
- **jcolors**, 13 palettes
- **LaCroixColoR**, 21 palettes
- **NineteenEightyR**, 12 palettes
- **nord**, 16 palettes
- **ochRe**, 16 palettes
- **palettetown**, 389 palettes
- **pals**, 8 palettes
- **Polychrome**, 7 palettes
- **quickpalette**, 17 palettes
- **rcartocolor**, 34 palettes
- **RColorBrewer**, 35 palettes
- **Redmonder**, 41 palettes
- **wesanderson**, 19 palettes
- **yarrr**, 21 palettes

Value

An object of class `gt_tbl`.

Examples

Get a table of info on just the `"ggthemes"` color palette (easily accessible from the **paletteer** package).

```
info_paletteer(color_pkgs = "ggthemes")
```

Function ID

10-5

See Also

Other information functions: `info_currencies()`, `info_date_style()`, `info_google_fonts()`, `info_locales()`, `info_time_style()`

`info_time_style` *View a table with info on time styles*

Description

The `fmt_time()` function lets us format time-based values in a convenient manner using preset styles. The table generated by the `info_time_style()` function provides a quick reference to all styles, with associated format names and example outputs using a fixed time (`14:35`).

Usage

```
info_time_style()
```

Value

An object of class `gt_tbl`.

Examples

Get a table of info on the different time-formatting styles (which are used by supplying a number code to the `fmt_time()` function).

```
info_time_style()
```

Function ID

10-2

See Also

Other information functions: `info_currencies()`, `info_date_style()`, `info_google_fonts()`, `info_locales()`, `info_paletteer()`

local_image *Helper function for adding a local image*

Description

We can flexibly add a local image (i.e., an image residing on disk) inside of a table with `local_image()` function. The function provides a convenient way to generate an HTML fragment using an on-disk PNG or SVG. Because this function is currently HTML-based, it is only useful for HTML table output. To use this function inside of data cells, it is recommended that the `text_transform()` function is used. With that function, we can specify which data cells to target and then include a `local_image()` call within the required user-defined function (for the `fn` argument). If we want to include an image in other places (e.g., in the header, within footnote text, etc.) we need to use `local_image()` within the `html()` helper function.

Usage

```
local_image(filename, height = 30)
```

Arguments

`filename`	A path to an image file.
`height`	The absolute height (px) of the image in the table cell.

Details

By itself, the function creates an HTML image tag with an image URI embedded within. We can easily experiment with a local PNG or SVG image that's available in the **gt** package using the `test_image()` function. Using that, the call `local_image(file = test_image(type = "png"))` evaluates to:
`<img src=<data URI> style=\"height:30px;\">`

where a height of `30px` is a default height chosen to work well within the heights of most table rows.

Value

A character object with an HTML fragment that can be placed inside of a cell.

Examples

Create a tibble that contains heights of an image in pixels (one column as a string, the other as numerical values), then, create a **gt** table. Use the `text_transform()` function to insert a local test image (PNG) image with the various sizes.

```
dplyr::tibble(
  pixels = px(seq(10, 35, 5)),
  image = seq(10, 35, 5)
) %>%
  gt() %>%
```

```
  text_transform(
    locations = cells_body(columns = image),
    fn = function(x) {
      local_image(
        filename = test_image(type = "png"),
        height = as.numeric(x)
      )
    }
  )
```

Function ID

8-2

See Also

Other image addition functions: ggplot_image(), test_image(), web_image()

---

md *Interpret input text as Markdown-formatted text*

Description

Markdown! It's a wonderful thing. We can use it in certain places (e.g., footnotes, source notes, the table title, etc.) and expect it to render to HTML as Markdown does. There is the html() helper that allows you to ferry in HTML but this function md()... it's almost like a two-for-one deal (you get to use Markdown plus any HTML fragments *at the same time*).

Usage

```
md(text)
```

Arguments

text	The text that is understood to contain Markdown formatting.

Value

A character object of class from_markdown. It's tagged as being Markdown text and it will undergo conversion to HTML.

Examples

Use exibble to create a **gt** table. When adding a title, use the md() helper to use Markdown formatting.

```
exibble %>%
  dplyr::select(currency, char) %>%
  gt() %>%
  tab_header(title = md("Using *Markdown*"))
```

Function ID

7-1

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), pct(), px(), random_id(), stub()

---

opt_align_table_header
Option to align the table header

Description

By default, a table header added to a **gt** table has center alignment for both the title and the subtitle elements. This function allows us to easily set the horizontal alignment of the title and subtitle to the left or right by using the "align" argument. This function serves as a convenient shortcut for <gt_tbl> %>% tab_options(heading.align = <align>).

Usage

```
opt_align_table_header(data, align = c("left", "center", "right"))
```

Arguments

data	A table object that is created using the gt() function.
align	The alignment of the title and subtitle elements in the table header. Options are "left" (the default), "center", or "right".

Value

An object of class gt_tbl.

Examples

Use exibble to create a **gt** table with a number of table parts added. The header (consisting of the title and the subtitle) are to be aligned to the left with the opt_align_table_header() function.

```
exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = c(num, currency),
```

```
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = currency,
    fns = list(
      total = ~sum(., na.rm = TRUE)
    )) %>%
  tab_source_note(source_note = "This is a source note.") %>%
  tab_footnote(
    footnote = "This is a footnote.",
    locations = cells_body(columns = 1, rows = 1)
  ) %>%
  tab_header(
    title = "The title of the table",
    subtitle = "The table's subtitle"
  ) %>%
  opt_align_table_header(align = "left")
```

Function ID

9-3

See Also

Other table option functions: opt_all_caps(), opt_css(), opt_footnote_marks(), opt_horizontal_padding(), opt_row_striping(), opt_stylize(), opt_table_font(), opt_table_lines(), opt_table_outline(), opt_vertical_padding()

---

opt_all_caps   *Option to use all caps in select table locations*

Description

Sometimes an all-capitalized look is suitable for a table. With the opt_all_caps() function, we can transform characters in the column labels, the stub, and in all row groups in this way (and there's control over which of these locations are transformed).

This function serves as a convenient shortcut for <gt_tbl> %>% tab_options(<location>.text_transform = "uppercas (for all locations selected).

Usage

```
opt_all_caps(
  data,
  all_caps = TRUE,
  locations = c("column_labels", "stub", "row_group")
)
```

Arguments

data	A table object that is created using the gt() function.
all_caps	A logical value to indicate whether the text transformation to all caps should be performed (TRUE, the default) or reset to default values (FALSE) for the locations targeted.
locations	Which locations should undergo this text transformation? By default it includes all of the "column_labels", the "stub", and the "row_group" locations. However, we could just choose one or two of those.

Value

An object of class gt_tbl.

Examples

Use exibble to create a **gt** table with a number of table parts added. All text in the column labels, the stub, and in all row groups is to be transformed to all caps using opt_all_caps().

```
exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = c(num, currency),
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = currency,
    fns = list(
      total = ~sum(., na.rm = TRUE)
    )) %>%
  tab_source_note(source_note = "This is a source note.") %>%
  tab_footnote(
    footnote = "This is a footnote.",
    locations = cells_body(columns = 1, rows = 1)
  ) %>%
  tab_header(
    title = "The title of the table",
    subtitle = "The table's subtitle"
  ) %>%
  opt_all_caps()
```

Function ID

9-6

See Also

Other table option functions: opt_align_table_header(), opt_css(), opt_footnote_marks(), opt_horizontal_padding(), opt_row_striping(), opt_stylize(), opt_table_font(), opt_table_lines(), opt_table_outline(), opt_vertical_padding()

---

opt_css *Option to add custom CSS for the table*

Description

The opt_css() function makes it possible to add CSS to a **gt** table. This CSS will be added after the compiled CSS that **gt** generates automatically when the object is transformed to an HTML output table. You can supply css as a vector of lines or as a single string.

Usage

```
opt_css(data, css, add = TRUE, allow_duplicates = FALSE)
```

Arguments

data	A table object that is created using the gt() function.
css	The CSS to include as part of the rendered table's <style> element.
add	If TRUE, the default, the CSS is added to any already-defined CSS (typically from previous calls of opt_table_font(), opt_css(), or, directly setting CSS the table.additional_css value in tab_options()). If this is set to FALSE, the CSS provided here will replace any previously-stored CSS.
allow_duplicates	When this is FALSE (the default), the CSS provided here won't be added (provided that add = TRUE) if it is seen in the already-defined CSS.

Value

An object of class gt_tbl.

Examples

Use exibble to create a **gt** table and format the data in both columns. With opt_css(), insert CSS rulesets as as string and be sure to set the table ID explicitly (here as "one").

```
exibble %>%
  dplyr::select(num, currency) %>%
  gt(id = "one") %>%
  fmt_currency(
    columns = currency,
    currency = "HKD"
  ) %>%
```

```
  fmt_scientific(
    columns = num
  ) %>%
  opt_css(
    css = "
    #one .gt_table {
      background-color: skyblue;
    }
    #one .gt_row {
      padding: 20px 30px;
    }
    #one .gt_col_heading {
      text-align: center !important;
    }
    "
  )
```

Function ID

9-11

See Also

Other table option functions: opt_align_table_header(), opt_all_caps(), opt_footnote_marks(), opt_horizontal_padding(), opt_row_striping(), opt_stylize(), opt_table_font(), opt_table_lines(), opt_table_outline(), opt_vertical_padding()

---

opt_footnote_marks   *Option to modify the set of footnote marks*

Description

Alter the footnote marks for any footnotes that may be present in the table. Either a vector of marks can be provided (including Unicode characters), or, a specific keyword could be used to signify a preset sequence. This function serves as a shortcut for using `tab_options(footnotes.marks = {marks})`

Usage

```
opt_footnote_marks(data, marks)
```

Arguments

data	A table object that is created using the gt() function.

marks Either a character vector of length greater than 1 (that will represent the series of marks) or a single keyword that represents a preset sequence of marks. The valid keywords are: `"numbers"` (for numeric marks), `"letters"` and `"LETTERS"` (for lowercase and uppercase alphabetic marks), `"standard"` (for a traditional set of four symbol marks), and `"extended"` (which adds two more symbols to the standard set).

Details

We can supply a vector of that will represent the series of marks. The series of footnote marks is recycled when its usage goes beyond the length of the set. At each cycle, the marks are simply doubled, tripled, and so on (e.g., `*` -> `**` -> `***`). The option exists for providing keywords for certain types of footnote marks. The keywords are:

- `"numbers"`: numeric marks, they begin from 1 and these marks are not subject to recycling behavior
- `"letters"`: miniscule alphabetic marks, internally uses the `letters` vector which contains 26 lowercase letters of the Roman alphabet
- `"LETTERS"`: majuscule alphabetic marks, using the `LETTERS` vector which has 26 uppercase letters of the Roman alphabet
- `"standard"`: symbolic marks, four symbols in total
- `"extended"`: symbolic marks, extends the standard set by adding two more symbols, making six

Value

An object of class `gt_tbl`.

Examples

Use `sza` to create a **gt** table, adding three footnotes. Call `opt_footnote_marks()` to specify which footnote marks to use.

```
sza %>%
  dplyr::filter(latitude == 30) %>%
  dplyr::group_by(tst) %>%
  dplyr::summarize(
    SZA.Max = if (
      all(is.na(sza))) {
      NA
    } else {
      max(sza, na.rm = TRUE)
    },
    SZA.Min = if (
      all(is.na(sza))) {
      NA
    } else {
      min(sza, na.rm = TRUE)
```

```
    },
    .groups = "drop"
  ) %>%
  gt(rowname_col = "tst") %>%
  tab_spanner_delim(delim = ".") %>%
  sub_missing(
    columns = everything(),
    missing_text = "90+"
  ) %>%
  tab_stubhead(label = "TST") %>%
  tab_footnote(
    footnote = "True solar time.",
    locations = cells_stubhead()
  ) %>%
  tab_footnote(
    footnote = "Solar zenith angle.",
    locations = cells_column_spanners(
      spanners = "spanner-SZA.Max"
    )
  ) %>%
  tab_footnote(
    footnote = "The Lowest SZA.",
    locations = cells_stub(rows = "1200")
  ) %>%
  opt_footnote_marks(marks = "standard")
```

Function ID

9-1

See Also

Other table option functions: `opt_align_table_header()`, `opt_all_caps()`, `opt_css()`, `opt_horizontal_padding()`, `opt_row_striping()`, `opt_stylize()`, `opt_table_font()`, `opt_table_lines()`, `opt_table_outline()`, `opt_vertical_padding()`

`opt_horizontal_padding` *Option to expand or contract horizontal padding*

Description

Increase or decrease the horizontal padding throughout all locations of a **gt** table by use of a `scale` factor, which here is defined by a real number between `0` and `3`. This function serves as a shortcut for setting the following eight options in `tab_options()`:

- `heading.padding.horizontal`

- column_labels.padding.horizontal
- data_row.padding.horizontal
- row_group.padding.horizontal
- summary_row.padding.horizontal
- grand_summary_row.padding.horizontal
- footnotes.padding.horizontal
- source_notes.padding.horizontal

Usage

```
opt_horizontal_padding(data, scale = 1)
```

Arguments

data	A table object that is created using the gt() function.
scale	A scale factor by which the horizontal padding will be adjusted. Must be a number between 0 and 3.

Value

An object of class gt_tbl.

Examples

Use exibble to create a **gt** table with a number of table parts added. Expand the horizontal padding across the entire table with opt_horizontal_padding().

```
exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = c(num, currency),
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = currency,
    fns = list(
      total = ~sum(., na.rm = TRUE)
    )) %>%
  tab_source_note(source_note = "This is a source note.") %>%
  tab_footnote(
    footnote = "This is a footnote.",
    locations = cells_body(columns = 1, rows = 1)
  ) %>%
  tab_header(
```

```
    title = "The title of the table",
    subtitle = "The table's subtitle"
  ) %>%
  opt_horizontal_padding(scale = 3)
```

Function ID

9-5

See Also

Other table option functions: opt_align_table_header(), opt_all_caps(), opt_css(), opt_footnote_marks(), opt_row_striping(), opt_stylize(), opt_table_font(), opt_table_lines(), opt_table_outline(), opt_vertical_padding()

---

opt_row_striping *Option to add or remove row striping*

Description

By default, a **gt** table does not have row striping enabled. However, this function allows us to easily enable or disable striped rows in the table body. This function serves as a convenient shortcut for `<gt_tbl> %>% tab_options(row.striping.include_table_body = TRUE|FALSE)`.

Usage

```
opt_row_striping(data, row_striping = TRUE)
```

Arguments

data	A table object that is created using the gt() function.
row_striping	A logical value to indicate whether row striping should be added or removed.

Value

An object of class gt_tbl.

Examples

Use exibble to create a **gt** table with a number of table parts added. Next, we add row striping to every second row with the opt_row_striping() function.

```
exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = c(num, currency),
    fns = list(
```

```
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = currency,
    fns = list(
      total = ~sum(., na.rm = TRUE)
    )) %>%
  tab_source_note(source_note = "This is a source note.") %>%
  tab_footnote(
    footnote = "This is a footnote.",
    locations = cells_body(columns = 1, rows = 1)
  ) %>%
  tab_header(
    title = "The title of the table",
    subtitle = "The table's subtitle"
  ) %>%
  opt_row_striping()
```

Function ID

9-2

See Also

Other table option functions: opt_align_table_header(), opt_all_caps(), opt_css(), opt_footnote_marks(), opt_horizontal_padding(), opt_stylize(), opt_table_font(), opt_table_lines(), opt_table_outline(), opt_vertical_padding()

---

opt_stylize   *Stylize your table with a colorful look*

Description

With opt_stylize() you can quickly style your **gt** table with a carefully curated set of background colors, line colors, and line styles. There are six styles to choose from and they largely vary in the extent of coloring applied to different table locations. Some have table borders applied, some apply darker colors to the table stub and summary sections, and, some even have vertical lines. In addition to choosing a style preset, there are six color variations that each use a range of five color tints. Each of the color tints have been fine-tuned to maximize the contrast between text and its background. There are 36 combinations of style and color to choose from.

Usage

```
opt_stylize(data, style = 1, color = "blue", add_row_striping = TRUE)
```

Arguments

- `data` — A table object that is created using the `gt()` function.
- `style` — Six numbered styles are available. Simply provide a number from `1` (the default) to `6` to choose a distinct look.
- `color` — There are six color variations: `"blue"` (the default), `"cyan"`, `"pink"`, `"green"`, `"red"`, and `"gray"`.
- `add_row_striping` — An option to enable row striping in the table body for the style chosen. By default, this is `TRUE`.

Value

an object of class `gt_tbl`.

Examples

Use `exibble` to create a **gt** table with a number of table parts added. Then, use the `opt_stylize()` function to give the table some additional style (using the `"cyan"` color variation and style number `6`).

```
exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = c(num, currency),
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = currency,
    fns = list(
      total = ~sum(., na.rm = TRUE)
    )) %>%
  tab_source_note(source_note = "This is a source note.") %>%
  tab_footnote(
    footnote = "This is a footnote.",
    locations = cells_body(columns = 1, rows = 1)
  ) %>%
  tab_header(
    title = "The title of the table",
    subtitle = "The table's subtitle"
  ) %>%
  opt_stylize(style = 6, color = "cyan")
```

Function ID

9-10

See Also

Other table option functions: `opt_align_table_header()`, `opt_all_caps()`, `opt_css()`, `opt_footnote_marks()`, `opt_horizontal_padding()`, `opt_row_striping()`, `opt_table_font()`, `opt_table_lines()`, `opt_table_outline()`, `opt_vertical_padding()`

`opt_table_font` *Option to define a custom font for the table*

Description

The `opt_table_font()` function makes it possible to define a custom font for the entire **gt** table. The standard fallback fonts are still set by default but the font defined here will take precedence. You could still have different fonts in select locations in the table, and for that you would need to use `tab_style()` in conjunction with the `cell_text()` helper function.

Usage

```
opt_table_font(data, font, weight = NULL, style = NULL, add = TRUE)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`font`	Either the name of a font available in the user system or a call to `google_font()`, which has a large selection of typefaces.
`weight`	The weight of the font. Can be a text-based keyword such as `"normal"`, `"bold"`, `"lighter"`, `"bolder"`, or, a numeric value between `1` and `1000`, inclusive. Note that only variable fonts may support the numeric mapping of weight.
`style`	The text style. Can be one of either `"normal"`, `"italic"`, or `"oblique"`.
`add`	Should this font be added to the front of the already-defined fonts for the table? By default, this is `TRUE` and is recommended since the list serves as fallbacks when certain fonts are not available.

Details

We have the option to supply either a system font for the `font_name`, or, a font available at the Google Fonts service by use of the `google_font()` helper function.

Value

An object of class `gt_tbl`.

Examples

Use sp500 to create a small **gt** table, using fmt_currency() to provide a dollar sign for the first row of monetary values. Then, set a larger font size for the table and use the "Merriweather" font (from *Google Fonts*, via google_font()) with two font fallbacks ("Cochin" and the catchall "Serif" group).

```
sp500 %>%
  dplyr::slice(1:10) %>%
  dplyr::select(-volume, -adj_close) %>%
  gt() %>%
  fmt_currency(
    columns = 2:5,
    rows = 1,
    currency = "USD",
    use_seps = FALSE
  ) %>%
  tab_options(table.font.size = px(18)) %>%
  opt_table_font(
    font = list(
      google_font(name = "Merriweather"),
      "Cochin", "Serif"
    )
  )
```

Use sza to create an eleven-row table. Within opt_table_font(), set up a preferred list of sans-serif fonts that are commonly available in macOS (using part of the default_fonts() vector as a fallback).

```
sza %>%
  dplyr::filter(
    latitude == 20 &
      month == "jan" &
      !is.na(sza)
  ) %>%
  dplyr::select(-latitude, -month) %>%
  gt() %>%
  opt_table_font(
    font = c(
      "Helvetica Neue", "Segoe UI",
      default_fonts()[-c(1:3)]
    )
  ) %>%
  opt_all_caps()
```

Function ID

9-9

See Also

Other table option functions: opt_align_table_header(), opt_all_caps(), opt_css(), opt_footnote_marks(), opt_horizontal_padding(), opt_row_striping(), opt_stylize(), opt_table_lines(), opt_table_outline(), opt_vertical_padding()

---

opt_table_lines *Option to set table lines to different extents*

Description

The opt_table_lines() function sets table lines in one of three possible ways: (1) all possible table lines drawn ("all"), (2) no table lines at all ("none"), and (3) resetting to the default line styles ("default"). This is great if you want to start off with lots of lines and subtract just a few of them with tab_options() or tab_style(). Or, use it to start with a completely lineless table, adding individual lines as needed.

Usage

```
opt_table_lines(data, extent = c("all", "none", "default"))
```

Arguments

data	A table object that is created using the gt() function.
extent	The extent to which lines will be visible in the table. Options are "all", "none", or "default".

Value

An object of class gt_tbl.

Examples

Use exibble to create a **gt** table with a number of table parts added. Then, use the opt_table_lines() function to haves lines everywhere there can possibly be lines.

```
exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = c(num, currency),
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = currency,
    fns = list(
```

```
      total = ~sum(., na.rm = TRUE)
    )) %>%
  tab_source_note(source_note = "This is a source note.") %>%
  tab_footnote(
    footnote = "This is a footnote.",
    locations = cells_body(columns = 1, rows = 1)
  ) %>%
  tab_header(
    title = "The title of the table",
    subtitle = "The table's subtitle"
  ) %>%
  opt_table_lines()
```

Function ID

9-7

See Also

Other table option functions: opt_align_table_header(), opt_all_caps(), opt_css(), opt_footnote_marks(), opt_horizontal_padding(), opt_row_striping(), opt_stylize(), opt_table_font(), opt_table_outline(), opt_vertical_padding()

---

opt_table_outline   *Option to wrap an outline around the entire table*

Description

This function puts an outline of consistent style, width, and color around the entire table. It'll write over any existing outside lines so long as the width is larger that of the existing lines. The default value of style ("solid") will draw a solid outline, whereas a value of "none" will remove any present outline.

Usage

```
opt_table_outline(data, style = "solid", width = px(3), color = "#D3D3D3")
```

Arguments

data
A table object that is created using the gt() function.

style, width, color
The style, width, and color properties for the table outline. By default, these are "solid", px(3) (or, "3px"), and "#D3D3D3". If "none" is used then the outline is removed and any values provided for width and color will be ignored (i.e., not set).

Value

An object of class gt_tbl.

Examples

Use exibble to create a **gt** table with a number of table parts added. Have an outline wrap around the entire table by using opt_table_outline().

```
tab_1 <-
  exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = c(num, currency),
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = currency,
    fns = list(
      total = ~sum(., na.rm = TRUE)
    )) %>%
  tab_source_note(source_note = "This is a source note.") %>%
  tab_footnote(
    footnote = "This is a footnote.",
    locations = cells_body(columns = 1, rows = 1)
  ) %>%
  tab_header(
    title = "The title of the table",
    subtitle = "The table's subtitle"
  ) %>%
  opt_table_outline()

tab_1
```

Remove the table outline with the style = "none" option.

```
tab_1 %>% opt_table_outline(style = "none")
```

Function ID

9-8

See Also

Other table option functions: opt_align_table_header(), opt_all_caps(), opt_css(), opt_footnote_marks(), opt_horizontal_padding(), opt_row_striping(), opt_stylize(), opt_table_font(), opt_table_lines(), opt_vertical_padding()

opt_vertical_padding *Option to expand or contract vertical padding*

Description

Increase or decrease the vertical padding throughout all locations of a **gt** table by use of a `scale` factor, which here is defined by a real number between `0` and `3`. This function serves as a shortcut for setting the following eight options in `tab_options()`:

- `heading.padding`
- `column_labels.padding`
- `data_row.padding`
- `row_group.padding`
- `summary_row.padding`
- `grand_summary_row.padding`
- `footnotes.padding`
- `source_notes.padding`

Usage

```
opt_vertical_padding(data, scale = 1)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`scale`	A scale factor by which the vertical padding will be adjusted. Must be a number between `0` and `3`.

Value

An object of class `gt_tbl`.

Examples

Use `exibble` to create a **gt** table with a number of table parts added. Contract the vertical padding across the entire table with `opt_vertical_padding()`.

```
exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = c(num, currency),
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
```

```
    )) %>%
  grand_summary_rows(
    columns = currency,
    fns = list(
      total = ~sum(., na.rm = TRUE)
    )) %>%
  tab_source_note(source_note = "This is a source note.") %>%
  tab_footnote(
    footnote = "This is a footnote.",
    locations = cells_body(columns = 1, rows = 1)
  ) %>%
  tab_header(
    title = "The title of the table",
    subtitle = "The table's subtitle"
  ) %>%
  opt_vertical_padding(scale = 0.25)
```

Function ID

9-4

See Also

Other table option functions: opt_align_table_header(), opt_all_caps(), opt_css(), opt_footnote_marks(), opt_horizontal_padding(), opt_row_striping(), opt_stylize(), opt_table_font(), opt_table_lines(), opt_table_outline()

---

pct *Helper for providing a numeric value as percentage*

Description

A percentage value acts as a length value that is relative to an initial state. For instance an 80 percent value for something will size the target to 80 percent the size of its 'previous' value. This type of sizing is useful for sizing up or down a length value with an intuitive measure. This helper function can be used for the setting of font sizes (e.g., in cell_text()) and altering the thicknesses of lines (e.g., in cell_borders()). Should a more exact definition of size be required, the analogous helper function pct() will be more useful.

Usage

```
pct(x)
```

Arguments

x	the numeric value to format as a string percentage for some tab_options() arguments that can take percentage values (e.g., table.width).

Value

A character vector with a single value in percentage units.

Examples

Use exibble to create a **gt** table. Use the pct() helper to define the font size for the column labels.

```
exibble %>%
  gt() %>%
  tab_style(
    style = cell_text(size = pct(75)),
    locations = cells_column_labels()
  )
```

Function ID

7-4

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), px(), random_id(), stub()

---

pizzaplace *A year of pizza sales from a pizza place*

Description

A synthetic dataset that describes pizza sales for a pizza place somewhere in the US. While the contents are artificial, the ingredients used to make the pizzas are far from it. There are 32 different pizzas that fall into 4 different categories: classic (classic pizzas: 'You probably had one like it before, but never like this!'), chicken (pizzas with chicken as a major ingredient: 'Try the Southwest Chicken Pizza! You'll love it!'), supreme (pizzas that try a little harder: 'My Soppressata pizza uses only the finest salami from my personal salumist!'), and, veggie (pizzas without any meats whatsoever: 'My Five Cheese pizza has so many cheeses, I can only offer it in Large Size!').

