Package 'rtables'

December 8, 2023

Title Reporting Tables

Version 0.6.6

Date 2023-12-07

Description Reporting tables often have structure that goes beyond simple rectangular data. The 'rtables' package provides a framework for declaring complex multi-level tabulations and then applying them to data. This framework models both tabulation and the resulting tables as hierarchical, tree-like objects which support sibling sub-tables, arbitrary splitting or grouping of data in row and column dimensions, cells containing multiple values, and the concept of contextual summary computations. A convenient pipe-able interface is provided for declaring table layouts and the corresponding computations, and then applying them to data.

License Apache License 2.0 | file LICENSE

URL https://github.com/insightsengineering/rtables, https://insightsengineering.github.io/rtables/

BugReports https://github.com/insightsengineering/rtables/issues

Depends formatters (>= 0.5.5), magrittr (>= 1.5), methods, R (>= 2.10)

Imports checkmate (>= 2.1.0), htmltools (>= 0.5.4), stats, stringi (>= 1.6)

Suggests broom (>= 0.7.10), car (>= 3.0-13), dplyr (>= 1.0.5), flextable (>= 0.8.4), knitr (>= 1.42), officer (>= 0.5.0), r2rtf (>= 0.3.2), survival (>= 3.3-1), testthat (>= 3.0.4), tibble (>= 3.2.1), tidyr (>= 1.1.3), xml2 (>= 1.1.0)

VignetteBuilder knitr

Config/Needs/verdepcheck insightsengineering/formatters, tidyverse/magrittr, rstudio/htmltools, tidymodels/broom, cran/car, mllg/checkmate, tidyverse/dplyr, davidgohel/flextable, yihui/knitr, davidgohel/officer, Merck/r2rtf, r-lib/testthat, tidyverse/tibble, tidyverse/tidyr, r-lib/xml2

Encoding UTF-8

Language en-US

RoxygenNote 7.2.3

Collate '00tabletrees.R' 'Viewer.R' 'argument_conventions.R' 'as_html.R' 'utils.R' 'colby_constructors.R' 'compare_rtables.R' 'deprecated.R' 'format_rcell.R' 'indent.R' 'make_subset_expr.R' 'simple_analysis.R' 'split_funs.R' 'make_split_fun.R' 'summary.R' 'tree_accessors.R' 'tt_afun_utils.R' 'tt_compare_tables.R' 'tt_compatibility.R' 'tt_dotabulation.R' 'tt_paginate.R' 'tt_pos_and_access.R' 'tt_showmethods.R' 'tt_sort.R' 'tt_test_afuns.R' 'tt_toString.R' 'tt_export.R' 'index_footnotes.R' 'tt_from_df.R' 'validate_table_struct.R' 'zzz_constants.R'

NeedsCompilation no

Author Gabriel Becker [aut] (Original creator of the package), Adrian Waddell [aut], Daniel Sabanés Bové [ctb], Maximilian Mordig [ctb], Davide Garolini [ctb], Emily de la Rua [ctb], Abinaya Yogasekaram [ctb], Joe Zhu [ctb, cre], F. Hoffmann-La Roche AG [cph, fnd]

Maintainer Joe Zhu <joe.zhu@roche.com>

Repository CRAN

Date/Publication 2023-12-08 10:40:02 UTC

R topics documented:

`additional_fun_params` *Additional parameters within analysis and content functions (*`afun/cfun`*)*

Description

It is possible to add specific parameters to `afun` and `cfun`, in analyze and summarize_row_groups respectively. These parameters grant access to relevant information like the row split structure (see spl_context) and the predefined baseline (`.ref_group`).

Details

We list and describe here all the parameters that can be added to a custom analysis function:

.N_col column-wise N (column count) for the full column being tabulated within

.N_total overall N (all observation count, defined as sum of column counts) for the tabulation

.N_row row-wise N (row group count) for the group of observations being analyzed (i.e. with no column-based subsetting)

.df_row data.frame for observations in the row group being analyzed (i.e. with no column-based subsetting)

.var variable that is analyzed

.ref_group data.frame or vector of subset corresponding to the `ref_group` column including subsetting defined by row-splitting. Optional and only required/meaningful if a `ref_group` column has been defined

.ref_full data.frame or vector of subset corresponding to the `ref_group` column without subsetting defined by row-splitting. Optional and only required/meaningful if a `ref_group` column has been defined

.in_ref_col boolean indicates if calculation is done for cells within the reference column

.spl_context data.frame, each row gives information about a previous/'ancestor' split state. See `spl_context`

.alt_df_row data.frame, i.e. the `alt_count_df` after row splitting. It can be used with `.all_col_exprs` and `.spl_context` information to retrieve current faceting, but for `alt_count_df`. It can be an empty table if all the entries are filtered out.

.alt_df data.frame, `.alt_df_row` but filtered by columns expression. This data present the same faceting of main data `df`. This also filters `NA`s out if related parameters are set to (e.g. `inclNAs` in analyze). Similarly to `.alt_df_row`, it can be an empty table if all the entries are filtered out.

.all_col_exprs list of expressions. Each of them represents a different column splitting.

.all_col_counts vector of integers. Each of them represents the global count for each column. It differs if `alt_counts_df` is used (see build_table).

Note

If any of these formals is specified incorrectly or not present in the tabulation machinery, it will be as if missing. For example `.ref_group` will be missing if no baseline is previously defined during data splitting (via `ref_group` parameters in, e.g., split_rows_by). Similarly, if no `alt_counts_df` is provided into build_table, `.alt_df_row` and `.alt_df` will not be present.

---

add_colcounts *Add the column population counts to the header*

Description

Add the data derived column counts.

Usage

```
add_colcounts(lyt, format = "(N=xx)")
```

Arguments

`lyt`	layout object pre-data used for tabulation
`format`	`FormatSpec`. Format associated with this split. Formats can be declared via strings (`"xx.x"`) or function. In cases such as `analyze` calls, they can character vectors or lists of functions.

Details

It is often the case that the the column counts derived from the input data to `build_table` is not representative of the population counts. For example, if events are counted in the table and the header should display the number of subjects and not the total number of events. In that case use the `col_count` argument in `build_table` to control the counts displayed in the table header.

Value

A `PreDataTableLayouts` object suitable for passing to further layouting functions, and to `build_table`.

Author(s)

Gabriel Becker

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  add_colcounts() %>%
  split_rows_by("RACE", split_fun = drop_split_levels) %>%
  analyze("AGE", afun = function(x) list(min = min(x), max = max(x)))
lyt
```

```
tbl <- build_table(lyt, DM)
tbl
```

`add_combo_facet` *Add a combination facet in postprocessing*

Description

Add a combination facet during postprocessing stage in a custom split fun.

Usage

```
add_combo_facet(name, label = name, levels, extra = list())

add_overall_facet(name, label, extra = list())
```

Arguments

`name`	character(1). Name for the resulting facet (for use in pathing, etc).
`label`	character(1). Label for the resulting facet.
`levels`	character. Vector of levels to combine within the resulting facet.
`extra`	list. Extra arguments to be passed to analysis functions applied within the resulting facet.

Details

For `add_combo_facet`, the data associated with the resulting facet will be the data associated with the facets for each level in `levels`, `rbound` together. In particular, this means that if those levels are overlapping, data that appears in both will be duplicated.

Value

a function which can be used within the `post` argument in `make_split_fun`.

See Also

`make_split_fun`

Other make_custom_split: `drop_facet_levels()`, `make_split_fun()`, `make_split_result()`, `trim_levels_in_facets()`

Examples

```
mysplfun <- make_split_fun(post = list(
  add_combo_facet("A_B",
    label = "Arms A+B",
    levels = c("A: Drug X", "B: Placebo")
  ),
  add_overall_facet("ALL", label = "All Arms")
))

lyt <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by("ARM", split_fun = mysplfun) %>%
  analyze("AGE")

tbl <- build_table(lyt, DM)
```

---

add_existing_table *Add an already calculated table to the layout*

Description

Add an already calculated table to the layout

Usage

```
add_existing_table(lyt, tt, indent_mod = 0)
```

Arguments

lyt	layout object pre-data used for tabulation
tt	TableTree (or related class). A TableTree object representing a populated table.
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.

Value

A PreDataTableLayouts object suitable for passing to further layouting functions, and to build_table.

Author(s)

Gabriel Becker

Examples

```
lyt1 <- basic_table() %>%
  split_cols_by("ARM") %>%
  analyze("AGE", afun = mean, format = "xx.xx")

tbl1 <- build_table(lyt1, DM)
tbl1

lyt2 <- basic_table() %>%
  split_cols_by("ARM") %>%
  analyze("AGE", afun = sd, format = "xx.xx") %>%
  add_existing_table(tbl1)

tbl2 <- build_table(lyt2, DM)
tbl2

table_structure(tbl2)
row_paths_summary(tbl2)
```

---

add_overall_col *Add Overall Column*

Description

This function will *only* add an overall column at the *top* level of splitting, NOT within existing column splits. See `add_overall_level` for the recommended way to add overall columns more generally within existing splits.

Usage

```
add_overall_col(lyt, label)
```

Arguments

`lyt`	layout object pre-data used for tabulation
`label`	character(1). A label (not to be confused with the name) for the object/structure.

Value

A `PreDataTableLayouts` object suitable for passing to further layouting functions, and to `build_table`.

See Also

`add_overall_level()`

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  add_overall_col("All Patients") %>%
  analyze("AGE")
lyt

tbl <- build_table(lyt, DM)
tbl
```

`add_overall_level` *Add an virtual 'overall' level to split*

Description

Add an virtual 'overall' level to split

Usage

```
add_overall_level(
  valname = "Overall",
  label = valname,
  extra_args = list(),
  first = TRUE,
  trim = FALSE
)
```

Arguments

`valname`	character(1). 'Value' to be assigned to the implicit all-observations split level. Defaults to `"Overall"`
`label`	character(1). A label (not to be confused with the name) for the object/structure.
`extra_args`	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.
`first`	logical(1). Should the implicit level appear first (`TRUE`) or last `FALSE`. Defaults to `TRUE`.
`trim`	logical(1). Should splits corresponding with 0 observations be kept when tabulating.

Value

a closure suitable for use as a splitting function (`splfun`) when creating a table layout

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM", split_fun = add_overall_level("All Patients",
    first = FALSE
  )) %>%
  analyze("AGE")

tbl <- build_table(lyt, DM)
tbl

lyt2 <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("RACE",
    split_fun = add_overall_level("All Ethnicities")
  ) %>%
  summarize_row_groups(label_fstr = "%s (n)") %>%
  analyze("AGE")
lyt2

tbl2 <- build_table(lyt2, DM)
tbl2
```

---

all_zero_or_na *Trimming and Pruning Criteria*

Description

Criteria functions (and constructors thereof) for trimming and pruning tables.

Usage

```
all_zero_or_na(tr)

all_zero(tr)

content_all_zeros_nas(tt, criteria = all_zero_or_na)

prune_empty_level(tt)

prune_zeros_only(tt)

low_obs_pruner(min, type = c("sum", "mean"))
```

Arguments

tr	TableRow (or related class). A TableRow object representing a single row within a populated table.

tt	TableTree (or related class). A TableTree object representing a populated table.
criteria	function. Function which takes a TableRow object and returns TRUE if that row should be removed. Defaults to all_zero_or_na
min	numeric(1). (low_obs_pruner only). Minimum aggregate count value. Subtables whose combined/average count are below this threshold will be pruned
type	character(1). How count values should be aggregated. Must be "sum" (the default) or "mean"

Details

all_zero_or_na returns TRUE (and thus indicates trimming/pruning) for any *non*-LabelRow TableRow which contain only any mix of NA (including NaN), 0, Inf and -Inf values.

all_zero returns TRUE for any non-Label row which contains only (non-missing) zero values.

content_all_zeros_nas Prunes a subtable if a) it has a content table with exactly one row in it, and b) all_zero_or_na returns TRUE for that single content row. In practice, when the default summary/content function is used, this represents pruning any subtable which corresponds to an empty set of the input data (e.g., because a factor variable was used in split_rows_by but not all levels were present in the data).

prune_empty_level combines all_zero_or_na behavior for TableRow objects, content_all_zeros_nas on content_table(tt) for TableTree objects, and an additional check that returns TRUE if the tt has no children.

prune_zeros_only behaves as prune_empty_level does, except that like all_zero it prunes only in the case of all non-missing zero values.

low_obs_pruner is a *constructor function* which, when called, returns a pruning criteria function which will prune on content rows by comparing sum or mean (dictated by type) of the count portions of the cell values (defined as the first value per cell regardless of how many values per cell there are) against min.

Value

A logical value indicating whether tr should be included (TRUE) or pruned (FALSE) during pruning.

See Also

prune_table(), trim_rows()

Examples

```
adsl <- ex_adsl
levels(adsl$SEX) <- c(levels(ex_adsl$SEX), "OTHER")
adsl$AGE[adsl$SEX == "UNDIFFERENTIATED"] <- 0
adsl$BMRKR1 <- 0

tbl_to_prune <- basic_table() %>%
  analyze("BMRKR1") %>%
  split_cols_by("ARM") %>%
  split_rows_by("SEX") %>%
```

```
  summarize_row_groups() %>%
  split_rows_by("STRATA1") %>%
  summarize_row_groups() %>%
  analyze("AGE") %>%
  build_table(adsl)

tbl_to_prune %>% prune_table(all_zero_or_na)

tbl_to_prune %>% prune_table(all_zero)

tbl_to_prune %>% prune_table(content_all_zeros_nas)

tbl_to_prune %>% prune_table(prune_empty_level)

tbl_to_prune %>% prune_table(prune_zeros_only)

min_prune <- low_obs_pruner(70, "sum")
tbl_to_prune %>% prune_table(min_prune)
```

`analyze` *Generate Rows Analyzing Variables Across Columns*

Description

Adding *analyzed variables* to our table layout defines the primary tabulation to be performed. We do this by adding calls to `analyze` and/or `analyze_colvars` into our layout pipeline. As with adding further splitting, the tabulation will occur at the current/next level of nesting by default.

Usage

```
analyze(
  lyt,
  vars,
  afun = simple_analysis,
  var_labels = vars,
  table_names = vars,
  format = NULL,
  na_str = NA_character_,
  nested = TRUE,
  inclNAs = FALSE,
  extra_args = list(),
  show_labels = c("default", "visible", "hidden"),
  indent_mod = 0L,
  section_div = NA_character_
)
```

Arguments

lyt	layout object pre-data used for tabulation
vars	character vector. Multiple variable names.
afun	function. Analysis function, must take x or df as its first parameter. Can optionally take other parameters which will be populated by the tabulation framework. See Details in analyze.
var_labels	character. Variable labels for 1 or more variables
table_names	character. Names for the tables representing each atomic analysis. Defaults to var.
format	FormatSpec. Format associated with this split. Formats can be declared via strings ("xx.x") or function. In cases such as analyze calls, they can character vectors or lists of functions.
na_str	character(1). String that should be displayed when the value of x is missing. Defaults to "NA".
nested	boolean. Should this layout instruction be applied within the existing layout structure *if possible* (TRUE, the default) or as a new top-level element ('FALSE). Ignored if it would nest a split underneath analyses, which is not allowed.
inclNAs	boolean. Should observations with NA in the var variable(s) be included when performing this analysis. Defaults to FALSE
extra_args	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.
show_labels	character(1). Should the variable labels for corresponding to the variable(s) in vars be visible in the resulting table.
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
section_div	character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.

Details

When non-NULL format is used to specify formats for all generated rows, and can be a character vector, a function, or a list of functions. It will be repped out to the number of rows once this is known during the tabulation process, but will be overridden by formats specified within rcell calls in afun.

The analysis function (afun) should take as its first parameter either x or df. Which of these the function accepts changes the behavior when tabulation is performed.

- If afun's first parameter is x, it will receive the corresponding subset *vector* of data from the relevant column (from var here) of the raw data being used to build the table.
- If afun's first parameter is df, it will receive the corresponding subset *data.frame* (i.e. all columns) of the raw data being tabulated

In addition to differentiation on the first argument, the analysis function can optionally accept a number of other parameters which, *if and only if* present in the formals will be passed to the function by the tabulation machinery. These are listed and described in additional_fun_params.

Value

A PreDataTableLayouts object suitable for passing to further layouting functions, and to build_table.

Note

None of the arguments described in the Details section can be overridden via extra_args or when calling make_afun. .N_col and .N_total can be overridden via the col_counts argument to build_table. Alternative values for the others must be calculated within afun based on a combination of extra arguments and the unmodified values provided by the tabulation framework.

Author(s)

Gabriel Becker

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  analyze("AGE", afun = list_wrap_x(summary), format = "xx.xx")
lyt

tbl <- build_table(lyt, DM)
tbl

lyt2 <- basic_table() %>%
  split_cols_by("Species") %>%
  analyze(head(names(iris), -1), afun = function(x) {
    list(
      "mean / sd" = rcell(c(mean(x), sd(x)), format = "xx.xx (xx.xx)"),
      "range" = rcell(diff(range(x)), format = "xx.xx")
    )
  })
lyt2

tbl2 <- build_table(lyt2, iris)
tbl2
```

AnalyzeVarSplit *Define a subset tabulation/analysis*

Description

Define a subset tabulation/analysis

Define a subset tabulation/analysis

Usage

```
AnalyzeVarSplit(
  var,
  split_label = var,
  afun,
  defrowlab = "",
  cfun = NULL,
  cformat = NULL,
  split_format = NULL,
  split_na_str = NA_character_,
  inclNAs = FALSE,
  split_name = var,
  extra_args = list(),
  indent_mod = 0L,
  label_pos = "default",
  cvar = "",
  section_div = NA_character_
)

AnalyzeColVarSplit(
  afun,
  defrowlab = "",
  cfun = NULL,
  cformat = NULL,
  split_format = NULL,
  split_na_str = NA_character_,
  inclNAs = FALSE,
  split_name = "",
  extra_args = list(),
  indent_mod = 0L,
  label_pos = "default",
  cvar = "",
  section_div = NA_character_
)

AnalyzeMultiVars(
  var,
  split_label = "",
  afun,
  defrowlab = "",
  cfun = NULL,
  cformat = NULL,
  split_format = NULL,
```

```
  split_na_str = NA_character_,
  inclNAs = FALSE,
  .payload = NULL,
  split_name = NULL,
  extra_args = list(),
  indent_mod = 0L,
  child_labels = c("default", "topleft", "visible", "hidden"),
  child_names = var,
  cvar = "",
  section_div = NA_character_
)
```

Arguments

`var`	string, variable name
`split_label`	string. Label string to be associated with the table generated by the split. Not to be confused with labels assigned to each child (which are based on the data and type of split during tabulation).
`afun`	function. Analysis function, must take `x` or `df` as its first parameter. Can optionally take other parameters which will be populated by the tabulation framework. See Details in `analyze`.
`defrowlab`	character. Default row labels if they are not specified by the return value of `afun`
`cfun`	list/function/NULL. tabulation function(s) for creating content rows. Must accept `x` or `df` as first parameter. Must accept `labelstr` as the second argument. Can optionally accept all optional arguments accepted by analysis functions. See `analyze`.
`cformat`	format spec. Format for content rows
`split_format`	`FormatSpec`. Default format associated with the split being created.
`split_na_str`	character. NA string vector for use with `split_format`.
`inclNAs`	boolean. Should observations with NA in the `var` variable(s) be included when performing this analysis. Defaults to `FALSE`
`split_name`	string. Name associated with this split (for pathing, etc)
`extra_args`	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.
`indent_mod`	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
`label_pos`	character(1). Location the variable label should be displayed, Accepts `"hidden"` (default for non-analyze row splits), `"visible"`, `"topleft"`, and - for analyze splits only - `"default"`. For analyze calls, `"default"` indicates that the variable should be visible if and only if multiple variables are analyzed at the same level of nesting.

cvar	character(1). The variable, if any, which the content function should accept. Defaults to NA.
section_div	character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.
.payload	Used internally, not intended to be set by end users.
child_labels	string. One of "default", "visible", "hidden". What should the display behavior be for the labels (i.e. label rows) of the children of this split. Defaults to "default" which flags the label row as visible only if the child has 0 content rows.
child_names	character. Names to be given to the sub splits contained by a compound split (typically a AnalyzeMultiVars split object).

