Package 'gt'

August 6, 2020

Type Package

Version 0.2.2

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://github.com/rstudio/gt

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

Encoding UTF-8

LazyData true

ByteCompile true

RoxygenNote 7.1.1

Depends R (>= 3.2.0)

Imports checkmate (>= 2.0.0), commonmark (>= 1.7), dplyr (>= 0.8.5), fs (>= 1.3.2), ggplot2 (>= 3.3.0), glue (>= 1.3.2), htmltools (>= 0.5.0), magrittr (>= 1.5), rlang (>= 0.4.5), sass (>= 0.1.1), scales (>= 1.1.0), stringr (>= 1.3.1), tibble (>= 3.0.0), tidyselect (>= 1.0.0)

Suggests covr, knitr, paletteer, testthat (>= 2.1.0), RColorBrewer, rmarkdown, rvest, shiny, tidyr, webshot, xml2

Collate 'as_data_frame.R' 'base64.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_options.R' 'dt_row_groups.R' 'dt_source_notes.R' 'dt_spanners.R' 'dt_stub_df.R' 'dt_stub_others.R'

'dt_stubhead.R' 'dt_styles.R' 'dt_summary.R' 'dt_transforms.R'
'export.R' 'format_data.R' 'gt-package.R' 'gt.R' 'gt_preview.R'
'helpers.R' 'image.R' 'info_tables.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' 'summary_rows.R' 'text_transform.R'
'utils.R' 'utils_formatters.R' 'utils_general_str_formatting.R'
'utils_pipe.R' 'utils_render_common.R'
'utils_render_footnotes.R' 'utils_render_html.R'
'utils_render_latex.R' 'utils_render_rtf.R' 'zzz.R'

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>),
RStudio [cph, fnd]

Maintainer Richard Iannone <rich@rstudio.com>

Repository CRAN

Date/Publication 2020-08-05 22:00:02 UTC

R topics documented:

`gt-package` *gt: 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.

Author(s)

Maintainer: Richard Iannone `<rich@rstudio.com>` (ORCID)

Authors:

- Joe Cheng `<joe@rstudio.com>`
- Barret Schloerke `<barret@rstudio.com>` (ORCID)

Other contributors:

- RStudio [copyright holder, funder]

See Also

Useful links:

- https://github.com/rstudio/gt
- Report bugs at https://github.com/rstudio/gt/issues

`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.

Figures

Function ID

7-20

See Also

Other Helper Functions: cell_borders(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_grand_summary(), cells_row_groups(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id()

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()`)
tab_1 <-
  dplyr::tibble(a = 1:8, b = 1:8, c = 1:8) %>%
  gt() %>%
  data_color(
    columns = vars(a),
    colors = scales::col_numeric(
      palette = pal_lighter,
      domain = c(1, 8)
    )
  ) %>%
  data_color(
    columns = vars(b),
    colors = scales::col_numeric(
      palette = pal,
      domain = c(1, 8)
    )
  ) %>%
  data_color(
    columns = vars(c),
    colors = scales::col_numeric(
      palette = pal_darker,
      domain = c(1, 8)
    )
  )
```

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.

Function ID

13-3

See Also

Other Export Functions: `as_raw_html()`, `as_rtf()`, `extract_summary()`, `gtsave()`

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()

# `tab_latex` 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
tab_latex %>%
```

```
as.character() %>%
cat()
```

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.

Function ID

13-2

See Also

Other Export Functions: `as_latex()`, `as_rtf()`, `extract_summary()`, `gtsave()`

Examples

```
# Use `gtcars` to create a gt table;
# add a header and then export as
# HTML code with CSS inlined
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()
```

```
# `tab_html` is a single-element vector
# containing inlined HTML for the table;
# it has only the `<table>...</table>` part
# so it's not a complete HTML document but
# rather an HTML fragment
tab_html %>%
  substr(1, 700) %>%
  cat()
```

---

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.

Function ID

13-4

See Also

Other Export Functions: `as_latex()`, `as_raw_html()`, `extract_summary()`, `gtsave()`

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()
```

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.

Usage

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

Arguments

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

Details

When using any of the location helper functions with an appropriate function that has a locations argument, multiple locations can be targeted by enclosing several cells_*() helper functions in a list(). The following helper functions can be used to target cells (roughly in order from the top to the 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, which appear above the column labels.
- cells_column_labels(): targets the column labels.
- 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

Value

A list object with the classes cells_body and location_cells.

Figures

Function ID

7-11

See Also

Other Helper Functions: adjust_luminance(), cell_borders(), cell_fill(), cell_text(), cells_column_labels(), cells_column_spanners(), cells_grand_summary(), cells_row_groups(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id()

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)
tab_1 <-
  gtcars %>%
  dplyr::filter(ctry_origin == "United Kingdom") %>%
  dplyr::select(mfr, model, year, hp) %>%
  gt() %>%
  tab_footnote(
    footnote = "Highest horsepower.",
    locations = cells_body(
      columns = vars(hp),
      rows = hp == max(hp))
  ) %>%
  opt_footnote_marks(marks = c("*", "+"))
```

---

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.

Usage

```
cells_column_labels(columns)
```

Arguments

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

Details

When using any of the location helper functions with an appropriate function that has a `locations` argument, multiple locations can be targeted by enclosing several cells_*() helper functions in a `list()`. The following helper functions can be used to target cells (roughly in order from the top to the 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, which appear above the column labels.
- `cells_column_labels()`: targets the column labels.
- `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`

Value

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

Figures

Function ID

7-8

See Also

Other Helper Functions: `adjust_luminance()`, `cell_borders()`, `cell_fill()`, `cell_text()`, `cells_body()`, `cells_column_spanners()`, `cells_grand_summary()`, `cells_row_groups()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`

Examples

```
# Use `sza` to create a gt table; add a
# header and then add footnotes to the
# column labels with `tab_footnote()` and
# `cells_column_labels()` in `locations`
tab_1 <-
  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 = vars(tst)
    )
  ) %>%
  tab_footnote(
    footnote = "Solar zenith angle.",
    locations = cells_column_labels(
      columns = vars(sza)
    )
  )
```

`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.

Usage

```
cells_column_spanners(spanners)
```

Arguments

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

Details

When using any of the location helper functions with an appropriate function that has a `locations` argument, multiple locations can be targeted by enclosing several cells_*() helper functions in a `list()`. The following helper functions can be used to target cells (roughly in order from the top to the 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, which appear above the column labels.
- `cells_column_labels()`: targets the column labels.
- `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`

Value

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

Figures

Function ID

7-7

See Also

Other Helper Functions: `adjust_luminance()`, `cell_borders()`, `cell_fill()`, `cell_text()`, `cells_body()`, `cells_column_labels()`, `cells_grand_summary()`, `cells_row_groups()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`

Examples

```
# Use `exibble` to create a gt table; add a
# spanner column label over three column
# labels and then use `tab_style()` to make
# the spanner label text bold
tab_1 <-
  exibble %>%
  dplyr::select(-fctr, -currency, -group) %>%
  gt(rowname_col = "row") %>%
  tab_spanner(
    label = "dates and times",
    columns = vars(date, time, datetime)
```

```
  ) %>%
  tab_style(
    style = cell_text(weight = "bold"),
    locations = cells_column_spanners(spanners = "dates and times")
  )
```

---

cells_data *Location helper for targeting data cells in the table body (deprecated)*

Description

Location helper for targeting data cells in the table body (deprecated)

Usage

```
cells_data(columns = TRUE, rows = TRUE)
```

Arguments

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

---

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.

Usage

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

Arguments

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

Details

When using any of the location helper functions with an appropriate function that has a `locations` argument, multiple locations can be targeted by enclosing several cells_*() helper functions in a `list()`. The following helper functions can be used to target cells (roughly in order from the top to the 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, which appear above the column labels.
- `cells_column_labels()`: targets the column labels.
- `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`

Value

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

Figures

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_row_groups()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`

Examples

```
# Use `countrypops` to create a gt table; add
# some styling to a grand summary cell with
# with `tab_style()` and `cells_grand_summary()`
tab_1 <-
```

```
countrypops %>%
dplyr::filter(
  country_name == "Spain",
  year < 1970
) %>%
dplyr::select(-contains("country")) %>%
gt(rowname_col = "year") %>%
fmt_number(
  columns = vars(population),
  decimals = 0
) %>%
grand_summary_rows(
  columns = vars(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 = vars(population),
    rows = 1)
)
```

`cells_group` *Location helper for targeting row groups (deprecated)*

Description

Location helper for targeting row groups (deprecated)

Usage

```
cells_group(groups = TRUE)
```

Arguments

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

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.