Usage

```
pizzaplace
```

Format

A tibble with 49574 rows and 7 variables:

id The ID for the order, which consists of one or more pizzas at a give `date` and `time`

date A character representation of the `order` date, expressed in the ISO 8601 date format (YYYY-MM-DD)

time A character representation of the `order` time, expressed as a 24-hour time the ISO 8601 extended time format (hh:mm:ss)

name The short name for the pizza

size The size of the pizza, which can either be `S`, `M`, `L`, `XL` (rare!), or `XXL` (even rarer!); most pizzas are available in the `S`, `M`, and `L` sizes but exceptions apply

type The category or type of pizza, which can either be `classic`, `chicken`, `supreme`, or `veggie`

price The price of the pizza and the amount that it sold for (in USD)

Details

Each pizza in the dataset is identified by a short `name`. The following listings provide the full names of each pizza and their main ingredients.

Classic Pizzas:

- `classic_dlx`: The Classic Deluxe Pizza (Pepperoni, Mushrooms, Red Onions, Red Peppers, Bacon)
- `big_meat`: The Big Meat Pizza (Bacon, Pepperoni, Italian Sausage, Chorizo Sausage)
- `pepperoni`: The Pepperoni Pizza (Mozzarella Cheese, Pepperoni)
- `hawaiian`: The Hawaiian Pizza (Sliced Ham, Pineapple, Mozzarella Cheese)
- `pep_msh_pep`: The Pepperoni, Mushroom, and Peppers Pizza (Pepperoni, Mushrooms, and Green Peppers)
- `ital_cpcllo`: The Italian Capocollo Pizza (Capocollo, Red Peppers, Tomatoes, Goat Cheese, Garlic, Oregano)
- `napolitana`: The Napolitana Pizza (Tomatoes, Anchovies, Green Olives, Red Onions, Garlic)
- `the_greek`: The Greek Pizza (Kalamata Olives, Feta Cheese, Tomatoes, Garlic, Beef Chuck Roast, Red Onions)

Chicken Pizzas:

- `thai_ckn`: The Thai Chicken Pizza (Chicken, Pineapple, Tomatoes, Red Peppers, Thai Sweet Chilli Sauce)
- `bbq_ckn`: The Barbecue Chicken Pizza (Barbecued Chicken, Red Peppers, Green Peppers, Tomatoes, Red Onions, Barbecue Sauce)
- `southw_ckn`: The Southwest Chicken Pizza (Chicken, Tomatoes, Red Peppers, Red Onions, Jalapeno Peppers, Corn, Cilantro, Chipotle Sauce)
- `cali_ckn`: The California Chicken Pizza (Chicken, Artichoke, Spinach, Garlic, Jalapeno Peppers, Fontina Cheese, Gouda Cheese)

- `ckn_pesto`: The Chicken Pesto Pizza (Chicken, Tomatoes, Red Peppers, Spinach, Garlic, Pesto Sauce)
- `ckn_alfredo`: The Chicken Alfredo Pizza (Chicken, Red Onions, Red Peppers, Mushrooms, Asiago Cheese, Alfredo Sauce)

Supreme Pizzas:

- `brie_carre`: The Brie Carre Pizza (Brie Carre Cheese, Prosciutto, Caramelized Onions, Pears, Thyme, Garlic)
- `calabrese`: The Calabrese Pizza ('Nduja Salami, Pancetta, Tomatoes, Red Onions, Friggitello Peppers, Garlic)
- `soppressata`: The Soppressata Pizza (Soppressata Salami, Fontina Cheese, Mozzarella Cheese, Mushrooms, Garlic)
- `sicilian`: The Sicilian Pizza (Coarse Sicilian Salami, Tomatoes, Green Olives, Luganega Sausage, Onions, Garlic)
- `ital_supr`: The Italian Supreme Pizza (Calabrese Salami, Capocollo, Tomatoes, Red Onions, Green Olives, Garlic)
- `peppr_salami`: The Pepper Salami Pizza (Genoa Salami, Capocollo, Pepperoni, Tomatoes, Asiago Cheese, Garlic)
- `prsc_argla`: The Prosciutto and Arugula Pizza (Prosciutto di San Daniele, Arugula, Mozzarella Cheese)
- `spinach_supr`: The Spinach Supreme Pizza (Spinach, Red Onions, Pepperoni, Tomatoes, Artichokes, Kalamata Olives, Garlic, Asiago Cheese)
- `spicy_ital`: The Spicy Italian Pizza (Capocollo, Tomatoes, Goat Cheese, Artichokes, Peperoncini verdi, Garlic)

Vegetable Pizzas

- `mexicana`: The Mexicana Pizza (Tomatoes, Red Peppers, Jalapeno Peppers, Red Onions, Cilantro, Corn, Chipotle Sauce, Garlic)
- `four_cheese`: The Four Cheese Pizza (Ricotta Cheese, Gorgonzola Piccante Cheese, Mozzarella Cheese, Parmigiano Reggiano Cheese, Garlic)
- `five_cheese`: The Five Cheese Pizza (Mozzarella Cheese, Provolone Cheese, Smoked Gouda Cheese, Romano Cheese, Blue Cheese, Garlic)
- `spin_pesto`: The Spinach Pesto Pizza (Spinach, Artichokes, Tomatoes, Sun-dried Tomatoes, Garlic, Pesto Sauce)
- `veggie_veg`: The Vegetables + Vegetables Pizza (Mushrooms, Tomatoes, Red Peppers, Green Peppers, Red Onions, Zucchini, Spinach, Garlic)
- `green_garden`: The Green Garden Pizza (Spinach, Mushrooms, Tomatoes, Green Olives, Feta Cheese)
- `mediterraneo`: The Mediterranean Pizza (Spinach, Artichokes, Kalamata Olives, Sun-dried Tomatoes, Feta Cheese, Plum Tomatoes, Red Onions)
- `spinach_fet`: The Spinach and Feta Pizza (Spinach, Mushrooms, Red Onions, Feta Cheese, Garlic)
- `ital_veggie`: The Italian Vegetables Pizza (Eggplant, Artichokes, Tomatoes, Zucchini, Red Peppers, Garlic, Pesto Sauce)

Examples

Here is a glimpse at the pizza data available in pizzaplace.

```
dplyr::glimpse(pizzaplace)
#> Rows: 49,574
#> Columns: 7
#> $ id    <chr> "2015-000001", "2015-000002", "2015-000002", "2015-000002", "201~
#> $ date  <chr> "2015-01-01", "2015-01-01", "2015-01-01", "2015-01-01", "2015-01~
#> $ time  <chr> "11:38:36", "11:57:40", "11:57:40", "11:57:40", "11:57:40", "11:~
#> $ name  <chr> "hawaiian", "classic_dlx", "mexicana", "thai_ckn", "five_cheese"~
#> $ size  <chr> "M", "M", "M", "L", "L", "L", "L", "M", "M", "M", "S", "S", "S",~
#> $ type  <chr> "classic", "classic", "veggie", "chicken", "veggie", "supreme", ~
#> $ price <dbl> 13.25, 16.00, 16.00, 20.75, 18.50, 20.75, 20.75, 16.50, 16.50, 1~
```

Function ID

11-5

See Also

Other datasets: countrypops, exibble, gtcars, sp500, sza

---

px *Helper for providing a numeric value as pixels value*

Description

For certain parameters, a length value is required. Examples include the setting of font sizes (e.g., in cell_text()) and thicknesses of lines (e.g., in cell_borders()). Setting a length in pixels with px() allows for an absolute definition of size as opposed to the analogous helper function pct().

Usage

```
px(x)
```

Arguments

x	the numeric value to format as a string (e.g., "12px") for some tab_options() arguments that can take values as units of pixels (e.g., table.font.size).

Value

A character vector with a single value in pixel units.

Examples

Use exibble to create a **gt** table. Use the px() helper to define the font size for the column labels.

```
exibble %>%
  gt() %>%
  tab_style(
    style = cell_text(size = px(20)),
    locations = cells_column_labels()
  )
```

Function ID

7-3

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), random_id(), stub()

---

random_id — *Helper for creating a random* id *for a* **gt** *table*

Description

This helper function can be used to create a random, character-based ID value argument of variable length (the default is 10 letters).

Usage

```
random_id(n = 10)
```

Arguments

n	The number of lowercase letters to use for the random ID.

Value

A character vector containing a single, random ID.

Function ID

7-24

See Also

Other helper functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_footnotes(), cells_grand_summary(), cells_row_groups(), cells_source_notes(), cells_stub_grand_summary(), cells_stub_summary(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), stub()

---

render_gt — *A **gt** display table render function for use in Shiny*

Description

With render_gt() we can create a reactive **gt** table that works wonderfully once assigned to an output slot (with gt_output()). This function is to be used within Shiny's server() component. We have some options for controlling the size of the container holding the **gt** table. The width and height arguments allow for sizing the container, and the align argument allows us to align the table within the container (some other fine-grained options for positioning are available in the tab_options() function).

Usage

```
render_gt(
  expr,
  width = NULL,
  height = NULL,
  align = NULL,
  env = parent.frame(),
  quoted = FALSE,
  outputArgs = list()
)
```

Arguments

expr	An expression that creates a **gt** table object. For sake of convenience, a data frame or tibble can be used here (it will be automatically introduced to gt() with its default options).
width, height	The width and height of the table's container. Either can be specified as a single-length character with units of pixels or as a percentage. If provided as a single-length numeric vector, it is assumed that the value is given in units of pixels. The px() and pct() helper functions can also be used to pass in numeric values and obtain values as pixel or percent units.
align	The alignment of the table in its container. By default, this is "center". Other options are "left" and "right".
env	The environment in which to evaluate the expr.
quoted	Is expr a quoted expression (with quote())? This is useful if you want to save an expression in a variable.

outputArgs — A list of arguments to be passed through to the implicit call to `gt_output()` when `render_gt` is used in an interactive R Markdown document.

Details

We need to ensure that we have the **shiny** package installed first. This is easily by using `install.packages("shiny")`. More information on creating Shiny apps can be found at the Shiny Site.

Examples

Here is a Shiny app (contained within a single file) that (1) prepares a **gt** table, (2) sets up the `ui` with `gt_output()`, and (3) sets up the `server` with a `render_gt()` that uses the `gt_tbl` object as the input expression.

```
library(shiny)

gt_tbl <-
  gtcars %>%
  gt() %>%
  cols_hide(contains("_"))

ui <- fluidPage(

  gt_output(outputId = "table")
)

server <- function(input,
                   output,
                   session) {

  output$table <-
    render_gt(
      expr = gt_tbl,
      height = px(600),
      width = px(600)
    )
}
```

Function ID

12-1

See Also

Other Shiny functions: `gt_output()`

rm_caption *Remove the table caption*

Description

We can easily remove the caption text from a **gt** table with `rm_caption()`. The caption may exist if it were set through the `gt()` `caption` argument or via `tab_caption()`.

This function for removal is useful if you have received a **gt** table (perhaps through an API that returns **gt** objects) but would prefer that the table not have a caption at all. This function is safe to use even if there is no table caption set in the input `gt_tbl` object.

Usage

```
rm_caption(data)
```

Arguments

`data`	A table object of class `gt_tbl`.

Value

An object of class `gt_tbl`.

Examples

Use `gtcars` to create a **gt** table. Add a header part with the `tab_header()` function, and, add a caption as well with `tab_caption()`.

```
gt_tbl <-
  gtcars %>%
  dplyr::select(mfr, model, msrp) %>%
  dplyr::slice(1:5) %>%
  gt() %>%
  tab_header(
    title = md("Data listing from **gtcars**"),
    subtitle = md("`gtcars` is an R dataset")
  ) %>%
  tab_caption(caption = md("**gt** table example."))

gt_tbl
```

If you decide that you don't want the caption in the `gt_tbl` object, it can be removed with the `rm_caption()` function.

```
rm_caption(data = gt_tbl)
```

Function ID

6-6

See Also

Other part removal functions: rm_footnotes(), rm_header(), rm_source_notes(), rm_spanners(), rm_stubhead()

---

rm_footnotes — *Remove table footnotes*

Description

If you have one or more footnotes that ought to be removed, the rm_footnotes() function allows for such a selective removal. The table footer is an optional table part that is positioned below the table body, containing areas for both the footnotes and source notes.

This function for removal is useful if you have received a **gt** table (perhaps through an API that returns **gt** objects) but would prefer that some or all of the footnotes be removed. This function is safe to use even if there are no footnotes in the input gt_tbl object so long as select helpers (such as the default everything()) are used instead of explicit integer values.

Usage

```
rm_footnotes(data, footnotes = everything())
```

Arguments

data	A table object of class gt_tbl.
footnotes	A specification of which footnotes should be removed. The footnotes to be removed can be given as a vector of integer values (they are stored as integer positions, in order of creation, starting at 1). A select helper can also be used and, by default, this is everything() (whereby all footnotes will be removed).

Value

An object of class gt_tbl.

Examples

Use sza to create a **gt** table. Color the sza column using the data_color() function, then, use tab_footnote() twice to add two footnotes (each one targeting a different column label).

```
gt_tbl <-
  sza %>%
  dplyr::filter(
    latitude == 20 &
      month == "jan" &
```

```
    !is.na(sza)
  ) %>%
  dplyr::select(-latitude, -month) %>%
  gt() %>%
  data_color(
    columns = sza,
    colors = scales::col_numeric(
      palette = c("white", "yellow", "navyblue"),
      domain = c(0, 90)
    )
  ) %>%
  tab_footnote(
    footnote = "Color indicates height of sun.",
    locations = cells_column_labels(
      columns = sza
    )
  ) %>%
  tab_footnote(
    footnote = "
    The true solar time at the given latitude
    and date (first of month) for which the
    solar zenith angle is calculated.
    ",
    locations = cells_column_labels(
      columns = tst
    )
  ) %>%
  cols_width(everything() ~ px(150))
```

```
gt_tbl
```

If you decide that you don't want the footnotes in the `gt_tbl` object, they can be removed with the `rm_footnotes()` function.

```
rm_footnotes(data = gt_tbl)
```

Individual footnotes can be selectively removed. Footnotes are identified by their index values. To remove the footnote concerning true solar time (footnote 2, since it was supplied to **gt** after the other footnote) we would give the correct index value to `footnotes`.

```
rm_footnotes(data = gt_tbl, footnotes = 2)
```

Function ID

6-4

See Also

Other part removal functions: `rm_caption()`, `rm_header()`, `rm_source_notes()`, `rm_spanners()`, `rm_stubhead()`

rm_header *Remove the table header*

Description

We can remove the table header from a **gt** table quite easily with `rm_header()`. The table header is an optional table part (positioned above the column labels) that can be added through the `tab_header()`.

This function for removal is useful if you have received a **gt** table (perhaps through an API that returns **gt** objects) but would prefer that the table not contain a header. This function is safe to use even if there is no header part in the input `gt_tbl` object.

Usage

```
rm_header(data)
```

Arguments

data A table object of class `gt_tbl`.

Value

An object of class `gt_tbl`.

Examples

Use `gtcars` to create a **gt** table. Add a header part with the `tab_header()` function; with that, we get a title and a subtitle for the table.

```
gt_tbl <-
  gtcars %>%
  dplyr::select(mfr, model, msrp) %>%
  dplyr::slice(1:5) %>%
  gt() %>%
  tab_header(
    title = md("Data listing from **gtcars**"),
    subtitle = md("`gtcars` is an R dataset")
  )

gt_tbl
```

If you decide that you don't want the header in the `gt_tbl` object, it can be removed with the `rm_header()` function.

```
rm_header(data = gt_tbl)
```

Function ID

6-1

See Also

Other part removal functions: rm_caption(), rm_footnotes(), rm_source_notes(), rm_spanners(), rm_stubhead()

---

rm_source_notes — *Remove table source notes*

Description

If you have one or more source notes that ought to be removed, the rm_source_notes() function allows for such a selective removal. The table footer is an optional table part that is positioned below the table body, containing areas for both the source notes and footnotes.

This function for removal is useful if you have received a **gt** table (perhaps through an API that returns **gt** objects) but would prefer that some or all of the source notes be removed. This function is safe to use even if there are no source notes in the input gt_tbl object so long as select helpers (such as the default everything()) are used instead of explicit integer values.

Usage

```
rm_source_notes(data, source_notes = everything())
```

Arguments

data	A table object of class gt_tbl.
source_notes	A specification of which source notes should be removed. The source notes to be removed can be given as a vector of integer values (they are stored as integer positions, in order of creation, starting at 1). A select helper can also be used and, by default, this is everything() (whereby all source notes will be removed).

Value

An object of class gt_tbl.

Examples

Use gtcars to create a **gt** table. Use tab_source_note() to add a source note to the table footer that cites the data source.

```
gt_tbl <-
  gtcars %>%
  dplyr::select(mfr, model, msrp) %>%
  dplyr::slice(1:5) %>%
  gt() %>%
  tab_source_note(source_note = "Data from the 'edmunds.com' site.") %>%
  tab_source_note(source_note = "Showing only the first five rows.") %>%
```

```
  cols_width(everything() ~ px(120))

gt_tbl
```

If you decide that you don't want the source notes in the `gt_tbl` object, they can be removed with the `rm_source_notes()` function.

```
rm_source_notes(data = gt_tbl)
```

Individual source notes can be selectively removed. Source notes are identified by their index values. To remove the source note concerning the extent of the data (source note 2, since it was supplied to **gt** after the other source note) we would give the correct index value to `source_notes`.

```
rm_source_notes(data = gt_tbl, source_notes = 2)
```

Function ID

6-5

See Also

Other part removal functions: `rm_caption()`, `rm_footnotes()`, `rm_header()`, `rm_spanners()`, `rm_stubhead()`

`rm_spanners` *Remove column spanner labels*

Description

If you would like to remove column spanner labels then the `rm_spanners()` function can make this possible. Column spanner labels appear above the column labels and can occupy several levels via stacking either though `tab_spanner()` or `tab_spanner_delim()`. Spanner column labels are distinguishable and accessible by their ID values.

This function for removal is useful if you have received a **gt** table (perhaps through an API that returns **gt** objects) but would prefer that some or all of the column spanner labels be removed. This function is safe to use even if there are no column spanner labels in the input `gt_tbl` object so long as select helpers (such as the default `everything()`) are used instead of explicit ID values.

Usage

```
rm_spanners(data, spanners = everything(), levels = NULL)
```

Arguments

`data`	A table object of class `gt_tbl`.
`spanners`	A specification of which spanner column labels should be removed. Those to be removed can be given as a vector of spanner ID values (every spanner column label has one, either set by the user or by **gt** when using `tab_spanner_delim()`). A select helper can also be used and, by default, this is `everything()` (whereby all spanner column labels will be removed).
`levels`	Instead of removing spanner column labels by ID values, entire levels of spanners can instead be removed. Supply a numeric vector of level values (the first level is `1`) and, if they are present, they will be removed. Any input given to `level` will mean that `spanners` is ignored.

Value

An object of class `gt_tbl`.

Examples

Use `gtcars` to create a **gt** table. With the `tab_spanner()` function, we can group several related columns together under spanner column labels. In this example, that is done with several calls of `tab_spanner()` in order to create two levels of spanner column labels.

```
gt_tbl <-
  gtcars %>%
  dplyr::select(
    -mfr, -trim, bdy_style, drivetrain,
    -drivetrain, -trsmn, -ctry_origin
  ) %>%
  dplyr::slice(1:8) %>%
  gt(rowname_col = "model") %>%
  tab_spanner(label = "HP", columns = c(hp, hp_rpm)) %>%
  tab_spanner(label = "Torque", columns = c(trq, trq_rpm)) %>%
  tab_spanner(label = "MPG", columns = c(mpg_c, mpg_h)) %>%
  tab_spanner(
    label = "Performance",
    columns = c(
      hp, hp_rpm, trq, trq_rpm,
      mpg_c, mpg_h
    )
  )

gt_tbl
```

If you decide that you don't want any of the spanners in the `gt_tbl` object, they can all be removed with the `rm_spanners()` function.

```
rm_spanners(data = gt_tbl)
```

Individual spanner column labels can be removed by ID value. In all the above uses of `tab_spanner()`, the `label` value *is* the ID value (you can alternately set a different ID value though the `id` argument). Let's remove the `"HP"` and `"MPG"` spanner column labels with `rm_spanners()`.

```
rm_spanners(data = gt_tbl, spanners = c("HP", "MPG"))
```

We can also remove spanner column labels by level with `rm_spanners()`. Provide a vector of one or more values greater than or equal to `1` (the first level starts there). In the next example, we'll remove the first level of spanner column labels. Any levels not being removed will collapse down accordingly.

```
rm_spanners(data = gt_tbl, levels = 1)
```

Function ID

6-3

See Also

Other part removal functions: `rm_caption()`, `rm_footnotes()`, `rm_header()`, `rm_source_notes()`, `rm_stubhead()`

`rm_stubhead` *Remove the stubhead label*

Description

We can easily remove the stubhead label from a **gt** table with `rm_stubhead()`. The stubhead location only exists if there is a table stub and the text in that cell is added through the `tab_stubhead()` function.

This function for removal is useful if you have received a **gt** table (perhaps through an API that returns **gt** objects) but would prefer that the table not contain any content in the stubhead. This function is safe to use even if there is no stubhead label in the input `gt_tbl` object.

Usage

```
rm_stubhead(data)
```

Arguments

`data`	A table object of class `gt_tbl`.

Value

An object of class `gt_tbl`.

Examples

Use gtcars to create a **gt** table. With tab_stubhead(), it's possible to add a stubhead label. This appears in the top-left and can be used to describe what is in the stub.

```
gt_tbl <-
  gtcars %>%
  dplyr::select(model, year, hp, trq) %>%
  dplyr::slice(1:5) %>%
  gt(rowname_col = "model") %>%
  tab_stubhead(label = "car")

gt_tbl
```

If you decide that you don't want the stubhead label in the gt_tbl object, it can be removed with the rm_stubhead() function.

```
rm_stubhead(data = gt_tbl)
```

Function ID

6-2

See Also

Other part removal functions: rm_caption(), rm_footnotes(), rm_header(), rm_source_notes(), rm_spanners()

---

row_group_order *Modify the ordering of any row groups*

Description

We can modify the display order of any row groups in a **gt** object with the row_group_order() function. The groups argument takes a vector of row group ID values. After this function is invoked, the row groups will adhere to this revised ordering. It isn't necessary to provide all row ID values in groups, rather, what is provided will assume the specified ordering at the top of the table and the remaining row groups will follow in their original ordering.

Usage

```
row_group_order(data, groups)
```

Arguments

data	A table object that is created using the gt() function.
groups	A character vector of row group ID values corresponding to the revised ordering. While this vector must contain valid group ID values, it is not required to have all of the row group IDs within it; any omitted values will be added to the end while preserving the original ordering.

Value

An object of class gt_tbl.

Examples

Use exibble to create a **gt** table with a stub and with row groups. Modify the order of the row groups with row_group_order(), specifying the new ordering in groups.

```
exibble %>%
  dplyr::select(char, currency, row, group) %>%
  gt(
    rowname_col = "row",
    groupname_col = "group"
  ) %>%
  row_group_order(groups = c("grp_b", "grp_a"))
```

Function ID

5-3

See Also

Other row addition/modification functions: grand_summary_rows(), summary_rows()

---

sp500 *Daily S&P 500 Index data from 1950 to 2015*

Description

This dataset provides daily price indicators for the S&P 500 index from the beginning of 1950 to the end of 2015. The index includes 500 leading companies and captures about 80\

Usage

```
sp500
```

Format

A tibble with 16607 rows and 7 variables:

date The date expressed as Date values

open, high, low, close The day's opening, high, low, and closing prices in USD; the close price is adjusted for splits

volume the number of trades for the given date

adj_close The close price adjusted for both dividends and splits

Examples

Here is a glimpse at the data available in sp500.

```
dplyr::glimpse(sp500)
#> Rows: 16,607
#> Columns: 7
#> $ date      <date> 2015-12-31, 2015-12-30, 2015-12-29, 2015-12-28, 2015-12-24,~
#> $ open      <dbl> 2060.59, 2077.34, 2060.54, 2057.77, 2063.52, 2042.20, 2023.1~
#> $ high      <dbl> 2062.54, 2077.34, 2081.56, 2057.77, 2067.36, 2064.73, 2042.7~
#> $ low       <dbl> 2043.62, 2061.97, 2060.54, 2044.20, 2058.73, 2042.20, 2020.4~
#> $ close     <dbl> 2043.94, 2063.36, 2078.36, 2056.50, 2060.99, 2064.29, 2038.9~
#> $ volume    <dbl> 2655330000, 2367430000, 2542000000, 2492510000, 1411860000, ~
#> $ adj_close <dbl> 2043.94, 2063.36, 2078.36, 2056.50, 2060.99, 2064.29, 2038.9~
```

Function ID

11-4

See Also

Other datasets: countrypops, exibble, gtcars, pizzaplace, sza

---

stub — *Select helper for targeting the stub column*

Description

Should you need to target only the stub column for formatting or other operations, the `stub()` select helper can be used. This obviates the need to use the name of the column that was selected as the stub column.

Usage

```
stub()
```

Value

A character vector of class "stub_column".

Examples

Create a tibble that has a `row` column (values from `1` to `6`), a `group` column, and a `vals` column (containing the same values as in `row`).

```
tbl <-
  dplyr::tibble(
    row = 1:6,
    group = c(rep("Group A", 3), rep("Group B", 3)),
    vals = 1:6
  )
```

Create a **gt** table with a two-column stub (incorporating the `row` and `group` columns in that). Format the row labels of the stub with `fmt_roman()` to obtain Roman numerals in the stub; we're selecting the stub column here with the `stub()` select helper.

```
tbl %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  fmt_roman(columns = stub()) %>%
  tab_options(row_group.as_column = TRUE)
```

Function ID

7-5

See Also

Other helper functions: `adjust_luminance()`, `cell_borders()`, `cell_fill()`, `cell_text()`, `cells_body()`, `cells_column_labels()`, `cells_column_spanners()`, `cells_footnotes()`, `cells_grand_summary()`, `cells_row_groups()`, `cells_source_notes()`, `cells_stub_grand_summary()`, `cells_stub_summary()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`

---

sub_large_vals *Substitute large values in the table body*

Description

Wherever there are numerical data that are very large in value, replacement text may be better for explanatory purposes. The `sub_large_vals()` function allows for this replacement through specification of a `threshold`, a `large_pattern`, and the sign (positive or negative) of the values to be considered.