Value

An AnalyzeVarSplit object.

An AnalyzeMultiVars split object.

Author(s)

Gabriel Becker

---

analyze_colvars *Generate Rows Analyzing Different Variables Across Columns*

Description

Generate Rows Analyzing Different Variables Across Columns

Usage

```
analyze_colvars(
  lyt,
  afun,
  format = NULL,
  na_str = NA_character_,
  nested = TRUE,
  extra_args = list(),
  indent_mod = 0L,
  inclNAs = FALSE
)
```

Arguments

lyt	layout object pre-data used for tabulation
afun	function or list. Function(s) to be used to calculate the values in each column. The list will be repped out as needed and matched by position with the columns during tabulation. This functions accepts the same parameters as analyze like afun and format. For further information see additional_fun_params.
format	FormatSpec. Format associated with this split. Formats can be declared via strings ("xx.x") or function. In cases such as analyze calls, they can character vectors or lists of functions.
na_str	character(1). String that should be displayed when the value of x is missing. Defaults to "NA".
nested	boolean. Should this layout instruction be applied within the existing layout structure *if possible* (TRUE, the default) or as a new top-level element ('FALSE). Ignored if it would nest a split underneath analyses, which is not allowed.
extra_args	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
inclNAs	boolean. Should observations with NA in the var variable(s) be included when performing this analysis. Defaults to FALSE

Value

A PreDataTableLayouts object suitable for passing to further layouting functions, and to build_table.

Author(s)

Gabriel Becker

See Also

split_cols_by_multivar()

Examples

```
library(dplyr)
ANL <- DM %>% mutate(value = rnorm(n()), pctdiff = runif(n()))

## toy example where we take the mean of the first variable and the
## count of >.5 for the second.
colfuns <- list(
  function(x) rcell(mean(x), format = "xx.x"),
  function(x) rcell(sum(x > .5), format = "xx")
```

```
)

lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by_multivar(c("value", "pctdiff")) %>%
  split_rows_by("RACE",
    split_label = "ethnicity",
    split_fun = drop_split_levels
  ) %>%
  summarize_row_groups() %>%
  analyze_colvars(afun = colfuns)
lyt

tbl <- build_table(lyt, ANL)
tbl

lyt2 <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by_multivar(c("value", "pctdiff"),
    varlabels = c("Measurement", "Pct Diff")
  ) %>%
  split_rows_by("RACE",
    split_label = "ethnicity",
    split_fun = drop_split_levels
  ) %>%
  summarize_row_groups() %>%
  analyze_colvars(afun = mean, format = "xx.xx")

tbl2 <- build_table(lyt2, ANL)
tbl2
```

`append_topleft` *Append a description to the 'top-left' materials for the layout*

Description

This function *adds* `newlines` to the current set of "top-left materials".

Usage

```
append_topleft(lyt, newlines)
```

Arguments

`lyt`	layout object pre-data used for tabulation
`newlines`	character. The new line(s) to be added to the materials

Details

Adds `newlines` to the set of strings representing the 'top-left' materials declared in the layout (the content displayed to the left of the column labels when the resulting tables are printed).

Top-left material strings are stored and then displayed *exactly as is*, no structure or indenting is applied to them either when they are added or when they are displayed.

Value

A `PreDataTableLayouts` object suitable for passing to further layouting functions, and to `build_table`.

Note

Currently, where in the construction of the layout this is called makes no difference, as it is independent of the actual splitting keywords. This may change in the future.

This function is experimental, its name and the details of its behavior are subject to change in future versions.

See Also

`top_left()`

Examples

```
library(dplyr)

DM2 <- DM %>% mutate(RACE = factor(RACE), SEX = factor(SEX))

lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by("SEX") %>%
  split_rows_by("RACE") %>%
  append_topleft("Ethnicity") %>%
  analyze("AGE") %>%
  append_topleft("  Age")

tbl <- build_table(lyt, DM2)
tbl
```

---

asvec *convert to a vector*

Description

Convert an `rtables` framework object into a vector, if possible. This is unlikely to be useful in realistic scenarios.

Usage

```
## S4 method for signature 'VTableTree'
as.vector(x, mode = "any")
```

Arguments

x	ANY. The object to be converted to a vector
mode	character(1). Passed on to as.vector

Value

a vector of the chosen mode (or an error is raised if more than one row was present).

Note

This only works for a table with a single row or a row object.

---

as_html *Convert an* rtable *object to a* shiny.tag *html object*

Description

The returned html object can be immediately used in shiny and rmarkdown.

Usage

```
as_html(
  x,
  width = NULL,
  class_table = "table table-condensed table-hover",
  class_tr = NULL,
  class_th = NULL,
  link_label = NULL,
  bold = c("header"),
  header_sep_line = TRUE,
  no_spaces_between_cells = FALSE
)
```

Arguments

x	rtable object
width	a string to indicate the desired width of the table. Common input formats include a percentage of the viewer window width (e.g. "100%") or a distance value (e.g. "300px"). Defaults to NULL.
class_table	class for table tag
class_tr	class for tr tag

class_th — class for th tag

link_label — link anchor label (not including tab: prefix) for the table.

bold — elements in table output that should be bold. Options are "main_title", "subtitles", "header", "row_names", "label_rows", and "content_rows" (which includes any non-label rows). Defaults to "header".

header_sep_line — whether a black line should be printed to under the table header. Defaults to TRUE.

no_spaces_between_cells — whether spaces between table cells should be collapsed. Defaults to FALSE.

Value

A shiny.tag object representing x in HTML.

Examples

```
tbl <- rtable(
  header = LETTERS[1:3],
  format = "xx",
  rrow("r1", 1, 2, 3),
  rrow("r2", 4, 3, 2, indent = 1),
  rrow("r3", indent = 2)
)

as_html(tbl)

as_html(tbl, class_table = "table", class_tr = "row")

as_html(tbl, bold = c("header", "row_names"))

## Not run:
Viewer(tbl)

## End(Not run)
```

---

basic_table *Layout with 1 column and zero rows*

Description

Every layout must start with a basic table.

Usage

```
basic_table(
  title = "",
  subtitles = character(),
  main_footer = character(),
  prov_footer = character(),
  header_section_div = NA_character_,
  show_colcounts = FALSE,
  colcount_format = "(N=xx)",
  inset = 0L
)
```

Arguments

title	character(1). Main title (main_title()) is a single string. Ignored for subtables.
subtitles	character. Subtitles (subtitles()) can be vector of strings, where every element is printed in a separate line. Ignored for subtables.
main_footer	character. Main global (non-referential) footer materials (main_footer()). If it is a vector of strings, they will be printed on separate lines.
prov_footer	character. Provenance-related global footer materials (prov_footer()). It can be also a vector of strings, printed on different lines. Generally should not be modified by hand.
header_section_div	character(1). String which will be used to divide the header from the table. See header_section_div() for getter and setter of these. Please consider changing last element of section_div() when concatenating tables that need a divider between them.
show_colcounts	logical(1). Should column counts be displayed in the resulting table when this layout is applied to data
colcount_format	character(1). Format for use when displaying the column counts. Must be 1d, or 2d where one component is a percent. See details.
inset	numeric(1). Number of spaces to inset the table header, table body, referential footnotes, and main_footer, as compared to alignment of title, subtitle, and provenance footer. Defaults to 0 (no inset).

Details

colcount_format is ignored if show_colcounts is FALSE (the default). When show_colcounts is TRUE, and colcount_format is 2-dimensional with a percent component, the value component for the percent is always populated with 1 (i.e. 100%). 1d formats are used to render the counts exactly as they normally would be, while 2d formats which don't include a percent, and all 3d formats result in an error. Formats in the form of functions are not supported for colcount format. See formatters::list_valid_format_labels() for the list of valid format labels to select from.

Value

A PreDataTableLayouts object suitable for passing to further layouting functions, and to build_table.

Note

- Because percent components in `colcount_format` are *always* populated with the value 1, we can get arguably strange results, such as that individual arm columns and a combined "all patients" column all list "100%" as their percentage, even though the individual arm columns represent strict subsets of the all patients column.
- Note that subtitles (`subtitles()`) and footers (`main_footer()` and `prov_footer()`) that spans more than one line can be supplied as a character vector to maintain indentation on multiple lines.

Examples

```
lyt <- basic_table() %>%
  analyze("AGE", afun = mean)

tbl <- build_table(lyt, DM)
tbl

lyt2 <- basic_table(
  title = "Title of table",
  subtitles = c("a number", "of subtitles"),
  main_footer = "test footer",
  prov_footer = paste(
    "test.R program, executed at",
    Sys.time()
  )
) %>%
  split_cols_by("ARM") %>%
  analyze("AGE", mean)

tbl2 <- build_table(lyt2, DM)
tbl2

lyt3 <- basic_table(
  show_colcounts = TRUE,
  colcount_format = "xx. (xx.%)"
) %>%
  split_cols_by("ARM")
```

---

brackets *Retrieve and assign elements of a* `TableTree`

Description

Retrieve and assign elements of a `TableTree`

Usage

```
## S4 replacement method for signature 'VTableTree,ANY,ANY,list'
x[i, j, ...] <- value

## S4 method for signature 'VTableTree,logical,logical'
x[i, j, ..., drop = FALSE]
```

Arguments

x	TableTree
i	index
j	index
...	Includes
	keep_topleft logical(1) ([only) Should the top-left material for the table be retained after subsetting. Defaults to TRUE if all rows are included (i.e. subsetting was by column), and drops it otherwise.
	keep_titles logical(1) Should title information be retained. Defaults to FALSE.
	keep_footers logical(1) Should non-referential footer information be retained. Defaults to keep_titles.
	reindex_refs logical(1). Should referential footnotes be re-indexed as if the resulting subset is the entire table. Defaults to TRUE.
value	Replacement value (list, TableRow, or TableTree)
drop	logical(1). Should the value in the cell be returned if one cell is selected by the combination of i and j. It is not possible to return a vector of values. To do so please consider using cell_values(). Defaults to FALSE.

Details

by default, subsetting drops the information about title, subtitle, main footer, provenance footer, and topleft. If only a column is selected and all rows are kept, the topleft information remains as default. Any referential footnote is kept whenever the subset table contains the referenced element.

Value

a TableTree (or ElementaryTable) object, unless a single cell was selected with drop=TRUE, in which case the (possibly multi-valued) fully stripped raw value of the selected cell.

Note

subsetting always preserve the original order, even if provided indexes do not preserve it. If sorting is needed, please consider using sort_at_path(). Also note that character indices are treated as paths, not vectors of names in both [and [<-.

See Also

Regarding sorting: sort_at_path() and how to understand path structure: summarize_row_groups(), and summarize_col_groups().

Examples

```
lyt <- basic_table(
  title = "Title",
  subtitles = c("Sub", "titles"),
  prov_footer = "prov footer",
  main_footer = "main footer"
) %>%
  split_cols_by("ARM") %>%
  split_rows_by("SEX") %>%
  analyze(c("AGE"))

tbl <- build_table(lyt, DM)
top_left(tbl) <- "Info"
tbl

# As default header, footer, and topleft information is lost
tbl[1, ]
tbl[1:2, 2]

# Also boolean filters can work
tbl[, c(FALSE, TRUE, FALSE)]

# If drop = TRUE, the content values are directly retrieved
tbl[2, 1]
tbl[2, 1, drop = TRUE]

# Drop works also if vectors are selected, but not matrices
tbl[, 1, drop = TRUE]
tbl[2, , drop = TRUE]
tbl[1, 1, drop = TRUE] # NULL because it is a label row
tbl[2, 1:2, drop = TRUE] # vectors can be returned only with cell_values()
tbl[1:2, 1:2, drop = TRUE] # no dropping because it is a matrix

# If all rows are selected, topleft is kept by default
tbl[, 2]
tbl[, 1]

# It is possible to deselect values
tbl[-2, ]
tbl[, -1]

# Values can be reassigned
tbl[2, 1] <- rcell(999)
tbl[2, ] <- list(rrow("FFF", 888, 666, 777))
tbl[6, ] <- list(-111, -222, -333)
tbl

# We can keep some information from the original table if we need
tbl[1, 2, keep_titles = TRUE]
tbl[1, 2, keep_footers = TRUE, keep_titles = FALSE]
tbl[1, 2, keep_footers = FALSE, keep_titles = TRUE]
tbl[1, 2, keep_footers = TRUE]
```

```
tbl[1, 2, keep_topleft = TRUE]

# Keeps the referential footnotes when subset contains them
fnotes_at_path(tbl, rowpath = c("SEX", "M", "AGE", "Mean")) <- "important"
tbl[4, 1]
tbl[2, 1] # None present

# We can reindex referential footnotes, so that the new table does not depend
#  on the original one
fnotes_at_path(tbl, rowpath = c("SEX", "U", "AGE", "Mean")) <- "important"
tbl[, 1] # both present
tbl[5:6, 1] # {1} because it has been indexed again
tbl[5:6, 1, reindex_refs = FALSE] # {2} -> not reindexed

# Note that order can not be changed with subsetting
tbl[c(4, 3, 1), c(3, 1)] # It preserves order and wanted selection
```

`build_table` *Create a table from a layout and data*

Description

Layouts are used to describe a table pre-data. `build_table` is used to create a table using a layout and a dataset.

Usage

```
build_table(
  lyt,
  df,
  alt_counts_df = NULL,
  col_counts = NULL,
  col_total = if (is.null(alt_counts_df)) nrow(df) else nrow(alt_counts_df),
  topleft = NULL,
  hsep = default_hsep(),
  ...
)
```

Arguments

`lyt`	layout object pre-data used for tabulation
`df`	dataset (`data.frame` or `tibble`)
`alt_counts_df`	dataset (`data.frame` or `tibble`). Alternative full data the rtables framework will use (*only*) when calculating column counts.
`col_counts`	numeric (or `NULL`). Deprecated. If non-null, column counts which override those calculated automatically during tabulation. Must specify "counts" for *all* resulting columns if non-NULL. `NA` elements will be replaced with the automatically calculated counts.

col_total	integer(1). The total observations across all columns. Defaults to nrow(df).
topleft	character. Override values for the "top left" material to be displayed during printing.
hsep	character(1). Set of character(s) to be repeated as the separator between the header and body of the table when rendered as text. Defaults to a connected horizontal line (unicode 2014) in locals that use a UTF charset, and to - elsewhere (with a once per session warning). See formatters::set_default_hsep() for further information.
...	currently ignored.

Details

When alt_counts_df is specified, column counts are calculated by applying the exact column subsetting expressions determined when applying column splitting to the main data (df) to alt_counts_df and counting the observations in each resulting subset.

In particular, this means that in the case of splitting based on cuts of the data, any dynamic cuts will have been calculated based on df and simply re-used for the count calculation.

Value

A TableTree or ElementaryTable object representing the table created by performing the tabulations declared in lyt to the data df.

Note

When overriding the column counts or totals care must be taken that, e.g., length() or nrow() are not called within tabulation functions, because those will NOT give the overridden counts. Writing/using tabulation functions which accept .N_col and .N_total or do not rely on column counts at all (even implicitly) is the only way to ensure overridden counts are fully respected.

Author(s)

Gabriel Becker

Examples

```
lyt <- basic_table() %>%
  split_cols_by("Species") %>%
  analyze("Sepal.Length", afun = function(x) {
    list(
      "mean (sd)" = rcell(c(mean(x), sd(x)), format = "xx.xx (xx.xx)"),
      "range" = diff(range(x))
    )
  })

lyt

tbl <- build_table(lyt, iris)
tbl
```

```
# analyze multiple variables
lyt2 <- basic_table() %>%
  split_cols_by("Species") %>%
  analyze(c("Sepal.Length", "Petal.Width"), afun = function(x) {
    list(
      "mean (sd)" = rcell(c(mean(x), sd(x)), format = "xx.xx (xx.xx)"),
      "range" = diff(range(x))
    )
  })

tbl2 <- build_table(lyt2, iris)
tbl2

# an example more relevant for clinical trials with column counts
lyt3 <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by("ARM") %>%
  analyze("AGE", afun = function(x) {
    setNames(as.list(fivenum(x)), c(
      "minimum", "lower-hinge", "median",
      "upper-hinge", "maximum"
    ))
  })

tbl3 <- build_table(lyt3, DM)
tbl3

tbl4 <- build_table(lyt3, subset(DM, AGE > 40))
tbl4

# with column counts calculated based on different data
miniDM <- DM[sample(1:NROW(DM), 100), ]
tbl5 <- build_table(lyt3, DM, alt_counts_df = miniDM)
tbl5

tbl6 <- build_table(lyt3, DM, col_counts = 1:3)
tbl6
```

`cbind_rtables` cbind *two* rtables

Description

cbind two rtables

Usage

```
cbind_rtables(x, ...)
```

Arguments

x	A table or row object
...	1 or more further objects of the same class as x

Value

A formal table object.

Examples

```
x <- rtable(c("A", "B"), rrow("row 1", 1, 2), rrow("row 2", 3, 4))

y <- rtable("C", rrow("row 1", 5), rrow("row 2", 6))

z <- rtable("D", rrow("row 1", 9), rrow("row 2", 10))

t1 <- cbind_rtables(x, y)
t1

t2 <- cbind_rtables(x, y, z)
t2

col_paths_summary(t1)
col_paths_summary(t2)
```

CellValue *Cell Value constructor*

Description

Cell Value constructor

Usage

```
CellValue(
  val,
  format = NULL,
  colspan = 1L,
  label = NULL,
  indent_mod = NULL,
  footnotes = NULL,
  align = NULL,
  format_na_str = NULL
)
```

Arguments

val	ANY. value in the cell exactly as it should be passed to a formatter or returned when extracted
format	FormatSpec. Format associated with this split. Formats can be declared via strings ("xx.x") or function. In cases such as analyze calls, they can character vectors or lists of functions.
colspan	integer(1). Column span value.
label	character(1). A label (not to be confused with the name) for the object/structure.
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
footnotes	list or NULL. Referential footnote messages for the cell.
align	character(1) or NULL. Alignment the value should be rendered with. It defaults to "center" if NULL is used. See formatters::list_valid_aligns() for currently supported alignments.
format_na_str	character(1). String which should be displayed when formatted if this cell's value(s) are all NA.

Value

An object representing the value within a single cell within a populated table. The underlying structure of this object is an implementation detail and should not be relied upon beyond calling accessors for the class.

---

cell_values *Retrieve cell values by row and column path*

Description

Retrieve cell values by row and column path

Usage

```
cell_values(tt, rowpath = NULL, colpath = NULL, omit_labrows = TRUE)

value_at(tt, rowpath = NULL, colpath = NULL)

## S4 method for signature 'VTableTree'
value_at(tt, rowpath = NULL, colpath = NULL)
```

Arguments

tt	TableTree (or related class). A TableTree object representing a populated table.
rowpath	character. Path in row-split space to the desired row(s). Can include "@content".
colpath	character. Path in column-split space to the desired column(s). Can include "*".
omit_labrows	logical(1). Should label rows underneath rowpath be omitted (TRUE, the default), or return empty lists of cell "values" (FALSE).