Usage

```
cells_row_groups(groups = TRUE)
```

Arguments

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

Details

When using any of the location helper functions with an appropriate function that has a locations argument, multiple locations can be targeted by enclosing several cells_*() helper functions in a list(). The following helper functions can be used to target cells (roughly in order from the top to the 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, which appear above the column labels.
- cells_column_labels(): targets the column labels.
- 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

Value

A list object with the classes cells_row_groups and location_cells.

Figures

Function ID

7-9

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_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id()

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`
tab_1 <-
  pizzaplace %>%
  dplyr::filter(
    name %in% c("soppressata", "peppr_salami")
  ) %>%
  dplyr::group_by(name, size) %>%
  dplyr::summarize(
    `Pizzas Sold` = dplyr::n()
  ) %>%
  gt(rowname_col = "size") %>%
  summary_rows(
    groups = TRUE,
    columns = vars("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")
  )
```

`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.

Usage

```
cells_stub(rows = TRUE)
```

Arguments

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

Details

When using any of the location helper functions with an appropriate function that has a `locations` argument, multiple locations can be targeted by enclosing several cells_*() helper functions in a `list()`. The following helper functions can be used to target cells (roughly in order from the top to the 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, which appear above the column labels.
- `cells_column_labels()`: targets the column labels.
- `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`

Value

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

Figures

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_grand_summary()`, `cells_row_groups()`, `cells_stubhead()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`

Examples

```
library(tidyr)

# 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` (`rows = TRUE` targets
# all stub rows)
tab_1 <-
  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") %>%
  fmt_missing(
    columns = TRUE,
    missing_text = ""
  ) %>%
  tab_style(
    style = list(
      cell_fill(color = "darkblue"),
      cell_text(color = "white")
      ),
    locations = cells_stub(rows = TRUE)
  )
```

`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.

Usage

```
cells_stubhead()
```

Details

When using any of the location helper functions with an appropriate function that has a `locations` argument, multiple locations can be targeted by enclosing several cells_*() helper functions in a `list()`. The following helper functions can be used to target cells (roughly in order from the top to the 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, which appear above the column labels.
- `cells_column_labels()`: targets the column labels.
- `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`

Value

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

Figures

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_grand_summary()`, `cells_row_groups()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`

Examples

```
# Use `pizzaplace` to create a gt table;
# add a stubhead label and then style it
# with `tab_style()` and `cells_stubhead()`
tab_1 <-
  pizzaplace %>%
  dplyr::mutate(month = as.numeric(substr(date, 6, 7))) %>%
  dplyr::group_by(month, type) %>%
  dplyr::summarize(sold = dplyr::n()) %>%
  dplyr::ungroup() %>%
  dplyr::filter(month %in% 1:2) %>%
  gt(rowname_col = "type") %>%
  tab_stubhead(label = "type") %>%
  tab_style(
    style = cell_fill(color = "lightblue"),
    locations = cells_stubhead()
  )
```

`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.

Usage

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

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.

Details

When using any of the location helper functions with an appropriate function that has a `locations` argument, multiple locations can be targeted by enclosing several cells_*() helper functions in a `list()`. The following helper functions can be used to target cells (roughly in order from the top to the 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, which appear above the column labels.
- `cells_column_labels()`: targets the column labels.
- `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`

Value

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

Figures

Function ID

7-12

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_stubhead()`, `cells_stub()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`

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`
```

```
tab_1 <-
  countrypops %>%
  dplyr::filter(
    country_name == "Japan",
    year < 1970) %>%
  dplyr::select(-contains("country")) %>%
  dplyr::mutate(
    decade = paste0(substr(year, 1, 3), "0s")
  ) %>%
  dplyr::group_by(decade) %>%
  gt(
    rowname_col = "year",
    groupname_col = "decade"
  ) %>%
  fmt_number(
    columns = vars(population),
    decimals = 0
  ) %>%
  summary_rows(
    groups = "1960s",
    columns = vars(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 = vars(population),
      rows = 1)
  ) %>%
  tab_style(
    style = list(
      cell_text(style = "italic"),
      cell_fill(color = "lightgreen")
      ),
    locations = cells_summary(
      groups = "1960s",
      columns = vars(population),
      rows = 2)
  )
```

`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.

Usage

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

Arguments

`groups`	We can either specify `"title"` or `"subtitle"` to target the title element or the subtitle element.

Details

When using any of the location helper functions with an appropriate function that has a `locations` argument, multiple locations can be targeted by enclosing several cells_*() helper functions in a `list()`. The following helper functions can be used to target cells (roughly in order from the top to the 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, which appear above the column labels.
- `cells_column_labels()`: targets the column labels.
- `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`

Value

A list object of classes `cells_title` and `location_cells`.

Figures

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_grand_summary(), cells_row_groups(), cells_stubhead(), cells_stub(), cells_summary(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id()

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`)
tab_1 <-
  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")
  )
```

`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", or "dotted". 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.

Figures

Function ID

7-17

See Also

Other Helper Functions: adjust_luminance(), cell_fill(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_grand_summary(), cells_row_groups(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id()

Examples

```
# Add horizontal border lines for
# all table body rows in `exibble`
tab_1 <-
  exibble %>%
    gt() %>%
    tab_options(row.striping.include_table_body = FALSE) %>%
    tab_style(
      style = cell_borders(
        sides = c("top", "bottom"),
        color = "#BBBBBB",
        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
tab_2 <-
  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 = vars(num),
        rows = is.na(num)
      ),
      cells_body(
        columns = vars(currency),
        rows = is.na(currency)
      )
    )
  )
```

`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.

Figures

Function ID

7-16

See Also

Other Helper Functions: adjust_luminance(), cell_borders(), cell_text(), cells_body(), cells_column_labels(), cells_column_spanners(), cells_grand_summary(), cells_row_groups(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id()

Examples

```
# Use `exibble` to create a gt table;
# add styles with `tab_style()` and
# the `cell_fill()` helper function
tab_1 <-
  exibble %>%
  dplyr::select(num, currency) %>%
  gt() %>%
  fmt_number(
    columns = vars(num, currency),
    decimals = 1
  ) %>%
  tab_style(
    style = cell_fill(color = "lightblue"),
    locations = cells_body(
      columns = vars(num),
      rows = num >= 5000)
  ) %>%
  tab_style(
    style = cell_fill(color = "gray85"),
    locations = cells_body(
      columns = vars(currency),
      rows = currency < 100
```

```
    )
  )
```

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,
  indent = NULL,
  decorate = NULL,
  transform = 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.

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()`).

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"`.

Value

A list object of class `cell_styles`.

Figures

Function ID

7-15

See Also

Other Helper Functions: `adjust_luminance()`, `cell_borders()`, `cell_fill()`, `cells_body()`, `cells_column_labels()`, `cells_column_spanners()`, `cells_grand_summary()`, `cells_row_groups()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`

Examples

```
# Use `exibble` to create a gt table;
# add styles with `tab_style()` and
# the `cell_text()` helper function
tab_1 <-
  exibble %>%
  dplyr::select(num, currency) %>%
  gt() %>%
  fmt_number(
    columns = vars(num, currency),
    decimals = 1
  ) %>%
  tab_style(
    style = cell_text(weight = "bold"),
    locations = cells_body(
      columns = vars(num),
```

```
    rows = num >= 5000)
) %>%
tab_style(
  style = cell_text(style = "italic"),
  locations = cells_body(
    columns = vars(currency),
    rows = currency < 100
  )
)
```

`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 = TRUE)
```

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 center-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`	An optional vector of column names for which the alignment should be applied. If nothing is supplied, or if `columns` is `TRUE`, then 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`.

Figures

Function ID

4-1

See Also

Other Modify Columns: cols_hide(), cols_label(), cols_merge_range(), cols_merge_uncert(), cols_merge(), cols_move_to_end(), cols_move_to_start(), cols_move(), cols_width()

Examples

```
# Use `countrypops` to create a gt table;
# align the `population` column data to
# the left
tab_1 <-
  countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_align(
    align = "left",
    columns = vars(population)
  )
```

---

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. The order of the remaining columns will be preserved. 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.

Value

An object of class `gt_tbl`.

Figures

Function ID

4-7

See Also

Other Modify Columns: `cols_align()`, `cols_label()`, `cols_merge_range()`, `cols_merge_uncert()`, `cols_merge()`, `cols_move_to_end()`, `cols_move_to_start()`, `cols_move()`, `cols_width()`

Examples

```
# Use `countrypops` to create a gt table;
# Hide the columns `country_code_2` and
# `country_code_3`
tab_1 <-
  countrypops %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_hide(
    columns = vars(
      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
tab_2 <-
  countrypops %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_hide(
    columns = vars(country_code_3, population)
  ) %>%
```

```
  tab_footnote(
    footnote = "Population above 3,000,000.",
    locations = cells_body(
      columns = vars(year),
      rows = population > 3000000)
  )
```

`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`.