Usage

```
sub_large_vals(
  data,
  columns = everything(),
  rows = everything(),
  threshold = 1e+12,
  large_pattern = ">={x}",
  sign = "+"
)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`columns`	The columns to format. Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.
`rows`	Optional rows to format. Providing `everything()` (the default) results in all rows in `columns` being formatted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2] < 5`
`threshold`	The threshold value with which values should be considered large enough for replacement.
`large_pattern`	The pattern text to be used in place of the suitably large values in the rendered table.
`sign`	The sign of the numbers to be considered in the replacement. By default, we only consider positive values (`"+"`). The other option (`"-"`) can be used to consider only negative values.

Details

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the Arguments section for more information on this.

Value

An object of class `gt_tbl`.

Examples

Let's generate a simple, single-column tibble that contains an assortment of values that could potentially undergo some substitution.

```
tbl <- dplyr::tibble(num = c(0, NA, 10^(8:14)))

tbl
#> # A tibble: 9 x 1
#>     num
#>   <dbl>
#> 1  0
#> 2 NA
#> 3  1e 8
#> 4  1e 9
#> 5  1e10
#> 6  1e11
```

```
#> 7  1e12
#> 8  1e13
#> 9  1e14
```

The `tbl` object contains a variety of larger numbers and some might be larger enough to reformat with a threshold value. With `sub_large_vals()` we can do just that:

```
tbl %>%
  gt() %>%
  fmt_number(columns = num) %>%
  sub_large_vals()
```

Large negative values can also be handled but they are handled specially by the `sign` parameter. Setting that to `"-"` will format only the large values that are negative. Notice that with the default `large_pattern` value of `">={x}"` the `">="` is automatically changed to `"<="`.

```
tbl %>%
  dplyr::mutate(num = -num) %>%
  gt() %>%
  fmt_number(columns = num) %>%
  sub_large_vals(sign = "-")
```

You don't have to settle with the default `threshold` value or the default replacement pattern (in `large_pattern`). This can be changed and the `"{x}"` in `large_pattern` (which uses the `threshold` value) can even be omitted.

```
tbl %>%
  gt() %>%
  fmt_number(columns = num) %>%
  sub_large_vals(
    threshold = 5E10,
    large_pattern = "hugemongous"
  )
```

Function ID

3-20

See Also

Other data formatting functions: `data_color()`, `fmt_bytes()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_duration()`, `fmt_engineering()`, `fmt_fraction()`, `fmt_integer()`, `fmt_markdown()`, `fmt_number()`, `fmt_partsper()`, `fmt_passthrough()`, `fmt_percent()`, `fmt_roman()`, `fmt_scientific()`, `fmt_time()`, `fmt()`, `sub_missing()`, `sub_small_vals()`, `sub_values()`, `sub_zero()`, `text_transform()`

`sub_missing` *Substitute missing values in the table body*

Description

Wherever there is missing data (i.e., NA values) customizable content may present better than the standard NA text that would otherwise appear. The sub_missing() function allows for this replacement through its missing_text argument (where an em dash serves as the default).

Usage

```
sub_missing(
  data,
  columns = everything(),
  rows = everything(),
  missing_text = "---"
)
```

Arguments

data	A table object that is created using the gt() function.
columns	The columns to format. Can either be a series of column names provided in c(), a vector of column indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything().
rows	Optional rows to format. Providing everything() (the default) results in all rows in columns being formatted. Alternatively, we can supply a vector of row captions within c(), a vector of row indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 5
missing_text	The text to be used in place of NA values in the rendered table.

Details

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the rows argument. See the Arguments section for more information on this.

Value

An object of class gt_tbl.

Examples

Use exibble to create a **gt** table. The NA values in different columns will be given replacement text with two calls of sub_missing().

```
exibble %>%
  dplyr::select(-row, -group) %>%
  gt() %>%
  sub_missing(
    columns = 1:2,
    missing_text = "missing"
  ) %>%
  sub_missing(
    columns = 4:7,
    missing_text = "nothing"
  )
```

Function ID

3-17

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_small_vals(), sub_values(), sub_zero(), text_transform()

---

sub_small_vals *Substitute small values in the table body*

Description

Wherever there is numerical data that are very small in value, replacement text may be better for explanatory purposes. The sub_small_vals() function allows for this replacement through specification of a threshold, a small_pattern, and the sign of the values to be considered.

Usage

```
sub_small_vals(
  data,
  columns = everything(),
  rows = everything(),
  threshold = 0.01,
  small_pattern = if (sign == "+") "<{x}" else md("<*abs*(-{x})"),
  sign = "+"
)
```

Arguments

data
: A table object that is created using the gt() function.

columns
: The columns to format. Can either be a series of column names provided in c(), a vector of column indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything().

rows
: Optional rows to format. Providing everything() (the default) results in all rows in columns being formatted. Alternatively, we can supply a vector of row captions within c(), a vector of row indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 5

threshold
: The threshold value with which values should be considered small enough for replacement.

small_pattern
: The pattern text to be used in place of the suitably small values in the rendered table.

sign
: The sign of the numbers to be considered in the replacement. By default, we only consider positive values ("+"). The other option ("-") can be used to consider only negative values.

Details

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the rows argument. See the Arguments section for more information on this.

Value

An object of class gt_tbl.

Examples

Let's generate a simple, single-column tibble that contains an assortment of values that could potentially undergo some substitution.

```
tbl <- dplyr::tibble(num = c(10^(-4:2), 0, NA))

tbl
#> # A tibble: 9 x 1
#>         num
#>       <dbl>
#> 1   0.0001
#> 2   0.001
#> 3   0.01
#> 4   0.1
#> 5    1
#> 6   10
```

```
#> 7 100
#> 8   0
#> 9  NA
```

The tbl contains a variety of smaller numbers and some might be small enough to reformat with a threshold value. With sub_small_vals() we can do just that:

```
tbl %>%
  gt() %>%
  fmt_number(columns = num) %>%
  sub_small_vals()
```

Small and negative values can also be handled but they are handled specially by the sign parameter. Setting that to "-" will format only the small, negative values.

```
tbl %>%
  dplyr::mutate(num = -num) %>%
  gt() %>%
  fmt_number(columns = num) %>%
  sub_small_vals(sign = "-")
```

You don't have to settle with the default threshold value or the default replacement pattern (in small_pattern). This can be changed and the "{x}" in small_pattern (which uses the threshold value) can even be omitted.

```
tbl %>%
  gt() %>%
  fmt_number(columns = num) %>%
  sub_small_vals(
    threshold = 0.0005,
    small_pattern = "smol"
  )
```

Function ID

3-19

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_values(), sub_zero(), text_transform()

sub_values *Substitute targeted values in the table body*

Description

Should you need to replace specific cell values with custom text, the sub_values() function can be good choice. We can target cells for replacement though value, regex, and custom matching rules.

Usage

```
sub_values(
  data,
  columns = everything(),
  rows = everything(),
  values = NULL,
  pattern = NULL,
  fn = NULL,
  replacement = NULL,
  escape = TRUE
)
```

Arguments

data	A table object that is created using the gt() function.
columns	Optional columns for constraining the targeting process. Providing everything() (the default) results in cells in all columns being targeting (this can be limited by rows however). Can either be a series of column names provided in c(), a vector of column indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything().
rows	Optional rows for constraining the targeting process. Providing everything() (the default) results in all rows in columns being targeted. Alternatively, we can supply a vector of row captions within c(), a vector of row indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2]
values	The specific value or values that should be replaced with a replacement value. If pattern is also supplied then values will be ignored.
pattern	A regex pattern that can target solely those values in character-based columns. If values is also supplied, pattern will take precedence.
fn	A supplied function that operates on x (the data in a column) and should return a logical vector that matches the length of x (i.e., number of rows in the input table). If either of values or pattern is also supplied, fn will take precedence.
replacement	The replacement value for any cell values matched by either values or pattern. Must be a character or numeric vector of length 1.

escape — An option to escape replacement text according to the final output format of the table. For example, if a LaTeX table is to be generated then LaTeX escaping would be performed on the replacements during rendering. By default this is set to TRUE but setting to FALSE would be useful in the case where replacement text is crafted for a specific output format in mind.

Value

An object of class gt_tbl.

Examples

Let's create an input table with three columns. This contains an assortment of values that could potentially undergo some substitution via sub_values().

```
tbl <-
  dplyr::tibble(
    num_1 = c(-0.01, 74, NA, 0, 500, 0.001, 84.3),
    int_1 = c(1L, -100000L, 800L, 5L, NA, 1L, -32L),
    lett = LETTERS[1:7]
  )

tbl
#> # A tibble: 7 x 3
#>     num_1   int_1 lett
#>     <dbl>   <int> <chr>
#> 1  -0.01        1 A
#> 2  74     -100000 B
#> 3  NA         800 C
#> 4   0           5 D
#> 5 500          NA E
#> 6   0.001       1 F
#> 7  84.3       -32 G
```

Values in the table body cells can be replaced by specifying which values should be replaced (in values) and what the replacement value should be. It's okay to search for numerical or character values across all columns and the replacement value can also be of the numeric or character types.

```
tbl %>%
  gt() %>%
  sub_values(values = c(74, 500), replacement = 150) %>%
  sub_values(values = "B", replacement = "Bee") %>%
  sub_values(values = 800, replacement = "Eight hundred")
```

We can also use the pattern argument to target cell values for replacement in character-based columns.

```
tbl %>%
  gt() %>%
  sub_values(pattern = "A|C|E", replacement = "Ace")
```

For the most flexibility, it's best to use the fn argument. With that you need to ensure that the function you provide will return a logical vector when invoked on a column of cell values, taken as x (and, the length of that vector must match the length of x).

```
tbl %>%
  gt() %>%
  sub_values(
    fn = function(x) x >= 0 & x < 50,
    replacement = "Between 0 and 50"
  )
```

Function ID

3-21

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_zero(), text_transform()

---

sub_zero *Substitute zero values in the table body*

Description

Wherever there is numerical data that are zero in value, replacement text may be better for explanatory purposes. The sub_zero() function allows for this replacement through its zero_text argument.

Usage

```
sub_zero(data, columns = everything(), rows = everything(), zero_text = "nil")
```

Arguments

data	A table object that is created using the gt() function.
columns	The columns to format. Can either be a series of column names provided in c(), a vector of column indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything().
rows	Optional rows to format. Providing everything() (the default) results in all rows in columns being formatted. Alternatively, we can supply a vector of row captions within c(), a vector of row indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), num_range(), and everything(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 5
zero_text	The text to be used in place of zero values in the rendered table.

Details

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). Conditional formatting is possible by providing a conditional expression to the `rows` argument. See the Arguments section for more information on this.

Value

An object of class `gt_tbl`.

Examples

Let's generate a simple, single-column tibble that contains an assortment of values that could potentially undergo some substitution.

```
tbl <- dplyr::tibble(num = c(10^(-1:2), 0, 0, 10^(4:6)))

tbl
#> # A tibble: 9 x 1
#>         num
#>       <dbl>
#> 1       0.1
#> 2       1
#> 3      10
#> 4     100
#> 5       0
#> 6       0
#> 7   10000
#> 8  100000
#> 9 1000000
```

With this table, the zero values in will be given replacement text with a single call of `sub_zero()`.

```
tbl %>%
  gt() %>%
  fmt_number(columns = num) %>%
  sub_zero()
```

Function ID

3-18

See Also

Other data formatting functions: `data_color()`, `fmt_bytes()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_duration()`, `fmt_engineering()`, `fmt_fraction()`, `fmt_integer()`, `fmt_markdown()`, `fmt_number()`, `fmt_partsper()`, `fmt_passthrough()`, `fmt_percent()`, `fmt_roman()`, `fmt_scientific()`, `fmt_time()`, `fmt()`, `sub_large_vals()`, `sub_missing()`, `sub_small_vals()`, `sub_values()`, `text_transform()`

`summary_rows` *Add groupwise summary rows using aggregation functions*

Description

Add summary rows to one or more row groups by using the table data and any suitable aggregation functions. You choose how to format the values in the resulting summary cells by use of a `formatter` function (e.g, `fmt_number`, etc.) and any relevant options.

Usage

```
summary_rows(
  data,
  groups = NULL,
  columns = everything(),
  fns,
  missing_text = "---",
  formatter = fmt_number,
  ...
)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`groups`	The groups to consider for generation of groupwise summary rows. By default this is set to `NULL`, which results in the formation of grand summary rows (a grand summary operates on all table data). Providing the names of row groups in `c()` will create a groupwise summary and generate summary rows for the specified groups. Setting this to `TRUE` indicates that all available groups will receive groupwise summary rows.
`columns`	The columns for which the summaries should be calculated.
`fns`	Functions used for aggregations. This can include base functions like `mean`, `min`, `max`, `median`, `sd`, or `sum` or any other user-defined aggregation function. The function(s) should be supplied within a `list()`. Within that list, we can specify the functions by use of function names in quotes (e.g., `"sum"`), as bare functions (e.g., `sum`), or as one-sided R formulas using a leading `~`. In the formula representation, a `.` serves as the data to be summarized (e.g., `sum(., na.rm = TRUE)`). The use of named arguments is recommended as the names will serve as summary row labels for the corresponding summary rows data (the labels can derived from the function names but only when not providing bare function names).
`missing_text`	The text to be used in place of `NA` values in summary cells with no data outputs.
`formatter`	A formatter function name. These can be any of the `fmt_*()` functions available in the package (e.g., `fmt_number()`, `fmt_percent()`, etc.), or a custom function using `fmt()`. The default function is `fmt_number()` and its options can be accessed through `...`.

... Values passed to the formatter function, where the provided values are to be in the form of named vectors. For example, when using the default formatter function, fmt_number(), options such as decimals, use_seps, and locale can be used.

Details

Should we need to obtain the summary data for external purposes, the extract_summary() function can be used with a gt_tbl object where summary rows were added via summary_rows().

Value

An object of class gt_tbl.

Examples

Use sp500 to create a **gt** table with row groups. Create the summary rows labeled min, max, and avg by row group (where each each row group is a week number) with the summary_rows() function.

```
sp500 %>%
  dplyr::filter(date >= "2015-01-05" & date <="2015-01-16") %>%
  dplyr::arrange(date) %>%
  dplyr::mutate(week = paste0( "W", strftime(date, format = "%V"))) %>%
  dplyr::select(-adj_close, -volume) %>%
  gt(
    rowname_col = "date",
    groupname_col = "week"
  ) %>%
  summary_rows(
    groups = TRUE,
    columns = c(open, high, low, close),
    fns = list(
      min = ~min(.),
      max = ~max(.),
      avg = ~mean(.)),
    formatter = fmt_number,
    use_seps = FALSE
  )
```

Function ID

5-1

See Also

Other row addition/modification functions: grand_summary_rows(), row_group_order()

sza *Twice hourly solar zenith angles by month & latitude*

Description

This dataset contains solar zenith angles (in degrees, with the range of 0-90) every half hour from 04:00 to 12:00, true solar time. This set of values is calculated on the first of every month for 4 different northern hemisphere latitudes. For determination of afternoon values, the presented tabulated values are symmetric about noon.

Usage

```
sza
```

Format

A tibble with 816 rows and 4 variables:

latitude The latitude in decimal degrees for the observations

month The measurement month; all calculations where conducted for the first day of each month

tst The true solar time at the given `latitude` and date (first of `month`) for which the solar zenith angle is calculated

sza The solar zenith angle in degrees, where `NA`s indicate that sunrise hadn't yet occurred by the `tst` value

Details

The solar zenith angle (SZA) is one measure that helps to describe the sun's path across the sky. It's defined as the angle of the sun relative to a line perpendicular to the earth's surface. It is useful to calculate the SZA in relation to the true solar time. True solar time relates to the position of the sun with respect to the observer, which is different depending on the exact longitude. For example, two hours before the sun crosses the meridian (the highest point it would reach that day) corresponds to a true solar time of 10 a.m. The SZA has a strong dependence on the observer's latitude. For example, at a latitude of 50 degrees N at the start of January, the noontime SZA is 73.0 but a different observer at 20 degrees N would measure the noontime SZA to be 43.0 degrees.

Examples

Here is a glimpse at the data available in `sza`.

```
dplyr::glimpse(sza)
#> Rows: 816
#> Columns: 4
#> $ latitude <dbl> 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 2~
#> $ month    <fct> jan, jan, jan, jan, jan, jan, jan, jan, jan, jan, jan, jan, j~
#> $ tst     <chr> "0400", "0430", "0500", "0530", "0600", "0630", "0700", "0730~
#> $ sza     <dbl> NA, NA, NA, NA, NA, NA, 84.9, 78.7, 72.7, 66.1, 61.5, 56.5, 5~
```

Function ID

11-2

Source

Calculated Actinic Fluxes (290 - 700 nm) for Air Pollution Photochemistry Applications (Peterson, 1976), available at: `https://nepis.epa.gov/Exe/ZyPURL.cgi?Dockey=9100JA26.txt`.

See Also

Other datasets: `countrypops`, `exibble`, `gtcars`, `pizzaplace`, `sp500`

`tab_caption` *Add a table caption*

Description

Add a caption to a **gt** table, which is handled specially for a table within an R Markdown, Quarto, or **bookdown** context. The addition of captions makes tables cross-referencing across the containing document. The caption location (i.e., top, bottom, margin) is handled at the document level in each of these system.

Usage

```
tab_caption(data, caption)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`caption`	The table caption to use for cross-referencing in R Markdown, Quarto, or **bookdown**.

Value

An object of class `gt_tbl`.

Examples

Use `gtcars` to create a **gt** table. Add a header part with the `tab_header()` function, and, add a caption as well with `tab_caption()`.

```
gtcars %>%
  dplyr::select(mfr, model, msrp) %>%
  dplyr::slice(1:5) %>%
  gt() %>%
  tab_header(
    title = md("Data listing from **gtcars**"),
```

```
    subtitle = md("`gtcars` is an R dataset")
  ) %>%
  tab_caption(caption = md("**gt** table example."))
```

Function ID

2-9

See Also

Other part creation/modification functions: tab_footnote(), tab_header(), tab_info(), tab_options(), tab_row_group(), tab_source_note(), tab_spanner_delim(), tab_spanner(), tab_stub_indent(), tab_stubhead(), tab_style_body(), tab_style()

`tab_footnote` *Add a table footnote*

Description

The `tab_footnote()` function can make it a painless process to add a footnote to a **gt** table. There are two components to a footnote: (1) a footnote mark that is attached to the targeted cell text, and (2) the footnote text (that starts with the corresponding footnote mark) that is placed in the table's footer area. Each call of `tab_footnote()` will add a different note, and one or more cells can be targeted via the location helper functions (e.g., `cells_body()`, `cells_column_labels()`, etc.).

Usage

```
tab_footnote(
  data,
  footnote,
  locations = NULL,
  placement = c("auto", "right", "left")
)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`footnote`	The text to be used in the footnote. We can optionally use the `md()` and `html()` functions to style the text as Markdown or to retain HTML elements in the footnote text.
`locations`	The cell or set of cells to be associated with the footnote. Supplying any of the `cells_*()` helper functions is a useful way to target the location cells that are associated with the footnote text. These helper functions are: `cells_title()`, `cells_stubhead()`, `cells_column_spanners()`, `cells_column_labels()`, `cells_row_groups()`, `cells_stub()`, `cells_body()`, `cells_summary()`, `cells_grand_summary()`, `cells_stub_summary()`, and `cells_stub_grand_summary()`. Additionally, we can enclose several `cells_*()` calls within a `list()` if we wish to link the

footnote text to different types of locations (e.g., body cells, row group labels, the table title, etc.).

placement — Where to affix footnote marks to the table content. Two options for this are "left or "right", where the placement is to the absolute left or right of the cell content. By default, however, this is set to "auto" whereby **gt** will choose a preferred left-or-right placement depending on the alignment of the cell content.

Details

The formatting of the footnotes can be controlled through the use of various parameters in the tab_options() function:

- footnotes.multiline: a setting that determines whether footnotes each start on a new line or are combined into a single block.
- footnotes.sep: allows for a choice of the separator between consecutive footnotes in the table footer. By default, this is set to a single space character.
- footnotes.marks: the set of sequential characters or numbers used to identify the footnotes.
- footnotes.font.size: the size of the font used in the footnote section.
- footnotes.padding: the amount of padding to apply between the footnote and source note sections in the table footer.

Value

An object of class gt_tbl.

Examples

Use sza to create a **gt** table. Color the sza column using the data_color() function, then, use tab_footnote() to add a footnote to the sza column label (explaining what the color scale signifies).

```
sza %>%
  dplyr::filter(
    latitude == 20 &
      month == "jan" &
      !is.na(sza)
  ) %>%
  dplyr::select(-latitude, -month) %>%
  gt() %>%
  data_color(
    columns = sza,
    colors = scales::col_numeric(
      palette = c("white", "yellow", "navyblue"),
      domain = c(0, 90)
    )
  ) %>%
  tab_footnote(
    footnote = "Color indicates height of sun.",
```

```
    locations = cells_column_labels(
      columns = sza
    )
  )
```

Function ID

2-7

See Also

Other part creation/modification functions: tab_caption(), tab_header(), tab_info(), tab_options(), tab_row_group(), tab_source_note(), tab_spanner_delim(), tab_spanner(), tab_stub_indent(), tab_stubhead(), tab_style_body(), tab_style()

---

tab_header *Add a table header*

Description

We can add a table header to the **gt** table with a title and even a subtitle. A table header is an optional table part that is positioned above the column labels. We have the flexibility to use Markdown formatting for the header's title and subtitle. Furthermore, if the table is intended for HTML output, we can use HTML in either of the title or subtitle.

Usage

```
tab_header(data, title, subtitle = NULL, preheader = NULL)
```

Arguments

- `data` A table object that is created using the gt() function.
- `title, subtitle` Text to be used in the table title and, optionally, for the table subtitle. We can elect to use the md() and html() helper functions to style the text as Markdown or to retain HTML elements in the text.
- `preheader` Optional preheader content that is rendered above the table. Can be supplied as a vector of text.

Value

An object of class `gt_tbl`.

Examples

Use gtcars to create a **gt** table. Add a header part with the `tab_header()` function so that we get a title and a subtitle for the table.

```
gtcars %>%
  dplyr::select(mfr, model, msrp) %>%
  dplyr::slice(1:5) %>%
  gt() %>%
  tab_header(
    title = md("Data listing from **gtcars**"),
    subtitle = md("`gtcars` is an R dataset")
  )
```

Function ID

2-1

See Also

Other part creation/modification functions: `tab_caption()`, `tab_footnote()`, `tab_info()`, `tab_options()`, `tab_row_group()`, `tab_source_note()`, `tab_spanner_delim()`, `tab_spanner()`, `tab_stub_indent()`, `tab_stubhead()`, `tab_style_body()`, `tab_style()`

---

tab_info — *Understand what's been set inside of a **gt** table object*

Description

It can become increasingly difficult to recall the ID values associated with different labels in a **gt** table. Further to this, there are also situations where **gt** will generate ID values on your behalf (e.g., with `tab_spanner_delim()`, etc.) while ensuring that duplicate ID values aren't produced. For the latter case, it is impossible to know what those ID values are unless one were to carefully examine to correct component of the `gt_tbl` object.

Because it's so essential to know these ID values for targeting purposes (when styling with `tab_style()`, adding footnote marks with `tab_footnote()`, etc.), the `tab_info()` function can help with all of this. It summarizes (by location) all of the table's ID values and their associated labels. The product is an informational **gt** table, designed for easy retrieval of the necessary values.

Usage

```
tab_info(data)
```

Arguments

`data`	A table object that is created using the `gt()` function.

Value

An object of class gt_tbl.

Examples

Use gtcars to create a **gt** table. Use the tab_spanner() function to group two columns together under a spanner column with the ID and label "performance". Finally, use the tab_info() function to get a table that summarizes the ID values and their label text for all parts of the table.

```
gt_tbl <-
  gtcars %>%
  dplyr::select(model, year, starts_with("hp"), msrp) %>%
  dplyr::slice(1:4) %>%
  gt(rowname_col = "model") %>%
  tab_spanner(
    label = "performance",
    columns = starts_with("hp")
  )

gt_tbl %>% tab_info()
```

Function ID

2-12

See Also

Other part creation/modification functions: tab_caption(), tab_footnote(), tab_header(), tab_options(), tab_row_group(), tab_source_note(), tab_spanner_delim(), tab_spanner(), tab_stub_indent(), tab_stubhead(), tab_style_body(), tab_style()

---

tab_options *Modify the table output options*

Description

Modify the options available in a table. These options are named by the components, the subcomponents, and the element that can adjusted.