Value

for cell_values, a *list* (regardless of the type of value the cells hold). if rowpath defines a path to a single row, cell_values returns the list of cell values for that row, otherwise a list of such lists, one for each row captured underneath rowpath. This occurs after subsetting to colpath has occurred.

For value_at the "unwrapped" value of a single cell, or an error, if the combination of rowpath and colpath do not define the location of a single cell in tt.

Note

cell_values will return a single cell's value wrapped in a list. Use value_at to receive the "bare" cell value.

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by("SEX") %>%
  split_rows_by("RACE") %>%
  summarize_row_groups() %>%
  split_rows_by("STRATA1") %>%
  analyze("AGE")

library(dplyr) ## for mutate
tbl <- build_table(lyt, DM %>%
  mutate(SEX = droplevels(SEX), RACE = droplevels(RACE)))

row_paths_summary(tbl)
col_paths_summary(tbl)

cell_values(
  tbl, c("RACE", "ASIAN", "STRATA1", "B"),
  c("ARM", "A: Drug X", "SEX", "F")
)

# it's also possible to access multiple values by being less specific
cell_values(
  tbl, c("RACE", "ASIAN", "STRATA1"),
  c("ARM", "A: Drug X", "SEX", "F")
)
cell_values(tbl, c("RACE", "ASIAN"), c("ARM", "A: Drug X", "SEX", "M"))
```

```
## any arm, male columns from the ASIAN content (i.e. summary) row
cell_values(
  tbl, c("RACE", "ASIAN", "@content"),
  c("ARM", "B: Placebo", "SEX", "M")
)
cell_values(
  tbl, c("RACE", "ASIAN", "@content"),
  c("ARM", "*", "SEX", "M")
)

## all columns
cell_values(tbl, c("RACE", "ASIAN", "STRATA1", "B"))

## all columns for the Combination arm
cell_values(
  tbl, c("RACE", "ASIAN", "STRATA1", "B"),
  c("ARM", "C: Combination")
)

cvlist <- cell_values(
  tbl, c("RACE", "ASIAN", "STRATA1", "B", "AGE", "Mean"),
  c("ARM", "B: Placebo", "SEX", "M")
)
cvnolist <- value_at(
  tbl, c("RACE", "ASIAN", "STRATA1", "B", "AGE", "Mean"),
  c("ARM", "B: Placebo", "SEX", "M")
)
stopifnot(identical(cvlist[[1]], cvnolist))
```

---

clayout *Column information/structure accessors*

Description

Column information/structure accessors

Usage

```
clayout(obj)

## S4 method for signature 'VTableNodeInfo'
clayout(obj)

## S4 method for signature 'PreDataTableLayouts'
clayout(obj)

## S4 method for signature 'ANY'
```

```
clayout(obj)

clayout(object) <- value

## S4 replacement method for signature 'PreDataTableLayouts'
clayout(object) <- value

col_info(obj)

## S4 method for signature 'VTableNodeInfo'
col_info(obj)

col_info(obj) <- value

## S4 replacement method for signature 'TableRow'
col_info(obj) <- value

## S4 replacement method for signature 'ElementaryTable'
col_info(obj) <- value

## S4 replacement method for signature 'TableTree'
col_info(obj) <- value

coltree(obj, df = NULL, rtpos = TreePos())

## S4 method for signature 'InstantiatedColumnInfo'
coltree(obj, df = NULL, rtpos = TreePos())

## S4 method for signature 'PreDataTableLayouts'
coltree(obj, df = NULL, rtpos = TreePos())

## S4 method for signature 'PreDataColLayout'
coltree(obj, df = NULL, rtpos = TreePos())

## S4 method for signature 'LayoutColTree'
coltree(obj, df = NULL, rtpos = TreePos())

## S4 method for signature 'VTableTree'
coltree(obj, df = NULL, rtpos = TreePos())

## S4 method for signature 'TableRow'
coltree(obj, df = NULL, rtpos = TreePos())

col_exprs(obj, df = NULL)

## S4 method for signature 'PreDataTableLayouts'
col_exprs(obj, df = NULL)
```

```
## S4 method for signature 'PreDataColLayout'
col_exprs(obj, df = NULL)

## S4 method for signature 'InstantiatedColumnInfo'
col_exprs(obj, df = NULL)

col_counts(obj, path = NULL)

## S4 method for signature 'InstantiatedColumnInfo'
col_counts(obj, path = NULL)

## S4 method for signature 'VTableNodeInfo'
col_counts(obj, path = NULL)

col_counts(obj, path = NULL) <- value

## S4 replacement method for signature 'InstantiatedColumnInfo'
col_counts(obj, path = NULL) <- value

## S4 replacement method for signature 'VTableNodeInfo'
col_counts(obj, path = NULL) <- value

col_total(obj)

## S4 method for signature 'InstantiatedColumnInfo'
col_total(obj)

## S4 method for signature 'VTableNodeInfo'
col_total(obj)

col_total(obj) <- value

## S4 replacement method for signature 'InstantiatedColumnInfo'
col_total(obj) <- value

## S4 replacement method for signature 'VTableNodeInfo'
col_total(obj) <- value
```

Arguments

obj	ANY. The object for the accessor to access or modify
object	The object to modify in-place
value	The new value
df	data.frame/NULL. Data to use if the column information is being generated from a Pre-Data layout object
rtpos	TreePos. Root position.
path	character or NULL. col_counts getter and setter only. Path (in column structure).

Value

A LayoutColTree object.

Various column information, depending on the accessor used.

---

clear_indent_mods *Clear All Indent Mods from a Table*

Description

Clear All Indent Mods from a Table

Usage

```
clear_indent_mods(tt)

## S4 method for signature 'VTableTree'
clear_indent_mods(tt)

## S4 method for signature 'TableRow'
clear_indent_mods(tt)
```

Arguments

tt	TableTree (or related class). A TableTree object representing a populated table.

Value

The same class as tt, with all indent mods set to zero.

Examples

```
lyt1 <- basic_table() %>%
  summarize_row_groups("STUDYID", label_fstr = "overall summary") %>%
  split_rows_by("AEBODSYS", child_labels = "visible") %>%
  summarize_row_groups("STUDYID", label = "subgroup summary") %>%
  analyze("AGE", indent_mod = -1L)

tbl1 <- build_table(lyt1, ex_adae)
tbl1
clear_indent_mods(tbl1)
```

collect_leaves *Collect leaves of a table tree*

Description

Collect leaves of a table tree

Usage

```
collect_leaves(tt, incl.cont = TRUE, add.labrows = FALSE)
```

Arguments

tt	TableTree (or related class). A TableTree object representing a populated table.
incl.cont	logical. Include rows from content tables within the tree. Defaults to TRUE
add.labrows	logical. Include label rows. Defaults to FALSE

Value

A list of TableRow objects for all rows in the table

compare_rtables *Compare two rtables*

Description

Prints a matrix where . means cell matches, X means cell does cells do not match, + cell (row) is missing, and - cell (row) should not be there. If structure is set to TRUE, C indicates columnar structure mismatch, R indicates row-structure mismatch, and S indicates mismatch in both row and column structure.

Usage

```
compare_rtables(
  object,
  expected,
  tol = 0.1,
  comp.attr = TRUE,
  structure = FALSE
)
```

Arguments

object	rtable to test
expected	rtable expected
tol	numerical tolerance
comp.attr	boolean. Compare format of cells. Other attributes are silently ignored.
structure	boolean. Should structure (in the form of column and row paths to cells) be compared. Currently defaults to FALSE, but this is subject to change in future versions.

Value

a matrix of class "rtables_diff" representing the differences between object and expected as described above.

Note

In its current form compare_rtables does not take structure into account, only row and cell position.

Examples

```
t1 <- rtable(header = c("A", "B"), format = "xx", rrow("row 1", 1, 2))
t2 <- rtable(header = c("A", "B", "C"), format = "xx", rrow("row 1", 1, 2, 3))

compare_rtables(object = t1, expected = t2)

if (interactive()) {
  Viewer(t1, t2)
}

expected <- rtable(
  header = c("ARM A\nN=100", "ARM B\nN=200"),
  format = "xx",
  rrow("row 1", 10, 15),
  rrow(),
  rrow("section title"),
  rrow("row colspan", rcell(c(.345543, .4432423), colspan = 2, format = "(xx.xx, xx.xx)"))
)

expected

object <- rtable(
  header = c("ARM A\nN=100", "ARM B\nN=200"),
  format = "xx",
  rrow("row 1", 10, 15),
  rrow("section title"),
  rrow("row colspan", rcell(c(.345543, .4432423), colspan = 2, format = "(xx.xx, xx.xx)"))
)
```

```
compare_rtables(object, expected, comp.attr = FALSE)

object <- rtable(
  header = c("ARM A\nN=100", "ARM B\nN=200"),
  format = "xx",
  rrow("row 1", 10, 15),
  rrow(),
  rrow("section title")
)

compare_rtables(object, expected)

object <- rtable(
  header = c("ARM A\nN=100", "ARM B\nN=200"),
  format = "xx",
  rrow("row 1", 14, 15.03),
  rrow(),
  rrow("section title"),
  rrow("row colspan", rcell(c(.345543, .4432423), colspan = 2, format = "(xx.xx, xx.xx)"))
)

compare_rtables(object, expected)

object <- rtable(
  header = c("ARM A\nN=100", "ARM B\nN=200"),
  format = "xx",
  rrow("row 1", 10, 15),
  rrow(),
  rrow("section title"),
  rrow("row colspan", rcell(c(.345543, .4432423), colspan = 2, format = "(xx.x, xx.x)"))
)

compare_rtables(object, expected)
```

`compat_args` *Compatibility Arg Conventions*

Description

Compatibility Arg Conventions

Usage

```
compat_args(.lst, row.name, format, indent, label, inset)
```

Arguments

`.lst`	list. An already-collected list of arguments to be used instead of the elements of `....` Arguments passed via `...` will be ignored if this is specified.
`row.name`	if `NULL` then an empty string is used as `row.name` of the `rrow`.

format	character(1) or function. The format label (string) or formatter function to apply to the cell values passed via .... See list_valid_format_labels for currently supported format labels.
indent	deprecated.
label	character(1). A label (not to be confused with the name) for the object/structure.
inset	integer(1). The table inset for the row or table being constructed. See table_inset.

Value

NULL (this is an argument template dummy function)

See Also

Other conventions: constr_args(), gen_args(), lyt_args(), sf_args()

---

constr_args *Constructor Arg Conventions*

Description

Constructor Arg Conventions

Usage

```
constr_args(
  kids,
  cont,
  lev,
  iscontent,
  cinfo,
  labelrow,
  vals,
  cspan,
  label_pos,
  cindent_mod,
  cvar,
  label,
  cextra_args,
  child_names,
  title,
  subtitles,
  main_footer,
  prov_footer,
  footnotes,
  page_title,
  page_prefix,
```

```
  section_div,
  trailing_section_div,
  split_na_str,
  cna_str,
  inset,
  table_inset,
  header_section_div
)
```

Arguments

kids	list. List of direct children.
cont	ElementaryTable. Content table.
lev	integer. Nesting level (roughly, indentation level in practical terms).
iscontent	logical. Is the TableTree/ElementaryTable being constructed the content table for another TableTree.
cinfo	InstantiatedColumnInfo (or NULL). Column structure for the object being created.
labelrow	LabelRow. The LabelRow object to assign to this Table. Constructed from label by default if not specified.
vals	list. cell values for the row
cspan	integer. Column span. 1 indicates no spanning.
label_pos	character(1). Location the variable label should be displayed, Accepts "hidden" (default for non-analyze row splits), "visible", "topleft", and - for analyze splits only - "default". For analyze calls, "default" indicates that the variable should be visible if and only if multiple variables are analyzed at the same level of nesting.
cindent_mod	numeric(1). The indent modifier for the content tables generated by this split.
cvar	character(1). The variable, if any, which the content function should accept. Defaults to NA.
label	character(1). A label (not to be confused with the name) for the object/structure.
cextra_args	list. Extra arguments to be passed to the content function when tabulating row group summaries.
child_names	character. Names to be given to the sub splits contained by a compound split (typically a AnalyzeMultiVars split object).
title	character(1). Main title (main_title()) is a single string. Ignored for subtables.
subtitles	character. Subtitles (subtitles()) can be vector of strings, where every element is printed in a separate line. Ignored for subtables.
main_footer	character. Main global (non-referential) footer materials (main_footer()). If it is a vector of strings, they will be printed on separate lines.
prov_footer	character. Provenance-related global footer materials (prov_footer()). It can be also a vector of strings, printed on different lines. Generally should not be modified by hand.

footnotes — list or NULL. Referential footnotes to be applied at current level. In post-processing, this can be achieved with fnotes_at_path<-.

page_title — character. Page specific title(s).

page_prefix — character(1). Prefix, to be appended with the split value, when forcing pagination between the children of this split/table

section_div — character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.

trailing_section_div — character(1). String which will be used as a section divider after the printing of the last row contained in this (sub)-table, unless that row is also the last table row to be printed overall, or NA_character_ for none (the default). When generated via layouting, this would correspond to the section_div of the split under which this table represents a single facet.

split_na_str — character. NA string vector for use with split_format.

cna_str — character. NA string for use with cformat for content table.

inset — numeric(1). Number of spaces to inset the table header, table body, referential footnotes, and main_footer, as compared to alignment of title, subtitle, and provenance footer. Defaults to 0 (no inset).

table_inset — numeric(1). Number of spaces to inset the table header, table body, referential footnotes, and main_footer, as compared to alignment of title, subtitle, and provenance footer. Defaults to 0 (no inset).

header_section_div — character(1). String which will be used to divide the header from the table. See header_section_div() for getter and setter of these. Please consider changing last element of section_div() when concatenating tables that need a divider between them.

Value

NULL (this is an argument template dummy function)

See Also

Other conventions: compat_args(), gen_args(), lyt_args(), sf_args()

---

content_table *Retrieve or set Content Table from a* TableTree

---

Description

Returns the content table of obj if it is a TableTree object, or NULL otherwise

Usage

```
content_table(obj)

content_table(obj) <- value
```

Arguments

obj	TableTree. The TableTree
value	ElementaryTable. The new content table for obj.

Value

the ElementaryTable containing the (top level) *content rows* of obj (or NULL if obj is not a formal table object).

---

cont_n_allcols *Score functions for sorting* TableTrees

Description

Score functions for sorting TableTrees

Usage

```
cont_n_allcols(tt)

cont_n_onecol(j)
```

Arguments

tt	TableTree (or related class). A TableTree object representing a populated table.
j	numeric(1). Number of column used for scoring.

Value

A single numeric value indicating score according to the relevant metric for tt, to be used when sorting.

See Also

For examples and details please read main documentation sort_at_path() and relevant vignette ((Sorting and Pruning))

counts_wpcts *Analysis function to count levels of a factor with percentage of the column total*

Description

Analysis function to count levels of a factor with percentage of the column total

Usage

```
counts_wpcts(x, .N_col)
```

Arguments

x	factor. Vector of data, provided by rtables pagination machinery
.N_col	integer(1). Total count for the column, provided by rtables pagination machinery

Value

A RowsVerticalSection object with counts (and percents) for each level of the factor

Examples

```
counts_wpcts(DM$SEX, 400)
```

custom_split_funs *Custom Split Functions*

Description

Split functions provide the work-horse for rtables's generalized partitioning. These functions accept a (sub)set of incoming data, a split object, and return 'splits' of that data.

Custom Splitting Function Details

User-defined custom split functions can perform any type of computation on the incoming data provided that they meet the contract for generating 'splits' of the incoming data 'based on' the split object.

Split functions are functions that accept:

df data.frame of incoming data to be split

spl a Split object. this is largely an internal detail custom functions will not need to worry about, but obj_name(spl), for example, will give the name of the split as it will appear in paths in the resulting table

vals Any pre-calculated values. If given non-null values, the values returned should match these. Should be NULL in most cases and can likely be ignored

labels Any pre-calculated value labels. Same as above for `values`

trim If `TRUE`, resulting splits that are empty should be removed

(Optional) .spl_context a data.frame describing previously performed splits which collectively arrived at `df`

The function must then output a `named list` with the following elements:

values The vector of all values corresponding to the splits of `df`

datasplit a list of data.frames representing the groupings of the actual observations from `df`.

labels a character vector giving a string label for each value listed in the `values` element above

(Optional) extras If present, extra arguments are to be passed to summary and analysis functions whenever they are executed on the corresponding element of `datasplit` or a subset thereof

One way to generate custom splitting functions is to wrap existing split functions and modify either the incoming data before they are called or their outputs.

See Also

`make_split_fun()` for the API for creating custom split functions, and split_funcs for a variety of pre-defined split functions.

Examples

```
# Example of a picky split function. The number of values in the column variable
# var decrees if we are going to print also the column with all observation
# or not.

picky_splitter <- function(var) {
  # Main layout function
  function(df, spl, vals, labels, trim) {
    orig_vals <- vals

    # Check for number of levels if all are selected
    if (is.null(vals)) {
      vec <- df[[var]]
      vals <- unique(vec)
    }

    # Do a split with or without All obs
    if (length(vals) == 1) {
      do_base_split(spl = spl, df = df, vals = vals, labels = labels, trim = trim)
    } else {
      fnc_tmp <- add_overall_level("Overall", label = "All Obs", first = FALSE)
      fnc_tmp(df = df, spl = spl, vals = orig_vals, trim = trim)
    }
  }
}
```

```
# Data sub-set
d1 <- subset(ex_adsl, ARM == "A: Drug X" | (ARM == "B: Placebo" & SEX == "F"))
d1 <- subset(d1, SEX %in% c("M", "F"))
d1$SEX <- factor(d1$SEX)

# This table uses the number of values in the SEX column to add the overall col or not
lyt <- basic_table() %>%
  split_cols_by("ARM", split_fun = drop_split_levels) %>%
  split_cols_by("SEX", split_fun = picky_splitter("SEX")) %>%
  analyze("AGE", show_labels = "visible")
tbl <- build_table(lyt, d1)
tbl
```

`data.frame_export` *Generate a Result Data Frame*

Description

Collection of utilities to exctract `data.frame` from `TableTree` objects.

Usage

```
as_result_df(tt, spec = "v0_experimental", simplify = FALSE, ...)

result_df_specs()

path_enriched_df(tt, path_fun = collapse_path, value_fun = collapse_values)
```

Arguments

`tt`	`TableTree` (or related class). A `TableTree` object representing a populated table.
`spec`	character(1). The specification to use to extract the result data frame. See details
`simplify`	logical(1). If `TRUE`, the result data frame will have only visible labels and result columns.
`...`	Passed to spec-specific result data frame conversion function. Currently it can be one or more of the following parameters (valid only for `v0_experimental` spec for now): • `expand_colnames`: when `TRUE`, the result data frame will have expanded column names above the usual output. This is useful when the result data frame is used for further processing. • `simplify`: when `TRUE`, the result data frame will have only visible labels and result columns. • `as_strings`: when `TRUE`, the result data frame will have all values as strings, as they appear in the final table (it can also be retrieved from `matrix_form(tt)$strings`). This is also true for column counts if `expand_colnames = TRUE`.

- as_viewer: when TRUE, the result data frame will have all values as they appear in the final table, i.e. with the same precision and numbers, but in easy-to-use numeric form.

path_fun — function. Function to transform paths into single-string row/column names.

value_fun — function. Function to transform cell values into cells of the data.frame. Defaults to collapse_values which creates strings where multi-valued cells are collapsed together, separated by |.

Details

as_result_df(): Result data frame specifications may differ in the exact information they include and the form in which they represent it. Specifications whose names end in "_experimental" are subject to change without notice, but specifications without the "_experimental" suffix will remain available *including any bugs in their construction* indefinitely.

Value

result_df_specs(): returns a named list of result data frame extraction functions by "specification".

path_enriched_df(): returns a data frame of tt's cell values (processed by value_fun, with columns named by the full column paths (processed by path_fun and an additional row_path column with the row paths (processed by by path_fun).

Functions

- result_df_specs(): list of functions that extract result data frames from TableTrees.
- path_enriched_df(): transform TableTree object to Path-Enriched data.frame.

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("STRATA1") %>%
  analyze(c("AGE", "BMRKR2"))

tbl <- build_table(lyt, ex_adsl)
as_result_df(tbl)

result_df_specs()

lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  analyze(c("AGE", "BMRKR2"))

tbl <- build_table(lyt, ex_adsl)
path_enriched_df(tbl)
```

df_to_tt *Create* ElementaryTable *from data.frame*

Description

Create ElementaryTable from data.frame

Usage

```
df_to_tt(df)
```

Arguments

df	data.frame.