Figures

Function ID

4-3

See Also

Other Modify Columns: cols_align(), cols_hide(), cols_merge_range(), cols_merge_uncert(), cols_merge(), cols_move_to_end(), cols_move_to_start(), cols_move(), cols_width()

Examples

```
# Use `countrypops` to create a gt table;
# label all the table's columns to
# present better
tab_1 <-
  countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_label(
    country_name = "Name",
    year = "Year",
    population = "Population"
  )

# Use `countrypops` to create a gt table;
# label columns as before but make them
# bold with markdown formatting
tab_2 <-
  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**")
  )
```

---

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.
`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 two other column-merging functions that offer specialized behavior that is optimized for common table tasks: `cols_merge_range()` and `cols_merge_uncert()`. 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`.

Figures

Function ID

4-10

See Also

Other Modify Columns: cols_align(), cols_hide(), cols_label(), cols_merge_range(), cols_merge_uncert(), cols_move_to_end(), cols_move_to_start(), cols_move(), cols_width()

Examples

```
# Use `sp500` to create a gt table;
# merge the `open` & `close` columns
# together, and, the `low` & `high`
# columns (putting an em dash between
# both); rename the columns
tab_1 <-
  sp500 %>%
  dplyr::slice(50:55) %>%
  dplyr::select(-volume, -adj_close) %>%
  gt() %>%
  cols_merge(
    columns = vars(open, close),
    hide_columns = vars(close),
    pattern = "{1}&mdash;{2}"
  ) %>%
  cols_merge(
    columns = vars(low, high),
    hide_columns = vars(high),
    pattern = "{1}&mdash;{2}"
  ) %>%
  cols_label(
    open = "open/close",
    low = "low/high"
  )
```

---

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 fmt_missing() function. Separate calls of fmt_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 three column-merging functions. The other two are the general cols_merge() function and the specialized cols_merge_uncert() function. 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.

Figures

Function ID

4-9

See Also

Other Modify Columns: cols_align(), cols_hide(), cols_label(), cols_merge_uncert(), cols_merge(), cols_move_to_end(), cols_move_to_start(), cols_move(), cols_width()

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)
tab_1 <-
  gtcars %>%
  dplyr::select(model, starts_with("mpg")) %>%
  dplyr::slice(1:8) %>%
  gt() %>%
  cols_merge_range(
    col_begin = vars(mpg_c),
    col_end = vars(mpg_h)
  ) %>%
  cols_label(
    mpg_c = md("*MPG*")
  )
```

---

cols_merge_uncert — *Merge two columns to a value & uncertainty column*

Description

The `cols_merge_uncert()` function is a specialized variant of the `cols_merge()` function. It operates by taking a base value column (`col_val`) and an uncertainty column (`col_uncert`) and merges them into a single column. What results is a column with values and associated uncertainties (e.g., 12.0 ± 0.1), and, the column specified in `col_uncert` is dropped from 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`	A single column name that contains the uncertainty values. These values will be combined with those in `col_val`. We have the option to automatically hide the `col_uncert` column through `autohide`.
`sep`	The separator text that contains the uncertainty mark. 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 base `I()` function.

autohide
: An option to automatically hide the column specified as 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.

Details

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

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

This function is part of a set of three column-merging functions. The other two are the general cols_merge() function and the specialized cols_merge_range() function. 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.

Figures

Function ID

4-8

See Also

Other Modify Columns: cols_align(), cols_hide(), cols_label(), cols_merge_range(), cols_merge(), cols_move_to_end(), cols_move_to_start(), cols_move(), cols_width()

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)
tab_1 <-
  exibble %>%
  dplyr::select(currency, num) %>%
  dplyr::slice(1:7) %>%
```

```
gt() %>%
fmt_number(
  columns = vars(num),
  decimals = 3,
  use_seps = FALSE
) %>%
cols_merge_uncert(
  col_val = vars(currency),
  col_uncert = vars(num)
) %>%
cols_label(
  currency = "value + uncert."
)
```

`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.

Figures

Function ID

4-6

See Also

Other Modify Columns: cols_align(), cols_hide(), cols_label(), cols_merge_range(), cols_merge_uncert(), cols_merge(), cols_move_to_end(), cols_move_to_start(), cols_width()

Examples

```
# Use `countrypops` to create a gt table;
# With the remaining columns, position
# `population` after `country_name`
tab_1 <-
  countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_move(
    columns = vars(population),
    after = vars(country_name)
  )
```

---

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`.

Figures

Function ID

4-5

See Also

Other Modify Columns: `cols_align()`, `cols_hide()`, `cols_label()`, `cols_merge_range()`, `cols_merge_uncert()`, `cols_merge()`, `cols_move_to_start()`, `cols_move()`, `cols_width()`

Examples

```
# Use `countrypops` to create a gt table;
# With the remaining columns, move the
# `year` column to the end
tab_1 <-
  countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_move_to_end(
    columns = vars(year)
  )

# Use `countrypops` to create a gt table;
# With the remaining columns, move `year`
# and `country_name` to the end
tab_2 <-
  countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
```

```
tail(5) %>%
gt() %>%
cols_move_to_end(
  columns = vars(year, country_name)
)
```

`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`.

Figures

Function ID

4-4

See Also

Other Modify Columns: cols_align(), cols_hide(), cols_label(), cols_merge_range(), cols_merge_uncert(), cols_merge(), cols_move_to_end(), cols_move(), cols_width()

Examples

```
# Use `countrypops` to create a gt table;
# With the remaining columns, move the
# `year` column to the start
tab_1 <-
  countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_move_to_start(
    columns = vars(year)
  )

# Use `countrypops` to create a gt table;
# With the remaining columns, move `year`
# and `population` to the start
tab_2 <-
  countrypops %>%
  dplyr::select(-contains("code")) %>%
  dplyr::filter(country_name == "Mongolia") %>%
  tail(5) %>%
  gt() %>%
  cols_move_to_start(
    columns = vars(year, population)
  )
```

---

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 `vars()`. 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 `TRUE` or `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`.

Figures

Function ID

4-2

See Also

Other Modify Columns: `cols_align()`, `cols_hide()`, `cols_label()`, `cols_merge_range()`, `cols_merge_uncert()`, `cols_merge()`, `cols_move_to_end()`, `cols_move_to_start()`, `cols_move()`

Examples

```
# Use `exibble` to create a gt table;
# with named arguments in `...`, we
# can specify the exact widths for
# table columns (using `everything()`
# or `TRUE` at the end will capture
# all remaining columns)
tab_1 <-
```

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

---

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

Function ID

11-1

Source

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

See Also

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

Examples

```
# Here is a glimpse at the data
# available in `countrypops`
dplyr::glimpse(countrypops)
```

---

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`.

Figures

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_grand_summary()`, `cells_row_groups()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`

Examples

```
# Use `exibble` to create a gt table;
# format the `currency` column to have
# currency values in guilder (a defunct
# Dutch currency)
tab_1 <-
  exibble %>%
  gt() %>%
  fmt_currency(
    columns = vars(currency),
    currency = currency(
      html = "&fnof;",
      default = "f"),
    decimals = 2
  )
```

`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
)
```

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 in some cases yield more optimal text-to-background color contrast. By default, this is set to `TRUE`.

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`.

Figures

Function ID

3-13

See Also

Other Format Data: `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_markdown()`, `fmt_missing()`, `fmt_number()`, `fmt_passthrough()`, `fmt_percent()`, `fmt_scientific()`, `fmt_time()`, `fmt()`, `text_transform()`

Examples

```
# library(paletteer)

# 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
tab_1 <-
  countrypops %>%
  dplyr::filter(country_name == "Mongolia") %>%
  dplyr::select(-contains("code")) %>%
  tail(10) %>%
  gt() %>%
  data_color(
    columns = vars(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 `red_material`
# palette (in the `ggsci` pkg 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
tab_2 <-
  pizzaplace %>%
  dplyr::filter(
    type %in% c("chicken", "supreme")) %>%
  dplyr::group_by(type, size) %>%
  dplyr::summarize(
    sold = dplyr::n(),
    income = sum(price)
  ) %>%
  gt(rowname_col = "size") %>%
  data_color(
    columns = vars(sold, income),
    colors = scales::col_numeric(
      palette = paletteer::paletteer_d(
        palette = "ggsci::red_material"
        ) %>% as.character(),
      domain = NULL
      )
  )
```

`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.

Figures

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_grand_summary(), cells_row_groups(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id()

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)
tab_1 <-
  exibble %>%
  dplyr::select(char, time) %>%
  gt() %>%
  tab_style(
    style = cell_text(
      font = c(
        "Comic Sansa", "Menloa",
        default_fonts()
      )
    ),
    locations = cells_body(columns = vars(time))
  )
```

---

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

7-22

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_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`, `random_id()`

`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

Function ID

11-6

See Also

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

Examples

```
# Here is a glimpse at the data
# available in `exibble`
dplyr::glimpse(exibble)
```

`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 `groupname` 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.