Usage

```
tab_options(
  data,
  container.width = NULL,
  container.height = NULL,
  container.padding.x = NULL,
  container.padding.y = NULL,
```

```
container.overflow.x = NULL,
container.overflow.y = NULL,
table.width = NULL,
table.layout = NULL,
table.align = NULL,
table.margin.left = NULL,
table.margin.right = NULL,
table.background.color = NULL,
table.additional_css = NULL,
table.font.names = NULL,
table.font.size = NULL,
table.font.weight = NULL,
table.font.style = NULL,
table.font.color = NULL,
table.font.color.light = NULL,
table.border.top.style = NULL,
table.border.top.width = NULL,
table.border.top.color = NULL,
table.border.right.style = NULL,
table.border.right.width = NULL,
table.border.right.color = NULL,
table.border.bottom.style = NULL,
table.border.bottom.width = NULL,
table.border.bottom.color = NULL,
table.border.left.style = NULL,
table.border.left.width = NULL,
table.border.left.color = NULL,
heading.background.color = NULL,
heading.align = NULL,
heading.title.font.size = NULL,
heading.title.font.weight = NULL,
heading.subtitle.font.size = NULL,
heading.subtitle.font.weight = NULL,
heading.padding = NULL,
heading.padding.horizontal = NULL,
heading.border.bottom.style = NULL,
heading.border.bottom.width = NULL,
heading.border.bottom.color = NULL,
heading.border.lr.style = NULL,
heading.border.lr.width = NULL,
heading.border.lr.color = NULL,
column_labels.background.color = NULL,
column_labels.font.size = NULL,
column_labels.font.weight = NULL,
column_labels.text_transform = NULL,
column_labels.padding = NULL,
column_labels.padding.horizontal = NULL,
column_labels.vlines.style = NULL,
```

```
column_labels.vlines.width = NULL,
column_labels.vlines.color = NULL,
column_labels.border.top.style = NULL,
column_labels.border.top.width = NULL,
column_labels.border.top.color = NULL,
column_labels.border.bottom.style = NULL,
column_labels.border.bottom.width = NULL,
column_labels.border.bottom.color = NULL,
column_labels.border.lr.style = NULL,
column_labels.border.lr.width = NULL,
column_labels.border.lr.color = NULL,
column_labels.hidden = NULL,
row_group.background.color = NULL,
row_group.font.size = NULL,
row_group.font.weight = NULL,
row_group.text_transform = NULL,
row_group.padding = NULL,
row_group.padding.horizontal = NULL,
row_group.border.top.style = NULL,
row_group.border.top.width = NULL,
row_group.border.top.color = NULL,
row_group.border.bottom.style = NULL,
row_group.border.bottom.width = NULL,
row_group.border.bottom.color = NULL,
row_group.border.left.style = NULL,
row_group.border.left.width = NULL,
row_group.border.left.color = NULL,
row_group.border.right.style = NULL,
row_group.border.right.width = NULL,
row_group.border.right.color = NULL,
row_group.default_label = NULL,
row_group.as_column = NULL,
table_body.hlines.style = NULL,
table_body.hlines.width = NULL,
table_body.hlines.color = NULL,
table_body.vlines.style = NULL,
table_body.vlines.width = NULL,
table_body.vlines.color = NULL,
table_body.border.top.style = NULL,
table_body.border.top.width = NULL,
table_body.border.top.color = NULL,
table_body.border.bottom.style = NULL,
table_body.border.bottom.width = NULL,
table_body.border.bottom.color = NULL,
stub.background.color = NULL,
stub.font.size = NULL,
stub.font.weight = NULL,
stub.text_transform = NULL,
```

```
stub.border.style = NULL,
stub.border.width = NULL,
stub.border.color = NULL,
stub.indent_length = NULL,
stub_row_group.font.size = NULL,
stub_row_group.font.weight = NULL,
stub_row_group.text_transform = NULL,
stub_row_group.border.style = NULL,
stub_row_group.border.width = NULL,
stub_row_group.border.color = NULL,
data_row.padding = NULL,
data_row.padding.horizontal = NULL,
summary_row.background.color = NULL,
summary_row.text_transform = NULL,
summary_row.padding = NULL,
summary_row.padding.horizontal = NULL,
summary_row.border.style = NULL,
summary_row.border.width = NULL,
summary_row.border.color = NULL,
grand_summary_row.background.color = NULL,
grand_summary_row.text_transform = NULL,
grand_summary_row.padding = NULL,
grand_summary_row.padding.horizontal = NULL,
grand_summary_row.border.style = NULL,
grand_summary_row.border.width = NULL,
grand_summary_row.border.color = NULL,
footnotes.background.color = NULL,
footnotes.font.size = NULL,
footnotes.padding = NULL,
footnotes.padding.horizontal = NULL,
footnotes.border.bottom.style = NULL,
footnotes.border.bottom.width = NULL,
footnotes.border.bottom.color = NULL,
footnotes.border.lr.style = NULL,
footnotes.border.lr.width = NULL,
footnotes.border.lr.color = NULL,
footnotes.marks = NULL,
footnotes.multiline = NULL,
footnotes.sep = NULL,
source_notes.background.color = NULL,
source_notes.font.size = NULL,
source_notes.padding = NULL,
source_notes.padding.horizontal = NULL,
source_notes.border.bottom.style = NULL,
source_notes.border.bottom.width = NULL,
source_notes.border.bottom.color = NULL,
source_notes.border.lr.style = NULL,
source_notes.border.lr.width = NULL,
```

```
  source_notes.border.lr.color = NULL,
  source_notes.multiline = NULL,
  source_notes.sep = NULL,
  row.striping.background_color = NULL,
  row.striping.include_stub = NULL,
  row.striping.include_table_body = NULL,
  page.orientation = NULL,
  page.numbering = NULL,
  page.header.use_tbl_headings = NULL,
  page.footer.use_tbl_notes = NULL,
  page.width = NULL,
  page.height = NULL,
  page.margin.left = NULL,
  page.margin.right = NULL,
  page.margin.top = NULL,
  page.margin.bottom = NULL,
  page.header.height = NULL,
  page.footer.height = NULL
)
```

Arguments

`data` A table object that is created using the `gt()` function.

`container.width, container.height, container.padding.x, container.padding.y`
The width and height of the table's container, and, the vertical and horizontal padding of the table's container. The container width and height can be specified with units of pixels or as a percentage. The padding is to be specified as a length with units of pixels. If provided as a numeric value, it is assumed that the value is given in units of pixels. The `px()` and `pct()` helper functions can also be used to pass in numeric values and obtain values as pixel or percent units.

`container.overflow.x, container.overflow.y`
Options to enable scrolling in the horizontal and vertical directions when the table content overflows the container dimensions. Using `TRUE` (the default for both) means that horizontal or vertical scrolling is enabled to view the entire table in those directions. With `FALSE`, the table may be clipped if the table width or height exceeds the `container.width` or `container.height`.

`table.width` The width of the table. Can be specified as a single-length character with units of pixels or as a percentage. If provided as a single-length numeric vector, it is assumed that the value is given in units of pixels. The `px()` and `pct()` helper functions can also be used to pass in numeric values and obtain values as pixel or percent units.

`table.layout` The value for the `table-layout` CSS style in the HTML output context. By default, this is `"fixed"` but another valid option is `"auto"`.

`table.align` The horizontal alignment of the table in its container. By default, this is `"center"`. Other options are `"left"` and `"right"`. This will automatically set `table.margin.left` and `table.margin.right` to the appropriate values.

`table.margin.left`, `table.margin.right`
: The size of the margins on the left and right of the table within the container. Can be specified as a single-length character with units of pixels or as a percentage. If provided as a single-length numeric vector, it is assumed that the value is given in units of pixels. The `px()` and `pct()` helper functions can also be used to pass in numeric values and obtain values as pixel or percent units. Using `table.margin.left` or `table.margin.right` will overwrite any values set by `table.align`.

`table.background.color`, `heading.background.color`, `column_labels.background.color`, `row_group.backgrour`
: Background colors for the parent element `table` and the following child elements: `heading`, `column_labels`, `row_group`, `stub`, `summary_row`, `grand_summary_row`, `footnotes`, and `source_notes`. A color name or a hexadecimal color code should be provided.

`table.additional_css`
: This option can be used to supply an additional block of CSS rules to be applied after the automatically generated table CSS.

`table.font.names`
: The names of the fonts used for the table. This is a vector of several font names. If the first font isn't available, then the next font is tried (and so on).

`table.font.size`, `heading.title.font.size`, `heading.subtitle.font.size`, `column_labels.font.size`, `row_gr`
: The font sizes for the parent text element `table` and the following child elements: `heading.title`, `heading.subtitle`, `column_labels`, `row_group`, `footnotes`, and `source_notes`. Can be specified as a single-length character vector with units of pixels (e.g., `12px`) or as a percentage (e.g., `80\%`). If provided as a single-length numeric vector, it is assumed that the value is given in units of pixels. The `px()` and `pct()` helper functions can also be used to pass in numeric values and obtain values as pixel or percentage units.

`table.font.weight`, `heading.title.font.weight`, `heading.subtitle.font.weight`, `column_labels.font.weight`
: The font weights of the table, `heading.title`, `heading.subtitle`, `column_labels`, `row_group`, and `stub` text elements. Can be a text-based keyword such as `"normal"`, `"bold"`, `"lighter"`, `"bolder"`, or, a numeric value between `1` and `1000`, inclusive. Note that only variable fonts may support the numeric mapping of weight.

`table.font.style`
: The font style for the table. Can be one of either `"normal"`, `"italic"`, or `"oblique"`.

`table.font.color`, `table.font.color.light`
: The text color used throughout the table. There are two variants: `table.font.color` is for text overlaid on lighter background colors, and `table.font.color.light` is automatically used when text needs to be overlaid on darker background colors. A color name or a hexadecimal color code should be provided.

`table.border.top.style`, `table.border.top.width`, `table.border.top.color`, `table.border.right.style`, `tab`
: The style, width, and color properties of the table's absolute top and absolute bottom borders.

`heading.align`
: Controls the horizontal alignment of the heading title and subtitle. We can either use `"center"`, `"left"`, or `"right"`.

heading.padding, column_labels.padding, data_row.padding, row_group.padding, summary_row.padding, gran

The amount of vertical padding to incorporate in the heading (title and subtitle), the column_labels (this includes the column spanners), the row group labels (row_group.padding), in the body/stub rows (data_row.padding), in summary rows (summary_row.padding or grand_summary_row.padding), or in the footnotes and source notes (footnotes.padding and source_notes.padding).

heading.padding.horizontal, column_labels.padding.horizontal, data_row.padding.horizontal, row_group.

The amount of horizontal padding to incorporate in the heading (title and subtitle), the column_labels (this includes the column spanners), the row group labels (row_group.padding.horizontal), in the body/stub rows (data_row.padding), in summary rows (summary_row.padding.horizontal or grand_summary_row.padding.horizontal) or in the footnotes and source notes (footnotes.padding.horizontal and source_notes.padding.horizontal).

heading.border.bottom.style, heading.border.bottom.width, heading.border.bottom.color

The style, width, and color properties of the header's bottom border. This border shares space with that of the column_labels location. If the width of this border is larger, then it will be the visible border.

heading.border.lr.style, heading.border.lr.width, heading.border.lr.color

The style, width, and color properties for the left and right borders of the heading location.

column_labels.text_transform, row_group.text_transform, stub.text_transform, summary_row.text_transfo

Options to apply text transformations to the column_labels, row_group, stub, summary_row, and grand_summary_row text elements. Either of the "uppercase", "lowercase", or "capitalize" keywords can be used.

column_labels.vlines.style, column_labels.vlines.width, column_labels.vlines.color

The style, width, and color properties for all vertical lines ('vlines') of the the column_labels.

column_labels.border.top.style, column_labels.border.top.width, column_labels.border.top.color

The style, width, and color properties for the top border of the column_labels location. This border shares space with that of the heading location. If the width of this border is larger, then it will be the visible border.

column_labels.border.bottom.style, column_labels.border.bottom.width, column_labels.border.bottom.co

The style, width, and color properties for the bottom border of the column_labels location.

column_labels.border.lr.style, column_labels.border.lr.width, column_labels.border.lr.color

The style, width, and color properties for the left and right borders of the column_labels location.

column_labels.hidden

An option to hide the column labels. If providing TRUE then the entire column_labels location won't be seen and the table header (if present) will collapse downward.

row_group.border.top.style, row_group.border.top.width, row_group.border.top.color, row_group.border.

The style, width, and color properties for all top, bottom, left, and right borders of the row_group location.

row_group.default_label

An option to set a default row group label for any rows not formally placed in a row group named by group in any call of tab_row_group(). If this is set as NA_character and there are rows that haven't been placed into a row group (where one or more row groups already exist), those rows will be automatically placed into a row group without a label.

row_group.as_column
: How should row groups be structured? By default, they are separate rows that lie above the each of the groups. Setting this to TRUE will structure row group labels are columns to the far left of the table.

table_body.hlines.style, table_body.hlines.width, table_body.hlines.color, table_body.vlines.style, t
: The style, width, and color properties for all horizontal lines ('hlines') and vertical lines ('vlines') in the table_body.

table_body.border.top.style, table_body.border.top.width, table_body.border.top.color, table_body.bor
: The style, width, and color properties for all top and bottom borders of the table_body location.

stub.border.style, stub.border.width, stub.border.color
: The style, width, and color properties for the vertical border of the table stub.

stub.indent_length
: The width of each indentation level. By default this is "5px".

stub_row_group.font.size, stub_row_group.font.weight, stub_row_group.text_transform, stub_row_group.b
: Options for the row group column in the stub (made possible when using row_group.as_column = TRUE). The defaults for these options mirror that of the stub.* variants (except for stub_row_group.border.width, which is "1px" instead of "2px").

summary_row.border.style, summary_row.border.width, summary_row.border.color
: The style, width, and color properties for all horizontal borders of the summary_row location.

grand_summary_row.border.style, grand_summary_row.border.width, grand_summary_row.border.color
: The style, width, and color properties for the top borders of the grand_summary_row location.

footnotes.border.bottom.style, footnotes.border.bottom.width, footnotes.border.bottom.color
: The style, width, and color properties for the bottom border of the footnotes location.

footnotes.border.lr.style, footnotes.border.lr.width, footnotes.border.lr.color
: The style, width, and color properties for the left and right borders of the footnotes location.

footnotes.marks
: The set of sequential marks used to reference and identify each of the footnotes (same input as the opt_footnote_marks() function. We can supply a vector that represents the series of footnote marks. This vector is recycled when its usage goes beyond the length of the set. At each cycle, the marks are simply combined (e.g., * -> ** -> ***). The option exists for providing keywords for certain types of footnote marks. The keyword "numbers" (the default, indicating that we want to use numeric marks). We can use lowercase "letters" or uppercase "LETTERS". There is the option for using a traditional symbol set where "standard" provides four symbols, and, "extended" adds two more symbols, making six.

footnotes.multiline, source_notes.multiline
: An option to either put footnotes and source notes in separate lines (the default, or TRUE) or render them as a continuous line of text with footnotes.sep providing the separator (by default " ") between notes.

footnotes.sep, source_notes.sep
: The separating characters between adjacent footnotes and source notes in their respective footer sections when rendered as a continuous line of text (when footnotes.multiline == FALSE). The default value is a single space character (" ").

source_notes.border.bottom.style, source_notes.border.bottom.width, source_notes.border.bottom.color
: The style, width, and color properties for the bottom border of the source_notes location.

source_notes.border.lr.style, source_notes.border.lr.width, source_notes.border.lr.color
: The style, width, and color properties for the left and right borders of the source_notes location.

row.striping.background_color
: The background color for striped table body rows. A color name or a hexadecimal color code should be provided.

row.striping.include_stub
: An option for whether to include the stub when striping rows.

row.striping.include_table_body
: An option for whether to include the table body when striping rows.

page.orientation
: For RTF output, this provides an two options for page orientation: "portrait" (the default) and "landscape".

page.numbering
: Within RTF output, should page numbering be displayed? By default, this is set to FALSE but if TRUE then page numbering text will be added to the document header.

page.header.use_tbl_headings
: If TRUE then RTF output tables will migrate all table headings (including the table title and all column labels) to the page header. This page header content will repeat across pages. By default, this is FALSE.

page.footer.use_tbl_notes
: If TRUE then RTF output tables will migrate all table footer content (this includes footnotes and source notes) to the page footer. This page footer content will repeat across pages. By default, this is FALSE.

page.width, page.height
: The page width and height in the standard portrait orientation. This is for RTF table output and the default values (in inches) are 8.5in and 11.0in.

page.margin.left, page.margin.right, page.margin.top, page.margin.bottom
: For RTF table output, these options correspond to the left, right, top, and bottom page margins. The default values for each of these is 1.0in.

page.header.height, page.footer.height
: The heights of the page header and footer for RTF table outputs. Default values for both are 0.5in.

Value

An object of class gt_tbl.

Examples

Use `exibble` to create a **gt** table with all the main parts added. We can use this **gt** object going forward to demo some of what's available in the `tab_options()` function.

```
tab_1 <-
  exibble %>%
  dplyr::select(-c(fctr, date, time, datetime)) %>%
  gt(
    rowname_col = "row",
    groupname_col = "group"
  ) %>%
  tab_header(
    title = md("Data listing from **exibble**"),
    subtitle = md("`exibble` is an R dataset")
  ) %>%
  fmt_number(columns = num) %>%
  fmt_currency(columns = currency) %>%
  tab_footnote(
    footnote = "Using commas for separators.",
    locations = cells_body(
      columns = num,
      rows = num > 1000
    )
  ) %>%
  tab_footnote(
    footnote = "Using commas for separators.",
    locations = cells_body(
      columns = currency,
      rows = currency > 1000
    )
  ) %>%
  tab_footnote(
    footnote = "Alphabetical fruit.",
    locations = cells_column_labels(columns = char)
  )

tab_1
```

Modify the table width to be 100% (which spans the entire content width area).

```
tab_1 %>% tab_options(table.width = pct(100))
```

Modify the table's background color to be `"lightcyan"`.

```
tab_1 %>% tab_options(table.background.color = "lightcyan")
```

Use letters as the marks for footnote references. Also, separate footnotes in the footer by spaces instead of newlines.

```
tab_1 %>%
  tab_options(
    footnotes.marks = letters,
    footnotes.multiline = FALSE
  )
```

Change the padding of data rows to 5 px.

```
tab_1 %>%
  tab_options(
    data_row.padding = px(5)
  )
```

Reduce the size of the title and the subtitle text.

```
tab_1 %>%
  tab_options(
    heading.title.font.size = "small",
    heading.subtitle.font.size = "small"
  )
```

Function ID

2-12

See Also

Other part creation/modification functions: `tab_caption()`, `tab_footnote()`, `tab_header()`, `tab_info()`, `tab_row_group()`, `tab_source_note()`, `tab_spanner_delim()`, `tab_spanner()`, `tab_stub_indent()`, `tab_stubhead()`, `tab_style_body()`, `tab_style()`

`tab_row_group` *Add a row group to a* **gt** *table*

Description

Create a row group with a collection of rows. This requires specification of the rows to be included, either by supplying row labels, row indices, or through use of a select helper function like `starts_with()`. To modify the order of row groups, use the `row_group_order()` function.

To set a default row group label for any rows not formally placed in a row group, we can use a separate call to `tab_options(row_group.default_label = <label>)`. If this is not done and there are rows that haven't been placed into a row group (where one or more row groups already exist), those rows will be automatically placed into a row group without a label. To restore labels for row groups not explicitly assigned a group, `tab_options(row_group.default_label = "")` can be used.

Usage

```
tab_row_group(data, label, rows, id = label, others_label = NULL, group = NULL)
```

Arguments

data	A table object that is created using the gt() function.
label	The text to use for the row group label.
rows	The rows to be made components of the row group. Can either be a vector of row captions provided in c(), a vector of row indices, or a helper function focused on selections. The select helper functions are: starts_with(), ends_with(), contains(), matches(), one_of(), and everything().
id	The ID for the row group. When accessing a row group through cells_row_groups() (when using tab_style() or tab_footnote()) the id value is used as the reference (and not the label). If an id is not explicitly provided here, it will be taken from the label value. It is advisable to set an explicit id value if you plan to access this cell in a later function call and the label text is complicated (e.g., contains markup, is lengthy, or both). Finally, when providing an id value you must ensure that it is unique across all ID values set for row groups (the function will stop if id isn't unique).
others_label	This argument is deprecated. Instead use tab_options(row_group.default_label = <label>).
group	This argument is deprecated. Instead use label.

Value

An object of class gt_tbl.

Examples

Use gtcars to create a **gt** table and use tab_row_group() to add two row groups with the labels: numbered and NA. The row group with the NA label ends up being rendered without a label at all.

```
gtcars %>%
  dplyr::select(model, year, hp, trq) %>%
  dplyr::slice(1:8) %>%
  gt(rowname_col = "model") %>%
  tab_row_group(
    label = "numbered",
    rows = matches("^[0-9]")
  )
```

Use gtcars to create a **gt** table. Add two row groups with the labels powerful and super powerful. The distinction between the groups is whether hp is lesser or greater than 600 (governed by the expressions provided to the rows argument).

```
gtcars %>%
  dplyr::select(model, year, hp, trq) %>%
  dplyr::slice(1:8) %>%
```

```
gt(rowname_col = "model") %>%
tab_row_group(
  label = "powerful",
  rows = hp <= 600
) %>%
tab_row_group(
  label = "super powerful",
  rows = hp > 600
)
```

Function ID

2-4

See Also

Other part creation/modification functions: tab_caption(), tab_footnote(), tab_header(), tab_info(), tab_options(), tab_source_note(), tab_spanner_delim(), tab_spanner(), tab_stub_indent(), tab_stubhead(), tab_style_body(), tab_style()

---

tab_source_note *Add a source note citation*

Description

Add a source note to the footer part of the **gt** table. A source note is useful for citing the data included in the table. Several can be added to the footer, simply use multiple calls of tab_source_note() and they will be inserted in the order provided. We can use Markdown formatting for the note, or, if the table is intended for HTML output, we can include HTML formatting.

Usage

```
tab_source_note(data, source_note)
```

Arguments

data	A table object that is created using the gt() function.
source_note	Text to be used in the source note. We can optionally use the md() and html() functions to style the text as Markdown or to retain HTML elements in the text.

Value

An object of class gt_tbl.

Examples

Use gtcars to create a **gt** table. Use `tab_source_note()` to add a source note to the table footer that cites the data source.

```
gtcars %>%
  dplyr::select(mfr, model, msrp) %>%
  dplyr::slice(1:5) %>%
  gt() %>%
  tab_source_note(source_note = "From edmunds.com")
```

Function ID

2-8

See Also

Other part creation/modification functions: `tab_caption()`, `tab_footnote()`, `tab_header()`, `tab_info()`, `tab_options()`, `tab_row_group()`, `tab_spanner_delim()`, `tab_spanner()`, `tab_stub_indent()`, `tab_stubhead()`, `tab_style_body()`, `tab_style()`

`tab_spanner` *Add a spanner column label*

Description

Set a spanner column label by mapping it to columns already in the table. This label is placed above one or more column labels, spanning the width of those columns and column labels.

Usage

```
tab_spanner(
  data,
  label,
  columns = NULL,
  spanners = NULL,
  level = NULL,
  id = label,
  gather = TRUE,
  replace = FALSE
)
```

Arguments

data	A table object that is created using the `gt()` function.
label	The text to use for the spanner column label.
columns	The columns to be components of the spanner heading.

`spanners` The spanners that should be spanned over, should they already be defined.

`level` An explicit level to which the spanner should be placed. If not provided, **gt** will choose the level based on the inputs provided within `columns` and `spanners`, placing the spanner label where it will fit. The first spanner level (right above the column labels) is `1`.

`id` The ID for the spanner column label. When accessing a spanner column label through `cells_column_spanners()` (when using `tab_style()` or `tab_footnote()`) the `id` value is used as the reference (and not the `label`). If an `id` is not explicitly provided here, it will be taken from the `label` value. It is advisable to set an explicit `id` value if you plan to access this cell in a later function call and the label text is complicated (e.g., contains markup, is lengthy, or both). Finally, when providing an `id` value you must ensure that it is unique across all ID values set for column spanner labels (the function will stop if `id` isn't unique).

`gather` An option to move the specified `columns` such that they are unified under the spanner column label. Ordering of the moved-into-place columns will be preserved in all cases. By default, this is set to `TRUE`.

`replace` Should new spanners be allowed to partially or fully replace existing spanners? (This is a possibility if setting spanners at an already populated `level`.) By default, this is set to `FALSE` and an error will occur if some replacement is attempted.

Value

An object of class `gt_tbl`.

Examples

Use `gtcars` to create a **gt** table. Use the `tab_spanner()` function to effectively group several columns related to car performance under a spanner column with the label `"performance"`.

```
gtcars %>%
  dplyr::select(
    -mfr, -trim, bdy_style,
    -drivetrain, -trsmn, -ctry_origin
  ) %>%
  dplyr::slice(1:8) %>%
  gt(rowname_col = "model") %>%
  tab_spanner(
    label = "performance",
    columns = c(
      hp, hp_rpm, trq, trq_rpm,
      mpg_c, mpg_h
    )
  )
```

Function ID

2-2

See Also

Other part creation/modification functions: tab_caption(), tab_footnote(), tab_header(), tab_info(), tab_options(), tab_row_group(), tab_source_note(), tab_spanner_delim(), tab_stub_indent(), tab_stubhead(), tab_style_body(), tab_style()

---

tab_spanner_delim *Create column labels and spanners via delimited names*

Description

This function will split selected delimited column names such that the first components (LHS) are promoted to being spanner column labels, and the secondary components (RHS) will become the column labels. Please note that reference to individual columns must continue to be the column names from the input table data (which are unique by necessity).

Usage

```
tab_spanner_delim(
  data,
  delim,
  columns = everything(),
  split = c("last", "first")
)
```

Arguments

data	A table object that is created using the gt() function.
delim	The delimiter to use to split an input column name. The delimiter supplied will be autoescaped for the internal splitting procedure. The first component of the split will become the spanner column label (and its ID value, used for styling or for the addition of footnotes in those locations) and the second component will be the column label.
columns	An optional vector of column names that this operation should be limited to. The default is to consider all columns in the table.
split	Should the delimiter splitting occur from the "last" instance of the delim character or from the "first"? By default, column name splitting begins at the last instance of the delimiter.

Details

If we look to the column names in the iris dataset as an example of how tab_spanner_delim() might be useful, we find the names Sepal.Length, Sepal.Width, Petal.Length, Petal.Width. From this naming system, it's easy to see that the Sepal and Petal can group together the repeated common Length and Width values. In your own datasets, we can avoid a lengthy relabeling with cols_label() if column names can be fashioned beforehand to contain both the spanner column label and the column label. An additional advantage is that the column names in the input table data remain unique even though there may eventually be repeated column labels in the rendered output table).

Value

An object of class `gt_tbl`.

Examples

Use `iris` to create a **gt** table and use the `tab_spanner_delim()` function to automatically generate column spanner labels. This splits any columns that are dot-separated between column spanner labels (first part) and column labels (second part).

```
iris %>%
  dplyr::group_by(Species) %>%
  dplyr::slice(1:4) %>%
  gt() %>%
  tab_spanner_delim(delim = ".")
```

Function ID

2-3

See Also

Other part creation/modification functions: `tab_caption()`, `tab_footnote()`, `tab_header()`, `tab_info()`, `tab_options()`, `tab_row_group()`, `tab_source_note()`, `tab_spanner()`, `tab_stub_indent()`, `tab_stubhead()`, `tab_style_body()`, `tab_style()`

`tab_stubhead` *Add label text to the stubhead*

Description

Add a label to the stubhead of a **gt** table. The stubhead is the lone element that is positioned left of the column labels, and above the stub. If a stub does not exist, then there is no stubhead (so no change will be made when using this function in that case). We have the flexibility to use Markdown formatting for the stubhead label. Furthermore, if the table is intended for HTML output, we can use HTML for the stubhead label.

Usage

```
tab_stubhead(data, label)
```

Arguments

`data` A table object that is created using the `gt()` function.

`label` The text to be used as the stubhead label We can optionally use the `md()` and `html()` functions to style the text as Markdown or to retain HTML elements in the text.