Value

an ElementaryTable object with unnested columns corresponding to names(df) and row labels corresponding to row.names(df)

Examples

```
df_to_tt(mtcars)
```

do_base_split *Apply Basic Split (For Use In Custom Split Functions)*

Description

This function is intended for use inside custom split functions. It applies the current split *as if it had no custom splitting function* so that those default splits can be further manipulated.

Usage

```
do_base_split(spl, df, vals = NULL, labels = NULL, trim = FALSE)
```

Arguments

spl	A Split object defining a partitioning or analysis/tabulation of the data.
df	dataset (data.frame or tibble)
vals	ANY. Already calculated/known values of the split. Generally should be left as NULL.
labels	character. Labels associated with vals. Should be NULL when vals is, which should almost always be the case.
trim	logical(1). Should groups corresponding to empty data subsets be removed. Defaults to FALSE.

Value

the result of the split being applied as if it had no custom split function, see custom_split_funs

Examples

```
uneven_splfun <- function(df, spl, vals = NULL, labels = NULL, trim = FALSE) {
  ret <- do_base_split(spl, df, vals, labels, trim)
  if (NROW(df) == 0) {
    ret <- lapply(ret, function(x) x[1])
  }
  ret
}

lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by_multivar(c("USUBJID", "AESEQ", "BMRKR1"),
    varlabels = c("N", "E", "BMR1"),
    split_fun = uneven_splfun
  ) %>%
  analyze_colvars(list(
    USUBJID = function(x, ...) length(unique(x)),
    AESEQ = max,
    BMRKR1 = mean
  ))

tbl <- build_table(lyt, subset(ex_adae, as.numeric(ARM) <= 2))
tbl
```

---

drop_facet_levels *Preprocessing Functions for use in make_split_fun*

Description

This function is intended for use as a preprocessing component in make_split_fun, and should not be called directly by end users.

Usage

```
drop_facet_levels(df, spl, ...)
```

Arguments

df	data.frame. The incoming data corresponding with the parent facet
spl	Split.
...	dots. This is used internally to pass parameters.

See Also

make_split_fun

Other make_custom_split: add_combo_facet(), make_split_fun(), make_split_result(), trim_levels_in_facets()

ElementaryTable-class TableTree *classes*

Description

TableTree classes

Table Constructors and Classes

Usage

```
ElementaryTable(
  kids = list(),
  name = "",
  lev = 1L,
  label = "",
 labelrow = LabelRow(lev = lev, label = label, vis = !isTRUE(iscontent) && !is.na(label)
   && nzchar(label)),
  rspans = data.frame(),
  cinfo = NULL,
  iscontent = NA,
  var = NA_character_,
  format = NULL,
  na_str = NA_character_,
  indent_mod = 0L,
  title = "",
  subtitles = character(),
  main_footer = character(),
  prov_footer = character(),
  header_section_div = NA_character_,
  hsep = default_hsep(),
  trailing_section_div = NA_character_,
  inset = 0L
)

TableTree(
  kids = list(),
  name = if (!is.na(var)) var else "",
  cont = EmptyElTable,
  lev = 1L,
  label = name,
 labelrow = LabelRow(lev = lev, label = label, vis = nrow(cont) == 0 && !is.na(label) &&
   nzchar(label)),
```

```
  rspans = data.frame(),
  iscontent = NA,
  var = NA_character_,
  cinfo = NULL,
  format = NULL,
  na_str = NA_character_,
  indent_mod = 0L,
  title = "",
  subtitles = character(),
  main_footer = character(),
  prov_footer = character(),
  page_title = NA_character_,
  hsep = default_hsep(),
  header_section_div = NA_character_,
  trailing_section_div = NA_character_,
  inset = 0L
)
```

Arguments

kids	list. List of direct children.
name	character(1). Name of the split/table/row being created. Defaults to same as the corresponding label, but is not required to be.
lev	integer. Nesting level (roughly, indentation level in practical terms).
label	character(1). A label (not to be confused with the name) for the object/structure.
labelrow	LabelRow. The LabelRow object to assign to this Table. Constructed from label by default if not specified.
rspans	data.frame. Currently stored but otherwise ignored.
cinfo	InstantiatedColumnInfo (or NULL). Column structure for the object being created.
iscontent	logical. Is the TableTree/ElementaryTable being constructed the content table for another TableTree.
var	string, variable name
format	FormatSpec. Format associated with this split. Formats can be declared via strings ("xx.x") or function. In cases such as analyze calls, they can character vectors or lists of functions.
na_str	character(1). String that should be displayed when the value of x is missing. Defaults to "NA".
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
title	character(1). Main title (main_title()) is a single string. Ignored for subtables.
subtitles	character. Subtitles (subtitles()) can be vector of strings, where every element is printed in a separate line. Ignored for subtables.

main_footer character. Main global (non-referential) footer materials (main_footer()). If it is a vector of strings, they will be printed on separate lines.

prov_footer character. Provenance-related global footer materials (prov_footer()). It can be also a vector of strings, printed on different lines. Generally should not be modified by hand.

header_section_div
character(1). String which will be used to divide the header from the table. See header_section_div() for getter and setter of these. Please consider changing last element of section_div() when concatenating tables that need a divider between them.

hsep character(1). Set of character(s) to be repeated as the separator between the header and body of the table when rendered as text. Defaults to a connected horizontal line (unicode 2014) in locals that use a UTF charset, and to - elsewhere (with a once per session warning). See formatters::set_default_hsep() for further information.

trailing_section_div
character(1). String which will be used as a section divider after the printing of the last row contained in this (sub)-table, unless that row is also the last table row to be printed overall, or NA_character_ for none (the default). When generated via layouting, this would correspond to the section_div of the split under which this table represents a single facet.

inset numeric(1). Number of spaces to inset the table header, table body, referential footnotes, and main_footer, as compared to alignment of title, subtitle, and provenance footer. Defaults to 0 (no inset).

cont ElementaryTable. Content table.

page_title character. Page specific title(s).

Value

A formal object representing a populated table.

Author(s)

Gabriel Becker

EmptyColInfo *Empty table, column, split objects*

Description

Empty objects of various types to compare against efficiently.

export_as_docx *Export as word document*

Description

From a table, produce a self-contained word document or attach it to a template word file (`template_file`). This function is based on `tt_to_flextable()` transformer and `officer` package.

Usage

```
export_as_docx(
  tt,
  file,
  doc_metadata = NULL,
  titles_as_header = FALSE,
  footers_as_text = TRUE,
  template_file = NULL,
  section_properties = NULL
)

section_properties_portrait()

section_properties_landscape()

margins_potrait()

margins_landscape()
```

Arguments

`tt`	`TableTree` (or related class). A `TableTree` object representing a populated table.
`file`	character(1). String that indicates the final file output. It needs to have `.docx` extension.
`doc_metadata`	list of character(1)s. Any value that can be used as metadata by `?officer::set_doc_properties`. Important text values are `title`, `subject`, `creator`, `description` while `created` is a date object.
`titles_as_header`	logical(1). Defaults to `TRUE` for `tt_to_flextable()`, so the table is self-contained as it makes additional header rows for `main_title()` string and `subtitles()` character vector (one per element). `FALSE` is suggested for `export_as_docx()`. This adds titles and subtitles as a text paragraph above the table. Same style is applied.
`footers_as_text`	logical(1). Defaults to `FALSE` for `tt_to_flextable()`, so the table is self-contained with the flextable definition of footnotes. `TRUE` is used for `export_as_docx()`

to add the footers as a new paragraph after the table. Same style is applied, but with a smaller font.

`template_file` character(1). Template file that `officer` will use as a starting point for the final document. It will attach the table and use the defaults defined in the template file. Output will be doc `file` nonetheless.

`section_properties`
`officer::prop_section` object. Here you can set margins and page size.

Functions

- `section_properties_portrait()`: helper function that defines standard portrait properties for tables.
- `section_properties_landscape()`: helper function that defines standard landscape properties for tables.
- `margins_potrait()`: helper function that defines standard portrait margins for tables.
- `margins_landscape()`: helper function that defines standard landscape margins for tables.

Note

`export_as_docx()` does not have many options available. We suggest, if you need specific formats and details to use `tt_to_flextable()` first and then `export_as_docx`. Only `title_as_header` and `footer_as_text` need to be specified again if changed in `tt_to_flextable()`.

See Also

`tt_to_flextable()`

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  analyze(c("AGE", "BMRKR2", "COUNTRY"))

tbl <- build_table(lyt, ex_adsl)

# See how section_properties_portrait function is built for custom
## Not run:
tf <- tempfile(fileext = ".docx")
export_as_docx(tbl, file = tf, section_properties = section_properties_portrait())

## End(Not run)
```

export_as_tsv *Create Enriched flat value table with paths*

Description

This function creates a flat tabular file of cell values and corresponding paths via path_enriched_df(). I then writes that data.frame out as a tsv file.

By default (i.e. when value_func is not specified, List columns where at least one value has length > 1 are collapsed to character vectors by collapsing the list element with "|".

Usage

```
export_as_tsv(
  tt,
  file = NULL,
  path_fun = collapse_path,
  value_fun = collapse_values
)

import_from_tsv(file)
```

Arguments

tt	TableTree (or related class). A TableTree object representing a populated table.
file	character(1). The path of the file to written to or read from.
path_fun	function. Function to transform paths into single-string row/column names.
value_fun	function. Function to transform cell values into cells of the data.frame. Defaults to collapse_values which creates strings where multi-valued cells are collapsed together, separated by |.

Value

NULL silently for export_as_tsv, a data.frame with re-constituted list values for export_as_tsv.

Note

There is currently no round-trip capability for this type of export. You can read values exported this way back in via import_from_tsv but you will receive only the data.frame version back, NOT a TableTree.

See Also

path_enriched_df() for the underlying function that does the work.

find_degen_struct *Find degenerate (sub)structures within a table (Experimental)*

Description

Find degenerate (sub)structures within a table (Experimental)

Usage

```
find_degen_struct(tt)
```

Arguments

tt	TableTree This function returns a list with the row-paths to all structural subtables which contain no data rows (even if they have associated content rows).

Value

a list of character vectors representing the row paths, if any, to degenerate substructures within the table.

Examples

```
find_degen_struct(rtable("hi"))
```

format_rcell *Format* rcell

Description

This is a wrapper around formatters::format_value for use with CellValue objects

Usage

```
format_rcell(
  x,
  format,
  output = c("ascii", "html"),
  na_str = obj_na_str(x) %||% "NA",
  pr_row_format = NULL,
  pr_row_na_str = NULL,
  shell = FALSE
)
```

Arguments

x	an object of class CellValue, or a raw value.
format	character(1) or function. The format label (string) or formatter function to apply to x.
output	character(1). Output type.
na_str	character(1). String that should be displayed when the value of x is missing. Defaults to "NA".
pr_row_format	list of default format coming from the general row.
pr_row_na_str	list of default "NA" string coming from the general row.
shell	logical(1). Should the formats themselves be returned instead of the values with formats applied. Defaults to FALSE.

Value

formatted text representing the cell

Examples

```
cll <- CellValue(pi, format = "xx.xxx")
format_rcell(cll)

# Cell values precedes the row values
cll <- CellValue(pi, format = "xx.xxx")
format_rcell(cll, pr_row_format = "xx.x")

# Similarly for NA values
cll <- CellValue(NA, format = "xx.xxx", format_na_str = "This is THE NA")
format_rcell(cll, pr_row_na_str = "This is NA")
```

---

gen_args *General Argument Conventions*

Description

General Argument Conventions

Usage

```
gen_args(
  df,
  alt_counts_df,
  spl,
  pos,
  tt,
  tr,
```

```
  verbose,
  colwidths,
  obj,
  x,
  value,
  object,
  path,
  label,
  label_pos,
  cvar,
  topleft,
  page_prefix,
  hsep,
  indent_size,
  section_div,
  na_str,
  inset,
  table_inset,
  ...
)
```

Arguments

df	dataset (data.frame or tibble)
alt_counts_df	dataset (data.frame or tibble). Alternative full data the rtables framework will use (*only*) when calculating column counts.
spl	A Split object defining a partitioning or analysis/tabulation of the data.
pos	numeric. Which top-level set of nested splits should the new layout feature be added to. Defaults to the current
tt	TableTree (or related class). A TableTree object representing a populated table.
tr	TableRow (or related class). A TableRow object representing a single row within a populated table.
verbose	logical(1). Should extra debugging messages be shown. Defaults to FALSE.
colwidths	numeric vector. Column widths for use with vertical pagination.
obj	ANY. The object for the accessor to access or modify
x	An object
value	The new value
object	The object to modify in-place
path	character. A vector path for a position within the structure of a tabletree. Each element represents a subsequent choice amongst the children of the previous choice.
label	character(1). A label (not to be confused with the name) for the object/structure.

label_pos	character(1). Location the variable label should be displayed, Accepts "hidden" (default for non-analyze row splits), "visible", "topleft", and - for analyze splits only - "default". For analyze calls, "default" indicates that the variable should be visible if and only if multiple variables are analyzed at the same level of nesting.
cvar	character(1). The variable, if any, which the content function should accept. Defaults to NA.
topleft	character. Override values for the "top left" material to be displayed during printing.
page_prefix	character(1). Prefix, to be appended with the split value, when forcing pagination between the children of this split/table
hsep	character(1). Set of character(s) to be repeated as the separator between the header and body of the table when rendered as text. Defaults to a connected horizontal line (unicode 2014) in locals that use a UTF charset, and to - elsewhere (with a once per session warning). See formatters::set_default_hsep() for further information.
indent_size	numeric(1). Number of spaces to use per indent level. Defaults to 2
section_div	character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.
na_str	character(1). String that should be displayed when the value of x is missing. Defaults to "NA".
inset	numeric(1). Number of spaces to inset the table header, table body, referential footnotes, and main_footer, as compared to alignment of title, subtitle, and provenance footer. Defaults to 0 (no inset).
table_inset	numeric(1). Number of spaces to inset the table header, table body, referential footnotes, and main_footer, as compared to alignment of title, subtitle, and provenance footer. Defaults to 0 (no inset).
...	Passed on to methods or tabulation functions.

Value

NULL (this is an argument template dummy function)

See Also

Other conventions: compat_args(), constr_args(), lyt_args(), sf_args()

`get_formatted_cells` *get formatted cells*

Description

get formatted cells

Usage

```
get_formatted_cells(obj, shell = FALSE)

## S4 method for signature 'TableTree'
get_formatted_cells(obj, shell = FALSE)

## S4 method for signature 'ElementaryTable'
get_formatted_cells(obj, shell = FALSE)

## S4 method for signature 'TableRow'
get_formatted_cells(obj, shell = FALSE)

## S4 method for signature 'LabelRow'
get_formatted_cells(obj, shell = FALSE)

get_cell_aligns(obj)

## S4 method for signature 'TableTree'
get_cell_aligns(obj)

## S4 method for signature 'ElementaryTable'
get_cell_aligns(obj)

## S4 method for signature 'TableRow'
get_cell_aligns(obj)

## S4 method for signature 'LabelRow'
get_cell_aligns(obj)
```

Arguments

`obj`	ANY. The object for the accessor to access or modify
`shell`	logical(1). Should the formats themselves be returned instead of the values with formats applied. Defaults to `FALSE`.

Value

the formatted print-strings for all (body) cells in `obj`.

Examples

```
library(dplyr)

iris2 <- iris %>%
  group_by(Species) %>%
  mutate(group = as.factor(rep_len(c("a", "b"), length.out = n()))) %>%
  ungroup()

tbl <- basic_table() %>%
  split_cols_by("Species") %>%
  split_cols_by("group") %>%
 analyze(c("Sepal.Length", "Petal.Width"), afun = list_wrap_x(summary), format = "xx.xx") %>%
  build_table(iris2)

get_formatted_cells(tbl)
```

---

head *Head and tail methods*

Description

Head and tail methods

Usage

```
head(x, ...)

## S4 method for signature 'VTableTree'
head(
  x,
  n = 6,
  ...,
  keep_topleft = TRUE,
  keep_titles = TRUE,
  keep_footers = keep_titles,
  reindex_refs = FALSE
)

tail(x, ...)