Figures

Function ID

13-5

See Also

Other Export Functions: as_latex(), as_raw_html(), as_rtf(), gtsave()

Examples

```
# Use `sp500` to create a gt table with
# row groups; create summary rows by row
# group (`min`, `max`, `avg`) and 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 = vars(open, high, low, close),
    fns = list(
      min = ~min(.),
      max = ~max(.),
      avg = ~mean(.)),
    formatter = fmt_number,
    use_seps = FALSE
  ) %>%
  extract_summary()

# 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()` (the `groupname` and
# `rowname` magic column names create
# row groups and a stub)
tab_1 <-
  summary_extracted %>%
  unlist(recursive = FALSE) %>%
  dplyr::bind_rows() %>%
```

`gt()`

`fmt` *Set a column format with a formatter function*

Description

The `fmt()` function provides greater control in formatting raw data values than any of the specialized fmt_*() functions that are available in **gt**. 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`, 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 = NULL, rows = NULL, 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 `vars()`, 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()`, and `everything()`.
`rows`	Optional rows to format. Not providing any value results in all rows in `columns` being formatted. Can either be a vector of row captions provided `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()`. We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 50).
`fns`	Either a single formatting function or a named list of functions.

Details

As with all of the fmt_*() functions, targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). A number of helper functions exist to make targeting more effective. 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.

Figures

Function ID

3-11

See Also

Other Format Data: data_color(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_markdown(), fmt_missing(), fmt_number(), fmt_passthrough(), fmt_percent(), fmt_scientific(), fmt_time(), text_transform()

Examples

```
# Use `exibble` to create a gt table;
# format the numeric values in the `num`
# column with a function supplied to
# the `fns` argument
tab_1 <-
  exibble %>%
  dplyr::select(-row, -group) %>%
  gt() %>%
  fmt(
    columns = vars(num),
    fns = function(x) {
      paste0("'", x * 1000, "'")
    }
  )
```

---

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 = NULL,
  currency = "USD",
  use_subunits = TRUE,
  accounting = FALSE,
  decimals = NULL,
  drop_trailing_dec_mark = TRUE,
  use_seps = TRUE,
  scale_by = 1,
  suffixing = FALSE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  placement = "left",
  incl_space = 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 `vars()`, 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()`, and `everything()`.

rows
: Optional rows to format. Not providing any value results in all rows in columns being formatted. Can either be a vector of row captions provided 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(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 50).

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.

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

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.

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).

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_US" for English (United States) and "fr_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.

Details

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). A number of helper functions exist to make targeting more effective. 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.

Figures

Function ID

3-4

See Also

Other Format Data: data_color(), fmt_datetime(), fmt_date(), fmt_markdown(), fmt_missing(), fmt_number(), fmt_passthrough(), fmt_percent(), fmt_scientific(), fmt_time(), fmt(), text_transform()

Examples

```
# Use `exibble` to create a gt table;
# format the `currency` column to have
# currency values in euros (EUR)
tab_1 <-
  exibble %>%
  gt() %>%
  fmt_currency(
    columns = vars(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
tab_2 <-
  exibble %>%
  dplyr::select(num, currency) %>%
  gt() %>%
  fmt_currency(
    columns = vars(num),
    currency = "CNY"
  ) %>%
  fmt_currency(
    columns = vars(currency),
    currency = "GBP"
  )
```

---

fmt_date *Format values as dates*

Description

Format input date values that are either of the `Date` type, or, are character-based and expressed according to the ISO 8601 date format (`YYYY-MM-DD`). Once the appropriate data cells are targeted with `columns` (and, optionally, `rows`), we can simply apply a preset date style to format the dates. The following date styles are available for simpler formatting of ISO dates (all using the input date of `2000-02-29` in the example output dates):

1. iso: `2000-02-29`
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. year: `2000`
11. month: `February`
12. day: `29`
13. year.mn.day: `2000/02/29`
14. y.mn.day: `0/02/29`

We can use the `info_date_style()` function for a useful reference on all of the possible inputs to `date_style`.

Usage

```
fmt_date(data, columns, rows = NULL, date_style = 2)
```

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 `vars()`, 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()`, and `everything()`.
`rows`	Optional rows to format. Not providing any value results in all rows in `columns` being formatted. Can either be a vector of row captions provided `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()`. We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 50).
`date_style`	The date style to use. Supply a number (from `1` to `14`) that corresponds to the preferred date style. Use `info_date_style()` to see the different numbered and named date presets.

Details

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). A number of helper functions exist to make targeting more effective. 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`.

Figures

Function ID

3-5

See Also

Other Format Data: data_color(), fmt_currency(), fmt_datetime(), fmt_markdown(), fmt_missing(), fmt_number(), fmt_passthrough(), fmt_percent(), fmt_scientific(), fmt_time(), fmt(), text_transform()

Examples

```
# Use `exibble` to create a gt table;
# keep only the `date` and `time` columns;
# format the `date` column to have
# dates formatted as `month_day_year`
# (date style `5`)
tab_1 <-
  exibble %>%
  dplyr::select(date, time) %>%
  gt() %>%
  fmt_date(
    columns = vars(date),
    date_style = 5
  )

# 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)
tab_2 <-
  exibble %>%
  dplyr::select(date, time) %>%
  gt() %>%
  fmt_date(
    columns = vars(date),
    rows =
      as.Date(date) > as.Date("2015-04-01"),
    date_style = 6
  ) %>%
  fmt_date(
    columns = vars(date),
    rows =
      as.Date(date) <= as.Date("2015-04-01"),
    date_style = 7
  )
```

fmt_datetime *Format values as date-times*

Description

Format input date-time values that are character-based and expressed according to the ISO 8601 date-time format (YYYY-MM-DD HH:MM:SS). Once the appropriate data cells are targeted with `columns` (and, optionally, `rows`), we can simply apply preset date and time styles to format the date-time values. The following date styles are available for simpler formatting of the date portion (all using the input date of `2000-02-29` in the example output dates):

1. iso: `2000-02-29`
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. year: `2000`
11. month: `February`
12. day: `29`
13. year.mn.day: `2000/02/29`
14. y.mn.day: `0/02/29`

The following time styles are available for simpler formatting of the time portion (all using the input time of `14:35:00` in the example output times):

1. hms: `14:35:00`
2. hm: `14:35`
3. hms_p: 2:35:00 PM
4. hm_p: 2:35 PM
5. h_p: 2 PM

We can use the `info_date_style()` and `info_time_style()` functions as useful references for all of the possible inputs to `date_style` and `time_style`.

Usage

```
fmt_datetime(data, columns, rows = NULL, date_style = 2, time_style = 2)
```

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 vars(), 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(), and everything().
rows	Optional rows to format. Not providing any value results in all rows in columns being formatted. Can either be a vector of row captions provided 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(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 50).
date_style	The date style to use. Supply a number (from 1 to 14) that corresponds to the preferred date style. Use info_date_style() to see the different numbered and named date presets.
time_style	The time style to use. Supply a number (from 1 to 5) that corresponds to the preferred time style. Use info_time_style() to see the different numbered and named time presets.

Details

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). A number of helper functions exist to make targeting more effective. 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.

Figures

Function ID

3-7

See Also

Other Format Data: data_color(), fmt_currency(), fmt_date(), fmt_markdown(), fmt_missing(), fmt_number(), fmt_passthrough(), fmt_percent(), fmt_scientific(), fmt_time(), fmt(), text_transform()

Examples

```
# Use `exibble` to create a gt table;
# keep only the `datetime` column;
# format the column to have dates
# formatted as `month_day_year` and
# times to be `hms_p`
tab_1 <-
  exibble %>%
  dplyr::select(datetime) %>%
  gt() %>%
  fmt_datetime(
    columns = vars(datetime),
    date_style = 5,
    time_style = 3
  )
```

`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 = 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 `vars()`, 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()`, and `everything()`.
`rows`	Optional rows to format. Not providing any value results in all rows in `columns` being formatted. Can either be a vector of row captions provided `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()`. We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 50).

Details

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). A number of helper functions exist to make targeting more effective. 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`.