Value

An object of class `gt_tbl`.

Examples

Use `gtcars` to create a **gt** table. With `tab_stubhead()` we can add a stubhead label. This appears in the top-left and can be used to describe what is in the stub.

```
gtcars %>%
  dplyr::select(model, year, hp, trq) %>%
  dplyr::slice(1:5) %>%
  gt(rowname_col = "model") %>%
  tab_stubhead(label = "car")
```

Function ID

2-5

See Also

Other part creation/modification functions: `tab_caption()`, `tab_footnote()`, `tab_header()`, `tab_info()`, `tab_options()`, `tab_row_group()`, `tab_source_note()`, `tab_spanner_delim()`, `tab_spanner()`, `tab_stub_indent()`, `tab_style_body()`, `tab_style()`

`tab_stub_indent` *Control indentation of row labels in the stub*

Description

Indentation of row labels is an effective way for establishing structure in a table stub. The `tab_stub_indent()` function allows for fine control over row label indentation through either explicit definition of an indentation level, or, by way of an indentation directive using keywords.

Usage

```
tab_stub_indent(data, rows, indent = "increase")
```

Arguments

`data`	A table object that is created using the `gt()` function.
`rows`	The rows to consider for the indentation change. Can either be a vector of row captions provided in `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, and `everything()`.

indent — An indentation directive either as a keyword describing the indentation change or as an explicit integer value for directly setting the indentation level. The keyword "increase" (the default) will increase the indentation level by one; "decrease" will do the same in the reverse direction. The starting indentation level of 0 means no indentation and this values serves as a lower bound. The upper bound for indentation is at level 5.

Value

An object of class gt_tbl.

Examples

Use pizzaplace to create a **gt** table. With tab_stub_indent() we can add indentation to targeted row labels in the stub. Here we target the different pizza sizes and avoid selecting the repeating "All Sizes" row label.

```
dplyr::bind_rows(
  pizzaplace %>%
    dplyr::group_by(type, size) %>%
    dplyr::summarize(
      sold = n(),
      income = sum(price),
      .groups = "drop_last"
    ) %>%
    dplyr::summarize(
      sold = sum(sold),
      income = sum(income),
      size = "All Sizes",
      .groups = "drop"
    ),
  pizzaplace %>%
    dplyr::group_by(type, size) %>%
    dplyr::summarize(
      sold = n(),
      income = sum(price),
      .groups = "drop"
    )
) %>%
  gt(rowname_col = "size", groupname_col = "type") %>%
  tab_header(title = "Pizzas Sold in 2015") %>%
  fmt_number(
    columns = sold,
    decimals = 0,
    use_seps = TRUE
  ) %>%
  fmt_currency(
    columns = income,
    currency = "USD"
```

```
) %>%
tab_options(
  summary_row.background.color = "#ACEACE",
  row_group.background.color = "#FFEFDB",
  row_group.as_column = TRUE
) %>%
tab_stub_indent(
  rows = matches("^L|^M|^S|^XL|^XXL"),
  indent = 2
) %>%
tab_style(
  style = cell_fill(color = "gray95"),
  locations = list(
    cells_body(rows = matches("^All")),
    cells_stub(rows = matches("^All"))
  )
)
```

Function ID

2-6

See Also

Other part creation/modification functions: tab_caption(), tab_footnote(), tab_header(), tab_info(), tab_options(), tab_row_group(), tab_source_note(), tab_spanner_delim(), tab_spanner(), tab_stubhead(), tab_style_body(), tab_style()

`tab_style` *Add custom styles to one or more cells*

Description

With the `tab_style()` function we can target specific cells and apply styles to them. This is best done in conjunction with the helper functions `cell_text()`, `cell_fill()`, and `cell_borders()`. At present this function is focused on the application of styles for HTML output only (as such, other output formats will ignore all `tab_style()` calls). Using the aforementioned helper functions, here are some of the styles we can apply:

- the background color of the cell (`cell_fill()`: `color`)
- the cell's text color, font, and size (`cell_text()`: `color`, `font`, `size`)
- the text style (`cell_text()`: `style`), enabling the use of italics or oblique text.
- the text weight (`cell_text()`: `weight`), allowing the use of thin to bold text (the degree of choice is greater with variable fonts)
- the alignment and indentation of text (`cell_text()`: `align` and `indent`)
- the cell borders (`cell_borders()`)

Usage

```
tab_style(data, style, locations)
```

Arguments

data	A table object that is created using the gt() function.
style	a vector of styles to use. The cell_text(), cell_fill(), and cell_borders() helper functions can be used here to more easily generate valid styles. If using more than one helper function to define styles, all calls must be enclosed in a list(). Custom CSS declarations can be used for HTML output by including a css()-based statement as a list item.
locations	the cell or set of cells to be associated with the style. Supplying any of the cells_*() helper functions is a useful way to target the location cells that are associated with the styling. These helper functions are: cells_title(), cells_stubhead(), cells_column_spanners(), cells_column_labels(), cells_row_groups(), cells_stub(), cells_body(), cells_summary(), cells_grand_summary(), cells_stub_summary(), cells_stub_grand_summary(), cells_footnotes(), and cells_source_notes(). Additionally, we can enclose several cells_*() calls within a list() if we wish to apply styling to different types of locations (e.g., body cells, row group labels, the table title, etc.).

Value

An object of class gt_tbl.

Examples

Use exibble to create a **gt** table. Add styles that are to be applied to data cells that satisfy a condition (using tab_style()).

```
exibble %>%
  dplyr::select(num, currency) %>%
  gt() %>%
  fmt_number(
    columns = c(num, currency),
    decimals = 1
  ) %>%
  tab_style(
    style = list(
      cell_fill(color = "lightcyan"),
      cell_text(weight = "bold")
      ),
    locations = cells_body(
      columns = num,
      rows = num >= 5000
    )
  ) %>%
  tab_style(
```

```
    style = list(
      cell_fill(color = "#F9E3D6"),
      cell_text(style = "italic")
      ),
    locations = cells_body(
      columns = currency,
      rows = currency < 100
    )
  )
```

Use sp500 to create a **gt** table. Color entire rows of cells based on values in a particular column.

```
sp500 %>%
  dplyr::filter(
    date >= "2015-12-01" &
    date <= "2015-12-15"
  ) %>%
  dplyr::select(-c(adj_close, volume)) %>%
  gt() %>%
  tab_style(
    style = cell_fill(color = "lightgreen"),
    locations = cells_body(rows = close > open)
  ) %>%
  tab_style(
    style = list(
      cell_fill(color = "red"),
      cell_text(color = "white")
      ),
    locations = cells_body(rows = open > close)
  )
```

Use exibble to create a **gt** table. Replace missing values with the sub_missing() function and then add styling to the char column with cell_fill() and with a CSS style declaration.

```
exibble %>%
  dplyr::select(char, fctr) %>%
  gt() %>%
  sub_missing() %>%
  tab_style(
    style = list(
      cell_fill(color = "lightcyan"),
      "font-variant: small-caps;"
    ),
    locations = cells_body(columns = char)
  )
```

Function ID

2-10

See Also

`cell_text()`, `cell_fill()`, and `cell_borders()` as helpers for defining custom styles and `cells_body()` as one of many useful helper functions for targeting the locations to be styled.

Other part creation/modification functions: `tab_caption()`, `tab_footnote()`, `tab_header()`, `tab_info()`, `tab_options()`, `tab_row_group()`, `tab_source_note()`, `tab_spanner_delim()`, `tab_spanner()`, `tab_stub_indent()`, `tab_stubhead()`, `tab_style_body()`

`tab_style_body` *Target cells in the table body and style accordingly*

Description

With the `tab_style_body()` function we can target cells though value, regex, and custom matching rules and apply styles to them and their surrounding context (i.e., styling an entire row or column wherein the match is found). Just as with the general `tab_style()` function, this function is focused on the application of styles for HTML output only (as such, other output formats will ignore all `tab_style()` calls).

With the collection of `cell_*()` helper functions available in **gt**, we can modify:

- the background color of the cell (`cell_fill()`: `color`)
- the cell's text color, font, and size (`cell_text()`: `color`, `font`, `size`)
- the text style (`cell_text()`: `style`), enabling the use of italics or oblique text.
- the text weight (`cell_text()`: `weight`), allowing the use of thin to bold text (the degree of choice is greater with variable fonts)
- the alignment and indentation of text (`cell_text()`: `align` and `indent`)
- the cell borders (`cell_borders()`)

Usage

```
tab_style_body(
  data,
  style,
  columns = everything(),
  rows = everything(),
  values = NULL,
  pattern = NULL,
  fn = NULL,
  targets = "cell",
  extents = "body"
)
```

Arguments

- `data` A table object that is created using the `gt()` function.
- `style` a vector of styles to use. The `cell_text()`, `cell_fill()`, and `cell_borders()` helper functions can be used here to more easily generate valid styles. If using more than one helper function to define styles, all calls must be enclosed in a `list()`. Custom CSS declarations can be used for HTML output by including a `css()`-based statement as a list item.
- `columns` Optional columns for constraining the targeting process. Providing `everything()` (the default) results in cells in all `columns` being targeting (this can be limited by `rows` however). Can either be a series of column names provided in `c()`, a vector of column indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`.
- `rows` Optional rows for constraining the targeting process. Providing `everything()` (the default) results in all rows in `columns` being targeted. Alternatively, we can supply a vector of row captions within `c()`, a vector of row indices, or a helper function focused on selections. The select helper functions are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()`, `num_range()`, and `everything()`. We can also use expressions to filter down to the rows we need (e.g., `[colname_1] > 100 & [colname_2]`
- `values` The specific value or values that should be targeted for styling. If `pattern` is also supplied then `values` will be ignored.
- `pattern` A regex pattern that can target solely those values in `character`-based columns. If `values` is also supplied, `pattern` will take precedence.
- `fn` A supplied function that operates on each cell of each column specified through `columns` and `rows`. The function should be fashioned such that a single logical value is returned. If either of `values` or `pattern` is also supplied, `fn` will take precedence.
- `targets` A vector of styling target keywords to contain or expand the target of each cell. By default, this is a vector just containing `"cell"`. However, the keywords `"row"` and `"column"` may be used separately or in combination to style the target cells' associated rows or columns.
- `extents` A vector of locations to project styling. By default, this is a vector just containing `"body"`, whereby styled rows or columns (facilitated via inclusion of the `"row"` and `"column"` keywords in `targets`) will not permeate into the stub. The additional keyword `"stub"` may be used alone or in conjunction with `"body"` to project or expand the styling into the stub.

Value

An object of class `gt_tbl`.

Examples

Use `exibble` to create a **gt** table with a stub and row groups. This contains an assortment of values that could potentially undergo some styling via `tab_style_body()`.

```
gt_tbl <-
  exibble %>%
  gt(
    rowname_col = "row",
    groupname_col = "group"
  )
```

Cells in the table body can be styled through specification of literal values in the `values` argument of `tab_style_body()`. It's okay to search for numerical, character, or logical values across all columns. Let's target the values `49.95` and `33.33` and style those cells with an orange fill.

```
gt_tbl %>%
  tab_style_body(
    style = cell_fill(color = "orange"),
    values = c(49.95, 33.33)
  )
```

Multiple styles can be combined in a `list`, here's an example of that using the same cell targets:

```
gt_tbl %>%
  tab_style_body(
    style = list(
      cell_text(font = google_font("Inter"), color = "white"),
      cell_fill(color = "red"),
      cell_borders(
        sides = c("left", "right"),
        color = "steelblue",
        weight = px(4)
      )
    ),
    values = c(49.95, 33.33)
  )
```

You can opt to color entire rows or columns (or both, should you want to) with those specific keywords in the `targets` argument. For the `49.95` value we will style the entire row and with `33.33` the entire column will get the same styling.

```
gt_tbl %>%
  tab_style_body(
    style = cell_fill(color = "lightblue"),
    values = 49.95,
    targets = "row"
  ) %>%
  tab_style_body(
    style = cell_fill(color = "lightblue"),
    values = 33.33,
    targets = "column"
  )
```

In a minor variation to the prior example, it's possible to extend the styling to other locations, or, entirely project the styling elsewhere. This is done with the `extents` argument. Valid keywords that can be included in the vector are: `"body"` (the default) and `"stub"`. Let's take the previous example and extend the styling of the row into the stub.

```
gt_tbl %>%
  tab_style_body(
    style = cell_fill(color = "lightblue"),
    values = 49.95,
    targets = "row",
    extents = c("body", "stub")
  ) %>%
  tab_style_body(
    style = cell_fill(color = "lightblue"),
    values = 33.33,
    targets = "column"
  )
```

We can also use the `pattern` argument to target cell values in `character`-based columns. The `"fctr"` column is skipped because it is in fact a factor-based column.

```
gt_tbl %>%
  tab_style_body(
    style = cell_fill(color = "green"),
    pattern = "ne|na"
  )
```

For the most flexibility in targeting, it's best to use the `fn` argument. The function you give to `fn` will be invoked separately on all cells so the `columns` argument of `tab_style_body()` might be useful to limit which cells should be evaluated. For this next example, the supplied function should only be used on numeric values and we can make sure of this by using `columns = where(is.numeric)`.

```
gt_tbl %>%
  tab_style_body(
    columns = where(is.numeric),
    style = cell_fill(color = "pink"),
    fn = function(x) x >= 0 && x < 50
  )
```

Function ID

2-11

See Also

Other part creation/modification functions: `tab_caption()`, `tab_footnote()`, `tab_header()`, `tab_info()`, `tab_options()`, `tab_row_group()`, `tab_source_note()`, `tab_spanner_delim()`, `tab_spanner()`, `tab_stub_indent()`, `tab_stubhead()`, `tab_style()`

`test_image` *Generate a path to a test image*

Description

Two test images are available within the **gt** package. Both contain the same imagery (sized at 200px by 200px) but one is a PNG file while the other is an SVG file. This function is most useful when paired with `local_image()` since we test various sizes of the test image within that function.

Usage

```
test_image(type = c("png", "svg"))
```

Arguments

`type`	The type of the image, which can either be `png` (the default) or `svg`.

Value

A character vector with a single path to an image file.

Function ID

8-4

See Also

Other image addition functions: `ggplot_image()`, `local_image()`, `web_image()`

`text_transform` *Perform targeted text transformation with a function*

Description

Perform targeted text transformation with a function

Usage

```
text_transform(data, locations, fn)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`locations`	The cell or set of cells to be associated with the text transformation. Only the `cells_body()`, `cells_stub()`, `cells_column_labels()`, and `cells_row_groups()` helper functions can be used here. We can enclose several of these calls within a `list()` if we wish to make the transformation happen at different locations.
`fn`	The function to use for text transformation.

Value

An object of class gt_tbl.

Examples

Use exibble to create a **gt** table. transform the formatted text in the num column using a function supplied to text_transform() (via the fn argument). Note that the x in the fn = function (x) part is a formatted vector of column values from the num column.

```
exibble %>%
  dplyr::select(num, char, currency) %>%
  dplyr::slice(1:4) %>%
  gt() %>%
  fmt_number(columns = num) %>%
  fmt_currency(columns = currency) %>%
  text_transform(
    locations = cells_body(columns = num),
    fn = function(x) {
      paste0(
        x, " (",
        dplyr::case_when(
          x > 20   ~ "large",
          x <= 20  ~ "small"),
        ")")
    }
  )
```

Function ID

3-22

See Also

Other data formatting functions: data_color(), fmt_bytes(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_duration(), fmt_engineering(), fmt_fraction(), fmt_integer(), fmt_markdown(), fmt_number(), fmt_partsper(), fmt_passthrough(), fmt_percent(), fmt_roman(), fmt_scientific(), fmt_time(), fmt(), sub_large_vals(), sub_missing(), sub_small_vals(), sub_values(), sub_zero()

---

vec_fmt_bytes *Format a vector as values in terms of bytes*

Description

With numeric values in a vector, we can transform each into byte values with human readable units. The `vec_fmt_bytes()` function allows for the formatting of byte sizes to either of two common representations: (1) with decimal units (powers of 1000, examples being `"kB"` and `"MB"`), and (2) with binary units (powers of 1024, examples being `"KiB"` and `"MiB"`).

It is assumed the input numeric values represent the number of bytes and automatic truncation of values will occur. The numeric values will be scaled to be in the range of 1 to <1000 and then decorated with the correct unit symbol according to the standard chosen. For more control over the formatting of byte sizes, we can use the following options:

- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol
- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
vec_fmt_bytes(
  x,
  standard = c("decimal", "binary"),
  decimals = 1,
  n_sigfig = NULL,
  drop_trailing_zeros = TRUE,
  drop_trailing_dec_mark = TRUE,
  use_seps = TRUE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  incl_space = TRUE,
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x`	A numeric vector.
`standard`	The way to express large byte sizes.
`decimals`	An option to specify the exact number of decimal places to use. The default number of decimal places is `1`.
`n_sigfig`	A option to format numbers to *n* significant figures. By default, this is `NULL` and thus number values will be formatted according to the number of decimal places set via `decimals`. If opting to format according to the rules of significant figures,

n_sigfig must be a number greater than or equal to 1. Any values passed to the decimals and drop_trailing_zeros arguments will be ignored.

drop_trailing_zeros
: A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).

drop_trailing_dec_mark
: A logical value that determines whether decimal marks should always appear even if there are no decimal digits to display after formatting (e.g, 23 becomes 23.). The default for this is TRUE, which means that trailing decimal marks are not shown.

use_seps
: An option to use digit group separators. The type of digit group separator is set by sep_mark and overridden if a locale ID is provided to locale. This setting is TRUE by default.

pattern
: A formatting pattern that allows for decoration of the formatted value. The value itself is represented by {x} and all other characters are taken to be string literals.

sep_mark
: The mark to use as a separator between groups of digits (e.g., using sep_mark = "," with 1000 would result in a formatted value of 1,000).

dec_mark
: The character to use as a decimal mark (e.g., using dec_mark = "," with 0.152 would result in a formatted value of 0,152).

force_sign
: Should the positive sign be shown for positive numbers (effectively showing a sign for all numbers except zero)? If so, use TRUE for this option. The default is FALSE, where only negative numbers will display a minus sign.

incl_space
: An option for whether to include a space between the value and the units. The default of TRUE uses a space character for separation.

locale
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include "en" for English (United States) and "fr" for French (France). The use of a valid locale ID will override any values provided in sep_mark and dec_mark. We can use the info_locales() function as a useful reference for all of the locales that are supported.

output
: The output style of the resulting character vector. This can either be "auto" (the default), "plain", "html", "latex", "rtf", or "word". In **knitr** rendering (i.e., Quarto or R Markdown), the "auto" option will choose the correct output value

Value

A character vector.

Examples

Let's create a numeric vector for the next few examples:

```
num_vals <- c(3.24294e14, 8, 1362902, -59027, NA)
```

Using vec_fmt_bytes() with the default options will create a character vector with values in bytes. Any NA values remain as NA values. The rendering context will be autodetected unless specified in the output argument (here, it is of the "plain" output type).

```
vec_fmt_bytes(num_vals)
```

```
#> [1] "324.3 TB" "8 B" "1.4 MB" "-59 kB" "NA"
```

We can change the number of decimal places with the `decimals` option:

```
vec_fmt_bytes(num_vals, decimals = 2)
```

```
#> [1] "324.29 TB" "8 B" "1.36 MB" "-59.03 kB" "NA"
```

If we are formatting for a different locale, we could supply the locale ID and **gt** will handle any locale-specific formatting options:

```
vec_fmt_bytes(num_vals, locale = "fi")
```

```
#> [1] "324,3 TB" "8 B" "1,4 MB" "-59 kB" "NA"
```

Should you need to have positive and negative signs on each of the output values, use `force_sign = TRUE`:

```
vec_fmt_bytes(num_vals, force_sign = TRUE)
```

```
#> [1] "+324.3 TB" "+8 B" "+1.4 MB" "-59 kB" "NA"
```

As a last example, one can wrap the values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_bytes(num_vals, pattern = "[{x}]")
```

```
#> [1] "[324.3 TB]" "[8 B]" "[1.4 MB]" "[-59 kB]" "NA"
```

Function ID

14-10

See Also

Other vector formatting functions: `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_scientific()`, `vec_fmt_time()`

`vec_fmt_currency` *Format a vector as currency values*

Description

With numeric values in a vector, we can perform currency-based formatting. This function supports both automatic formatting with a three-letter or numeric currency code. We can also specify a custom currency that is formatted according to the output context with the `currency()` helper function. We have fine control over the conversion from numeric values to currency values, where we could take advantage of the following options:

- the currency: providing a currency code or common currency name will procure the correct currency symbol and number of currency subunits; we could also use the `currency()` helper function to specify a custom currency
- currency symbol placement: the currency symbol can be placed before or after the values
- decimals/subunits: choice of the number of decimal places, and a choice of the decimal symbol, and an option on whether to include or exclude the currency subunits (decimal portion)
- negative values: choice of a negative sign or parentheses for values less than zero
- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- scaling: we can choose to scale targeted values by a multiplier value
- large-number suffixing: larger figures (thousands, millions, etc.) can be autoscaled and decorated with the appropriate suffixes
- pattern: option to use a text pattern for decoration of the formatted currency values
- locale-based formatting: providing a locale ID will result in currency formatting specific to the chosen locale

We can use the `info_currencies()` function for a useful reference on all of the possible inputs to the `currency` argument.

Usage

```
vec_fmt_currency(
  x,
  currency = "USD",
  use_subunits = TRUE,
  decimals = NULL,
  drop_trailing_dec_mark = TRUE,
  use_seps = TRUE,
  accounting = FALSE,
  scale_by = 1,
  suffixing = FALSE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
```

```
  force_sign = FALSE,
  placement = "left",
  incl_space = FALSE,
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x` A numeric vector.

`currency` The currency to use for the numeric value. This input can be supplied as a 3-letter currency code (e.g., "USD" for U.S. Dollars, "EUR" for the Euro currency). Use `info_currencies()` to get an information table with all of the valid currency codes and examples of each. Alternatively, we can provide a common currency name (e.g., "dollar", "pound", "yen", etc.) to simplify the process. Use `info_currencies()` with the `type == "symbol"` option to view an information table with all of the supported currency symbol names along with examples.

We can also use the `currency()` helper function to specify a custom currency, where the string could vary across output contexts. For example, using `currency(html = "ƒ", default = "f")` would give us a suitable glyph for the Dutch guilder in an HTML output table, and it would simply be the letter "f" in all other output contexts). Please note that `decimals` will default to `2` when using the `currency()` helper function.

If nothing is provided to `currency` then "USD" (U.S. dollars) will be used.

`use_subunits` An option for whether the subunits portion of a currency value should be displayed. By default, this is `TRUE`.

`decimals` An option to specify the exact number of decimal places to use. The default number of decimal places is `2`.

`drop_trailing_dec_mark` A logical value that determines whether decimal marks should always appear even if there are no decimal digits to display after formatting (e.g, `23` becomes `23.`). The default for this is `TRUE`, which means that trailing decimal marks are not shown.

`use_seps` An option to use digit group separators. The type of digit group separator is set by `sep_mark` and overridden if a locale ID is provided to `locale`. This setting is `TRUE` by default.

`accounting` An option to use accounting style for values. With `FALSE` (the default), negative values will be shown with a minus sign. Using `accounting = TRUE` will put negative values in parentheses.

`scale_by` A value to scale the input. The default is `1.0`. All numeric values will be multiplied by this value first before undergoing formatting. This value will be ignored if using any of the `suffixing` options (i.e., where `suffixing` is not set to `FALSE`).

`suffixing` An option to scale and apply suffixes to larger numbers (e.g., `1924000` can be transformed to `1.92M`). This option can accept a logical value, where `FALSE` (the default) will not perform this transformation and `TRUE` will apply thousands (`K`), millions (M), billions (B), and trillions (T) suffixes after automatic value scaling. We can also specify which symbols to use for each of the value ranges by using a character vector of the preferred symbols to replace the defaults (e.g., c("k", "Ml", "Bn", "Tr")).

Including NA values in the vector will ensure that the particular range will either not be included in the transformation (e.g, c(NA, "M", "B", "T") won't modify numbers in the thousands range) or the range will inherit a previous suffix (e.g., with c("K", "M", NA, "T"), all numbers in the range of millions and billions will be in terms of millions).

Any use of suffixing (where it is not set expressly as FALSE) means that any value provided to scale_by will be ignored.

pattern
: A formatting pattern that allows for decoration of the formatted value. The value itself is represented by {x} and all other characters are taken to be string literals.

sep_mark
: The mark to use as a separator between groups of digits (e.g., using sep_mark = "," with 1000 would result in a formatted value of 1,000).

dec_mark
: The character to use as a decimal mark (e.g., using dec_mark = "," with 0.152 would result in a formatted value of 0,152).

force_sign
: Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use TRUE for this option. The default is FALSE, where only negative numbers will display a minus sign. This option is disregarded when using accounting notation with accounting = TRUE.

placement
: The placement of the currency symbol. This can be either be left (the default) or right.

incl_space
: An option for whether to include a space between the value and the currency symbol. The default is to not introduce a space character.

locale
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include "en" for English (United States) and "fr" for French (France). The use of a valid locale ID will override any values provided in sep_mark and dec_mark. We can use the info_locales() function as a useful reference for all of the locales that are supported.

output
: The output style of the resulting character vector. This can either be "auto" (the default), "plain", "html", "latex", "rtf", or "word". In **knitr** rendering (i.e., Quarto or R Markdown), the "auto" option will choose the correct output value

Value

A character vector.

Examples

Let's create a numeric vector for the next few examples:

```
num_vals <- c(5.2, 8.65, 0, -5.3, NA)
```

Using `vec_fmt_currency()` with the default options will create a character vector where the numeric values have been transformed to U.S. Dollars ("USD"). Furthermore, the rendering context will be autodetected unless specified in the output argument (here, it is of the "plain" output type).

```
vec_fmt_currency(num_vals)
```

```
#> [1] "$5.20" "$8.65" "$0.00" "-$5.30" "NA"
```

We can supply a currency code to the currency argument. Let's use British Pounds through currency = "GBP":

```
vec_fmt_currency(num_vals, currency = "GBP")
```

```
#> [1] "GBP5.20" "GBP8.65" "GBP0.00" "-GBP5.30" "NA"
```

If we are formatting for a different locale, we could supply the locale ID and let **gt** handle all locale-specific formatting options:

```
vec_fmt_currency(num_vals, currency = "EUR", locale = "fr")
```

```
#> [1] "EUR5,20" "EUR8,65" "EUR0,00" "-EUR5,30" "NA"
```

There are many options for formatting values. Perhaps you need to have explicit positive and negative signs? Use force_sign = TRUE for that.