## S4 method for signature 'VTableTree'
tail(
  x,
  n = 6,
  ...,
  keep_topleft = TRUE,
  keep_titles = TRUE,
```

```
  keep_footers = keep_titles,
  reindex_refs = FALSE
)
```

Arguments

x	an object
...	arguments to be passed to or from other methods.
n	an integer vector of length up to dim(x) (or 1, for non-dimensioned objects). A logical is silently coerced to integer. Values specify the indices to be selected in the corresponding dimension (or along the length) of the object. A positive value of n[i] includes the first/last n[i] indices in that dimension, while a negative value excludes the last/first abs(n[i]), including all remaining indices. NA or non-specified values (when length(n) < length(dim(x))) select all indices in that dimension. Must contain at least one non-missing value.
keep_topleft	logical(1). If TRUE (the default), top_left material for the table will be carried over to the subset.
keep_titles	logical(1). If TRUE (the default), all title material for the table will be carried over to the subset.
keep_footers	logical(1). If TRUE, all footer material for the table will be carried over to the subset. It defaults to keep_titles.
reindex_refs	logical(1). Defaults to FALSE. If TRUE, referential footnotes will be reindexed for the subset.

---

horizontal_sep *Access or recursively set header-body separator for tables*

Description

Access or recursively set header-body separator for tables

Usage

```
horizontal_sep(obj)

## S4 method for signature 'VTableTree'
horizontal_sep(obj)

horizontal_sep(obj) <- value

## S4 replacement method for signature 'VTableTree'
horizontal_sep(obj) <- value

## S4 replacement method for signature 'TableRow'
horizontal_sep(obj) <- value
```

Arguments

obj	ANY. The object for the accessor to access or modify
value	character(1). String to use as new header/body separator.

Value

for horizontal_sep the string acting as the header separator. for horizontal_sep<-, the obj, with the new header separator applied recursively to it and all its subtables.

---

indent *Change indentation of all* rrows *in an* rtable

Description

Change indentation of all rrows in an rtable

Usage

```
indent(x, by = 1)
```

Arguments

x	rtable object
by	integer to increase indentation of rows. Can be negative. If final indentation is smaller than 0 then the indentation is set to 0.

Value

x with its indent modifier incremented by by.

Examples

```
is_setosa <- iris$Species == "setosa"
m_tbl <- rtable(
  header = rheader(
   rrow(row.name = NULL, rcell("Sepal.Length", colspan = 2), rcell("Petal.Length", colspan = 2)),
    rrow(NULL, "mean", "median", "mean", "median")
  ),
  rrow(
    row.name = "All Species",
    mean(iris$Sepal.Length), median(iris$Sepal.Length),
    mean(iris$Petal.Length), median(iris$Petal.Length),
    format = "xx.xx"
  ),
  rrow(
    row.name = "Setosa",
    mean(iris$Sepal.Length[is_setosa]), median(iris$Sepal.Length[is_setosa]),
    mean(iris$Petal.Length[is_setosa]), median(iris$Petal.Length[is_setosa]),
```

```
    format = "xx.xx"
  )
)
indent(m_tbl)
indent(m_tbl, 2)
```

`indent_string` *Indent Strings*

Description

Used in rtables to indent row names for the ASCII output.

Usage

```
indent_string(x, indent = 0, incr = 2, including_newline = TRUE)
```

Arguments

`x`	a character vector
`indent`	a vector of length `length(x)` with non-negative integers
`incr`	non-negative integer: number of spaces per indent level
`including_newline`	boolean: should newlines also be indented

Value

`x` indented by left-padding with `indent*incr` white-spaces.

Examples

```
indent_string("a", 0)
indent_string("a", 1)
indent_string(letters[1:3], 0:2)
indent_string(paste0(letters[1:3], "\n", LETTERS[1:3]), 0:2)
```

`insert_row_at_path` *Insert Row at Path*

Description

Insert a row into an existing table directly before or directly after an existing data (i.e., non-content and non-label) row, specified by its path.

Usage

```
insert_row_at_path(tt, path, value, after = FALSE)

## S4 method for signature 'VTableTree,DataRow'
insert_row_at_path(tt, path, value, after = FALSE)

## S4 method for signature 'VTableTree,ANY'
insert_row_at_path(tt, path, value)
```

Arguments

`tt`	`TableTree` (or related class). A `TableTree` object representing a populated table.
`path`	character. A vector path for a position within the structure of a `tabletree`. Each element represents a subsequent choice amongst the children of the previous choice.
`value`	The new value
`after`	logical(1). Should `value` be added as a row directly before (`FALSE`, the default) or after (`TRUE`) the row specified by `path`.

See Also

`DataRow()` `rrow()`

Examples

```
lyt <- basic_table() %>%
  split_rows_by("COUNTRY", split_fun = keep_split_levels(c("CHN", "USA"))) %>%
  analyze("AGE")

tbl <- build_table(lyt, DM)

tbl2 <- insert_row_at_path(
  tbl, c("COUNTRY", "CHN", "AGE", "Mean"),
  rrow("new row", 555)
)
tbl2
```

```
tbl3 <- insert_row_at_path(tbl2, c("COUNTRY", "CHN", "AGE", "Mean"),
  rrow("new row redux", 888),
  after = TRUE
)
tbl3
```

`insert_rrow` *[DEPRECATED] insert* `rrows` *at (before) a specific location*

Description

This function is deprecated and will be removed in a future release of `rtables`. Please use `insert_row_at_path` or `label_at_path` instead.

Usage

```
insert_rrow(tbl, rrow, at = 1, ascontent = FALSE)
```

Arguments

`tbl`	`rtable`
`rrow`	`rrow` to append to `rtable`
`at`	position into which to put the `rrow`, defaults to beginning (i.e. 1)
`ascontent`	logical. Currently ignored.

Value

A `TableTree` of the same specific class as `tbl`

Note

Label rows (i.e. a row with no data values, only a `row.name`) can only be inserted at positions which do not already contain a label row when there is a non-trivial nested row structure in `tbl`

Examples

```
o <- options(warn = 0)
lyt <- basic_table() %>%
  split_cols_by("Species") %>%
  analyze("Sepal.Length")

tbl <- build_table(lyt, iris)

insert_rrow(tbl, rrow("Hello World"))
insert_rrow(tbl, rrow("Hello World"), at = 2)

lyt2 <- basic_table() %>%
```

```
  split_cols_by("Species") %>%
  split_rows_by("Species") %>%
  analyze("Sepal.Length")

tbl2 <- build_table(lyt2, iris)

insert_rrow(tbl2, rrow("Hello World"))
insert_rrow(tbl2, rrow("Hello World"), at = 2)
insert_rrow(tbl2, rrow("Hello World"), at = 4)

insert_rrow(tbl2, rrow("new row", 5, 6, 7))

insert_rrow(tbl2, rrow("new row", 5, 6, 7), at = 3)

options(o)
```

`InstantiatedColumnInfo-class` `InstantiatedColumnInfo`

Description

`InstantiatedColumnInfo`

Usage

```
InstantiatedColumnInfo(
  treelyt = LayoutColTree(),
  csubs = list(expression(TRUE)),
  extras = list(list()),
  cnts = NA_integer_,
  total_cnt = NA_integer_,
  dispcounts = FALSE,
  countformat = "(N=xx)",
  count_na_str = "",
  topleft = character()
)
```

Arguments

`treelyt`	`LayoutColTree`.
`csubs`	list. List of subsetting expressions
`extras`	list. Extra arguments associated with the columns
`cnts`	integer. Counts.
`total_cnt`	integer(1). Total observations represented across all columns.
`dispcounts`	logical(1). Should the counts be displayed as header info when the associated table is printed.

countformat	character(1). Format for the counts if they are displayed
count_na_str	character. NA string to be used when formatting counts. Defaults to "".
topleft	character. Override values for the "top left" material to be displayed during printing.

Value

an InstantiateadColumnInfo object.

---

in_rows *Create multiple rows in analysis or summary functions*

Description

define the cells that get placed into multiple rows in afun

Usage

```
in_rows(
  ...,
  .list = NULL,
  .names = NULL,
  .labels = NULL,
  .formats = NULL,
  .indent_mods = NULL,
  .cell_footnotes = list(NULL),
  .row_footnotes = list(NULL),
  .aligns = NULL,
  .format_na_strs = NULL
)
```

Arguments

...	single row defining expressions
.list	list. list cell content, usually rcells, the .list is concatenated to ...
.names	character or NULL. Names of the returned list/structure.
.labels	character or NULL. labels for the defined rows
.formats	character or NULL. Formats for the values
.indent_mods	integer or NULL. Indent modifications for the defined rows.
.cell_footnotes	list. Referential footnote messages to be associated by name with *cells*.
.row_footnotes	list. Referential footnotes messages to be associated by name with *rows*.
.aligns	character or NULL. Alignments for the cells. Standard for NULL is "center". See formatters::list_valid_aligns() for currently supported alignments.
.format_na_strs	character or NULL. NA strings for the cells

Value

an RowsVerticalSection object (or NULL). The details of this object should be considered an internal implementation detail.

Note

In post-processing, referential footnotes can also be added using row and column paths with fnotes_at_path<-.

See Also

analyze()

Examples

```
in_rows(1, 2, 3, .names = c("a", "b", "c"))
in_rows(1, 2, 3, .labels = c("a", "b", "c"))
in_rows(1, 2, 3, .names = c("a", "b", "c"), .labels = c("AAA", "BBB", "CCC"))

in_rows(.list = list(a = 1, b = 2, c = 3))
in_rows(1, 2, .list = list(3), .names = c("a", "b", "c"))

lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  analyze("AGE", afun = function(x) {
    in_rows(
      "Mean (sd)" = rcell(c(mean(x), sd(x)), format = "xx.xx (xx.xx)"),
      "Range" = rcell(range(x), format = "xx.xx - xx.xx")
    )
  })

tbl <- build_table(lyt, ex_adsl)
tbl
```

---

is_rtable *Check if an object is a valid* rtable

Description

Check if an object is a valid rtable

Usage

```
is_rtable(x)
```

Arguments

x	an object

Value

TRUE if x is a formal Table object, FALSE otherwise.

Examples

```
is_rtable(build_table(basic_table(), iris))
```

LabelRow *Row classes and constructors*

Description

Row classes and constructors

Row constructors and Classes

Usage

```
LabelRow(
  lev = 1L,
  label = "",
  name = label,
  vis = !is.na(label) && nzchar(label),
  cinfo = EmptyColInfo,
  indent_mod = 0L,
  table_inset = 0L,
  trailing_section_div = NA_character_
)

.tablerow(
  vals = list(),
  name = "",
  lev = 1L,
  label = name,
  cspan = rep(1L, length(vals)),
  cinfo = EmptyColInfo,
  var = NA_character_,
  format = NULL,
  na_str = NA_character_,
  klass,
  indent_mod = 0L,
  footnotes = list(),
  table_inset = 0L,
  trailing_section_div = NA_character_
)

DataRow(...)

ContentRow(...)
```

Arguments

lev	integer. Nesting level (roughly, indentation level in practical terms).
label	character(1). A label (not to be confused with the name) for the object/structure.
name	character(1). Name of the split/table/row being created. Defaults to same as the corresponding label, but is not required to be.
vis	logical. Should the row be visible (LabelRow only).
cinfo	InstantiatedColumnInfo (or NULL). Column structure for the object being created.
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
table_inset	numeric(1). Number of spaces to inset the table header, table body, referential footnotes, and main_footer, as compared to alignment of title, subtitle, and provenance footer. Defaults to 0 (no inset).
trailing_section_div	character(1). String which will be used as a section divider after the printing of the last row contained in this (sub)-table, unless that row is also the last table row to be printed overall, or NA_character_ for none (the default). When generated via layouting, this would correspond to the section_div of the split under which this table represents a single facet.
vals	list. cell values for the row
cspan	integer. Column span. 1 indicates no spanning.
var	string, variable name
format	FormatSpec. Format associated with this split. Formats can be declared via strings ("xx.x") or function. In cases such as analyze calls, they can character vectors or lists of functions.
na_str	character(1). String that should be displayed when the value of x is missing. Defaults to "NA".
klass	Internal detail.
footnotes	list or NULL. Referential footnotes to be applied at current level. In post-processing, this can be achieved with fnotes_at_path<-.
...	passed to shared constructor (.tablerow).

Value

A formal object representing a table row of the constructed type.

Author(s)

Gabriel Becker

label_at_path *Label at Path*

Description

Gets or sets the label at a path

Usage

```
label_at_path(tt, path)

label_at_path(tt, path) <- value
```

Arguments

tt	TableTree (or related class). A TableTree object representing a populated table.
path	character. A vector path for a position within the structure of a tabletree. Each element represents a subsequent choice amongst the children of the previous choice.
value	The new value

Details

If path resolves to a single row, the label for that row is retrieved or set. If, instead, path resolves to a subtable, the text for the row-label associated with that path is retrieved or set. In the subtable case, if the label text is set to a non-NA value, the labelrow will be set to visible, even if it was not before. Similarly, if the label row text for a subtable is set to NA, the label row will bet set to non-visible, so the row will not appear at all when the table is printed.

Note

When changing the row labels for content rows, it is important to path all the way to the *row*. Paths ending in "@content" will not exhibit the behavior you want, and are thus an error. See row_paths for help determining the full paths to content rows.

Examples

```
lyt <- basic_table() %>%
  split_rows_by("COUNTRY", split_fun = keep_split_levels(c("CHN", "USA"))) %>%
  analyze("AGE")

tbl <- build_table(lyt, DM)

label_at_path(tbl, c("COUNTRY", "CHN"))

label_at_path(tbl, c("COUNTRY", "USA")) <- "United States"
tbl
```

length,CellValue-method *Length of a Cell value*

Description

Length of a Cell value

Usage

```
## S4 method for signature 'CellValue'
length(x)
```

Arguments

x	x.

Value

Always returns 1L

list_wrap_x *Returns a function that coerces the return values of f to a list*

Description

Returns a function that coerces the return values of f to a list

Usage

```
list_wrap_x(f)

list_wrap_df(f)
```

Arguments

f	The function to wrap.

Details

`list_wrap_x` generates a wrapper which takes `x` as its first argument, while `list_wrap_df` generates an otherwise identical wrapper function whose first argument is named `df`.

We provide both because when using the functions as tabulation in `analyze`, functions which take `df` as their first argument are passed the full subset dataframe, while those which accept anything else notably including `x` are passed only the relevant subset of the variable being analyzed.

Value

A function which calls `f` and converts the result to a list of `CellValue` objects.

Author(s)

Gabriel Becker

Examples

```
summary(iris$Sepal.Length)

f <- list_wrap_x(summary)
f(x = iris$Sepal.Length)

f2 <- list_wrap_df(summary)
f2(df = iris$Sepal.Length)
```

---

lyt_args *Layouting Function Arg Conventions*

Description

Layouting Function Arg Conventions

Usage

```
lyt_args(
  lyt,
  var,
  vars,
  label,
  labels_var,
  varlabels,
  varnames,
  split_format,
  split_na_str,
  nested,
  format,
  cfun,
  cformat,
  cna_str,
  split_fun,
  split_name,
  split_label,
  afun,
  inclNAs,
```

```
  valorder,
  ref_group,
  compfun,
  label_fstr,
  child_labels,
  extra_args,
  name,
  cuts,
  cutlabels,
  cutfun,
  cutlabelfun,
  cumulative,
  indent_mod,
  show_labels,
  label_pos,
  var_labels,
  cvar,
  table_names,
  topleft,
  align,
  page_by,
  page_prefix,
  format_na_str,
  section_div,
  na_str
)
```

Arguments

lyt	layout object pre-data used for tabulation
var	string, variable name
vars	character vector. Multiple variable names.
label	character(1). A label (not to be confused with the name) for the object/structure.
labels_var	string, name of variable containing labels to be displayed for the values of var
varlabels	character vector. Labels for vars
varnames	character vector. Names for vars which will appear in pathing. When vars are all unique this will be the variable names. If not, these will be variable names with suffixes as necessary to enforce uniqueness.
split_format	FormatSpec. Default format associated with the split being created.
split_na_str	character. NA string vector for use with split_format.
nested	boolean. Should this layout instruction be applied within the existing layout structure *if possible* (TRUE, the default) or as a new top-level element ('FALSE). Ignored if it would nest a split underneath analyses, which is not allowed.
format	FormatSpec. Format associated with this split. Formats can be declared via strings ("xx.x") or function. In cases such as analyze calls, they can character vectors or lists of functions.

cfun	list/function/NULL. tabulation function(s) for creating content rows. Must accept x or df as first parameter. Must accept labelstr as the second argument. Can optionally accept all optional arguments accepted by analysis functions. See analyze.
cformat	format spec. Format for content rows
cna_str	character. NA string for use with cformat for content table.
split_fun	function/NULL. custom splitting function See custom_split_funs
split_name	string. Name associated with this split (for pathing, etc)
split_label	string. Label string to be associated with the table generated by the split. Not to be confused with labels assigned to each child (which are based on the data and type of split during tabulation).
afun	function. Analysis function, must take x or df as its first parameter. Can optionally take other parameters which will be populated by the tabulation framework. See Details in analyze.
inclNAs	boolean. Should observations with NA in the var variable(s) be included when performing this analysis. Defaults to FALSE
valorder	character vector. Order that the split children should appear in resulting table.
ref_group	character. Value of var to be taken as the ref_group/control to be compared against.
compfun	function/string. The comparison function which accepts the analysis function outputs for two different partitions and returns a single value. Defaults to subtraction. If a string, taken as the name of a function.
label_fstr	string. An sprintf style format string containing. For non-comparison splits, it can contain up to one "%s" which takes the current split value and generates the row/column label. Comparison-based splits it can contain up to two "%s".
child_labels	string. One of "default", "visible", "hidden". What should the display behavior be for the labels (i.e. label rows) of the children of this split. Defaults to "default" which flags the label row as visible only if the child has 0 content rows.
extra_args	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.
name	character(1). Name of the split/table/row being created. Defaults to same as the corresponding label, but is not required to be.
cuts	numeric. Cuts to use
cutlabels	character (or NULL). Labels for the cuts
cutfun	function. Function which accepts the *full vector* of var values and returns cut points to be used (via cut) when splitting data during tabulation
cutlabelfun	function. Function which returns either labels for the cuts or NULL when passed the return value of cutfun
cumulative	logical. Should the cuts be treated as cumulative. Defaults to FALSE

indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
show_labels	character(1). Should the variable labels for corresponding to the variable(s) in vars be visible in the resulting table.
label_pos	character(1). Location the variable label should be displayed, Accepts "hidden" (default for non-analyze row splits), "visible", "topleft", and - for analyze splits only - "default". For analyze calls, "default" indicates that the variable should be visible if and only if multiple variables are analyzed at the same level of nesting.
var_labels	character. Variable labels for 1 or more variables
cvar	character(1). The variable, if any, which the content function should accept. Defaults to NA.
table_names	character. Names for the tables representing each atomic analysis. Defaults to var.
topleft	character. Override values for the "top left" material to be displayed during printing.
align	character(1) or NULL. Alignment the value should be rendered with. It defaults to "center" if NULL is used. See formatters::list_valid_aligns() for currently supported alignments.
page_by	logical(1). Should pagination be forced between different children resulting form this split. An error will rise if the selected split does not contain at least one value that is not NA.
page_prefix	character(1). Prefix, to be appended with the split value, when forcing pagination between the children of this split/table
format_na_str	character(1). String which should be displayed when formatted if this cell's value(s) are all NA.
section_div	character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.
na_str	character(1). String that should be displayed when the value of x is missing. Defaults to "NA".

Value

NULL (this is an argument template dummy function)

See Also

Other conventions: compat_args(), constr_args(), gen_args(), sf_args()

make_afun *Create custom analysis function wrapping existing function*

Description

Create custom analysis function wrapping existing function

Usage

```
make_afun(
  fun,
  .stats = NULL,
  .formats = NULL,
  .labels = NULL,
  .indent_mods = NULL,
  .ungroup_stats = NULL,
  .format_na_strs = NULL,
  ...,
  .null_ref_cells = ".in_ref_col" %in% names(formals(fun))
)
```

Arguments

fun	function. The function to be wrapped in a new customized analysis fun. Should return named list.
.stats	character. Names of elements to keep from fun's full output.
.formats	ANY. vector/list of formats to override any defaults applied by fun.
.labels	character. Vector of labels to override defaults returned by fun
.indent_mods	integer. Named vector of indent modifiers for the generated rows.
.ungroup_stats	character. Vector of names, which must match elements of .stats
.format_na_strs	ANY. vector/list of na strings to override any defaults applied by fun.
...	dots. Additional arguments to fun which effectively become new defaults. These can still be overridden by extra_args within a split.
.null_ref_cells	logical(1). Should cells for the reference column be NULL-ed by the returned analysis function. Defaults to TRUE if fun accepts .in_ref_col as a formal argument. Note this argument occurs after ... so it must be *fully* specified by name when set.

Value

A function suitable for use in analyze with element selection, reformatting, and relabeling performed automatically.

Note

setting `.ungroup_stats` to non-null changes the *structure* of the value(s) returned by `fun`, rather than just labeling (`.labels`), formatting (`.formats`), and selecting amongst (`.stats`) them. This means that subsequent `make_afun` calls to customize the output further both can and must operate on the new structure, *NOT* the original structure returned by `fun`. See the final pair of examples below.

See Also

`analyze()`

Examples