Figures

Function ID

3-8

See Also

Other Format Data: `data_color()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_missing()`, `fmt_number()`, `fmt_passthrough()`, `fmt_percent()`, `fmt_scientific()`, `fmt_time()`, `fmt()`, `text_transform()`

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()`
tab_1 <-
  dplyr::tribble(
    ~Markdown, ~md,
    text_1a,   text_2a,
    text_1b,   text_2b,
  ) %>%
    gt() %>%
    fmt_markdown(columns = TRUE) %>%
    tab_options(table.width = px(400))
```

`fmt_missing` *Format missing values*

Description

Wherever there is missing data (i.e., `NA` values) a customizable mark may present better than the standard `NA` text that would otherwise appear. The `fmt_missing()` function allows for this replacement through its `missing_text` argument (where an em dash serves as the default).

Usage

```
fmt_missing(data, columns, rows = NULL, 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 `vars()`, 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()`, and `everything()`.
`rows`	Optional rows to format. Not providing any value results in all rows in `columns` being formatted. Can either be a vector of row captions provided `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()`. We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 50).
`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). A number of helper functions exist to make targeting more effective. 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.

Figures

Function ID

3-10

See Also

Other Format Data: data_color(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_markdown(), fmt_number(), fmt_passthrough(), fmt_percent(), fmt_scientific(), fmt_time(), fmt(), text_transform()

Examples

```
# Use `exibble` to create a gt table;
# NA values in different columns will
# be given replacement text
tab_1 <-
  exibble %>%
  dplyr::select(-row, -group) %>%
  gt() %>%
  fmt_missing(
    columns = 1:2,
    missing_text = "missing"
  ) %>%
  fmt_missing(
    columns = 4:7,
    missing_text = "nothing"
  )
```

---

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 = NULL,
  decimals = 2,
  n_sigfig = NULL,
  drop_trailing_zeros = FALSE,
  drop_trailing_dec_mark = TRUE,
  use_seps = TRUE,
  scale_by = 1,
  suffixing = FALSE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  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 `vars()`, 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()`, and `everything()`.
`rows`	Optional rows to format. Not providing any value results in all rows in `columns` being formatted. Can either be a vector of row captions provided `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()`. We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 50).
`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.

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).

locale
: An optional locale ID that can be used for formatting the value according the locale's rules. Examples include `"en_US"` for English (United States) and `"fr_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.

Details

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). A number of helper functions exist to make targeting more effective. 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`.

Figures

Function ID

3-1

See Also

Other Format Data: `data_color()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_markdown()`, `fmt_missing()`, `fmt_passthrough()`, `fmt_percent()`, `fmt_scientific()`, `fmt_time()`, `fmt()`, `text_transform()`

Examples

```
library(tidyr)

# Use `exibble` to create a gt table;
# format the `num` column as numeric
# with three decimal places and with no
# use of digit separators
tab_1 <-
  exibble %>%
  gt() %>%
  fmt_number(
    columns = vars(num),
    decimals = 3,
    use_seps = FALSE
  )

# Use `countrypops` to create a gt
# table; format all numeric columns
# to use large-number suffixing
tab_2 <-
  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
```

```
)
```

`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 = NULL, 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 `vars()`, 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()`, and `everything()`.
`rows`	Optional rows to format. Not providing any value results in all rows in `columns` being formatted. Can either be a vector of row captions provided `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()`. We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 50).
`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` is useful in the case where LaTeX-formatted text should be passed through to the output LaTeX table unchanged.
`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.

Details

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). A number of helper functions exist to make targeting more effective. 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`.

Figures

Function ID

3-9

See Also

Other Format Data: `data_color()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_markdown()`, `fmt_missing()`, `fmt_number()`, `fmt_percent()`, `fmt_scientific()`, `fmt_time()`, `fmt()`, `text_transform()`

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
tab_1 <-
  exibble %>%
  dplyr::select(char) %>%
  gt() %>%
  fmt_passthrough(
    columns = vars(char),
    rows = !is.na(char),
    pattern = "{x}s"
  )
```

`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
- 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 = NULL,
  decimals = 2,
  drop_trailing_zeros = FALSE,
  drop_trailing_dec_mark = TRUE,
  scale_values = TRUE,
  use_seps = TRUE,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  incl_space = FALSE,
  placement = "right",
  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 `vars()`, 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()`, and `everything()`.

`rows` Optional rows to format. Not providing any value results in all rows in `columns` being formatted. Can either be a vector of row captions provided `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()`. We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 50).

`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.
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).
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_US" for English (United States) and "fr_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.

Details

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). A number of helper functions exist to make targeting more effective. 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.

Figures

Function ID

3-3

See Also

Other Format Data: data_color(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_markdown(), fmt_missing(), fmt_number(), fmt_passthrough(), fmt_scientific(), fmt_time(), fmt(), text_transform()

Examples

```
# Use `pizzaplace` to create a gt table;
# format the `frac_of_quota` column to
# display values as percentages
tab_1 <-
  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 = vars(frac_of_quota),
    decimals = 1
  )
```

---

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 = NULL,
  decimals = 2,
  drop_trailing_zeros = FALSE,
  scale_by = 1,
  pattern = "{x}",
  sep_mark = ",",
  dec_mark = ".",
  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 vars(), 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(), and everything().
rows	Optional rows to format. Not providing any value results in all rows in columns being formatted. Can either be a vector of row captions provided 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(). We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 50).
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. This value will be ignored if using any of the suffixing options (i.e., where suffixing is not set to FALSE).
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).
locale	An optional locale ID that can be used for formatting the value according the locale's rules. Examples include "en_US" for English (United States) and "fr_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.

Details

Targeting of values is done through columns and additionally by rows (if nothing is provided for rows then entire columns are selected). A number of helper functions exist to make targeting more effective. 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.

Figures

Function ID

3-2

See Also

Other Format Data: data_color(), fmt_currency(), fmt_datetime(), fmt_date(), fmt_markdown(), fmt_missing(), fmt_number(), fmt_passthrough(), fmt_percent(), fmt_time(), fmt(), text_transform()

Examples

```
# Use `exibble` to create a gt table;
# format the `num` column as partially
# numeric and partially in scientific
# notation
tab_1 <-
  exibble %>%
  gt() %>%
  fmt_number(
    columns = vars(num),
    rows = num > 500,
    decimals = 1,
    scale_by = 1/1000,
    pattern = "{x}K"
  ) %>%
  fmt_scientific(
    columns = vars(num),
    rows = num <= 500,
    decimals = 1
  )
```

---

fmt_time *Format values as times*

Description

Format input time values that are character-based and expressed according to the ISO 8601 time format (HH:MM:SS). Once the appropriate data cells are targeted with columns (and, optionally, rows), we can simply apply a preset time style to format the times. The following time styles are available for simpler formatting of ISO times (all using the input time of 14:35:00 in the example output times):

1. hms: 14:35:00
2. hm: 14:35
3. hms_p: 2:35:00 PM
4. hm_p: 2:35 PM
5. h_p: 2 PM

We can use the info_time_style() function for a useful reference on all of the possible inputs to time_style.

Usage

```
fmt_time(data, columns, rows = NULL, time_style = 2)
```

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 `vars()`, 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()`, and `everything()`.
`rows`	Optional rows to format. Not providing any value results in all rows in `columns` being formatted. Can either be a vector of row captions provided `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()`. We can also use expressions to filter down to the rows we need (e.g., [colname_1] > 100 & [colname_2] < 50).
`time_style`	The time style to use. Supply a number (from `1` to `5`) that corresponds to the preferred time style. Use `info_time_style()` to see the different numbered and named time presets.

Details

Targeting of values is done through `columns` and additionally by `rows` (if nothing is provided for `rows` then entire columns are selected). A number of helper functions exist to make targeting more effective. 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`.

Figures

Function ID

3-6

See Also

Other Format Data: `data_color()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_markdown()`, `fmt_missing()`, `fmt_number()`, `fmt_passthrough()`, `fmt_percent()`, `fmt_scientific()`, `fmt()`, `text_transform()`

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`)
tab_1 <-
  exibble %>%
  dplyr::select(date, time) %>%
  gt() %>%
  fmt_time(
    columns = vars(time),
    time_style = 3
  )

# 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)
tab_2 <-
  exibble %>%
  dplyr::select(date, time) %>%
  gt() %>%
  fmt_time(
    columns = vars(time),
    rows =
      time > "16:00",
    time_style = 3
  ) %>%
  fmt_time(
    columns = vars(time),
    rows =
      time <= "16:00",
    time_style = 4
  )
```

`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 object> evaluates to:
<img cid=<random CID> 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.

Figures

Function ID

8-3

See Also

Other Image Addition Functions: `local_image()`, `test_image()`, `web_image()`

Examples

```
library(ggplot2)

# Create a ggplot plot
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
tab_1 <-
  dplyr::tibble(
    text = "Here is a ggplot:",
    ggplot = NA
  ) %>%
  gt() %>%
  text_transform(
    locations = cells_body(vars(ggplot)),
    fn = function(x) {
      plot_object %>%
        ggplot_image(height = px(200))
    }
  )
```

`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`.