```
vec_fmt_currency(num_vals, force_sign = TRUE)
```

```
#> [1] "+$5.20" "+$8.65" "$0.00" "-$5.30" "NA"
```

As a last example, one can wrap the values in a pattern with the pattern argument. Note here that NA values won't have the pattern applied.

```
vec_fmt_currency(num_vals, pattern = "`{x}`")
```

```
#> [1] "`$5.20`" "`$8.65`" "`$0.00`" "`-$5.30`" "NA"
```

Function ID

14-8

See Also

Other vector formatting functions: vec_fmt_bytes(), vec_fmt_datetime(), vec_fmt_date(), vec_fmt_duration(), vec_fmt_engineering(), vec_fmt_fraction(), vec_fmt_integer(), vec_fmt_markdown(), vec_fmt_number(), vec_fmt_partsper(), vec_fmt_percent(), vec_fmt_roman(), vec_fmt_scientific(), vec_fmt_time()

`vec_fmt_date` *Format a vector as date values*

Description

Format vector values to date values using one of 41 preset date styles. Input can be in the form of `POSIXt` (i.e., datetimes), the `Date` type, or `character` (must be in the ISO 8601 form of `YYYY-MM-DD HH:MM:SS` or `YYYY-MM-DD`).

Usage

```
vec_fmt_date(
  x,
  date_style = "iso",
  pattern = "{x}",
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x`	A numeric vector.
`date_style`	The date style to use. By default this is `"iso"` which corresponds to ISO 8601 date formatting. The other date styles can be viewed using `info_date_style()`.
`pattern`	A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.
`locale`	An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported.
`output`	The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Formatting with the `date_style` argument

We need to supply a preset date style to the `date_style` argument. The date styles are numerous and can handle localization to any supported locale. A large segment of date styles are termed flexible date formats and this means that their output will adapt to any `locale` provided. That feature makes the flexible date formats a better option for locales other than `"en"` (the default locale).

The following table provides a listing of all date styles and their output values (corresponding to an input date of `2000-02-29`).

	Date Style	Output	Notes
1	"iso"	"2000-02-29"	ISO 8601
2	"wday_month_day_year"	"Tuesday, February 29, 2000"	
3	"wd_m_day_year"	"Tue, Feb 29, 2000"	
4	"wday_day_month_year"	"Tuesday 29 February 2000"	
5	"month_day_year"	"February 29, 2000"	
6	"m_day_year"	"Feb 29, 2000"	
7	"day_m_year"	"29 Feb 2000"	
8	"day_month_year"	"29 February 2000"	
9	"day_month"	"29 February"	
10	"day_m"	"29 Feb"	
11	"year"	"2000"	
12	"month"	"February"	
13	"day"	"29"	
14	"year.mn.day"	"2000/02/29"	
15	"y.mn.day"	"00/02/29"	
16	"year_week"	"2000-W09"	
17	"year_quarter"	"2000-Q1"	
18	"yMd"	"2/29/2000"	flexible
19	"yMEd"	"Tue, 2/29/2000"	flexible
20	"yMMM"	"Feb 2000"	flexible
21	"yMMMM"	"February 2000"	flexible
22	"yMMMd"	"Feb 29, 2000"	flexible
23	"yMMMEd"	"Tue, Feb 29, 2000"	flexible
24	"GyMd"	"2/29/2000 A"	flexible
25	"GyMMMd"	"Feb 29, 2000 AD"	flexible
26	"GyMMMEd"	"Tue, Feb 29, 2000 AD"	flexible
27	"yM"	"2/2000"	flexible
28	"Md"	"2/29"	flexible
29	"MEd"	"Tue, 2/29"	flexible
30	"MMMd"	"Feb 29"	flexible
31	"MMMEd"	"Tue, Feb 29"	flexible
32	"MMMMd"	"February 29"	flexible
33	"GyMMM"	"Feb 2000 AD"	flexible
34	"yQQQ"	"Q1 2000"	flexible
35	"yQQQQ"	"1st quarter 2000"	flexible
36	"Gy"	"2000 AD"	flexible
37	"y"	"2000"	flexible
38	"M"	"2"	flexible
39	"MMM"	"Feb"	flexible
40	"d"	"29"	flexible
41	"Ed"	"29 Tue"	flexible

We can use the `info_date_style()` within the console to view a similar table of date styles with example output.

Examples

Let's create a character vector of dates in the ISO-8601 format for the next few examples:

```
str_vals <- c("2022-06-13", "2019-01-25", "2015-03-23", NA)
```

Using `vec_fmt_date()` (here with the `"wday_month_day_year"` date style) will result in a character vector of formatted dates. Any `NA` values remain as `NA` values. The rendering context will be autodetected unless specified in the `output` argument (here, it is of the `"plain"` output type).

```
vec_fmt_date(str_vals, date_style = "wday_month_day_year")
```

```
#> [1] "Monday, June 13, 2022" "Friday, January 25, 2019"
#> [3] "Monday, March 23, 2015" NA
```

We can choose from any of 41 different date formatting styles. Many of these styles are flexible, meaning that the structure of the format will adapt to different locales. Let's use the `"yMMMEd"` date style to demonstrate this (first in the default locale of `"en"`):

```
vec_fmt_date(str_vals, date_style = "yMMMEd")
```

```
#> [1] "Mon, Jun 13, 2022" "Fri, Jan 25, 2019" "Mon, Mar 23, 2015" NA
```

Let's perform the same type of formatting in the French (`"fr"`) locale:

```
vec_fmt_date(str_vals, date_style = "yMMMEd", locale = "fr")
```

```
#> [1] "lun. 13 juin 2022" "ven. 25 janv. 2019" "lun. 23 mars 2015" NA
```

We can always use `info_date_style()` to call up an info table that serves as a handy reference to all of the `date_style` options.

As a last example, one can wrap the date values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_date(str_vals, pattern = "Date: {x}")
```

```
#> [1] "Date: 2022-06-13" "Date: 2019-01-25" "Date: 2015-03-23" NA
```

Function ID

14-11

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_scientific()`, `vec_fmt_time()`

vec_fmt_datetime *Format a vector as datetime values*

Description

Format values in a vector to datetime values using either presets for the date and time components or a formatting directive (this can either use a *CLDR* datetime pattern or `strptime` formatting). Input can be in the form of `POSIXt` (i.e., datetimes), the `Date` type, or `character` (must be in the ISO 8601 form of `YYYY-MM-DD HH:MM:SS` or `YYYY-MM-DD`).

Usage

```
vec_fmt_datetime(
  x,
  date_style = "iso",
  time_style = "iso",
  sep = " ",
  format = NULL,
  tz = NULL,
  pattern = "{x}",
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x`	A numeric vector.
`date_style`	The date style to use. By default this is `"iso"` which corresponds to ISO 8601 date formatting. The other date styles can be viewed using `info_date_style()`.
`time_style`	The time style to use. By default this is `"iso"` which corresponds to how times are formatted within ISO 8601 datetime values. The other time styles can be viewed using `info_time_style()`.
`sep`	The separator string to use between the date and time components. By default, this is a single space character (`" "`). Only used when not specifying a `format` code.
`format`	An optional format code used for generating custom dates/times. If used then the arguments governing preset styles (`date_style` and `time_style`) will be ignored in favor of formatting via the `format` string.
`tz`	The time zone for printing dates/times (i.e., the output). The default of `NULL` will preserve the time zone of the input data in the output. If providing a time zone, it must be one that is recognized by the user's operating system (a vector of all valid `tz` values can be produced with `OlsonNames()`).
`pattern`	A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

`locale` An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported.

`output` The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Formatting with the `date_style` argument

We can supply a preset date style to the `date_style` argument to separately handle the date portion of the output. The date styles are numerous and can handle localization to any supported locale. A large segment of date styles are termed flexible date formats and this means that their output will adapt to any `locale` provided. That feature makes the flexible date formats a better option for locales other than `"en"` (the default locale).

The following table provides a listing of all date styles and their output values (corresponding to an input date of `2000-02-29`).

	Date Style	Output	Notes
1	`"iso"`	`"2000-02-29"`	ISO 8601
2	`"wday_month_day_year"`	`"Tuesday, February 29, 2000"`	
3	`"wd_m_day_year"`	`"Tue, Feb 29, 2000"`	
4	`"wday_day_month_year"`	`"Tuesday 29 February 2000"`	
5	`"month_day_year"`	`"February 29, 2000"`	
6	`"m_day_year"`	`"Feb 29, 2000"`	
7	`"day_m_year"`	`"29 Feb 2000"`	
8	`"day_month_year"`	`"29 February 2000"`	
9	`"day_month"`	`"29 February"`	
10	`"day_m"`	`"29 Feb"`	
11	`"year"`	`"2000"`	
12	`"month"`	`"February"`	
13	`"day"`	`"29"`	
14	`"year.mn.day"`	`"2000/02/29"`	
15	`"y.mn.day"`	`"00/02/29"`	
16	`"year_week"`	`"2000-W09"`	
17	`"year_quarter"`	`"2000-Q1"`	
18	`"yMd"`	`"2/29/2000"`	flexible
19	`"yMEd"`	`"Tue, 2/29/2000"`	flexible
20	`"yMMM"`	`"Feb 2000"`	flexible
21	`"yMMMM"`	`"February 2000"`	flexible
22	`"yMMMd"`	`"Feb 29, 2000"`	flexible
23	`"yMMMEd"`	`"Tue, Feb 29, 2000"`	flexible
24	`"GyMd"`	`"2/29/2000 A"`	flexible

25	"GyMMMd"	"Feb 29, 2000 AD"	flexible
26	"GyMMMEd"	"Tue, Feb 29, 2000 AD"	flexible
27	"yM"	"2/2000"	flexible
28	"Md"	"2/29"	flexible
29	"MEd"	"Tue, 2/29"	flexible
30	"MMMd"	"Feb 29"	flexible
31	"MMMEd"	"Tue, Feb 29"	flexible
32	"MMMMd"	"February 29"	flexible
33	"GyMMM"	"Feb 2000 AD"	flexible
34	"yQQQ"	"Q1 2000"	flexible
35	"yQQQQ"	"1st quarter 2000"	flexible
36	"Gy"	"2000 AD"	flexible
37	"y"	"2000"	flexible
38	"M"	"2"	flexible
39	"MMM"	"Feb"	flexible
40	"d"	"29"	flexible
41	"Ed"	"29 Tue"	flexible

We can use the `info_date_style()` within the console to view a similar table of date styles with example output.

Formatting with the `time_style` argument

We can supply a preset time style to the `time_style` argument to separately handle the time portion of the output. There are many time styles and all of them can handle localization to any supported locale. Many of the time styles are termed flexible time formats and this means that their output will adapt to any `locale` provided. That feature makes the flexible time formats a better option for locales other than `"en"` (the default locale).

The following table provides a listing of all time styles and their output values (corresponding to an input time of `14:35:00`). It is noted which of these represent 12- or 24-hour time. Some of the flexible formats (those that begin with `"E"`) include the the day of the week. Keep this in mind when pairing such `time_style` values with a `date_style` so as to avoid redundant or repeating information.

	Time Style	Output	Notes
1	"iso"	"14:35:00"	ISO 8601, 24h
2	"iso-short"	"14:35"	ISO 8601, 24h
3	"h_m_s_p"	"2:35:00 PM"	12h
4	"h_m_p"	"2:35 PM"	12h
5	"h_p"	"2 PM"	12h
6	"Hms"	"14:35:00"	flexible, 24h
7	"Hm"	"14:35"	flexible, 24h
8	"H"	"14"	flexible, 24h
9	"EHm"	"Thu 14:35"	flexible, 24h
10	"EHms"	"Thu 14:35:00"	flexible, 24h
11	"Hmsv"	"14:35:00 GMT+00:00"	flexible, 24h
12	"Hmv"	"14:35 GMT+00:00"	flexible, 24h

13	`"hms"`	`"2:35:00 PM"`	flexible, 12h
14	`"hm"`	`"2:35 PM"`	flexible, 12h
15	`"h"`	`"2 PM"`	flexible, 12h
16	`"Ehm"`	`"Thu 2:35 PM"`	flexible, 12h
17	`"Ehms"`	`"Thu 2:35:00 PM"`	flexible, 12h
18	`"EBhms"`	`"Thu 2:35:00 in the afternoon"`	flexible, 12h
19	`"Bhms"`	`"2:35:00 in the afternoon"`	flexible, 12h
20	`"EBhm"`	`"Thu 2:35 in the afternoon"`	flexible, 12h
21	`"Bhm"`	`"2:35 in the afternoon"`	flexible, 12h
22	`"Bh"`	`"2 in the afternoon"`	flexible, 12h
23	`"hmsv"`	`"2:35:00 PM GMT+00:00"`	flexible, 12h
24	`"hmv"`	`"2:35 PM GMT+00:00"`	flexible, 12h
25	`"ms"`	`"35:00"`	flexible

We can use the `info_time_style()` within the console to view a similar table of time styles with example output.

Formatting with a *CLDR* datetime pattern

We can use a *CLDR* datetime pattern with the `format` argument to create a highly customized and locale-aware output. This is a character string that consists of two types of elements:

- Pattern fields, which repeat a specific pattern character one or more times. These fields are replaced with date and time data when formatting. The character sets of `A-Z` and `a-z` are reserved for use as pattern characters.
- Literal text, which is output verbatim when formatting. This can include:
  - Any characters outside the reserved character sets, including spaces and punctuation.
  - Any text between single vertical quotes (e.g., `'text'`).
  - Two adjacent single vertical quotes (''), which represent a literal single quote, either inside or outside quoted text.

The number of pattern fields is quite sizable so let's first look at how some *CLDR* datetime patterns work. We'll use the datetime string `"2018-07-04T22:05:09.2358(America/Vancouver)"` for all of the examples that follow.

- `"mm/dd/y"` -> `"05/04/2018"`
- `"EEEE, MMMM d, y"` -> `"Wednesday, July 4, 2018"`
- `"MMM d E"` -> `"Jul 4 Wed"`
- `"HH:mm"` -> `"22:05"`
- `"h:mm a"` -> `"10:05 PM"`
- `"EEEE, MMMM d, y 'at' h:mm a"` -> `"Wednesday, July 4, 2018 at 10:05 PM"`

Here are the individual pattern fields:

Year:

Calendar Year:
This yields the calendar year, which is always numeric. In most cases the length of the "y" field specifies the minimum number of digits to display, zero-padded as necessary. More digits will be displayed if needed to show the full year. There is an exception: "yy" gives use just the two low-order digits of the year, zero-padded as necessary. For most use cases, "y" or "yy" should be good enough.

Field Patterns	Output
"y"	"2018"
"yy"	"18"
"yyy" to "yyyyyyyyy"	"2018" to "000002018"

Year in the Week in Year Calendar:
This is the year in 'Week of Year' based calendars in which the year transition occurs on a week boundary. This may differ from calendar year "y" near a year transition. This numeric year designation is used in conjunction with pattern character "w" in the ISO year-week calendar as defined by ISO 8601.

Field Patterns	Output
"Y"	"2018"
"YY"	"18"
"YYY" to "YYYYYYYYY"	"2018" to "000002018"

Quarter:

Quarter of the Year: formatting and standalone versions:
The quarter names are identified numerically, starting at 1 and ending at 4. Quarter names may vary along two axes: the width and the context. The context is either 'formatting' (taken as a default), which the form used within a complete date format string, or, 'standalone', the form for date elements used independently (such as in calendar headers). The standalone form may be used in any other date format that shares the same form of the name. Here, the formatting form for quarters of the year consists of some run of "Q" values whereas the standalone form uses "q".

Field Patterns	Output	Notes
"Q"/"q"	"3"	Numeric, one digit
"QQ"/"qq"	"03"	Numeric, two digits (zero padded)
"QQQ"/"qqq"	"Q3"	Abbreviated
"QQQQ"/"qqqq"	"3rd quarter"	Wide
"QQQQQ"/"qqqqq"	"3"	Narrow

Month:

Month: formatting and standalone versions:
The month names are identified numerically, starting at 1 and ending at 12. Month names may vary along two axes: the width and the context. The context is either 'formatting' (taken as a default), which the form used within a complete date format string, or, 'standalone', the form for date elements used independently (such as in calendar headers). The standalone form may

be used in any other date format that shares the same form of the name. Here, the formatting form for months consists of some run of "M" values whereas the standalone form uses "L".

Field Patterns	Output	Notes
"M"/"L"	"7"	Numeric, minimum digits
"MM"/"LL"	"07"	Numeric, two digits (zero padded)
"MMM"/"LLL"	"Jul"	Abbreviated
"MMMM"/"LLLL"	"July"	Wide
"MMMMM"/"LLLLL"	"J"	Narrow

Week:

Week of Year:
Values calculated for the week of year range from 1 to 53. Week 1 for a year is the first week that contains at least the specified minimum number of days from that year. Weeks between week 1 of one year and week 1 of the following year are numbered sequentially from 2 to 52 or 53 (if needed).
There are two available field lengths. Both will display the week of year value but the "ww" width will always show two digits (where weeks 1 to 9 are zero padded).

Field Patterns	Output	Notes
"w"	"27"	Minimum digits
"ww"	"27"	Two digits (zero padded)

Week of Month:
The week of a month can range from 1 to 5. The first day of every month always begins at week 1 and with every transition into the beginning of a week, the week of month value is incremented by 1.

Field Pattern	Output
"W"	"1"

Day:

Day of Month:
The day of month value is always numeric and there are two available field length choices in its formatting. Both will display the day of month value but the "dd" formatting will always show two digits (where days 1 to 9 are zero padded).

Field Patterns	Output	Notes
"d"	"4"	Minimum digits
"dd"	"04"	Two digits, zero padded

Day of Year:
The day of year value ranges from 1 (January 1) to either 365 or 366 (December 31), where the higher value of the range indicates that the year is a leap year (29 days in February, instead of 28). The field length specifies the minimum number of digits, with zero-padding as necessary.

Field Patterns	Output	Notes
`"D"`	`"185"`	
`"DD"`	`"185"`	Zero padded to minimum width of 2
`"DDD"`	`"185"`	Zero padded to minimum width of 3

Day of Week in Month:
The day of week in month returns a numerical value indicating the number of times a given weekday had occurred in the month (e.g., '2nd Monday in March'). This conveniently resolves to predicable case structure where ranges of day of the month values return predictable day of week in month values:

- days 1 - 7 -> 1
- days 8 - 14 -> 2
- days 15 - 21 -> 3
- days 22 - 28 -> 4
- days 29 - 31 -> 5

Field Pattern	Output
`"F"`	`"1"`

Modified Julian Date:
The modified version of the Julian date is obtained by subtracting 2,400,000.5 days from the Julian date (the number of days since January 1, 4713 BC). This essentially results in the number of days since midnight November 17, 1858. There is a half day offset (unlike the Julian date, the modified Julian date is referenced to midnight instead of noon).

Field Patterns	Output
`"g"` to `"ggggggggg"`	`"58303"` -> `"000058303"`

Weekday:

Day of Week Name:
The name of the day of week is offered in four different widths.

Field Patterns	Output	Notes
`"E"`, `"EE"`, or `"EEE"`	`"Wed"`	Abbreviated
`"EEEE"`	`"Wednesday"`	Wide
`"EEEEE"`	`"W"`	Narrow
`"EEEEEE"`	`"We"`	Short

Periods:

AM/PM Period of Day:
This denotes before noon and after noon time periods. May be upper or lowercase depending on the locale and other options. The wide form may be the same as the short form if the 'real' long form (e.g. 'ante meridiem') is not customarily used. The narrow form must be unique, unlike some other fields.

Field Patterns	Output	Notes
"a", "aa", or "aaa"	"PM"	Abbreviated
"aaaa"	"PM"	Wide
"aaaaa"	"p"	Narrow

AM/PM Period of Day Plus Noon and Midnight:
Provide AM and PM as well as phrases for exactly noon and midnight. May be upper or lowercase depending on the locale and other options. If the locale doesn't have the notion of a unique 'noon' (i.e., 12:00), then the PM form may be substituted. A similar behavior can occur for 'midnight' (00:00) and the AM form. The narrow form must be unique, unlike some other fields.
(a) `input_midnight`: "2020-05-05T00:00:00" (b) `input_noon`: "2020-05-05T12:00:00"

Field Patterns	Output	Notes
"b", "bb", or "bbb"	(a) "midnight"	Abbreviated
	(b) "noon"	
"bbbb"	(a) "midnight"	Wide
	(b) "noon"	
"bbbbb"	(a) "mi"	Narrow
	(b) "n"	

Flexible Day Periods:
Flexible day periods denotes things like 'in the afternoon', 'in the evening', etc., and the flexibility comes from a locale's language and script. Each locale has an associated rule set that specifies when the day periods start and end for that locale.
(a) `input_morning`: "2020-05-05T00:08:30" (b) `input_afternoon`: "2020-05-05T14:00:00"

Field Patterns	Output	Notes
"B", "BB", or "BBB"	(a) "in the morning"	Abbreviated
	(b) "in the afternoon"	
"BBBB"	(a) "in the morning"	Wide
	(b) "in the afternoon"	
"BBBBB"	(a) "in the morning"	Narrow
	(b) "in the afternoon"	

Hours, Minutes, and Seconds:

Hour 0-23:
Hours from `0` to `23` are for a standard 24-hour clock cycle (midnight plus 1 minute is `00:01`) when using "HH" (which is the more common width that indicates zero-padding to 2 digits).
Using "2015-08-01T08:35:09":

Field Patterns	Output	Notes
"H"	"8"	Numeric, minimum digits
"HH"	"08"	Numeric, 2 digits (zero padded)

Hour 1-12:
Hours from `1` to `12` are for a standard 12-hour clock cycle (midnight plus 1 minute is `12:01`) when using `"hh"` (which is the more common width that indicates zero-padding to 2 digits).
Using `"2015-08-01T08:35:09"`:

Field Patterns	Output	Notes
`"h"`	`"8"`	Numeric, minimum digits
`"hh"`	`"08"`	Numeric, 2 digits (zero padded)

Hour 1-24:
Using hours from `1` to `24` is a less common way to express a 24-hour clock cycle (midnight plus 1 minute is `24:01`) when using `"kk"` (which is the more common width that indicates zero-padding to 2 digits).
Using `"2015-08-01T08:35:09"`:

Field Patterns	Output	Notes
`"k"`	`"9"`	Numeric, minimum digits
`"kk"`	`"09"`	Numeric, 2 digits (zero padded)

Hour 0-11:
Using hours from `0` to `11` is a less common way to express a 12-hour clock cycle (midnight plus 1 minute is `00:01`) when using `"KK"` (which is the more common width that indicates zero-padding to 2 digits).
Using `"2015-08-01T08:35:09"`:

Field Patterns	Output	Notes
`"K"`	`"7"`	Numeric, minimum digits
`"KK"`	`"07"`	Numeric, 2 digits (zero padded)

Minute:
The minute of the hour which can be any number from `0` to `59`. Use `"m"` to show the minimum number of digits, or `"mm"` to always show two digits (zero-padding, if necessary).

Field Patterns	Output	Notes
`"m"`	`"5"`	Numeric, minimum digits
`"mm"`	`"06"`	Numeric, 2 digits (zero padded)

Seconds:
The second of the minute which can be any number from `0` to `59`. Use `"s"` to show the minimum number of digits, or `"ss"` to always show two digits (zero-padding, if necessary).

Field Patterns	Output	Notes
`"s"`	`"9"`	Numeric, minimum digits
`"ss"`	`"09"`	Numeric, 2 digits (zero padded)

Fractional Second:
The fractional second truncates (like other time fields) to the width requested (i.e., count of letters). So using pattern "SSSS" will display four digits past the decimal (which, incidentally, needs to be added manually to the pattern).

Field Patterns	Output
"S" to "SSSSSSSSS"	"2" -> "235000000"

Milliseconds Elapsed in Day:
There are 86,400,000 milliseconds in a day and the "A" pattern will provide the whole number. The width can go up to nine digits with "AAAAAAAAA" and these higher field widths will result in zero padding if necessary.
Using "2011-07-27T00:07:19.7223":

Field Patterns	Output
"A" to "AAAAAAAAA"	"439722" -> "000439722"

Era:

The Era Designator:
This provides the era name for the given date. The Gregorian calendar has two eras: AD and BC. In the AD year numbering system, AD 1 is immediately preceded by 1 BC, with nothing in between them (there was no year zero).

Field Patterns	Output	Notes
"G", "GG", or "GGG"	"AD"	Abbreviated
"GGGG"	"Anno Domini"	Wide
"GGGGG"	"A"	Narrow

Time Zones:

TZ // Short and Long Specific non-Location Format:
The short and long specific non-location formats for time zones are suggested for displaying a time with a user friendly time zone name. Where the short specific format is unavailable, it will fall back to the short localized GMT format ("O"). Where the long specific format is unavailable, it will fall back to the long localized GMT format ("OOOO").

Field Patterns	Output	Notes
"z", "zz", or "zzz"	"PDT"	Short Specific
"zzzz"	"Pacific Daylight Time"	Long Specific

TZ // Common UTC Offset Formats:
The ISO8601 basic format with hours, minutes and optional seconds fields is represented by "Z", "ZZ", or "ZZZ". The format is equivalent to RFC 822 zone format (when the optional seconds field is absent). This is equivalent to the "xxxx" specifier. The field pattern "ZZZZ" represents the long localized GMT format. This is equivalent to the "OOOO" specifier. Finally, "ZZZZZ" pattern yields the ISO8601 extended format with hours, minutes and optional seconds

fields. The ISO8601 UTC indicator `Z` is used when local time offset is `0`. This is equivalent to the `"XXXXX"` specifier.

Field Patterns	Output	Notes
`"Z"`, `"ZZ"`, or `"ZZZ"`	`"-0700"`	ISO 8601 basic format
`"ZZZZ"`	`"GMT-7:00"`	Long localized GMT format
`"ZZZZZ"`	`"-07:00"`	ISO 8601 extended format

TZ // Short and Long Localized GMT Formats:
The localized GMT formats come in two widths `"O"` (which removes the minutes field if it's `0`) and `"OOOO"` (which always contains the minutes field). The use of the `GMT` indicator changes according to the locale.