```
s_summary <- function(x) {
  stopifnot(is.numeric(x))

  list(
    n = sum(!is.na(x)),
    mean_sd = c(mean = mean(x), sd = sd(x)),
    min_max = range(x)
  )
}

s_summary(iris$Sepal.Length)

a_summary <- make_afun(
  fun = s_summary,
  .formats = c(n = "xx", mean_sd = "xx.xx (xx.xx)", min_max = "xx.xx - xx.xx"),
  .labels = c(n = "n", mean_sd = "Mean (sd)", min_max = "min - max")
)

a_summary(x = iris$Sepal.Length)

a_summary2 <- make_afun(a_summary, .stats = c("n", "mean_sd"))

a_summary2(x = iris$Sepal.Length)

a_summary3 <- make_afun(a_summary, .formats = c(mean_sd = "(xx.xxx, xx.xxx)"))



s_foo <- function(df, .N_col, a = 1, b = 2) {
  list(
    nrow_df = nrow(df),
    .N_col = .N_col,
    a = a,
    b = b
  )
}

s_foo(iris, 40)
```

```
a_foo <- make_afun(s_foo,
  b = 4,
  .formats = c(nrow_df = "xx.xx", ".N_col" = "xx.", a = "xx", b = "xx.x"),
  .labels = c(
    nrow_df = "Nrow df",
    ".N_col" = "n in cols", a = "a value", b = "b value"
  ),
  .indent_mods = c(nrow_df = 2L, a = 1L)
)

a_foo(iris, .N_col = 40)
a_foo2 <- make_afun(a_foo, .labels = c(nrow_df = "Number of Rows"))
a_foo2(iris, .N_col = 40)

# grouping and further customization
s_grp <- function(df, .N_col, a = 1, b = 2) {
  list(
    nrow_df = nrow(df),
    .N_col = .N_col,
    letters = list(
      a = a,
      b = b
    )
  )
}
a_grp <- make_afun(s_grp,
  b = 3,
  .labels = c(
    nrow_df = "row count",
    .N_col = "count in column"
  ),
  .formats = c(nrow_df = "xx.", .N_col = "xx."),
  .indent_mod = c(letters = 1L),
  .ungroup_stats = "letters"
)
a_grp(iris, 40)
a_aftergrp <- make_afun(a_grp,
  .stats = c("nrow_df", "b"),
  .formats = c(b = "xx.")
)
a_aftergrp(iris, 40)

s_ref <- function(x, .in_ref_col, .ref_group) {
  list(
    mean_diff = mean(x) - mean(.ref_group)
  )
}

a_ref <- make_afun(s_ref,
  .labels = c(mean_diff = "Mean Difference from Ref")
)
a_ref(iris$Sepal.Length, .in_ref_col = TRUE, 1:10)
```

```
a_ref(iris$Sepal.Length, .in_ref_col = FALSE, 1:10)
```

`make_col_df` *Column Layout Summary*

Description

Generate a structural summary of the columns of an rtables table and return it as a data.frame.

Usage

```
make_col_df(tt, colwidths = NULL, visible_only = TRUE)
```

Arguments

`tt`	ANY. Object representing the table-like object to be summarized.
`colwidths`	numeric. Internal detail do not set manually.
`visible_only`	logical(1). Should only visible aspects of the table structure be reflected in this summary. Defaults to `TRUE`. May not be supported by all methods.

Details

Used for Pagination

`make_split_fun` *Create a Custom Splitting Function*

Description

Create a Custom Splitting Function

Usage

```
make_split_fun(pre = list(), core_split = NULL, post = list())
```

Arguments

`pre`	list. Zero or more functions which operate on the incoming data and return a new data frame that should split via `core_split`. They will be called on the data in the order they appear in the list.
`core_split`	function or NULL. If not NULL, a function which accepts the same arguments do_base_split does, and returns the same type of named list. Custom functions which override this behavior cannot be used in column splits.
`post`	list. Zero or more functions which should be called on the list output by splitting.

Details

Custom split functions can be thought of as (up to) 3 different types of manipulations of the splitting process

1. Preprocessing of the incoming data to be split
2. (Row-splitting only) Customization of the core mapping of incoming data to facets, and
3. Postprocessing operations on the set of facets (groups) generated by the split.

This function provides an interface to create custom split functions by implementing and specifying sets of operations in each of those classes of customization independently.

Preprocessing functions (1), must accept: `df`, `spl`, `vals`, `labels`, and can optionally accept `.spl_context`. They then manipulate `df` (the incoming data for the split) and return a modified data.frame. This modified data.frame *must* contain all columns present in the incoming data.frame, but can add columns if necessary (though we note that these new columns cannot be used in the layout as split or analysis variables, because they will not be present when validity checking is done).

The preprocessing component is useful for things such as manipulating factor levels, e.g., to trim unobserved ones or to reorder levels based on observed counts, etc.

Customization of core splitting (2) is currently only supported in row splits. Core splitting functions override the fundamental splitting procedure, and are only necessary in rare cases. These must accept `spl`, `df`, `vals`, `labels`, and can optionally accept `.spl_context`. They must return a named list with elements, all of the same length, as follows: - `datasplit` (containing a list of data.frames), - `values` containing values associated with the facets, which must be `character` or `SplitValue` objects. These values will appear in the paths of the resulting table. - `labels` containing the character labels associated with `values`

Postprocessing functions (3) must accept the result of the core split as their first argument (which as of writing can be anything), in addition to `spl`, and `fulldf`, and can optionally accept `.spl_context`. They must each return a modified version of the same structure specified above for core splitting.

In both the pre- and post-processing cases, multiple functions can be specified. When this happens, they are applied sequentially, in the order they appear in the list passed to the relevant argument (`pre` and `post`, respectively).

Value

A function for use as a custom split function.

See Also

custom_split_funs for a more detailed discussion on what custom split functions do.

Other make_custom_split: `add_combo_facet()`, `drop_facet_levels()`, `make_split_result()`, `trim_levels_in_facets()`

Examples

```
mysplitfun <- make_split_fun(
  pre = list(drop_facet_levels),
  post = list(add_overall_facet("ALL", "All Arms"))
```

```
)


basic_table(show_colcounts = TRUE) %>%
  split_cols_by("ARM", split_fun = mysplitfun) %>%
  analyze("AGE") %>%
  build_table(subset(DM, ARM %in% c("B: Placebo", "C: Combination")))

## post (and pre) arguments can take multiple functions, here
## we add an overall facet and the reorder the facets
reorder_facets <- function(splret, spl, fulldf, ...) {
  ord <- order(names(splret$values))
  make_split_result(
    splret$values[ord],
    splret$datasplit[ord],
    splret$labels[ord]
  )
}

mysplitfun2 <- make_split_fun(
  pre = list(drop_facet_levels),
  post = list(
    add_overall_facet("ALL", "All Arms"),
    reorder_facets
  )
)
basic_table(show_colcounts = TRUE) %>%
  split_cols_by("ARM", split_fun = mysplitfun2) %>%
  analyze("AGE") %>%
  build_table(subset(DM, ARM %in% c("B: Placebo", "C: Combination")))

very_stupid_core <- function(spl, df, vals, labels, .spl_context) {
  make_split_result(c("stupid", "silly"),
    datasplit = list(df[1:10, ], df[11:30, ]),
    labels = c("first 10", "second 20")
  )
}

dumb_30_facet <- add_combo_facet("dumb",
  label = "thirty patients",
  levels = c("stupid", "silly")
)
nonsense_splfun <- make_split_fun(
  core_split = very_stupid_core,
  post = list(dumb_30_facet)
)

## recall core split overriding is not supported in column space
## currently, but we can see it in action in row space

lyt_silly <- basic_table() %>%
  split_rows_by("ARM", split_fun = nonsense_splfun) %>%
  summarize_row_groups() %>%
```

```
  analyze("AGE")
silly_table <- build_table(lyt_silly, DM)
silly_table
```

`make_split_result` *Construct split result object*

Description

These functions can be used to create or add to a split result in functions which implement core splitting or post-processing within a custom split function.

Usage

```
make_split_result(values, datasplit, labels, extras = NULL)

add_to_split_result(splres, values, datasplit, labels, extras = NULL)
```

Arguments

`values`	character or `list(SplitValue)`. The values associated with each facet
`datasplit`	`list(data.frame)`. The facet data for each facet generated in the split
`labels`	character. The labels associated with each facet
`extras`	NULL or list. Extra values associated with each of the facets which will be passed to analysis functions applied within the facet.
`splres`	list. A list representing the result of splitting.

Details

These functions does various housekeeping to ensure that the split result list is as the rtables internals expect it, most of which are not relevant to end users.

Value

a named list representing the facets generated by the split with elements `values`, `datasplit`, and `labels`, which are the same length and correspond to each other elementwise.

See Also

Other make_custom_split: `add_combo_facet()`, `drop_facet_levels()`, `make_split_fun()`, `trim_levels_in_facets()`

Examples

```
splres <- make_split_result(
  values = c("hi", "lo"),
  datasplit = list(hi = mtcars, lo = mtcars[1:10, ]),
  labels = c("more data", "less data")
)

splres2 <- add_to_split_result(splres,
  values = "med",
  datasplit = list(med = mtcars[1:20, ]),
  labels = "kinda some data"
)
```

ManualSplit *Manually defined split*

Description

Manually defined split

Usage

```
ManualSplit(
  levels,
  label,
  name = "manual",
  extra_args = list(),
  indent_mod = 0L,
  cindent_mod = 0L,
  cvar = "",
  cextra_args = list(),
  label_pos = "visible",
  page_prefix = NA_character_,
  section_div = NA_character_
)
```

Arguments

levels	character. Levels of the split (i.e. the children of the manual split)
label	character(1). A label (not to be confused with the name) for the object/structure.
name	character(1). Name of the split/table/row being created. Defaults to same as the corresponding label, but is not required to be.
extra_args	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.

indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
cindent_mod	numeric(1). The indent modifier for the content tables generated by this split.
cvar	character(1). The variable, if any, which the content function should accept. Defaults to NA.
cextra_args	list. Extra arguments to be passed to the content function when tabulating row group summaries.
label_pos	character(1). Location the variable label should be displayed, Accepts "hidden" (default for non-analyze row splits), "visible", "topleft", and - for analyze splits only - "default". For analyze calls, "default" indicates that the variable should be visible if and only if multiple variables are analyzed at the same level of nesting.
page_prefix	character(1). Prefix, to be appended with the split value, when forcing pagination between the children of this split/table
section_div	character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.

Value

A ManualSplit object.

Author(s)

Gabriel Becker

---

manual_cols *Manual column declaration*

Description

Manual column declaration

Usage

```
manual_cols(..., .lst = list(...))
```

Arguments

...	One or more vectors of levels to appear in the column space. If more than one set of levels is given, the values of the second are nested within each value of the first, and so on.
.lst	A list of sets of levels, by default populated via list(...).

Value

An InstantiatedColumnInfo object, suitable for use declaring the column structure for a manually constructed table.

Author(s)

Gabriel Becker

Examples

```
# simple one level column space
rows <- lapply(1:5, function(i) {
  DataRow(rep(i, times = 3))
})
tbl <- TableTree(kids = rows, cinfo = manual_cols(split = c("a", "b", "c")))
tbl

# manually declared nesting
tbl2 <- TableTree(
  kids = list(DataRow(as.list(1:4))),
  cinfo = manual_cols(
    Arm = c("Arm A", "Arm B"),
    Gender = c("M", "F")
  )
)
tbl2
```

---

matrix_form,VTableTree-method

Transform rtable *to a list of matrices which can be used for outputting*

Description

Although rtables are represented as a tree data structure when outputting the table to ASCII or HTML it is useful to map the rtable to an in between state with the formatted cells in a matrix form.

Usage

```
## S4 method for signature 'VTableTree'
matrix_form(
  obj,
  indent_rownames = FALSE,
  expand_newlines = TRUE,
  indent_size = 2
)
```

Arguments

obj	ANY. The object for the accessor to access or modify
indent_rownames	logical(1), if TRUE the column with the row names in the strings matrix of has indented row names (strings pre-fixed)
expand_newlines	logical(1). Should the matrix form generated expand rows whose values contain newlines into multiple 'physical' rows (as they will appear when rendered into ASCII). Defaults to TRUE
indent_size	numeric(1). Number of spaces to use per indent level. Defaults to 2

Details

The strings in the return object are defined as follows: row labels are those determined by make_row_df and cell values are determined using get_formatted_cells. (Column labels are calculated using a non-exported internal function.

Value

A list with the following elements:

strings The content, as it should be printed, of the top-left material, column headers, row labels , and cell values of tt

spans The column-span information for each print-string in the strings matrix

aligns The text alignment for each print-string in the strings matrix

display Whether each print-string in the strings matrix should be printed or not.

row_info the data.frame generated by make_row_df

With an additional nrow_header attribute indicating the number of pseudo "rows" the column structure defines.

Examples

```
library(dplyr)

iris2 <- iris %>%
  group_by(Species) %>%
  mutate(group = as.factor(rep_len(c("a", "b"), length.out = n()))) %>%
  ungroup()

lyt <- basic_table() %>%
  split_cols_by("Species") %>%
  split_cols_by("group") %>%
  analyze(c("Sepal.Length", "Petal.Width"),
    afun = list_wrap_x(summary), format = "xx.xx"
  )

lyt
```

```
tbl <- build_table(lyt, iris2)

matrix_form(tbl)
```

`MultiVarSplit` *Split between two or more different variables*

Description

Split between two or more different variables

Usage

```
MultiVarSplit(
  vars,
  split_label = "",
  varlabels = NULL,
  varnames = NULL,
  cfun = NULL,
  cformat = NULL,
  cna_str = NA_character_,
  split_format = NULL,
  split_na_str = NA_character_,
  split_name = "multivars",
  child_labels = c("default", "visible", "hidden"),
  extra_args = list(),
  indent_mod = 0L,
  cindent_mod = 0L,
  cvar = "",
  cextra_args = list(),
  label_pos = "visible",
  split_fun = NULL,
  page_prefix = NA_character_,
  section_div = NA_character_
)
```

Arguments

`vars`	character vector. Multiple variable names.
`split_label`	string. Label string to be associated with the table generated by the split. Not to be confused with labels assigned to each child (which are based on the data and type of split during tabulation).
`varlabels`	character vector. Labels for `vars`
`varnames`	character vector. Names for `vars` which will appear in pathing. When `vars` are all unique this will be the variable names. If not, these will be variable names with suffixes as necessary to enforce uniqueness.

cfun	list/function/NULL. tabulation function(s) for creating content rows. Must accept x or df as first parameter. Must accept labelstr as the second argument. Can optionally accept all optional arguments accepted by analysis functions. See analyze.
cformat	format spec. Format for content rows
cna_str	character. NA string for use with cformat for content table.
split_format	FormatSpec. Default format associated with the split being created.
split_na_str	character. NA string vector for use with split_format.
split_name	string. Name associated with this split (for pathing, etc)
child_labels	string. One of "default", "visible", "hidden". What should the display behavior be for the labels (i.e. label rows) of the children of this split. Defaults to "default" which flags the label row as visible only if the child has 0 content rows.
extra_args	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
cindent_mod	numeric(1). The indent modifier for the content tables generated by this split.
cvar	character(1). The variable, if any, which the content function should accept. Defaults to NA.
cextra_args	list. Extra arguments to be passed to the content function when tabulating row group summaries.
label_pos	character(1). Location the variable label should be displayed, Accepts "hidden" (default for non-analyze row splits), "visible", "topleft", and - for analyze splits only - "default". For analyze calls, "default" indicates that the variable should be visible if and only if multiple variables are analyzed at the same level of nesting.
split_fun	function/NULL. custom splitting function See custom_split_funs
page_prefix	character(1). Prefix, to be appended with the split value, when forcing pagination between the children of this split/table
section_div	character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.

Value

A MultiVarSplit object.

Author(s)

Gabriel Becker

names,VTableNodeInfo-method *Names of a* TableTree

Description

Names of a TableTree

Usage

```
## S4 method for signature 'VTableNodeInfo'
names(x)

## S4 method for signature 'InstantiatedColumnInfo'
names(x)

## S4 method for signature 'LayoutColTree'
names(x)

## S4 method for signature 'VTableTree'
row.names(x)
```

Arguments

x	the object.

Details

For TableTrees with more than one level of splitting in columns, the names are defined to be the top-level split values repped out across the columns that they span.

Value

The column names of x, as defined in the details above.

no_colinfo *Exported for use in tern*

Description

Does the table/row/InstantiatedColumnInfo object contain no column structure information?

Usage

```
no_colinfo(obj)

## S4 method for signature 'VTableNodeInfo'
no_colinfo(obj)

## S4 method for signature 'InstantiatedColumnInfo'
no_colinfo(obj)
```

Arguments

obj	ANY. The object for the accessor to access or modify

Value

TRUE if the object has no/empty instantiated column information, FALSE otherwise.

---

nrow,VTableTree-method
Table Dimensions

Description

Table Dimensions

Usage

```
## S4 method for signature 'VTableTree'
nrow(x)

## S4 method for signature 'VTableNodeInfo'
ncol(x)

## S4 method for signature 'VTableNodeInfo'
dim(x)
```

Arguments

x	TableTree or ElementaryTable object

Value

the number of rows (nrow), columns (ncol) or both (dim) of the object.

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  analyze(c("SEX", "AGE"))

tbl <- build_table(lyt, ex_adsl)

dim(tbl)
nrow(tbl)
ncol(tbl)

NROW(tbl)
NCOL(tbl)
```

---

obj_avar *Row attribute accessors*

Description

Row attribute accessors

Usage

```
obj_avar(obj)

## S4 method for signature 'TableRow'
obj_avar(obj)

## S4 method for signature 'ElementaryTable'
obj_avar(obj)

row_cells(obj)

## S4 method for signature 'TableRow'
row_cells(obj)

row_cells(obj) <- value

## S4 replacement method for signature 'TableRow'
row_cells(obj) <- value

row_values(obj)

## S4 method for signature 'TableRow'
row_values(obj)

row_values(obj) <- value
```

```
## S4 replacement method for signature 'TableRow'
row_values(obj) <- value

## S4 replacement method for signature 'LabelRow'
row_values(obj) <- value
```

Arguments

obj	ANY. The object for the accessor to access or modify
value	The new value

Value

various, depending on the accessor called.

`obj_name,VNodeInfo-method`

Methods for generics in the `formatters` *package*

Description

See the `formatters` documentation for descriptions of these generics.