Figures

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_grand_summary(), cells_row_groups(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), gt_latex_dependencies(), html(), md(), pct(), px(), random_id()

Examples

```
if (interactive()) {

# 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)
tab_1 <-
  exibble %>%
  dplyr::select(char, time) %>%
  gt() %>%
  fmt_missing(columns = everything()) %>%
  tab_style(
    style = cell_text(
      font = c(
        google_font(name = "IBM Plex Mono"),
        default_fonts()
      )
    ),
    locations = cells_body(columns = vars(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)
tab_2 <-
  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"
    )
  )
}
```

`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 = TRUE,
  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.

Figures

Function ID

6-2

See Also

Other Add Rows: summary_rows()

Examples

```
# Use `sp500` to create a gt table with
# row groups; create grand summary rows
# (`min`, `max`, `avg`) for the table
tab_1 <-
  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 = vars(open, high, low, close),
    fns = list(
      min = ~min(.),
      max = ~max(.),
      avg = ~mean(.)),
    formatter = fmt_number,
    use_seps = FALSE
  )
```

---

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),
  rownames_to_stub = FALSE,
  auto_align = TRUE,
  id = 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.

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.

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.

Figures

Function ID

1-1

See Also

Other Create Table: gt_preview()

Examples

```
# Create a table object using the
# `exibble` dataset; use the `row`
# and `group` columns to add a stub
# and row groups
tab_1 <-
  exibble %>%
  gt(
    rowname_col = "row",
    groupname_col = "group"
  )

# The resulting object can be used
# in transformations (with `tab_*()`,
# `fmt_*()`, `cols_*()` functions)
tab_2 <-
  tab_1 %>%
  tab_header(
    title = "Table Title",
    subtitle = "Subtitle"
  ) %>%
  fmt_number(
    columns = vars(num),
    decimals = 2
  ) %>%
  cols_label(num = "number")
```

`gt-options` **gt** *package options*

Description

gt package options

Package options

gt uses the following `options()` to configure behavior:

- `gt.row_group.sep`: a separator between groups for the row group label.

`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

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.

Function ID

11-3

See Also

Other Datasets: countrypops, exibble, pizzaplace, sp500, sza

Examples

```
# Here is a glimpse at the data
# available in `gtcars`
dplyr::glimpse(gtcars)
```

---

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`). 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.

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 **webshot** 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 `webshot::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). There are several more options available so have a look at the **webshot** 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 `...`.

Function ID

13-1

See Also

Other Export Functions: `as_latex()`, `as_raw_html()`, `as_rtf()`, `extract_summary()`

Examples

```
if (interactive()) {

# Use `gtcars` to create a gt table; add
# a stubhead label 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")

# Get an HTML file with inlined CSS
# (which is necessary for including the
# table as part of an HTML email)
tab_1 %>%
  gtsave(
    "tab_1.html", inline_css = TRUE,
    path = tempdir()
  )

# By leaving out the `inline_css` option,
# we get a more conventional HTML file
# with embedded CSS styles
tab_1 %>%
  gtsave("tab_1.html", path = tempdir())

# Saving as PNG file results in a cropped
# image of an HTML table; the amount of
# whitespace can be set
tab_1 %>%
  gtsave(
    "tab_1.png", expand = 10,
    path = tempdir()
  )
```

```
# Any use of the `.tex`, `.ltx`, or `.rnw`
# will result in the output of a LaTeX
# document
tab_1 %>%
  gtsave("tab_1.tex", path = tempdir())
}
```

`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

7-23

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_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), html(), md(), pct(), px(), random_id()

---

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.

Function ID

12-2

See Also

Other Shiny functions: render_gt()

Examples

```
library(shiny)

# 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

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)
    )
}

if (interactive()) {
  shinyApp(ui, server)
}
```

---

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`.

Figures

Function ID

1-2

See Also

Other Create Table: `gt()`

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
tab_1 <-
  gtcars %>%
  dplyr::select(mfr, model, year) %>%
  gt_preview()
```

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.

Figures

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_grand_summary()`, `cells_row_groups()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `md()`, `pct()`, `px()`, `random_id()`

Examples

```
# Use `exibble` to create a gt table;
# when adding a title, use the `html()`
# helper to use html formatting
tab_1 <-
  exibble %>%
  dplyr::select(currency, char) %>%
  gt() %>%
  tab_header(
    title = html("<em>HTML</em>"))
```

`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`.

Figures

Function ID

10-3

See Also

Other Information Functions: info_date_style(), info_google_fonts(), info_locales(), info_paletteer(), info_time_style()

Examples

```
# Get a table of info on all of
# the currencies where the three-
# letter code begins with a "h"
tab_1 <- 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()`
tab_2 <- info_currencies(type = "symbol")
```

---

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 14 styles, with associated number codes, the format names, and example outputs using a fixed date (2000-02-29).

Usage

```
info_date_style()
```

Value

An object of class gt_tbl.

Figures

Function ID

10-1

See Also

Other Information Functions: info_currencies(), info_google_fonts(), info_locales(), info_paletteer(), info_time_style()

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)
tab_1 <- info_date_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`.

Function ID

10-6

See Also

Other Information Functions: `info_currencies()`, `info_date_style()`, `info_locales()`, `info_paletteer()`, `info_time_style()`

Examples

```
# Get a table of info on some of the
# recommended Google Fonts for tables
tab_1 <- info_google_fonts()
```

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`.

Function ID

10-4

See Also

Other Information Functions: `info_currencies()`, `info_date_style()`, `info_google_fonts()`, `info_paletteer()`, `info_time_style()`

Examples

```
# Get a table of info on all of
# the locales where the base
# locale ID begins with a "v"
tab_1 <- info_locales(begins_with = "v")
```

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`.

Figures

Function ID

10-5

See Also

Other Information Functions: `info_currencies()`, `info_date_style()`, `info_google_fonts()`, `info_locales()`, `info_time_style()`

Examples

```
# Get a table of info on just the
# `ggthemes` color palette (easily
# accessible from the paletteer pkg)
tab_1 <-
  info_paletteer(
    color_pkgs = "ggthemes")
```

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 five styles, with associated number codes, the format names, and example outputs using a fixed time (14:35).

Usage

```
info_time_style()
```

Value

An object of class gt_tbl.

Figures

Function ID

10-2

See Also

Other Information Functions: info_currencies(), info_date_style(), info_google_fonts(), info_locales(), info_paletteer()

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)
tab_1 <- info_time_style()
```

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 cid=<random CID> 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.

Figures

Function ID

8-2

See Also

Other Image Addition Functions: `ggplot_image()`, `test_image()`, `web_image()`

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
tab_1 <-
  dplyr::tibble(
    pixels = px(seq(10, 35, 5)),
    image = seq(10, 35, 5)
  ) %>%
  gt() %>%
  text_transform(
    locations = cells_body(vars(image)),
    fn = function(x) {
      local_image(
        filename = test_image(type = "png"),
        height = as.numeric(x)
      )
    }
  )
```

---

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.

Figures

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_grand_summary()`, `cells_row_groups()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `pct()`, `px()`, `random_id()`

Examples

```
# Use `exibble` to create a gt table;
# when adding a title, use the `md()`
# helper to use Markdown formatting
tab_1 <-
  exibble %>%
  dplyr::select(currency, char) %>%
  gt() %>%
  tab_header(
    title = md("Using *Markdown*"))
```

`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.

Figures

Function ID

9-3

See Also

Other Table Option Functions: opt_all_caps(), opt_css(), opt_footnote_marks(), opt_row_striping(), opt_table_font(), opt_table_lines(), opt_table_outline()

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
tab_1 <-
  exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = vars(num, currency),
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = vars(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")
```

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).

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.

Details

This function serves as a convenient shortcut for <gt_tbl> %>% tab_options(<location>.text_transform = "uppercase", <location>.font.size = pct(80), <location>.font.weight = "bolder") (for all `locations` selected).

Value

An object of class `gt_tbl`.

Figures

Function ID

9-4

See Also

Other Table Option Functions: opt_align_table_header(), opt_css(), opt_footnote_marks(), opt_row_striping(), opt_table_font(), opt_table_lines(), opt_table_outline()

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()`
tab_1 <-
  exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = vars(num, currency),
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = vars(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()
```

---

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.

Figures

Function ID

9-8

See Also

Other Table Option Functions: opt_align_table_header(), opt_all_caps(), opt_footnote_marks(), opt_row_striping(), opt_table_font(), opt_table_lines(), opt_table_outline()

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")
tab_1 <-
  exibble %>%
  dplyr::select(num, currency) %>%
  gt(id = "one") %>%
  fmt_currency(
    columns = vars(currency),
    currency = "HKD"
  ) %>%
  fmt_scientific(
    columns = vars(num)
  ) %>%
  opt_css(
    css = "
    #one .gt_table {
      background-color: skyblue;
```

```
    }
    #one .gt_row {
      padding: 20px 30px;
    }
    #one .gt_col_heading {
      text-align: center !important;
    }
    "
  )
```

`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.