Field Patterns	Output	Notes
`"O"`	`"GMT-7"`	Short localized GMT format
`"OOOO"`	`"GMT-07:00"`	Long localized GMT format

TZ // Short and Long Generic non-Location Formats:
The generic non-location formats are useful for displaying a recurring wall time (e.g., events, meetings) or anywhere people do not want to be overly specific. Where either of these is unavailable, there is a fallback to the generic location format (`"VVVV"`), then the short localized GMT format as the final fallback.

Field Patterns	Output	Notes
`"v"`	`"PT"`	Short generic non-location format
`"vvvv"`	`"Pacific Time"`	Long generic non-location format

TZ // Short Time Zone IDs and Exemplar City Formats:
These formats provide variations of the time zone ID and often include the exemplar city. The widest of these formats, `"VVVV"`, is useful for populating a choice list for time zones, because it supports 1-to-1 name/zone ID mapping and is more uniform than other text formats.

Field Patterns	Output	Notes
`"V"`	`"cavan"`	Short time zone ID
`"VV"`	`"America/Vancouver"`	Long time zone ID
`"VVV"`	`"Vancouver"`	The tz exemplar city
`"VVVV"`	`"Vancouver Time"`	Generic location format

TZ // ISO 8601 Formats with Z for +0000:
The `"X"`-`"XXX"` field patterns represent valid ISO 8601 patterns for time zone offsets in datetimes. The final two widths, `"XXXX"` and `"XXXXX"` allow for optional seconds fields. The seconds field is *not* supported by the ISO 8601 specification. For all of these, the ISO 8601 UTC indicator `Z` is used when the local time offset is `0`.

Field Patterns	Output	Notes
`"X"`	`"-07"`	ISO 8601 basic format (h, optional m)

`"XX"`	`"-0700"`	ISO 8601 basic format (h & m)
`"XXX"`	`"-07:00"`	ISO 8601 extended format (h & m)
`"XXXX"`	`"-0700"`	ISO 8601 basic format (h & m, optional s)
`"XXXXX"`	`"-07:00"`	ISO 8601 extended format (h & m, optional s)

TZ // ISO 8601 Formats (no use of Z for +0000):
The `"x"`-`"xxxxx"` field patterns represent valid ISO 8601 patterns for time zone offsets in datetimes. They are similar to the `"X"`-`"XXXXX"` field patterns except that the ISO 8601 UTC indicator `Z` *will not* be used when the local time offset is `0`.

Field Patterns	Output	Notes
`"x"`	`"-07"`	ISO 8601 basic format (h, optional m)
`"xx"`	`"-0700"`	ISO 8601 basic format (h & m)
`"xxx"`	`"-07:00"`	ISO 8601 extended format (h & m)
`"xxxx"`	`"-0700"`	ISO 8601 basic format (h & m, optional s)
`"xxxxx"`	`"-07:00"`	ISO 8601 extended format (h & m, optional s)

Formatting with a `strptime` format code

Performing custom date/time formatting with the `format` argument can also occur with a `strptime` format code. This works by constructing a string of individual format codes representing formatted date and time elements. These are all indicated with a leading `%`, literal characters are interpreted as any characters not starting with a `%` character.

First off, let's look at a few format code combinations that work well together as a `strptime` format. This will give us an intuition on how these generally work. We'll use the datetime `"2015-06-08 23:05:37.48"` for all of the examples that follow.

- `"%m/%d/%Y"` -> `"06/08/2015"`
- `"%A, %B %e, %Y"` -> `"Monday, June 8, 2015"`
- `"%b %e %a"` -> `"Jun 8 Mon"`
- `"%H:%M"` -> `"23:05"`
- `"%I:%M %p"` -> `"11:05 pm"`
- `"%A, %B %e, %Y at %I:%M %p"` -> `"Monday, June 8, 2015 at 11:05 pm"`

Here are the individual format codes for the date components:

- `"%a"` -> `"Mon"` (abbreviated day of week name)
- `"%A"` -> `"Monday"` (full day of week name)
- `"%w"` -> `"1"` (day of week number in `0..6`; Sunday is `0`)
- `"%u"` -> `"1"` (day of week number in `1..7`; Monday is `1`, Sunday `7`)
- `"%y"` -> `"15"` (abbreviated year, using the final two digits)
- `"%Y"` -> `"2015"` (full year)
- `"%b"` -> `"Jun"` (abbreviated month name)
- `"%B"` -> `"June"` (full month name)

- "%m" -> "06" (month number)
- "%d" -> "08" (day number, zero-padded)
- "%e" -> "8" (day number without zero padding)
- "%j" -> "159" (day of the year, always zero-padded)
- "%W" -> "23" (week number for the year, always zero-padded)
- "%V" -> "24" (week number for the year, following the ISO 8601 standard)
- "%C" -> "20" (the century number)

Here are the individual format codes for the time components:

- "%H" -> "23" (24h hour)
- "%I" -> "11" (12h hour)
- "%M" -> "05" (minute)
- "%S" -> "37" (second)
- "%OS3" -> "37.480" (seconds with decimals; 3 decimal places here)
- %p -> "pm" (AM or PM indicator)

Here are some extra formats that you may find useful:

- "%z" -> "+0000" (signed time zone offset, here using UTC)
- "%F" -> "2015-06-08" (the date in the ISO 8601 date format)
- "%%" -> "%" (the literal "%" character, in case you need it)

Examples

Let's create a character vector of datetime values in the ISO-8601 format for the next few examples:

```
str_vals <- c("2022-06-13 18:36", "2019-01-25 01:08", NA)
```

Using vec_fmt_datetime() with different date_style and time_style options (here, date_style = "yMMMEd" and time_style = "Hm") will result in a character vector of formatted datetime values. Any NA values remain as NA values. The rendering context will be autodetected unless specified in the output argument (here, it is of the "plain" output type).

```
vec_fmt_datetime(
  str_vals,
  date_style = "yMMMEd",
  time_style = "Hm"
)
```

```
#> [1] "Mon, Jun 13, 2022 18:36" "Fri, Jan 25, 2019 01:08" NA
```

We can choose from any of 41 different date styles and 25 time formatting styles. Many of these styles are flexible, meaning that the structure of the format will adapt to different locales. Let's use a combination of the the "yMMMd" and "hms" date and time styles to demonstrate this (first in the default locale of "en"):

```
vec_fmt_datetime(
  str_vals,
  date_style = "yMMMd",
  time_style = "hms"
)
```

```
#> [1] "Jun 13, 2022 6:36:00 PM" "Jan 25, 2019 1:08:00 AM" NA
```

Let's perform the same type of formatting in the Italian ("it") locale:

```
vec_fmt_datetime(
  str_vals,
  date_style = "yMMMd",
  time_style = "hms",
  locale = "it"
)
```

```
#> [1] "13 giu 2022 6:36:00 PM" "25 gen 2019 1:08:00 AM" NA
```

We can always use `info_date_style()` or `info_time_style()` to call up info tables that serve as handy references to all of the `date_style` and `time_style` options.

It's possible to supply our own time formatting pattern within the `format` argument. One way is with a CLDR pattern, which is locale-aware:

```
vec_fmt_datetime(str_vals, format = "EEEE, MMMM d, y, h:mm a")
```

```
#> [1] "Monday, June 13, 2022, 06:36 PM"
#> [2] "Friday, January 25, 2019, 01:08 AM"
#> [3] NA
```

By using the `locale` argument, this can be formatted as Dutch datetime values:

```
vec_fmt_datetime(
  str_vals,
  format = "EEEE, MMMM d, y, h:mm a",
  locale = "nl"
)
```

```
#> [1] "maandag, juni 13, 2022, 6:36 p.m."
#> [2] "vrijdag, januari 25, 2019, 1:08 a.m."
#> [3] NA
```

It's also possible to use a `strptime` format code with `format` (however, any value provided to `locale` will be ignored).

```
vec_fmt_datetime(str_vals, format = "%A, %B %e, %Y at %I:%M %p")
```

```
#> [1] "Monday, June 13, 2022 at 06:36 pm"
#> [2] "Friday, January 25, 2019 at 01:08 am"
#> [3] NA
```

As a last example, one can wrap the datetime values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_datetime(
  str_vals,
  sep = " at ",
  pattern = "Date and Time: {x}"
)
```

```
#> [1] "Date and Time: 2022-06-13 at 18:36:00"
#> [2] "Date and Time: 2019-01-25 at 01:08:00"
#> [3] NA
```

Function ID

14-13

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_scientific()`, `vec_fmt_time()`

`vec_fmt_duration` *Format a vector of numeric or duration values as styled time duration strings*

Description

Format input values to time duration values whether those input values are numbers or of the `difftime` class. We can specify which time units any numeric input values have (as weeks, days, hours, minutes, or seconds) and the output can be customized with a duration style (corresponding to narrow, wide, colon-separated, and ISO forms) and a choice of output units ranging from weeks to seconds.

Usage

```
vec_fmt_duration(
  x,
  input_units = NULL,
  output_units = NULL,
  duration_style = c("narrow", "wide", "colon-sep", "iso"),
```

```
  trim_zero_units = TRUE,
  max_output_units = NULL,
  pattern = "{x}",
  use_seps = TRUE,
  sep_mark = ",",
  force_sign = FALSE,
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

x
: A numeric vector.

input_units
: If one or more selected columns contains numeric values, a keyword must be provided for `input_units` for **gt** to determine how those values are to be interpreted in terms of duration. The accepted units are: `"seconds"`, `"minutes"`, `"hours"`, `"days"`, and `"weeks"`.

output_units
: Controls the output time units. The default, `NULL`, means that **gt** will automatically choose time units based on the input duration value. To control which time units are to be considered for output (before trimming with `trim_zero_units`) we can specify a vector of one or more of the following keywords: `"weeks"`, `"days"`, `"hours"`, `"minutes"`, or `"seconds"`.

duration_style
: A choice of four formatting styles for the output duration values. With `"narrow"` (the default style), duration values will be formatted with single letter time-part units (e.g., 1.35 days will be styled as `"1d 8h 24m`). With `"wide"`, this example value will be expanded to `"1 day 8 hours 24 minutes"` after formatting. The `"colon-sep"` style will put days, hours, minutes, and seconds in the `"([D]/)[HH]:[MM]:[SS]"` format. The `"iso"` style will produce a value that conforms to the ISO 8601 rules for duration values (e.g., 1.35 days will become `"P1DT8H24M"`).

trim_zero_units
: Provides methods to remove output time units that have zero values. By default this is `TRUE` and duration values that might otherwise be formatted as `"0w 1d 0h 4m 19s"` with `trim_zero_units = FALSE` are instead displayed as `"1d 4m 19s"`. Aside from using `TRUE`/`FALSE` we could provide a vector of keywords for more precise control. These keywords are: (1) `"leading"`, to omit all leading zero-value time units (e.g., `"0w 1d"` -> `"1d"`), (2) `"trailing"`, to omit all trailing zero-value time units (e.g., `"3d 5h 0s"` -> `"3d 5h"`), and `"internal"`, which removes all internal zero-value time units (e.g., `"5d 0h 33m"` -> `"5d 33m"`).

max_output_units
: If `output_units` is `NULL`, where the output time units are unspecified and left to **gt** to handle, a numeric value provided for `max_output_units` will be taken as the maximum number of time units to display in all output time duration values. By default, this is `NULL` and all possible time units will be displayed. This option has no effect when `duration_style = "colon-sep"` (only `output_units` can be used to customize that type of duration output).

pattern
: A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

use_seps
: An option to use digit group separators. The type of digit group separator is set by `sep_mark` and overridden if a locale ID is provided to `locale`. This setting is `TRUE` by default.

sep_mark
: The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

force_sign
: Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative value will display a minus sign.

locale
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported.

output
: The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Output units for the colon-separated duration style

The colon-separated duration style (enabled when `duration_style = "colon-sep"`) is essentially a clock-based output format which uses the display logic of chronograph watch functionality. It will, by default, display duration values in the `(D/)HH:MM:SS` format. Any duration values greater than or equal to 24 hours will have the number of days prepended with an adjoining slash mark. While this output format is versatile, it can be changed somewhat with the `output_units` option. The following combinations of output units are permitted:

- `c("minutes", "seconds")` -> `MM:SS`
- `c("hours", "minutes")` -> `HH:MM`
- `c("hours", "minutes", "seconds")` -> `HH:MM:SS`
- `c("days", "hours", "minutes")` -> `(D/)HH:MM`

Any other specialized combinations will result in the default set being used, which is `c("days", "hours", "minutes", "seconds")`

Examples

Let's create a `difftime`-based vector for the next few examples:

```
difftimes <-
  difftime(
    lubridate::ymd("2017-01-15"),
    lubridate::ymd(c("2015-06-25", "2016-03-07", "2017-01-10"))
  )
```

Using `vec_fmt_duration()` with its defaults provides us with a succinct vector of formatted durations.

```
vec_fmt_duration(difftimes)

#> [1] "81w 3d" "44w 6d" "5d"
```

We can elect to use just only the time units of days to describe the duration values.

```
vec_fmt_duration(difftimes, output_units = "days")

#> [1] "570d" "314d" "5d"
```

We can also use numeric values in the input vector `vec_fmt_duration()`. Here's a numeric vector for use with examples:

```
num_vals <- c(3.235, 0.23, 0.005, NA)
```

The necessary thing with numeric values as an input is defining what time unit those values have.

```
vec_fmt_duration(num_vals, input_units = "days")

#> [1] "3d 5h 38m 23s" "5h 31m 12s" "7m 12s" "NA"
```

We can define a set of output time units that we want to see.

```
vec_fmt_duration(
  num_vals,
  input_units = "days",
  output_units = c("hours", "minutes")
)

#> [1] "77h 38m" "5h 31m" "7m" "NA"
```

There are many duration 'styles' to choose from. We could opt for the `"wide"` style.

```
vec_fmt_duration(
  num_vals,
  input_units = "days",
  duration_style = "wide"
)

#> [1] "3 days 5 hours 38 minutes 23 seconds"
#> [2] "5 hours 31 minutes 12 seconds"
#> [3] "7 minutes 12 seconds"
#> [4] "NA"
```

We can always perform locale-specific formatting with `vec_fmt_duration()`. Let's attempt the same type of duration formatting as before with the `"nl"` locale.

```
vec_fmt_duration(
  num_vals,
  input_units = "days",
  duration_style = "wide",
  locale = "nl"
)
```

```
#> [1] "3 dagen 5 uur 38 minuten 23 seconden"
#> [2] "5 uur 31 minuten 12 seconden"
#> [3] "7 minuten 12 seconden"
#> [4] "NA"
```

Function ID

14-14

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_scientific()`, `vec_fmt_time()`

`vec_fmt_engineering` *Format a vector as values in engineering notation*

Description

With numeric values in a vector, we can perform formatting so that the input values are rendered into engineering notation within the output character vector. The following major options are available:

- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol
- digit grouping separators: choice of separator symbol
- scaling: we can choose to scale targeted values by a multiplier value
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in formatting specific to the chosen locale

Usage

```
vec_fmt_engineering(
  x,
  decimals = 2,
  drop_trailing_zeros = FALSE,
  scale_by = 1,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x` A numeric vector.

`decimals` An option to specify the exact number of decimal places to use. The default number of decimal places is `2`.

`drop_trailing_zeros` A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).

`scale_by` A value to scale the input. The default is `1.0`. All numeric values will be multiplied by this value first before undergoing formatting.

`pattern` A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

`sep_mark` The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

`dec_mark` The character to use as a decimal mark (e.g., using `dec_mark = ","` with `0.152` would result in a formatted value of `0,152`).

`force_sign` Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign.

`locale` An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported.

`output` The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Examples

Let's create a numeric vector for the next few examples:

```
num_vals <- c(3.24e-4, 8.65, 1362902.2, -59027.3, NA)
```

Using `vec_fmt_engineering()` with the default options will create a character vector with values engineering notation. Any `NA` values remain as `NA` values. The rendering context will be autodetected unless specified in the `output` argument (here, it is of the `"plain"` output type).

```
vec_fmt_engineering(num_vals)
```

```
#> [1] "324.00 × 10^-6" "8.65" "1.36 × 10^6" "-59.03 × 10^3" "NA"
```

We can change the number of decimal places with the `decimals` option:

```
vec_fmt_engineering(num_vals, decimals = 1)
```

```
#> [1] "324.0 × 10^-6" "8.7" "1.4 × 10^6" "-59.0 × 10^3" "NA"
```

If we are formatting for a different locale, we could supply the locale ID and **gt** will handle any locale-specific formatting options:

```
vec_fmt_engineering(num_vals, locale = "da")
```

```
#> [1] "324,00 × 10^-6" "8,65" "1,36 × 10^6" "-59,03 × 10^3" "NA"
```

Should you need to have positive and negative signs on each of the output values, use `force_sign = TRUE`:

```
vec_fmt_engineering(num_vals, force_sign = TRUE)
```

```
#> [1] "+324.00 × 10^-6" "+8.65" "+1.36 × 10^6" "-59.03 × 10^3" "NA"
```

As a last example, one can wrap the values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_engineering(num_vals, pattern = "/{x}/")
```

```
#> [1] "/324.00 × 10^-6/" "/8.65/" "/1.36 × 10^6/" "/-59.03 × 10^3/" "NA"
```

Function ID

14-4

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_scientific()`, `vec_fmt_time()`

vec_fmt_fraction   *Format a vector as mixed fractions*

Description

With numeric values in vector, we can perform mixed-fraction-based formatting. There are several options for setting the accuracy of the fractions. Furthermore, there is an option for choosing a layout (i.e., typesetting style) for the mixed-fraction output.

The following options are available for controlling this type of formatting:

- accuracy: how to express the fractional part of the mixed fractions; there are three keyword options for this and an allowance for arbitrary denominator settings
- simplification: an option to simplify fractions whenever possible
- layout: We can choose to output values with diagonal or inline fractions
- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol for the whole number portion
- pattern: option to use a text pattern for decoration of the formatted mixed fractions
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
vec_fmt_fraction(
  x,
  accuracy = NULL,
  simplify = TRUE,
  layout = c("inline", "diagonal"),
  use_seps = TRUE,
  pattern = "{x}",
  sep_mark = ",",
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

x	A numeric vector.
accuracy	The type of fractions to generate. This can either be one of the keywords `"low"`, `"med"`, or `"high"` (to generate fractions with denominators of up to 1, 2, or 3 digits, respectively) or an integer value greater than zero to obtain fractions with a fixed denominator (`2` yields halves, `3` is for thirds, `4` is quarters, etc.). For the latter option, using `simplify = TRUE` will simplify fractions where possible (e.g., `2/4` will be simplified as `1/2`). By default, the `"low"` option is used.
simplify	If choosing to provide a numeric value for `accuracy`, the option to simplify the fraction (where possible) can be taken with `TRUE` (the default). With `FALSE`, denominators in fractions will be fixed to the value provided in `accuracy`.

layout
: For HTML output, the "inline" layout is the default. This layout places the numerals of the fraction on the baseline and uses a standard slash character. The "diagonal" layout will generate fractions that are typeset with raised/lowered numerals and a virgule.

use_seps
: An option to use digit group separators. The type of digit group separator is set by sep_mark and overridden if a locale ID is provided to locale. This setting is TRUE by default.

pattern
: A formatting pattern that allows for decoration of the formatted value. The value itself is represented by {x} and all other characters are taken to be string literals.

sep_mark
: The mark to use as a separator between groups of digits (e.g., using sep_mark = "," with 1000 would result in a formatted value of 1,000).

locale
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include "en" for English (United States) and "fr" for French (France). The use of a valid locale ID will override any values provided in sep_mark and dec_mark. We can use the info_locales() function as a useful reference for all of the locales that are supported.

output
: The output style of the resulting character vector. This can either be "auto" (the default), "plain", "html", "latex", "rtf", or "word". In **knitr** rendering (i.e., Quarto or R Markdown), the "auto" option will choose the correct output value

Value

A character vector.

Examples

Let's create a numeric vector for the next few examples:

```
num_vals <- c(0.0052, 0.08, 0, -0.535, NA)
```

Using vec_fmt_fraction() will create a character vector of fractions. Any NA values will render as "NA". The rendering context will be autodetected unless specified in the output argument (here, it is of the "plain" output type).

```
vec_fmt_fraction(num_vals)
```

```
#> [1] "0" "1/9" "0" "-5/9" "NA"
```

There are many options for formatting as fractions. If you'd like a higher degree of accuracy in the computation of fractions we can supply the "med" or "high" keywords to the accuracy argument:

```
vec_fmt_fraction(num_vals, accuracy = "high")
```

```
#> [1] "1/200" "2/25" "0" "-107/200" "NA"
```

As a last example, one can wrap the values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_fraction(num_vals, accuracy = 8, pattern = "[{x}]")
```

```
#> [1] "[0]" "[1/8]" "[0]" "[-1/2]" "NA"
```

Function ID

14-7

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_scientific()`, `vec_fmt_time()`

---

vec_fmt_integer — *Format a vector as integer values*

Description

With numeric values in a vector, we can perform number-based formatting so that the input values are always rendered as integer values within a character vector. The following major options are available:

- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- scaling: we can choose to scale targeted values by a multiplier value
- large-number suffixing: larger figures (thousands, millions, etc.) can be autoscaled and decorated with the appropriate suffixes
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
vec_fmt_integer(
  x,
  use_seps = TRUE,
  accounting = FALSE,
  scale_by = 1,
  suffixing = FALSE,
  pattern = "{x}",
  sep_mark = ",",
```

```
  force_sign = FALSE,
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

x
: A numeric vector.

use_seps
: An option to use digit group separators. The type of digit group separator is set by `sep_mark` and overridden if a locale ID is provided to `locale`. This setting is `TRUE` by default.

accounting
: An option to use accounting style for values. With `FALSE` (the default), negative values will be shown with a minus sign. Using `accounting = TRUE` will put negative values in parentheses.

scale_by
: A value to scale the input. The default is `1.0`. All numeric values will be multiplied by this value first before undergoing formatting. This value will be ignored if using any of the `suffixing` options (i.e., where `suffixing` is not set to `FALSE`).

suffixing
: An option to scale and apply suffixes to larger numbers (e.g., `1924000` can be transformed to `2M`). This option can accept a logical value, where `FALSE` (the default) will not perform this transformation and `TRUE` will apply thousands (`K`), millions (`M`), billions (`B`), and trillions (`T`) suffixes after automatic value scaling. We can also specify which symbols to use for each of the value ranges by using a character vector of the preferred symbols to replace the defaults (e.g., `c("k", "Ml", "Bn", "Tr")`).

  Including `NA` values in the vector will ensure that the particular range will either not be included in the transformation (e.g, `c(NA, "M", "B", "T")` won't modify numbers in the thousands range) or the range will inherit a previous suffix (e.g., with `c("K", "M", NA, "T")`, all numbers in the range of millions and billions will be in terms of millions).

  Any use of `suffixing` (where it is not set expressly as `FALSE`) means that any value provided to `scale_by` will be ignored.

pattern
: A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

sep_mark
: The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

force_sign
: Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign. This option is disregarded when using accounting notation with `accounting = TRUE`.

locale
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported.

`output` The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Examples

Let's create a numeric vector for the next few examples:

```
num_vals <- c(5.2, 8.65, 13602, -5.3, NA)
```

Using `vec_fmt_integer()` with the default options will create a character vector where the input values undergo rounding to become integers and `NA` values will render as `"NA"`. Also, the rendering context will be autodetected unless specified in the `output` argument (here, it is of the `"plain"` output type).

```
vec_fmt_integer(num_vals)
```

```
#> [1] "5" "9" "13,602" "-5" "NA"
```

We can change the digit separator mark to a period with the `sep_mark` option:

```
vec_fmt_integer(num_vals, sep_mark = ".")
```

```
#> [1] "5" "9" "13.602" "-5" "NA"
```

Many options abound for formatting values. If you have a need for positive and negative signs in front of each and every value, use `force_sign = TRUE`:

```
vec_fmt_integer(num_vals, force_sign = TRUE)
```

```
#> [1] "+5" "+9" "+13,602" "-5" "NA"
```

As a last example, one can wrap the values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_integer(num_vals, pattern = "`{x}`")
```

```
#> [1] "`5`" "`9`" "`13,602`" "`-5`" "NA"
```

Function ID

14-2

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_scientific()`, `vec_fmt_time()`

---

vec_fmt_markdown *Format a vector containing Markdown text*

Description

Any Markdown-formatted text in the input vector will be transformed to the appropriate output type.

Usage

```
vec_fmt_markdown(
  x,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x`	A numeric vector.
`output`	The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Examples

Create a vector of Markdown-based text snippets.