Usage

```
## S4 method for signature 'VNodeInfo'
obj_name(obj)

## S4 method for signature 'Split'
obj_name(obj)

## S4 replacement method for signature 'VNodeInfo'
obj_name(obj) <- value

## S4 replacement method for signature 'Split'
obj_name(obj) <- value

## S4 method for signature 'Split'
obj_label(obj)

## S4 method for signature 'TableRow'
obj_label(obj)

## S4 method for signature 'VTableTree'
obj_label(obj)
```

```
## S4 method for signature 'ValueWrapper'
obj_label(obj)

## S4 replacement method for signature 'Split'
obj_label(obj) <- value

## S4 replacement method for signature 'TableRow'
obj_label(obj) <- value

## S4 replacement method for signature 'ValueWrapper'
obj_label(obj) <- value

## S4 replacement method for signature 'VTableTree'
obj_label(obj) <- value

## S4 method for signature 'VTableNodeInfo'
obj_format(obj)

## S4 method for signature 'CellValue'
obj_format(obj)

## S4 method for signature 'Split'
obj_format(obj)

## S4 replacement method for signature 'VTableNodeInfo'
obj_format(obj) <- value

## S4 replacement method for signature 'Split'
obj_format(obj) <- value

## S4 replacement method for signature 'CellValue'
obj_format(obj) <- value

## S4 method for signature 'Split'
obj_na_str(obj)

## S4 method for signature 'VTitleFooter'
main_title(obj)

## S4 replacement method for signature 'VTitleFooter'
main_title(obj) <- value

## S4 method for signature 'TableRow'
main_title(obj)

## S4 method for signature 'VTitleFooter'
subtitles(obj)
```

```
## S4 replacement method for signature 'VTitleFooter'
subtitles(obj) <- value

## S4 method for signature 'TableRow'
subtitles(obj)

## S4 method for signature 'VTitleFooter'
main_footer(obj)

## S4 replacement method for signature 'VTitleFooter'
main_footer(obj) <- value

## S4 method for signature 'TableRow'
main_footer(obj)

## S4 method for signature 'VTitleFooter'
prov_footer(obj)

## S4 replacement method for signature 'VTitleFooter'
prov_footer(obj) <- value

## S4 method for signature 'TableRow'
prov_footer(obj)

## S4 method for signature 'VTableNodeInfo'
table_inset(obj)

## S4 method for signature 'PreDataTableLayouts'
table_inset(obj)

## S4 replacement method for signature 'VTableNodeInfo'
table_inset(obj) <- value

## S4 replacement method for signature 'PreDataTableLayouts'
table_inset(obj) <- value

## S4 replacement method for signature 'InstantiatedColumnInfo'
table_inset(obj) <- value

## S4 method for signature 'TableRow'
nlines(x, colwidths = NULL, max_width = NULL)

## S4 method for signature 'LabelRow'
nlines(x, colwidths = NULL, max_width = NULL)

## S4 method for signature 'RefFootnote'
nlines(x, colwidths = NULL, max_width = NULL)
```

```
## S4 method for signature 'InstantiatedColumnInfo'
nlines(x, colwidths = NULL, max_width = NULL)

## S4 method for signature 'VTableTree'
make_row_df(
  tt,
  colwidths = NULL,
  visible_only = TRUE,
  rownum = 0,
  indent = 0L,
  path = character(),
  incontent = FALSE,
  repr_ext = 0L,
  repr_inds = integer(),
  sibpos = NA_integer_,
  nsibs = NA_integer_,
  max_width = NULL
)

## S4 method for signature 'TableRow'
make_row_df(
  tt,
  colwidths = NULL,
  visible_only = TRUE,
  rownum = 0,
  indent = 0L,
  path = "root",
  incontent = FALSE,
  repr_ext = 0L,
  repr_inds = integer(),
  sibpos = NA_integer_,
  nsibs = NA_integer_,
  max_width = NULL
)

## S4 method for signature 'LabelRow'
make_row_df(
  tt,
  colwidths = NULL,
  visible_only = TRUE,
  rownum = 0,
  indent = 0L,
  path = "root",
  incontent = FALSE,
  repr_ext = 0L,
  repr_inds = integer(),
  sibpos = NA_integer_,
  nsibs = NA_integer_,
```

```
  max_width = NULL
)
```

Arguments

obj	ANY. The object for the accessor to access or modify
value	The new value
x	An object
colwidths	numeric vector. Column widths for use with vertical pagination.
max_width	numeric(1). Width strings should be wrapped to when determining how many lines they require.
tt	TableTree (or related class). A TableTree object representing a populated table.
visible_only	logical(1). Should only visible aspects of the table structure be reflected in this summary. Defaults to TRUE. May not be supported by all methods.
rownum	numeric(1). Internal detail do not set manually.
indent	integer(1). Internal detail do not set manually.
path	character. A vector path for a position within the structure of a tabletree. Each element represents a subsequent choice amongst the children of the previous choice.
incontent	logical(1). Internal detail do not set manually.
repr_ext	integer(1). Internal detail do not set manually.
repr_inds	integer. Internal detail do not set manually.
sibpos	integer(1). Internal detail do not set manually.
nsibs	integer(1). Internal detail do not set manually.

Details

When visible_only is TRUE (the default), methods should return a data.frame with exactly one row per visible row in the table-like object. This is useful when reasoning about how a table will print, but does not reflect the full pathing space of the structure (though the paths which are given will all work as is).

If supported, when visible_only is FALSE, every structural element of the table (in row-space) will be reflected in the returned data.frame, meaning the full pathing-space will be represented but some rows in the layout summary will not represent printed rows in the table as it is displayed.

Most arguments beyond tt and visible_only are present so that make_row_df methods can call make_row_df recursively and retain information, and should not be set during a top-level call

Value

for getters, the current value of the component being accessed on obj, for setters, a modified copy of obj with the new value.

Note

the technically present root tree node is excluded from the summary returned by both `make_row_df` and `make_col_df` (see `rtables::make_col_df`), as it is simply the row/column structure of `tt` and thus not useful for pathing or pagination.

`pag_tt_indices` *Pagination of a* `TableTree`

Description

Paginate an `rtables` table in the vertical and/or horizontal direction, as required for the specified page size.

Usage

```
pag_tt_indices(
  tt,
  lpp = 15,
  min_siblings = 2,
  nosplitin = character(),
  colwidths = NULL,
  max_width = NULL,
  verbose = FALSE
)

paginate_table(
  tt,
  page_type = "letter",
  font_family = "Courier",
  font_size = 8,
  lineheight = 1,
  landscape = FALSE,
  pg_width = NULL,
  pg_height = NULL,
  margins = c(top = 0.5, bottom = 0.5, left = 0.75, right = 0.75),
  lpp = NA_integer_,
  cpp = NA_integer_,
  min_siblings = 2,
  nosplitin = character(),
  colwidths = NULL,
  tf_wrap = FALSE,
  max_width = NULL,
  verbose = FALSE
)
```

Arguments

tt	TableTree (or related class). A TableTree object representing a populated table.
lpp	numeric. Maximum lines per page including (re)printed header and context rows
min_siblings	numeric. Minimum sibling rows which must appear on either side of pagination row for a mid-subtable split to be valid. Defaults to 2.
nosplitin	character. List of names of sub-tables where page-breaks are not allowed, regardless of other considerations. Defaults to none.
colwidths	numeric vector. Column widths for use with vertical pagination.
max_width	integer(1), character(1) or NULL. Width that title and footer (including footnotes) materials should be word-wrapped to. If NULL, it is set to the current print width of the session (getOption("width")). If set to "auto", the width of the table (plus any table inset) is used. Ignored completely if tf_wrap is FALSE.
verbose	logical(1). Should extra debugging messages be shown. Defaults to FALSE.
page_type	character(1). Name of a page type. See page_types. Ignored when pg_width and pg_height are set directly.
font_family	character(1). Name of a font family. An error will be thrown if the family named is not monospaced. Defaults to Courier.
font_size	numeric(1). Font size, defaults to 12.
lineheight	numeric(1). Line height, defaults to 1.
landscape	logical(1). Should the dimensions of page_type be inverted for landscape? Defaults to FALSE, ignored when pg_width and pg_height are set directly.
pg_width	numeric(1). Page width in inches.
pg_height	numeric(1). Page height in inches.
margins	numeric(4). Named numeric vector containing 'bottom', 'left', 'top', and 'right' margins in inches. Defaults to .5 inches for both vertical margins and .75 for both horizontal margins.
cpp	numeric(1) or NULL. Width (in characters) of the pages for horizontal pagination. NA (the default) indicates cpp should be inferred from the page size; NULL indicates no horizontal pagination should be done regardless of page size.
tf_wrap	logical(1). Should the texts for title, subtitle, and footnotes be wrapped?

Details

rtables pagination is context aware, meaning that label rows and row-group summaries (content rows) are repeated after (vertical) pagination, as appropriate. This allows the reader to immediately understand where they are in the table after turning to a new page, but does also mean that a rendered, paginated table will take up more lines of text than rendering the table without pagination would.

Pagination also takes into account word-wrapping of title, footer, column-label, and formatted cell value content.

Vertical pagination information (pagination data.frame) is created using (make_row_df)

Horizontal pagination is performed by creating a pagination dataframe for the columns, and then applying the same algorithm used for vertical pagination to it.

If physical page size and font information are specified, these are used to derive lines-per-page (lpp) and characters-per-page (cpp) values.

The full multi-direction pagination algorithm then is as follows:

1. Adjust lpp and cpp to account for rendered elements that are not rows (columns)

- titles/footers/column labels, and horizontal dividers in the vertical pagination case
- row-labels, table_inset, and top-left materials in the horizontal case

1. Perform 'forced pagination' representing page-by row splits, generating 1 or more tables
2. Perform vertical pagination separately on each table generated in (1)
3. Perform horizontal pagination **on the entire table** and apply the results to each table page generated in (1)-(2)
4. Return a list of subtables representing full bi-directional pagination

Pagination in both directions is done using the *Core Pagination Algorithm* implemented in the formatters package:

Value

for pag_tt_indices a list of paginated-groups of row-indices of tt. For paginate_table, The subtables defined by subsetting by the indices defined by pag_tt_indices.

Pagination Algorithm

Pagination is performed independently in the vertical and horizontal directions based solely on a *pagination data.frame*, which includes the following information for each row/column:

- number of lines/characters rendering the row will take **after word-wrapping** (self_extent)
- the indices (reprint_inds) and number of lines (par_extent) of the rows which act as **context** for the row
- the row's number of siblings and position within its siblings

Given lpp (cpp) already adjusted for rendered elements which are not rows/columns and a dataframe of pagination information, pagination is performed via the following algorithm, and with a start = 1:

Core Pagination Algorithm:

1. Initial guess for pagination point is start + lpp (start + cpp)
2. While the guess is not a valid pagination position, and guess > start, decrement guess and repeat

- an error is thrown if all possible pagination positions between start and start + lpp (start + cpp) would ever be < start after decrementing

1. Retain pagination index

2. if pagination point was less than `NROW(tt)` (`ncol(tt)`), set `start` to `pos + 1`, and repeat steps (1) - (4).

Validating pagination position:

Given an (already adjusted) `lpp` or `cpp` value, a pagination is invalid if:

- The rows/columns on the page would take more than (adjusted) `lpp` lines/`cpp` characters to render **including**
  - word-wrapping
  - (vertical only) context repetition
- (vertical only) footnote messages and or section divider lines take up too many lines after rendering rows
- (vertical only) row is a label or content (row-group summary) row
- (vertical only) row at the pagination point has siblings, and it has less than `min_siblings` preceding or following siblings
- pagination would occur within a sub-table listed in `nosplitin`

Examples

```
s_summary <- function(x) {
  if (is.numeric(x)) {
    in_rows(
      "n" = rcell(sum(!is.na(x)), format = "xx"),
      "Mean (sd)" = rcell(c(mean(x, na.rm = TRUE), sd(x, na.rm = TRUE)),
        format = "xx.xx (xx.xx)"
      ),
      "IQR" = rcell(IQR(x, na.rm = TRUE), format = "xx.xx"),
      "min - max" = rcell(range(x, na.rm = TRUE), format = "xx.xx - xx.xx")
    )
  } else if (is.factor(x)) {
    vs <- as.list(table(x))
    do.call(in_rows, lapply(vs, rcell, format = "xx"))
  } else {
    (
      stop("type not supported")
    )
  }
}


lyt <- basic_table() %>%
  split_cols_by(var = "ARM") %>%
  analyze(c("AGE", "SEX", "BEP01FL", "BMRKR1", "BMRKR2", "COUNTRY"), afun = s_summary)

tbl <- build_table(lyt, ex_adsl)
tbl

nrow(tbl)
```

```
row_paths_summary(tbl)

tbls <- paginate_table(tbl, lpp = 15)
mf <- matrix_form(tbl, indent_rownames = TRUE)
w_tbls <- propose_column_widths(mf) # so that we have the same column widths


tmp <- lapply(tbls, function(tbli) {
  cat(toString(tbli, widths = w_tbls))
  cat("\n\n")
  cat("~~~~ PAGE BREAK ~~~~")
  cat("\n\n")
})
```

---

prune_table *Recursively prune a* TableTree

Description

Recursively prune a TableTree

Usage

```
prune_table(
  tt,
  prune_func = prune_empty_level,
  stop_depth = NA_real_,
  depth = 0
)
```

Arguments

tt	TableTree (or related class). A TableTree object representing a populated table.
prune_func	function. A Function to be called on each subtree which returns TRUE if the entire subtree should be removed.
stop_depth	numeric(1). The depth after which subtrees should not be checked for pruning. Defaults to NA which indicates pruning should happen at all levels
depth	numeric(1). Used internally, not intended to be set by the end user.

Value

A TableTree pruned via recursive application of prune_func.

See Also

prune_empty_level() for details on this and several other basic pruning functions included in the rtables package.

Examples

```
adsl <- ex_adsl
levels(adsl$SEX) <- c(levels(ex_adsl$SEX), "OTHER")

tbl_to_prune <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("SEX") %>%
  summarize_row_groups() %>%
  split_rows_by("STRATA1") %>%
  summarize_row_groups() %>%
  analyze("AGE") %>%
  build_table(adsl)

tbl_to_prune %>% prune_table()
```

---

qtable_layout *Generalized Frequency Table*

Description

This function provides a convenience interface for generating generalizations of a 2-way frequency table. Row and column space can be facetted by variables, and an analysis function can be specified.

The function then builds a layout with the specified layout and applies it to the data provided.

Usage

```
qtable_layout(
  data,
  row_vars = character(),
  col_vars = character(),
  avar = NULL,
  row_labels = NULL,
  afun = NULL,
  summarize_groups = FALSE,
  title = "",
  subtitles = character(),
  main_footer = character(),
  prov_footer = character(),
  show_colcounts = TRUE,
  drop_levels = TRUE,
  ...,
  .default_rlabel = NULL
)

qtable(
  data,
```

```
  row_vars = character(),
  col_vars = character(),
  avar = NULL,
  row_labels = NULL,
  afun = NULL,
  summarize_groups = FALSE,
  title = "",
  subtitles = character(),
  main_footer = character(),
  prov_footer = character(),
  show_colcounts = TRUE,
  drop_levels = TRUE,
  ...
)
```

Arguments

data	data.frame. The data to tabulate.
row_vars	character. The names of variables to be used in row facetting.
col_vars	character. The names of variables to be used in column facetting.
avar	character(1). The variable to be analyzed. Defaults to the first variable in data.
row_labels	character or NULL. Row label(s) which should be applied to the analysis rows. length must match the number of rows generated by afun. See details.
afun	function. The function to generate the analysis row cell values. This can be a proper analysis function, or a function which returns a vector or list. Vectors are taken as multi-valued single cells, whereas lists are interpreted as multiple cells.
summarize_groups	logical(1). Should each level of nesting include marginal summary rows. Defaults to FALSE
title	character(1). Main title (main_title()) is a single string. Ignored for subtables.
subtitles	character. Subtitles (subtitles()) can be vector of strings, where every element is printed in a separate line. Ignored for subtables.
main_footer	character. Main global (non-referential) footer materials (main_footer()). If it is a vector of strings, they will be printed on separate lines.
prov_footer	character. Provenance-related global footer materials (prov_footer()). It can be also a vector of strings, printed on different lines. Generally should not be modified by hand.
show_colcounts	logical(1). Should column counts be displayed in the resulting table when this layout is applied to data
drop_levels	logical(1). Should unobserved factor levels be dropped during facetting. Defaults to TRUE.
...	passed to afun, if specified. Otherwise ignored.
.default_rlabel	character(1). This is an implementation detail that should not be set by end users.

Details

This function creates a table with a single top-level structure in both row and column dimensions involving faceting by 0 or more variables in each.

The display of the table depends on certain details of the tabulation. In the case of an `afun` which returns a single cell's contents (either a scalar or a vector of 2 or 3 elements), the label rows for the deepest-nested row facets will be hidden and the labels used there will be used as the analysis row labels. In the case of an `afun` which returns a list (corresponding to multiple cells), the names of the list will be used as the analysis row labels and the deepest-nested facet row labels will be visible.

The table will be annotated in the top-left area with an informative label displaying the analysis variable (`avar`), if set, and the function used (captured via substitute) where possible, or 'count' if not. One exception where the user may directly modify the top-left area (via `row_labels`) is the case of a table with row facets and an `afun` which returns a single row.

Value

for `qtable` a built TableTree object representing the desired table, for `qtable_layout`, a `PreDataTableLayouts` object declaring the structure of the desired table, suitable for passing to `build_table`.

Examples

```
qtable(ex_adsl)
qtable(ex_adsl, row_vars = "ARM")
qtable(ex_adsl, col_vars = "ARM")
qtable(ex_adsl, row_vars = "SEX", col_vars = "ARM")
qtable(ex_adsl, row_vars = c("COUNTRY", "SEX"), col_vars = c("ARM", "STRATA1"))
qtable(ex_adsl,
  row_vars = c("COUNTRY", "SEX"),
  col_vars = c("ARM", "STRATA1"), avar = "AGE", afun = mean
)
summary_list <- function(x, ...) as.list(summary(x))
qtable(ex_adsl, row_vars = "SEX", col_vars = "ARM", avar = "AGE", afun = summary_list)
suppressWarnings(qtable(ex_adsl,
  row_vars = "SEX",
  col_vars = "ARM", avar = "AGE", afun = range
))
```

---

rbindl_rtables `rbind TableTree` *and related objects*

Description

`rbind TableTree` and related objects

Usage

```
rbindl_rtables(x, gap = 0, check_headers = TRUE)

## S4 method for signature 'VTableNodeInfo'
rbind(..., deparse.level = 1)

## S4 method for signature 'VTableNodeInfo,ANY'
rbind2(x, y)
```

Arguments

x	VTableNodeInfo. TableTree, ElementaryTable or TableRow object.
gap	deprecated. Ignored.
check_headers	deprecated. Ignored.
...	ANY. Elements to be stacked.
deparse.level	numeric(1). Currently Ignored.
y	VTableNodeInfo. TableTree, ElementaryTable or TableRow object.

Value

A formal table object.

Note

When objects are rbinded, titles and footer information is retained from the first object (if any exists) if all other objects have no titles/footers or have identical titles/footers. Otherwise, all titles/footers are removed and must be set for the bound table via the main_title(), subtitles(), main_footer(), and prov_footer() functions.

Examples

```
mtbl <- rtable(
  header = rheader(
   rrow(row.name = NULL, rcell("Sepal.Length", colspan = 2), rcell("Petal.Length", colspan = 2)),
    rrow(NULL, "mean", "median", "mean", "median")
  ),
  rrow(
    row.name = "All Species",
    mean(iris$Sepal.Length), median(iris$Sepal.Length),
    mean(iris$Petal.Length), median(iris$Petal.Length),
    format = "xx.xx"
  )
)

mtbl2 <- with(subset(iris, Species == "setosa"), rtable(
  header = rheader(
   rrow(row.name = NULL, rcell("Sepal.Length", colspan = 2), rcell("Petal.Length", colspan = 2)),
    rrow(NULL, "mean", "median", "mean", "median")
  ),
```

```
  rrow(
    row.name = "Setosa",
    mean(Sepal.Length), median(Sepal.Length),
    mean(Petal.Length), median(Petal.Length),
    format = "xx.xx"
  )
))

rbind(mtbl, mtbl2)
rbind(mtbl, rrow(), mtbl2)
rbind(mtbl, rrow("aaa"), indent(mtbl2))
```

`rcell` *Cell value constructors*

Description

Construct a cell value and associate formatting, labeling, indenting, and column spanning information with it.