Figures

Function ID

9-1

See Also

Other Table Option Functions: opt_align_table_header(), opt_all_caps(), opt_css(), opt_row_striping(), opt_table_font(), opt_table_lines(), opt_table_outline()

Examples

```
# Use `sza` to create a gt table,
# adding three footnotes; call
# `opt_footnote_marks()` to specify
# which footnote marks to use
tab_1 <-
  sza %>%
  dplyr::group_by(latitude, tst) %>%
  dplyr::summarize(
    SZA.Max = max(sza),
    SZA.Min = min(sza, na.rm = TRUE)
  ) %>%
  dplyr::ungroup() %>%
  dplyr::filter(latitude == 30, !is.infinite(SZA.Min)) %>%
  dplyr::select(-latitude) %>%
  gt(rowname_col = "tst") %>%
  tab_spanner_delim(delim = ".") %>%
  fmt_missing(
    columns = everything(),
    missing_text = "90+"
  ) %>%
  tab_stubhead("TST") %>%
  tab_footnote(
    footnote = "True solar time.",
    locations = cells_stubhead()
  ) %>%
  tab_footnote(
    footnote = "Solar zenith angle.",
    locations = cells_column_spanners(spanners = "SZA")
  ) %>%
```

```
  tab_footnote(
    footnote = "The Lowest SZA.",
    locations = cells_stub(rows = "1200")
  ) %>%
  opt_footnote_marks(marks = "standard")
```

`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`.

Figures

Function ID

9-2

See Also

Other Table Option Functions: `opt_align_table_header()`, `opt_all_caps()`, `opt_css()`, `opt_footnote_marks()`, `opt_table_font()`, `opt_table_lines()`, `opt_table_outline()`

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
tab_1 <-
  exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = vars(num, currency),
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = vars(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()
```

`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.

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.

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.

Value

An object of class gt_tbl.

Figures

Function ID

9-7

See Also

Other Table Option Functions: opt_align_table_header(), opt_all_caps(), opt_css(), opt_footnote_marks(), opt_row_striping(), opt_table_lines(), opt_table_outline()

Examples

```
if (interactive()) {

# 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)
tab_1 <-
  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)
# and Windows 10
tab_2 <-
  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()

}
```

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`.

Figures

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_table_font()`, `opt_table_outline()`

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
tab_1 <-
  exibble %>%
  gt(rowname_col = "row", groupname_col = "group") %>%
  summary_rows(
    groups = "grp_a",
    columns = vars(num, currency),
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = vars(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()
```

`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`.

Figures

Function ID

9-6

See Also

Other Table Option Functions: `opt_align_table_header()`, `opt_all_caps()`, `opt_css()`, `opt_footnote_marks()`, `opt_row_striping()`, `opt_table_font()`, `opt_table_lines()`

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 = vars(num, currency),
    fns = list(
      min = ~min(., na.rm = TRUE),
      max = ~max(., na.rm = TRUE)
    )) %>%
  grand_summary_rows(
    columns = vars(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()

# Remove the table outline with the
# `style = "none"` option
tab_2 <-
  tab_1 %>%
  opt_table_outline(style = "none")
```

`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.

Figures

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_grand_summary(), cells_row_groups(), cells_stubhead(), cells_stub(), cells_summary(), cells_title(), currency(), default_fonts(), escape_latex(), google_font(), gt_latex_dependencies(), html(), md(), px(), random_id()

Examples

```
# Use `exibble` to create a gt table;
# use the `pct()` helper to define the
# font size for the column labels
tab_1 <-
  exibble %>%
  gt() %>%
  tab_style(
    style = cell_text(size = pct(75)),
    locations = cells_column_labels(columns = TRUE)
  )
```

`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)

Function ID

11-5

See Also

Other Datasets: `countrypops`, `exibble`, `gtcars`, `sp500`, `sza`

Examples

```
# Here is a glimpse at the data
# available in `pizzaplace`
dplyr::glimpse(pizzaplace)
```

---

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.

Figures

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_grand_summary()`, `cells_row_groups()`, `cells_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `random_id()`

Examples

```
# Use `exibble` to create a gt table;
# use the `px()` helper to define the
# font size for the column labels
tab_1 <-
  exibble %>%
  gt() %>%
  tab_style(
    style = cell_text(size = px(20)),
    locations = cells_column_labels(columns = TRUE)
  )
```

`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

7-21

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_stubhead()`, `cells_stub()`, `cells_summary()`, `cells_title()`, `currency()`, `default_fonts()`, `escape_latex()`, `google_font()`, `gt_latex_dependencies()`, `html()`, `md()`, `pct()`, `px()`

`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.

Function ID

12-1

See Also

Other Shiny functions: `gt_output()`

Examples

```
library(shiny)

# 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

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)
    )
}

if (interactive()) {
  shinyApp(ui, server)
}
```

`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 can either take a vector of row group names or a numeric vector of row group indices; whichever is provided, the row groups will adhere to this revised ordering. It isn't necessary to provide all row group names 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 vector of row group names, or, a numeric vector of indices corresponding to the new ordering. Either vector must correspond to assigned group names or the index positions. Also, either type of vector is not required to have all of the row group names or available index positions within it; any omitted values will be added to the end while preserving the original ordering.

Value

An object of class `gt_tbl`.

Figures

Function ID

5-1

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`
tab_1 <-
  exibble %>%
  dplyr::select(char, currency, row, group) %>%
    gt(
      rowname_col = "row",
      groupname_col = "group"
    ) %>%
    row_group_order(
      groups = c("grp_b", "grp_a")
    )
```

---

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

Function ID

11-4

See Also

Other Datasets: countrypops, exibble, gtcars, pizzaplace, sza

Examples

```
# Here is a glimpse at the data
# available in `sp500`
dplyr::glimpse(sp500)
```

---

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 = TRUE,
  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`.

Figures

Function ID

6-1

See Also

Other Add Rows: `grand_summary_rows()`

Examples

```
# Use `sp500` to create a gt table with
# row groups; create summary rows (`min`,
# `max`, `avg`) by row group, where each
# each row group is a week number
tab_1 <-
  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 = vars(open, high, low, close),
  fns = list(
    min = ~min(.),
    max = ~max(.),
    avg = ~mean(.)),
  formatter = fmt_number,
  use_seps = FALSE
)
```

`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.

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`

Examples

```
# Here is a glimpse at the data
# available in `sza`
dplyr::glimpse(sza)
```

`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)
```

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()`, and `cells_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.).

Details

The formatting of the footnotes can be controlled through the use of various parameters in the `tab_options()` function:

- `footnotes.sep`: allows for a choice of the separator between consecutive footnotes in the table footer. By default, this is set to a linebreak.
- `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`.

Figures

Function ID

2-6

See Also

Other Create or Modify Parts: `tab_header()`, `tab_options()`, `tab_row_group()`, `tab_source_note()`, `tab_spanner_delim()`, `tab_spanner()`, `tab_stubhead()`, `tab_style()`

Examples

```
# Use `sza` to create a gt table; color
# the `sza` column using the `data_color()`
# function, then, add a footnote to the
# `sza` column label explaining what the
# color scale signifies
tab_1 <-
  sza %>%
  dplyr::filter(
    latitude == 20 &
      month == "jan" &
      !is.na(sza)
  ) %>%
  dplyr::select(-latitude, -month) %>%
  gt() %>%
  data_color(
    columns = vars(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 = vars(sza))
  )
```

---

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)
```

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.

Value

An object of class gt_tbl.

Figures

Function ID

2-1

See Also

Other Create or Modify Parts: tab_footnote(), tab_options(), tab_row_group(), tab_source_note(), tab_spanner_delim(), tab_spanner(), tab_stubhead(), tab_style()

Examples

```
# Use `gtcars` to create a gt table;
# add a header part to contain a title
# and subtitle
tab_1 <-
  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_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.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.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.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.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,
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,
data_row.padding = NULL,
summary_row.background.color = NULL,
summary_row.text_transform = NULL,
summary_row.padding = 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.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.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.sep = NULL,
  footnotes.marks = NULL,
  source_notes.background.color = NULL,
  source_notes.font.size = NULL,
  source_notes.padding = 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,
  row.striping.background_color = NULL,
  row.striping.include_stub = NULL,
  row.striping.include_table_body = NULL
)
```

Arguments

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

`container.width`, `container.height`
: The width and height of the table's 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.

`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.backgroun
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.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 dthe `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.

`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.

data_row.padding, row_group.padding, summary_row.padding, grand_summary_row.padding, footnotes.paddin
: The amount of vertical padding to incorporate in the data_row, row_group, summary_row, grand_summary_row, footnotes, and source_notes locations.