```
text_vec <-
  c(
    "This **is** *Markdown*.",
    "Info on Markdown syntax can be found
[here](https://daringfireball.net/projects/markdown/).",
    "The **gt** package has these datasets:
- `countrypops`
- `sza`
- `gtcars`
- `sp500`
```

```
- `pizzaplace`
- `exibble`"
  )
```

With `vec_fmt_markdown()` we can easily convert these to different output types, like HTML

```
vec_fmt_markdown(text_vec, output = "html")
#> [1] "<p>This <strong>is</strong> <em>Markdown</em>.</p>"
#> [2] "<p>Info on Markdown syntax can be found\n<a href=\"https://daringfireball.net/projects/markdow
#> [3] "<p>The <strong>gt</strong> package has these datasets:</p>\n<ul>\n<li><code>countrypops</code>
```

or LaTeX

```
vec_fmt_markdown(text_vec, output = "latex")
#> [1] "This \\textbf{is} \\emph{Markdown}."
#> [2] "Info on Markdown syntax can be found\n\\href{https://daringfireball.net/projects/markdown/}{he
#> [3] "The \\textbf{gt} package has these datasets:\n\n\\begin{itemize}\n\\item \\texttt{countrypops}
```

Function ID

14-15

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_scientific()`, `vec_fmt_time()`

`vec_fmt_number` *Format a vector as numeric values*

Description

With numeric values in a vector, we can perform number-based formatting so that the values are rendered to a character vector with some level of precision. The following major options are available:

- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol
- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- scaling: we can choose to scale targeted values by a multiplier value
- large-number suffixing: larger figures (thousands, millions, etc.) can be autoscaled and decorated with the appropriate suffixes
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
vec_fmt_number(
  x,
  decimals = 2,
  n_sigfig = NULL,
  drop_trailing_zeros = FALSE,
  drop_trailing_dec_mark = TRUE,
  use_seps = TRUE,
  accounting = FALSE,
  scale_by = 1,
  suffixing = FALSE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x`	A numeric vector.
`decimals`	An option to specify the exact number of decimal places to use. The default number of decimal places is `2`.
`n_sigfig`	A option to format numbers to *n* significant figures. By default, this is `NULL` and thus number values will be formatted according to the number of decimal places set via `decimals`. If opting to format according to the rules of significant figures, `n_sigfig` must be a number greater than or equal to `1`. Any values passed to the `decimals` and `drop_trailing_zeros` arguments will be ignored.
`drop_trailing_zeros`	A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).
`drop_trailing_dec_mark`	A logical value that determines whether decimal marks should always appear even if there are no decimal digits to display after formatting (e.g, `23` becomes `23.`). The default for this is `TRUE`, which means that trailing decimal marks are not shown.
`use_seps`	An option to use digit group separators. The type of digit group separator is set by `sep_mark` and overridden if a locale ID is provided to `locale`. This setting is `TRUE` by default.
`accounting`	An option to use accounting style for values. With `FALSE` (the default), negative values will be shown with a minus sign. Using `accounting = TRUE` will put negative values in parentheses.
`scale_by`	A value to scale the input. The default is `1.0`. All numeric values will be multiplied by this value first before undergoing formatting. This value will be ignored if using any of the `suffixing` options (i.e., where `suffixing` is not set to `FALSE`).

suffixing — An option to scale and apply suffixes to larger numbers (e.g., `1924000` can be transformed to `1.92M`). This option can accept a logical value, where `FALSE` (the default) will not perform this transformation and `TRUE` will apply thousands (`K`), millions (`M`), billions (`B`), and trillions (`T`) suffixes after automatic value scaling. We can also specify which symbols to use for each of the value ranges by using a character vector of the preferred symbols to replace the defaults (e.g., `c("k", "Ml", "Bn", "Tr")`).

Including `NA` values in the vector will ensure that the particular range will either not be included in the transformation (e.g, `c(NA, "M", "B", "T")` won't modify numbers in the thousands range) or the range will inherit a previous suffix (e.g., with `c("K", "M", NA, "T")`, all numbers in the range of millions and billions will be in terms of millions).

Any use of `suffixing` (where it is not set expressly as `FALSE`) means that any value provided to `scale_by` will be ignored.

pattern — A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

sep_mark — The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

dec_mark — The character to use as a decimal mark (e.g., using `dec_mark = ","` with `0.152` would result in a formatted value of `0,152`).

force_sign — Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign. This option is disregarded when using accounting notation with `accounting = TRUE`.

locale — An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported.

output — The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Examples

Let's create a numeric vector for the next few examples:

```
num_vals <- c(5.2, 8.65, 0, -5.3, NA)
```

Using `vec_fmt_number()` with the default options will create a character vector where the numeric values have two decimal places and `NA` values will render as `"NA"`. Also, the rendering context will be autodetected unless specified in the `output` argument (here, it is of the `"plain"` output type).

```
vec_fmt_number(num_vals)

#> [1] "5.20" "8.65" "0.00" "-5.30" "NA"
```

We can change the decimal mark to a comma, and we have to be sure to change the digit separator mark from the default comma to something else (a period works here):

```
vec_fmt_number(num_vals, sep_mark = ".", dec_mark = ",")

#> [1] "5,20" "8,65" "0,00" "-5,30" "NA"
```

If we are formatting for a different locale, we could supply the locale ID and let **gt** handle these locale-specific formatting options:

```
vec_fmt_number(num_vals, locale = "fr")

#> [1] "5,20" "8,65" "0,00" "-5,30" "NA"
```

There are many options for formatting values. Perhaps you need to have explicit positive and negative signs? Use `force_sign = TRUE` for that.

```
vec_fmt_number(num_vals, force_sign = TRUE)

#> [1] "+5.20" "+8.65" "0.00" "-5.30" "NA"
```

Those trailing zeros past the decimal mark can be stripped out by using the `drop_trailing_zeros` option.

```
vec_fmt_number(num_vals, drop_trailing_zeros = TRUE)

#> [1] "5.2" "8.65" "0" "-5.3" "NA"
```

As a last example, one can wrap the values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_number(num_vals, pattern = "`{x}`")

#> [1] "`5.20`" "`8.65`" "`0.00`" "`-5.30`" "NA"
```

Function ID

14-1

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_scientific()`, `vec_fmt_time()`

`vec_fmt_partsper` *Format a vector as parts-per quantities*

Description

With numeric values in a vector, we can format the values so that they are rendered as *per mille*, *ppm*, *ppb*, etc., quantities. The following list of keywords (with associated naming and scaling factors) is available to use within `vec_fmt_partsper()`:

- `"per-mille"`: Per mille, (1 part in `1,000`)
- `"per-myriad"`: Per myriad, (1 part in `10,000`)
- `"pcm"`: Per cent mille (1 part in `100,000`)
- `"ppm"`: Parts per million, (1 part in `1,000,000`)
- `"ppb"`: Parts per billion, (1 part in `1,000,000,000`)
- `"ppt"`: Parts per trillion, (1 part in `1,000,000,000,000`)
- `"ppq"`: Parts per quadrillion, (1 part in `1,000,000,000,000,000`)

The function provides a lot of formatting control and we can use the following options:

- custom symbol/units: we can override the automatic symbol or units display with our own choice as the situation warrants
- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol
- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- value scaling toggle: choose to disable automatic value scaling in the situation that values are already scaled coming in (and just require the appropriate symbol or unit display)
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
vec_fmt_partsper(
  x,
  to_units = c("per-mille", "per-myriad", "pcm", "ppm", "ppb", "ppt", "ppq"),
  symbol = "auto",
  decimals = 2,
  drop_trailing_zeros = FALSE,
  drop_trailing_dec_mark = TRUE,
  scale_values = TRUE,
  use_seps = TRUE,
  pattern = "{x}",
  sep_mark = ",",
```

```
  dec_mark = ".",
  force_sign = FALSE,
  incl_space = "auto",
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x` A numeric vector.

`to_units` A keyword that signifies the desired output quantity. This can be any from the following set: "per-mille", "per-myriad", "pcm", "ppm", "ppb", "ppt", or "ppq".

`symbol` The symbol/units to use for the quantity. By default, this is set to "auto" and **gt** will choose the appropriate symbol based on the to_units keyword and the output context. However, this can be changed by supplying a string (e.g, using symbol = "ppbV" when to_units = "ppb").

`decimals` An option to specify the exact number of decimal places to use. The default number of decimal places is 2.

`drop_trailing_zeros`
A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).

`drop_trailing_dec_mark`
A logical value that determines whether decimal marks should always appear even if there are no decimal digits to display after formatting (e.g, 23 becomes 23.). The default for this is TRUE, which means that trailing decimal marks are not shown.

`scale_values` Should the values be scaled through multiplication according to the keyword set in to_units? By default this is TRUE since the expectation is that normally values are proportions. Setting to FALSE signifies that the values are already scaled and require only the appropriate symbol/units when formatted.

`use_seps` An option to use digit group separators. The type of digit group separator is set by sep_mark and overridden if a locale ID is provided to locale. This setting is TRUE by default.

`pattern` A formatting pattern that allows for decoration of the formatted value. The value itself is represented by {x} and all other characters are taken to be string literals.

`sep_mark` The mark to use as a separator between groups of digits (e.g., using sep_mark = "," with 1000 would result in a formatted value of 1,000).

`dec_mark` The character to use as a decimal mark (e.g., using dec_mark = "," with 0.152 would result in a formatted value of 0,152).

`force_sign` Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use TRUE for this option. The default is FALSE, where only negative numbers will display a minus sign. This option is disregarded when using accounting notation with accounting = TRUE.

`incl_space` An option for whether to include a space between the value and the symbol/units. The default is `"auto"` which provides spacing dependent on the mark itself. This can be directly controlled by using either `TRUE` or `FALSE`.

`locale` An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported.

`output` The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Examples

Let's create a numeric vector for the next few examples:

```
num_vals <- c(10^(-3:-5), NA)
```

Using `vec_fmt_partsper()` with the default options will create a character vector where the resultant per mille values have two decimal places and `NA` values will render as `"NA"`. The rendering context will be autodetected unless specified in the `output` argument (here, it is of the `"plain"` output type).

```
vec_fmt_partsper(num_vals)
```

```
#> [1] "1.00%" "0.10%" "0.01%" "NA"
```

We can change the output units to a different measure. If ppm units are desired then `to_units = "ppm"` can be used.

```
vec_fmt_partsper(num_vals, to_units = "ppm")
```

```
#> [1] "1,000.00 ppm" "100.00 ppm" "10.00 ppm" "NA"
```

We can change the decimal mark to a comma, and we have to be sure to change the digit separator mark from the default comma to something else (a period works here):

```
vec_fmt_partsper(
  num_vals,
  to_units = "ppm",
  sep_mark = ".",
  dec_mark = ","
)
```

```
#> [1] "1.000,00 ppm" "100,00 ppm" "10,00 ppm" "NA"
```

If we are formatting for a different locale, we could supply the locale ID and let **gt** handle these locale-specific formatting options:

```
vec_fmt_partsper(num_vals, to_units = "ppm", locale = "es")
```

```
#> [1] "1.000,00 ppm" "100,00 ppm" "10,00 ppm" "NA"
```

As a last example, one can wrap the values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_partsper(num_vals, to_units = "ppm", pattern = "{x}V")
```

```
#> [1] "1,000.00 ppmV" "100.00 ppmV" "10.00 ppmV" "NA"
```

Function ID

14-6

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_scientific()`, `vec_fmt_time()`

`vec_fmt_percent` *Format a vector as percentage values*

Description

With numeric values in vector, we can perform percentage-based formatting. It is assumed that numeric values in the input vector are proportional values and, in this case, the values will be automatically multiplied by `100` before decorating with a percent sign (the other case is accommodated though setting the `scale_values` to `FALSE`). For more control over percentage formatting, we can use the following options:

- percent sign placement: the percent sign can be placed after or before the values and a space can be inserted between the symbol and the value.
- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol
- digit grouping separators: options to enable/disable digit separators and provide a choice of separator symbol
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in number formatting specific to the chosen locale

Usage

```
vec_fmt_percent(
  x,
  decimals = 2,
  drop_trailing_zeros = FALSE,
  drop_trailing_dec_mark = TRUE,
  scale_values = TRUE,
  use_seps = TRUE,
  accounting = FALSE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  incl_space = FALSE,
  placement = "right",
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x` A numeric vector.

`decimals` An option to specify the exact number of decimal places to use. The default number of decimal places is `2`.

`drop_trailing_zeros` A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).

`drop_trailing_dec_mark` A logical value that determines whether decimal marks should always appear even if there are no decimal digits to display after formatting (e.g, `23` becomes `23.`). The default for this is `TRUE`, which means that trailing decimal marks are not shown.

`scale_values` Should the values be scaled through multiplication by 100? By default this is `TRUE` since the expectation is that normally values are proportions. Setting to `FALSE` signifies that the values are already scaled and require only the percent sign when formatted.

`use_seps` An option to use digit group separators. The type of digit group separator is set by `sep_mark` and overridden if a locale ID is provided to `locale`. This setting is `TRUE` by default.

`accounting` An option to use accounting style for values. With `FALSE` (the default), negative values will be shown with a minus sign. Using `accounting = TRUE` will put negative values in parentheses.

`pattern` A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.

`sep_mark` The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).

- `dec_mark` The character to use as a decimal mark (e.g., using `dec_mark = ","` with `0.152` would result in a formatted value of `0,152`).
- `force_sign` Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign. This option is disregarded when using accounting notation with `accounting = TRUE`.
- `incl_space` An option for whether to include a space between the value and the percent sign. The default is to not introduce a space character.
- `placement` The placement of the percent sign. This can be either be `right` (the default) or `left`.
- `locale` An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported.
- `output` The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Examples

Let's create a numeric vector for the next few examples:

```
num_vals <- c(0.0052, 0.08, 0, -0.535, NA)
```

Using `vec_fmt_percent()` with the default options will create a character vector where the resultant percentage values have two decimal places and `NA` values will render as `"NA"`. The rendering context will be autodetected unless specified in the `output` argument (here, it is of the `"plain"` output type).

```
vec_fmt_percent(num_vals)
```

```
#> [1] "0.52%" "8.00%" "0.00%" "-53.50%" "NA"
```

We can change the decimal mark to a comma, and we have to be sure to change the digit separator mark from the default comma to something else (a period works here):

```
vec_fmt_percent(num_vals, sep_mark = ".", dec_mark = ",")
```

```
#> [1] "0,52%" "8,00%" "0,00%" "-53,50%" "NA"
```

If we are formatting for a different locale, we could supply the locale ID and let **gt** handle these locale-specific formatting options:

```
vec_fmt_percent(num_vals, locale = "pt")
```

```
#> [1] "0,52%" "8,00%" "0,00%" "-53,50%" "NA"
```

There are many options for formatting values. Perhaps you need to have explicit positive and negative signs? Use `force_sign = TRUE` for that.

```
vec_fmt_percent(num_vals, force_sign = TRUE)
```

```
#> [1] "+0.52%" "+8.00%" "0.00%" "-53.50%" "NA"
```

Those trailing zeros past the decimal mark can be stripped out by using the `drop_trailing_zeros` option.

```
vec_fmt_percent(num_vals, drop_trailing_zeros = TRUE)
```

```
#> [1] "0.52%" "8%" "0%" "-53.5%" "NA"
```

As a last example, one can wrap the values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_percent(num_vals, pattern = "{x}wt")
```

```
#> [1] "0.52%wt" "8.00%wt" "0.00%wt" "-53.50%wt" "NA"
```

Function ID

14-5

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_roman()`, `vec_fmt_scientific()`, `vec_fmt_time()`

`vec_fmt_roman` *Format a vector as Roman numerals*

Description

With numeric values in a vector, we can transform those to Roman numerals, rounding values as necessary.

Usage

```
vec_fmt_roman(
  x,
  case = c("upper", "lower"),
  pattern = "{x}",
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x`	A numeric vector.
`case`	Should Roman numerals should be rendered as uppercase (`"upper"`) or lowercase (`"lower"`) letters? By default, this is set to `"upper"`.
`pattern`	A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.
`output`	The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Examples

Let's create a numeric vector for the next few examples:

```
num_vals <- c(1, 4, 5, 8, 12, 20, 0, -5, 1.3, NA)
```

Using `vec_fmt_roman()` with the default options will create a character vector with values rendered as Roman numerals. Zero values will be rendered as `"N"`, any `NA` values remain as `NA` values, negative values will be automatically made positive, and values greater than or equal to 3900 will be rendered as `"ex terminis"`. The rendering context will be autodetected unless specified in the `output` argument (here, it is of the `"plain"` output type).

```
vec_fmt_roman(num_vals)
```

```
#> [1] "I" "IV" "V" "VIII" "XII" "XX" "N" "V" "I" "NA"
```

We can also use `vec_fmt_roman()` with the `case = "lower"` option to create a character vector with values rendered as lowercase Roman numerals.

```
vec_fmt_roman(num_vals, case = "lower")
```

```
#> [1] "i" "iv" "v" "viii" "xii" "xx" "n" "v" "i" "NA"
```

As a last example, one can wrap the values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_roman(num_vals, case = "lower", pattern = "{x}.")
```

```
#> [1] "i." "iv." "v." "viii." "xii." "xx." "n." "v." "i." "NA"
```

Function ID

14-9

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_scientific()`, `vec_fmt_time()`

`vec_fmt_scientific` *Format a vector as values in scientific notation*

Description

With numeric values in a vector, we can perform formatting so that the input values are rendered into scientific notation within the output character vector. The following major options are available:

- decimals: choice of the number of decimal places, option to drop trailing zeros, and a choice of the decimal symbol
- scaling: we can choose to scale targeted values by a multiplier value
- pattern: option to use a text pattern for decoration of the formatted values
- locale-based formatting: providing a locale ID will result in formatting specific to the chosen locale

Usage

```
vec_fmt_scientific(
  x,
  decimals = 2,
  drop_trailing_zeros = FALSE,
  scale_by = 1,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  force_sign = FALSE,
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x`	A numeric vector.
`decimals`	An option to specify the exact number of decimal places to use. The default number of decimal places is `2`.
`drop_trailing_zeros`	A logical value that allows for removal of trailing zeros (those redundant zeros after the decimal mark).
`scale_by`	A value to scale the input. The default is `1.0`. All numeric values will be multiplied by this value first before undergoing formatting.
`pattern`	A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.
`sep_mark`	The mark to use as a separator between groups of digits (e.g., using `sep_mark = ","` with `1000` would result in a formatted value of `1,000`).
`dec_mark`	The character to use as a decimal mark (e.g., using `dec_mark = ","` with `0.152` would result in a formatted value of `0,152`).
`force_sign`	Should the positive sign be shown for positive values (effectively showing a sign for all values except zero)? If so, use `TRUE` for this option. The default is `FALSE`, where only negative numbers will display a minus sign.
`locale`	An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported.
`output`	The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Examples

Let's create a numeric vector for the next few examples:

```
num_vals <- c(3.24e-4, 8.65, 1362902.2, -59027.3, NA)
```

Using `vec_fmt_scientific()` with the default options will create a character vector with values in scientific notation. Any `NA` values remain as `NA` values. The rendering context will be autodetected unless specified in the `output` argument (here, it is of the `"plain"` output type).

```
vec_fmt_scientific(num_vals)
```

```
#> [1] "3.24 × 10^-4" "8.65" "1.36 × 10^6" "-5.90 × 10^4" "NA"
```

We can change the number of decimal places with the `decimals` option:

```
vec_fmt_scientific(num_vals, decimals = 1)
```

```
#> [1] "3.2 × 10^-4" "8.7" "1.4 × 10^6" "-5.9 × 10^4" "NA"
```

If we are formatting for a different locale, we could supply the locale ID and **gt** will handle any locale-specific formatting options:

```
vec_fmt_scientific(num_vals, locale = "es")
```

```
#> [1] "3,24 × 10^-4" "8,65" "1,36 × 10^6" "-5,90 × 10^4" "NA"
```

Should you need to have positive and negative signs on each of the output values, use `force_sign = TRUE`:

```
vec_fmt_scientific(num_vals, force_sign = TRUE)
```

```
#> [1] "+3.24 × 10^-4" "+8.65" "+1.36 × 10^6" "-5.90 × 10^4" "NA"
```

As a last example, one can wrap the values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_scientific(num_vals, pattern = "[{x}]")
```

```
#> [1] "[3.24 × 10^-4]" "[8.65]" "[1.36 × 10^6]" "[-5.90 × 10^4]" "NA"
```

Function ID

14-3

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_time()`

vec_fmt_time *Format a vector as time values*

Description

Format vector values to time values using one of 25 preset time styles. Input can be in the form of `POSIXt` (i.e., datetimes), `character` (must be in the ISO 8601 forms of `HH:MM:SS` or `YYYY-MM-DD HH:MM:SS`), or `Date` (which always results in the formatting of `00:00:00`).

Usage

```
vec_fmt_time(
  x,
  time_style = "iso",
  pattern = "{x}",
  locale = NULL,
  output = c("auto", "plain", "html", "latex", "rtf", "word")
)
```

Arguments

`x`	A numeric vector.
`time_style`	The time style to use. By default this is `"iso"` which corresponds to how times are formatted within ISO 8601 datetime values. The other time styles can be viewed using `info_time_style()`.
`pattern`	A formatting pattern that allows for decoration of the formatted value. The value itself is represented by `{x}` and all other characters are taken to be string literals.
`locale`	An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en"` for English (United States) and `"fr"` for French (France). The use of a valid locale ID will override any values provided in `sep_mark` and `dec_mark`. We can use the `info_locales()` function as a useful reference for all of the locales that are supported.
`output`	The output style of the resulting character vector. This can either be `"auto"` (the default), `"plain"`, `"html"`, `"latex"`, `"rtf"`, or `"word"`. In **knitr** rendering (i.e., Quarto or R Markdown), the `"auto"` option will choose the correct `output` value

Value

A character vector.

Formatting with the `time_style` argument

We need to supply a preset time style to the `time_style` argument. There are many time styles and all of them can handle localization to any supported locale. Many of the time styles are termed flexible time formats and this means that their output will adapt to any `locale` provided. That feature makes the flexible time formats a better option for locales other than "en" (the default locale).

The following table provides a listing of all time styles and their output values (corresponding to an input time of 14:35:00). It is noted which of these represent 12- or 24-hour time.

	Time Style	Output	Notes
1	"iso"	"14:35:00"	ISO 8601, 24h
2	"iso-short"	"14:35"	ISO 8601, 24h
3	"h_m_s_p"	"2:35:00 PM"	12h
4	"h_m_p"	"2:35 PM"	12h
5	"h_p"	"2 PM"	12h
6	"Hms"	"14:35:00"	flexible, 24h
7	"Hm"	"14:35"	flexible, 24h
8	"H"	"14"	flexible, 24h
9	"EHm"	"Thu 14:35"	flexible, 24h
10	"EHms"	"Thu 14:35:00"	flexible, 24h
11	"Hmsv"	"14:35:00 GMT+00:00"	flexible, 24h
12	"Hmv"	"14:35 GMT+00:00"	flexible, 24h
13	"hms"	"2:35:00 PM"	flexible, 12h
14	"hm"	"2:35 PM"	flexible, 12h
15	"h"	"2 PM"	flexible, 12h
16	"Ehm"	"Thu 2:35 PM"	flexible, 12h
17	"Ehms"	"Thu 2:35:00 PM"	flexible, 12h
18	"EBhms"	"Thu 2:35:00 in the afternoon"	flexible, 12h
19	"Bhms"	"2:35:00 in the afternoon"	flexible, 12h
20	"EBhm"	"Thu 2:35 in the afternoon"	flexible, 12h
21	"Bhm"	"2:35 in the afternoon"	flexible, 12h
22	"Bh"	"2 in the afternoon"	flexible, 12h
23	"hmsv"	"2:35:00 PM GMT+00:00"	flexible, 12h
24	"hmv"	"2:35 PM GMT+00:00"	flexible, 12h
25	"ms"	"35:00"	flexible

We can use the info_time_style() within the console to view a similar table of time styles with example output.

Examples

Let's create a character vector of datetime values in the ISO-8601 format for the next few examples:

```
str_vals <- c("2022-06-13 18:36", "2019-01-25 01:08", NA)
```

Using vec_fmt_time() (here with the "iso-short" time style) will result in a character vector of formatted times. Any NA values remain as NA values. The rendering context will be autodetected unless specified in the output argument (here, it is of the "plain" output type).

```
vec_fmt_time(str_vals, time_style = "iso-short")
```

```
#> [1] "18:36" "01:08" NA
```

We can choose from any of 25 different time formatting styles. Many of these styles are flexible, meaning that the structure of the format will adapt to different locales. Let's use the "Bhms" time style to demonstrate this (first in the default locale of "en"):

```
vec_fmt_time(str_vals, time_style = "Bhms")
```

```
#> [1] "6:36:00 in the evening" "1:08:00 at night" NA
```

Let's perform the same type of formatting in the German ("de") locale:

```
vec_fmt_time(str_vals, time_style = "Bhms", locale = "de")
```

```
#> [1] "6:36:00 abends" "1:08:00 nachts" NA
```

We can always use `info_time_style()` to call up an info table that serves as a handy reference to all of the `time_style` options.

As a last example, one can wrap the time values in a pattern with the `pattern` argument. Note here that `NA` values won't have the pattern applied.

```
vec_fmt_time(
  str_vals,
  time_style = "hm",
  pattern = "temps: {x}",
  locale = "fr_CA"
)
```

```
#> [1] "temps: 6:36 PM" "temps: 1:08 AM" NA
```

Function ID

14-12

See Also

Other vector formatting functions: `vec_fmt_bytes()`, `vec_fmt_currency()`, `vec_fmt_datetime()`, `vec_fmt_date()`, `vec_fmt_duration()`, `vec_fmt_engineering()`, `vec_fmt_fraction()`, `vec_fmt_integer()`, `vec_fmt_markdown()`, `vec_fmt_number()`, `vec_fmt_partsper()`, `vec_fmt_percent()`, `vec_fmt_roman()`, `vec_fmt_scientific()`

web_image *Helper function for adding an image from the web*

Description

We can flexibly add a web image inside of a table with `web_image()` function. The function provides a convenient way to generate an HTML fragment with an image URL. Because this function is currently HTML-based, it is only useful for HTML table output. To use this function inside of data cells, it is recommended that the `text_transform()` function is used. With that function, we can specify which data cells to target and then include a `web_image()` call within the required user-defined function (for the `fn` argument). If we want to include an image in other places (e.g., in the header, within footnote text, etc.) we need to use `web_image()` within the `html()` helper function.

Usage

```
web_image(url, height = 30)
```

Arguments

`url`	A url that resolves to an image file.
`height`	The absolute height (px) of the image in the table cell.

Details

By itself, the function creates an HTML image tag, so, the call `web_image("http://example.com/image.png")` evaluates to:
`<img src=\"http://example.com/image.png\" style=\"height:30px;\">`

where a height of `30px` is a default height chosen to work well within the heights of most table rows.

Value

A character object with an HTML fragment that can be placed inside of a cell.

Examples

Get the PNG-based logo for the R Project from an image URL.

```
r_png_url <- "https://www.r-project.org/logo/Rlogo.png"
```

Create a tibble that contains heights of an image in pixels (one column as a string, the other as numerical values), then, create a **gt** table. Use the `text_transform()` function to insert the R logo PNG image with the various sizes.

```
dplyr::tibble(
  pixels = px(seq(10, 35, 5)),
  image = seq(10, 35, 5)
) %>%
```

```
  gt() %>%
  text_transform(
    locations = cells_body(columns = image),
    fn = function(x) {
      web_image(
        url = r_png_url,
        height = as.numeric(x)
      )
    }
  )
```

Get the SVG-based logo for the R Project from an image URL.

```
r_svg_url <- "https://www.r-project.org/logo/Rlogo.svg"
```

Create a tibble that contains heights of an image in pixels (one column as a string, the other as numerical values), then, create a **gt** table. Use the `tab_header()` function to insert the **R** logo SVG image once in the title and five times in the subtitle.

```
dplyr::tibble(
  pixels = px(seq(10, 35, 5)),
  image = seq(10, 35, 5)
) %>%
  gt() %>%
  tab_header(
    title = html(
      "<strong>R Logo</strong>",
      web_image(
        url = r_svg_url,
        height = px(50)
      )
    ),
    subtitle = html(
      web_image(
        url = r_svg_url,
        height = px(12)
      ) %>%
        rep(5)
    )
  )
```

Function ID

8-1

See Also

Other image addition functions: `ggplot_image()`, `local_image()`, `test_image()`

Index