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.sep
: The separating characters between adjacent footnotes in the footnotes section. The default value produces a linebreak.

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.

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.

Value

An object of class gt_tbl.

Figures

Function ID

2-9

See Also

Other Create or Modify Parts: tab_footnote(), tab_header(), tab_row_group(), tab_source_note(), tab_spanner_delim(), tab_spanner(), tab_stubhead(), tab_style()

Examples

```
# Use `exibble` to create a gt table with
# all the main parts added; we can use this
# going forward to demo some `tab_options()`
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 = vars(num)) %>%
  fmt_currency(columns = vars(currency)) %>%
  tab_footnote(
    footnote = "Using commas for separators.",
    locations = cells_body(
      columns = vars(num),
      rows = num > 1000)
  ) %>%
  tab_footnote(
    footnote = "Using commas for separators.",
    locations = cells_body(
      columns = vars(currency),
      rows = currency > 1000)
  ) %>%
  tab_footnote(
    footnote = "Alphabetical fruit.",
    locations = cells_column_labels(
      columns = vars(char))
  )

# Modify the table width to 100% (which
# spans the entire content width area)
```

```
tab_2 <-
  tab_1 %>%
  tab_options(
    table.width = pct(100)
  )

# Modify the table's background color
# to be "lightcyan"
tab_3 <-
  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_4 <-
  tab_1 %>%
  tab_options(
    footnotes.sep = " ",
    footnotes.marks = letters
  )

# Change the padding of data rows to 5px
tab_5 <-
  tab_1 %>%
  tab_options(
    data_row.padding = px(5)
  )

# Reduce the size of the title and the
# subtitle text
tab_6 <-
  tab_1 %>%
  tab_options(
    heading.title.font.size = "small",
    heading.subtitle.font.size = "small"
  )
```

`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()`.

Usage

```
tab_row_group(data, group = NULL, rows = NULL, others = NULL)
```

Arguments

`data`	A table object that is created using the `gt()` function.
`group`	The name of the row group. This text will also serve as 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()`.
`others`	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()`. A separate call to `tab_row_group()` with only a value to `others` is possible and makes explicit that the call is meant to provide a default row group label. If this is not set 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.

Value

An object of class `gt_tbl`.

Figures

Function ID

2-4

See Also

Other Create or Modify Parts: `tab_footnote()`, `tab_header()`, `tab_options()`, `tab_source_note()`, `tab_spanner_delim()`, `tab_spanner()`, `tab_stubhead()`, `tab_style()`

Examples

```
# Use `gtcars` to create a gt table and
# add two row groups with the labels:
# `numbered` and `NA` (a group without
# a title, or, the rest)
tab_1 <-
  gtcars %>%
  dplyr::select(model, year, hp, trq) %>%
  dplyr::slice(1:8) %>%
  gt(rowname_col = "model") %>%
  tab_row_group(
    group = "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 being `hp` lesser or
# greater than `600`
tab_2 <-
  gtcars %>%
  dplyr::select(model, year, hp, trq) %>%
  dplyr::slice(1:8) %>%
  gt(rowname_col = "model") %>%
  tab_row_group(
    group = "powerful",
    rows = hp <= 600
  ) %>%
  tab_row_group(
    group = "super powerful",
    rows = hp > 600
  )
```

`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`.

Figures

Function ID

2-7

See Also

Other Create or Modify Parts: tab_footnote(), tab_header(), tab_options(), tab_row_group(), tab_spanner_delim(), tab_spanner(), tab_stubhead(), tab_style()

Examples

```
# Use `gtcars` to create a gt table;
# add a source note to the table
# footer that cites the data source
tab_1 <-
  gtcars %>%
  dplyr::select(mfr, model, msrp) %>%
  dplyr::slice(1:5) %>%
  gt() %>%
  tab_source_note(
    source_note = "From edmunds.com"
  )
```

---

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, gather = TRUE)
```

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.
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.

Value

An object of class gt_tbl.

Figures

Function ID

2-2

See Also

Other Create or Modify Parts: tab_footnote(), tab_header(), tab_options(), tab_row_group(), tab_source_note(), tab_spanner_delim(), tab_stubhead(), tab_style()

Examples

```
# Use `gtcars` to create a gt table;
# Group several columns related to car
# performance under a spanner column
# with the label `performance`
tab_1 <-
  gtcars %>%
  dplyr::select(
    -mfr, -trim, bdy_style, drivetrain,
    -drivetrain, -trsmn, -ctry_origin
  ) %>%
  dplyr::slice(1:8) %>%
  gt(rowname_col = "model") %>%
  tab_spanner(
    label = "performance",
    columns = vars(
      hp, hp_rpm, trq, trq_rpm,
      mpg_c, mpg_h)
  )
```

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 = NULL, gather = TRUE)
```

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 group name 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.
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.

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.

Figures

Function ID

2-3

See Also

Other Create or Modify Parts: tab_footnote(), tab_header(), tab_options(), tab_row_group(), tab_source_note(), tab_spanner(), tab_stubhead(), tab_style()

Examples

```
# Use `iris` to create a gt table; split
# any columns that are dot-separated
# between column spanner labels (first
# part) and column labels (second part)
tab_1 <-
  iris %>%
  dplyr::group_by(Species) %>%
```

```
dplyr::slice(1:4) %>%
gt() %>%
tab_spanner_delim(delim = ".")
```

---

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.

Figures

Function ID

2-5

See Also

Other Create or Modify Parts: tab_footnote(), tab_header(), tab_options(), tab_row_group(), tab_source_note(), tab_spanner_delim(), tab_spanner(), tab_style()

Examples

```
# Use `gtcars` to create a gt table; add
# a stubhead label 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")
```

`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()`, and `cells_grand_summary()`. 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`.

Figures

Function ID

2-8

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 Create or Modify Parts: `tab_footnote()`, `tab_header()`, `tab_options()`, `tab_row_group()`, `tab_source_note()`, `tab_spanner_delim()`, `tab_spanner()`, `tab_stubhead()`

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()`)
tab_1 <-
  exibble %>%
  dplyr::select(num, currency) %>%
  gt() %>%
  fmt_number(
    columns = vars(num, currency),
    decimals = 1
  ) %>%
  tab_style(
    style = list(
      cell_fill(color = "lightcyan"),
      cell_text(weight = "bold")
      ),
    locations = cells_body(
      columns = vars(num),
      rows = num >= 5000)
  ) %>%
```

```
  tab_style(
    style = list(
      cell_fill(color = "#F9E3D6"),
      cell_text(style = "italic")
      ),
    locations = cells_body(
      columns = vars(currency),
      rows = currency < 100)
  )

# Use `sp500` to create a gt table;
# color entire rows of cells based
# on values in a particular column
tab_2 <-
  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
# `fmt_missing()` function and then
# add styling to the `char` column
# with `cell_fill()` and with a
# CSS style declaration
tab_3 <-
  exibble %>%
  dplyr::select(char, fctr) %>%
  gt() %>%
  fmt_missing(everything()) %>%
  tab_style(
    style = list(
      cell_fill(color = "lightcyan"),
      "font-variant: small-caps;"
    ),
    locations = cells_body(columns = vars(char))
  )
```

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()`, and `cells_column_labels()` 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`.

Figures

Function ID

3-12

See Also

Other Format Data: `data_color()`, `fmt_currency()`, `fmt_datetime()`, `fmt_date()`, `fmt_markdown()`, `fmt_missing()`, `fmt_number()`, `fmt_passthrough()`, `fmt_percent()`, `fmt_scientific()`, `fmt_time()`, `fmt()`

Examples

```
# Use `exibble` to create a gt table;
# transform the formatted text in the
# `num` and `currency` columns using
# a function within `text_transform()`,
# where `x` is a formatted vector of
# column values
tab_1 <-
  exibble %>%
  dplyr::select(num, char, currency) %>%
  dplyr::slice(1:4) %>%
  gt() %>%
  fmt_number(columns = vars(num)) %>%
  fmt_currency(columns = vars(currency)) %>%
  text_transform(
    locations = cells_body(
      columns = vars(num)),
    fn = function(x) {
      paste0(
        x, " (",
        dplyr::case_when(
          x > 20   ~ "large",
          x <= 20  ~ "small"),
```

```
        ")")
    }
  )
```

`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://some.web.site/image.png")` evaluates to:
<img src=\"http://some.web.site/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.

Figures

Function ID

8-1

See Also

Other Image Addition Functions: `ggplot_image()`, `local_image()`, `test_image()`

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
tab_1 <-
  dplyr::tibble(
    pixels = px(seq(10, 35, 5)),
    image = seq(10, 35, 5)
  ) %>%
  gt() %>%
  text_transform(
    locations = cells_body(vars(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
tab_2 <-
  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)
    )
  )
```

Index