Usage

```
rcell(
  x,
  format = NULL,
  colspan = 1L,
  label = NULL,
  indent_mod = NULL,
  footnotes = NULL,
  align = NULL,
  format_na_str = NULL
)

non_ref_rcell(
  x,
  is_ref,
  format = NULL,
  colspan = 1L,
  label = NULL,
  indent_mod = NULL,
  refval = NULL,
  align = "center",
  format_na_str = NULL
)
```

Arguments

x	ANY. Cell value.

format	character(1) or function. The format label (string) or formatters function to apply to x. See formatters::list_valid_format_labels() for currently supported format labels.
colspan	integer(1). Column span value.
label	character(1). Label or NULL. If non-null, it will be looked at when determining row labels.
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
footnotes	list or NULL. Referential footnote messages for the cell.
align	character(1) or NULL. Alignment the value should be rendered with. It defaults to "center" if NULL is used. See formatters::list_valid_aligns() for currently supported alignments.
format_na_str	character(1). String which should be displayed when formatted if this cell's value(s) are all NA.
is_ref	logical(1). Are we in the reference column (i.e. .in_ref_col should be passed to this argument)
refval	ANY. Value to use when in the reference column. Defaults to NULL

Details

non_ref_rcell provides the common *blank for cells in the reference column, this value otherwise*, and should be passed the value of .in_ref_col when it is used.

Value

An object representing the value within a single cell within a populated table. The underlying structure of this object is an implementation detail and should not be relied upon beyond calling accessors for the class.

Note

Currently column spanning is only supported for defining header structure.

---

rheader *Create a header*

Description

Create a header

Usage

```
rheader(..., format = "xx", .lst = NULL)
```

Arguments

...	row specifications (either as character vectors or the output from rrow or DataRow, LabelRow, etc.
format	character(1) or function. The format label (string) or formatter function to apply to the cell values passed via .... See list_valid_format_labels for currently supported format labels.
.lst	list. An already-collected list of arguments to be used instead of the elements of .... Arguments passed via ... will be ignored if this is specified.

Value

a InstantiatedColumnInfo object.

See Also

Other compatibility: rrowl(), rrow(), rtable()

Examples

```
h1 <- rheader(c("A", "B", "C"))

h2 <- rheader(
  rrow(NULL, rcell("group 1", colspan = 2), rcell("group 2", colspan = 2)),
  rrow(NULL, "A", "B", "A", "B")
)

h1

h2
```

---

row_footnotes *Referential Footnote Accessors*

Description

Get and set referential footnotes on aspects of a built table

Usage

```
row_footnotes(obj)

row_footnotes(obj) <- value

cell_footnotes(obj)
```

```
cell_footnotes(obj) <- value

col_fnotes_here(obj)

## S4 method for signature 'ANY'
col_fnotes_here(obj)

col_fnotes_here(obj) <- value

col_footnotes(obj)

col_footnotes(obj) <- value

ref_index(obj)

ref_index(obj) <- value

ref_symbol(obj)

ref_symbol(obj) <- value

ref_msg(obj)

fnotes_at_path(obj, rowpath = NULL, colpath = NULL, reset_idx = TRUE) <- value
```

Arguments

obj	ANY. The object for the accessor to access or modify
value	The new value
rowpath	character or NULL. Path within row structure. NULL indicates the footnote should go on the column rather than cell.
colpath	character or NULL. Path within column structure. NULL indicates footnote should go on the row rather than cell
reset_idx	logical(1). Should the numbering for referential footnotes be immediately recalculated. Defaults to TRUE.

See Also

row_paths(), col_paths(), row_paths_summary(), col_paths_summary()

Examples

```
# How to add referencial footnotes after having created a table
lyt <- basic_table() %>%
  split_rows_by("SEX", page_by = TRUE) %>%
  analyze("AGE")

tbl <- build_table(lyt, DM)
tbl <- trim_rows(tbl)
```

```
# Check the row and col structure to add precise references
# row_paths(tbl)
# col_paths(t)
# row_paths_summary(tbl)
# col_paths_summary(tbl)

# Add the citation numbers on the table and relative references in the footnotes
fnotes_at_path(tbl, rowpath = c("SEX", "F", "AGE", "Mean")) <- "Famous paper 1"
fnotes_at_path(tbl, rowpath = c("SEX", "UNDIFFERENTIATED")) <- "Unfamous paper 2"
# tbl
```

`row_paths` *Return List with Table Row/Col Paths*

Description

Return List with Table Row/Col Paths

Usage

```
row_paths(x)

col_paths(x)
```

Arguments

`x`	an `rtable` object

Value

a list of paths to each row/column within `x`

See Also

`cell_values()`, `fnotes_at_path<-`, `row_paths_summary()`, `col_paths_summary()`

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  analyze(c("SEX", "AGE"))

tbl <- build_table(lyt, ex_adsl)
tbl

row_paths(tbl)
col_paths(tbl)

cell_values(tbl, c("AGE", "Mean"), c("ARM", "B: Placebo"))
```

`row_paths_summary` *Print Row/Col Paths Summary*

Description

Print Row/Col Paths Summary

Usage

```
row_paths_summary(x)

col_paths_summary(x)
```

Arguments

`x`	an `rtable` object

Value

A data.frame summarizing the row- or column-structure of `x`.

Examples

```
library(dplyr)

ex_adsl_MF <- ex_adsl %>% filter(SEX %in% c("M", "F"))

lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by("SEX", split_fun = drop_split_levels) %>%
  analyze(c("AGE", "BMRKR2"))

tbl <- build_table(lyt, ex_adsl_MF)
tbl

df <- row_paths_summary(tbl)
df

col_paths_summary(tbl)

# manually constructed table
tbl2 <- rtable(
  rheader(
    rrow(
      "row 1", rcell("a", colspan = 2),
      rcell("b", colspan = 2)
    ),
    rrow("h2", "a", "b", "c", "d")
  ),
```

```
  rrow("r1", 1, 2, 1, 2), rrow("r2", 3, 4, 2, 1)
)
col_paths_summary(tbl2)
```

`rrow` *row*

Description

row

Usage

```
rrow(row.name = "", ..., format = NULL, indent = 0, inset = 0L)
```

Arguments

`row.name`	if `NULL` then an empty string is used as `row.name` of the `rrow`.
`...`	cell values
`format`	character(1) or function. The format label (string) or formatter function to apply to the cell values passed via `....` See `list_valid_format_labels` for currently supported format labels.
`indent`	deprecated.
`inset`	integer(1). The table inset for the row or table being constructed. See `table_inset`.

Value

A row object of the context-appropriate type (label or data)

See Also

Other compatibility: `rheader()`, `rrowl()`, `rtable()`

Examples

```
rrow("ABC", c(1, 2), c(3, 2), format = "xx (xx.%)")
rrow("")
```

rrowl rrowl

Description

rrowl

Usage

```
rrowl(row.name, ..., format = NULL, indent = 0, inset = 0L)
```

Arguments

row.name	if NULL then an empty string is used as row.name of the rrow.
...	values in vector/list form
format	character(1) or function. The format label (string) or formatter function to apply to the cell values passed via .... See list_valid_format_labels for currently supported format labels.
indent	deprecated.
inset	integer(1). The table inset for the row or table being constructed. See table_inset.

Value

A row object of the context-appropriate type (label or data)

See Also

Other compatibility: rheader(), rrow(), rtable()

Examples

```
rrowl("a", c(1, 2, 3), format = "xx")
rrowl("a", c(1, 2, 3), c(4, 5, 6), format = "xx")


rrowl("N", table(iris$Species))
rrowl("N", table(iris$Species), format = "xx")

x <- tapply(iris$Sepal.Length, iris$Species, mean, simplify = FALSE)

rrow(row.name = "row 1", x)
rrow("ABC", 2, 3)

rrowl(row.name = "row 1", c(1, 2), c(3, 4))
rrow(row.name = "row 2", c(1, 2), c(3, 4))
```

rtable *Create a Table*

Description

Create a Table

Usage

```
rtable(header, ..., format = NULL, hsep = default_hsep(), inset = 0L)

rtablel(header, ..., format = NULL, hsep = default_hsep(), inset = 0L)
```

Arguments

header	Information defining the header (column structure) of the table. This can be as row objects (legacy), character vectors or a InstantiatedColumnInfo object.
...	Rows to place in the table.
format	character(1) or function. The format label (string) or formatter function to apply to the cell values passed via .... See list_valid_format_labels for currently supported format labels.
hsep	character(1). Set of character(s) to be repeated as the separator between the header and body of the table when rendered as text. Defaults to a connected horizontal line (unicode 2014) in locals that use a UTF charset, and to - elsewhere (with a once per session warning). See formatters::set_default_hsep() for further information.
inset	integer(1). The table inset for the row or table being constructed. See table_inset.

Value

a formal table object of the appropriate type (ElementaryTable or TableTree)

See Also

Other compatibility: rheader(), rrowl(), rrow()

Examples

```
rtable(
  header = LETTERS[1:3],
  rrow("one to three", 1, 2, 3),
  rrow("more stuff", rcell(pi, format = "xx.xx"), "test", "and more")
)


# Table with multirow header
```

```
sel <- iris$Species == "setosa"
mtbl <- rtable(
  header = rheader(
    rrow(
      row.name = NULL, rcell("Sepal.Length", colspan = 2),
      rcell("Petal.Length", colspan = 2)
    ),
    rrow(NULL, "mean", "median", "mean", "median")
  ),
  rrow(
    row.name = "All Species",
    mean(iris$Sepal.Length), median(iris$Sepal.Length),
    mean(iris$Petal.Length), median(iris$Petal.Length),
    format = "xx.xx"
  ),
  rrow(
    row.name = "Setosa",
    mean(iris$Sepal.Length[sel]), median(iris$Sepal.Length[sel]),
    mean(iris$Petal.Length[sel]), median(iris$Petal.Length[sel])
  )
)

mtbl

names(mtbl) # always first row of header

# Single row header

tbl <- rtable(
  header = c("Treatement\nN=100", "Comparison\nN=300"),
  format = "xx (xx.xx%)",
  rrow("A", c(104, .2), c(100, .4)),
  rrow("B", c(23, .4), c(43, .5)),
  rrow(""),
  rrow("this is a very long section header"),
  rrow("estimate", rcell(55.23, "xx.xx", colspan = 2)),
  rrow("95% CI", indent = 1, rcell(c(44.8, 67.4), format = "(xx.x, xx.x)", colspan = 2))
)
tbl

row.names(tbl)
names(tbl)


# Subsetting
tbl[1, ]
tbl[, 1]

tbl[1, 2]
tbl[2, 1]

tbl[3, 2]
tbl[5, 1]
```

```
tbl[5, 2]

# # Data Structure methods
dim(tbl)
nrow(tbl)
ncol(tbl)
names(tbl)


# Colspans

tbl2 <- rtable(
  c("A", "B", "C", "D", "E"),
  format = "xx",
  rrow("r1", 1, 2, 3, 4, 5),
  rrow("r2", rcell("sp2", colspan = 2), "sp1", rcell("sp2-2", colspan = 2))
)

tbl2
```

`sanitize_table_struct` *Sanitize degenerate table structures (Experimental)*

Description

Experimental function to correct structure of degenerate tables by adding messaging rows to empty sub-structures.

Usage

```
sanitize_table_struct(tt, empty_msg = "-- This Section Contains No Data --")
```

Arguments

`tt`	`TableTree`
`empty_msg`	character(1). The string which should be spanned across the inserted empty rows.

Details

This function locates degenerate portions of the table (including the table overall in the case of a table with no data rows) and inserts a row which spans all columns with the message `empty_msg` at each one, generating a table guaranteed to be non-degenerate.

Value

If `tt` is already valid, it is returned unmodified. If `tt` is degenerate, a modified, non-degenerate version of the table is returned.

Examples

```
sanitize_table_struct(rtable("cool beans"))

lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("SEX") %>%
  summarize_row_groups()

## Degenerate because it doesn't have any analyze calls -> no data rows
badtab <- build_table(lyt, DM)
sanitize_table_struct(badtab)
```

---

section_div *Section dividers getter and setter*

Description

`section_div` can be used to set or get the section divider for a table object produced by `build_table()`. When assigned in post-processing (`section_div<-`) the table can have a section divider after every row, each assigned independently. If assigning during layout creation, only `split_rows_by()` (and its related row-wise splits) and `analyze()` have a `section_div` parameter that will produce separators between split sections and data subgroups, respectively.

Usage

```
section_div(obj)

## S4 method for signature 'VTableTree'
section_div(obj)

## S4 method for signature 'list'
section_div(obj)

## S4 method for signature 'TableRow'
section_div(obj)

section_div(obj, only_sep_sections = FALSE) <- value

## S4 replacement method for signature 'VTableTree'
section_div(obj, only_sep_sections = FALSE) <- value

## S4 replacement method for signature 'list'
section_div(obj, only_sep_sections = FALSE) <- value

## S4 replacement method for signature 'TableRow'
section_div(obj, only_sep_sections = FALSE) <- value
```

```
## S4 replacement method for signature 'LabelRow'
section_div(obj, only_sep_sections = FALSE) <- value

header_section_div(obj)

## S4 method for signature 'PreDataTableLayouts'
header_section_div(obj)

## S4 method for signature 'VTableTree'
header_section_div(obj)

header_section_div(obj) <- value

## S4 replacement method for signature 'PreDataTableLayouts'
header_section_div(obj) <- value

## S4 replacement method for signature 'VTableTree'
header_section_div(obj) <- value
```

Arguments

obj	Table object. This can be of any class that inherits from VTableTree or TableRow/LabelRow.
only_sep_sections	logical(1). Defaults to FALSE for section_div<-. Allows you to set the section divider only for sections that are splits or analyses if the number of values is less than the number of rows in the table. If TRUE, the section divider will be set for all rows of the table.
value	character. Vector of single characters to use as section dividers. Each character is repeated such that all section dividers span the width of the table. Each character that is not NA_character_ will produce a trailing separator for each row of the table. value length should reflect the number of rows, or be between 1 and the number of splits/levels. See the Details section below for more information.

Details

Assigned value to section divider must be a character vector. If any value is NA_character_ the section divider will be absent for that row or section. When you want to only affect sections or splits, please use only_sep_sections or provide a shorter vector than the number of rows. Ideally, the length of the vector should be less than the number of splits with, eventually, the leaf-level, i.e. DataRow where analyze results are. Note that if only one value is inserted, only the first split will be affected. If only_sep_sections = TRUE, which is the default for section_div() produced from the table construction, the section divider will be set for all the splits and eventually analyses, but not for the header or each row of the table. This can be set with header_section_div in basic_table() or, eventually, with hsep in build_table(). If FALSE, the section divider will be set for all the rows of the table.

Value

The section divider string. Each line that does not have a trailing separator will have `NA_character_` as section divider.

See Also

`basic_table()` parameter `header_section_div` for a global section divider.

Examples

```
# Data
df <- data.frame(
  cat = c(
    "really long thing its so ", "long"
  ),
  value = c(6, 3, 10, 1)
)
fast_afun <- function(x) list("m" = rcell(mean(x), format = "xx."), "m/2" = max(x) / 2)

tbl <- basic_table() %>%
  split_rows_by("cat", section_div = "~") %>%
  analyze("value", afun = fast_afun, section_div = " ") %>%
  build_table(df)

# Getter
section_div(tbl)

# Setter
section_div(tbl) <- letters[seq_len(nrow(tbl))]
tbl

# last letter can appear if there is another table
rbind(tbl, tbl)

# header_section_div
header_section_div(tbl) <- "+"
tbl
```

`select_all_levels` *Add Combination Levels to split*

Description

Add Combination Levels to split

Usage

```
select_all_levels

add_combo_levels(combosdf, trim = FALSE, first = FALSE, keep_levels = NULL)
```

Arguments

combosdf	data.frame/tbl_df. Columns valname, label, levelcombo, exargs. Of which levelcombo and exargs are list columns. Passing the select_all_levels object as a value in the comblevels column indicates that an overall/all-observations level should be created.
trim	logical(1). Should splits corresponding with 0 observations be kept when tabulating.
first	logical(1). Should the created split level be placed first in the levels (TRUE) or last (FALSE, the default).
keep_levels	character or NULL. If non-NULL, the levels to retain across both combination and individual levels.

Format

An object of class AllLevelsSentinel of length 0.

Value

a closure suitable for use as a splitting function (splfun) when creating a table layout

Note

Analysis or summary functions for which the order matters should never be used within the tabulation framework.

Examples

```
library(tibble)
combodf <- tribble(
  ~valname, ~label, ~levelcombo, ~exargs,
  "A_B", "Arms A+B", c("A: Drug X", "B: Placebo"), list(),
  "A_C", "Arms A+C", c("A: Drug X", "C: Combination"), list()
)

lyt <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by("ARM", split_fun = add_combo_levels(combodf)) %>%
  analyze("AGE")

tbl <- build_table(lyt, DM)
tbl

lyt1 <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by("ARM",
    split_fun = add_combo_levels(combodf,
      keep_levels = c(
        "A_B",
        "A_C"
      )
    )
  ) %>%
```

```
  analyze("AGE")

tbl1 <- build_table(lyt1, DM)
tbl1

smallerDM <- droplevels(subset(DM, SEX %in% c("M", "F") &
  grepl("^(A|B)", ARM)))
lyt2 <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by("ARM", split_fun = add_combo_levels(combodf[1, ])) %>%
  split_cols_by("SEX",
    split_fun = add_overall_level("SEX_ALL", "All Genders")
  ) %>%
  analyze("AGE")

lyt3 <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by("ARM", split_fun = add_combo_levels(combodf)) %>%
  split_rows_by("SEX",
    split_fun = add_overall_level("SEX_ALL", "All Genders")
  ) %>%
  summarize_row_groups() %>%
  analyze("AGE")

tbl3 <- build_table(lyt3, smallerDM)
tbl3
```

`sf_args` *Split Function Arg Conventions*

Description

Split Function Arg Conventions

Usage

```
sf_args(trim, label, first)
```

Arguments

`trim`	logical(1). Should splits corresponding with 0 observations be kept when tabulating.
`label`	character(1). A label (not to be confused with the name) for the object/structure.
`first`	logical(1). Should the created split level be placed first in the levels (`TRUE`) or last (`FALSE`, the default).

Value

NULL (this is an argument template dummy function)

See Also

Other conventions: compat_args(), constr_args(), gen_args(), lyt_args()

---

simple_analysis *Default tabulation*

Description

This function is used when analyze is invoked

Usage

```
simple_analysis(x, ...)

## S4 method for signature 'numeric'
simple_analysis(x, ...)

## S4 method for signature 'logical'
simple_analysis(x, ...)

## S4 method for signature 'factor'
simple_analysis(x, ...)

## S4 method for signature 'ANY'
simple_analysis(x, ...)
```

Arguments

x	the *already split* data being tabulated for a particular cell/set of cells
...	passed on directly

Details

This function has the following behavior given particular types of inputs:

numeric calls mean on x

logical calls sum on x

factor calls length on x

in_rows is called on the resulting value(s).

All other classes of input currently lead to an error.

Value

an RowsVerticalSection object (or NULL). The details of this object should be considered an internal implementation detail.

Author(s)

Gabriel Becker and Adrian Waddell

Examples

```
simple_analysis(1:3)
simple_analysis(iris$Species)
simple_analysis(iris$Species == "setosa")
```

---

sort_at_path *Sorting a Table at a Specific Path*

Description

Main sorting function to order the substructure of a `TableTree` at a particular Path in the table tree.

Usage

```
sort_at_path(
  tt,
  path,
  scorefun,
  decreasing = NA,
  na.pos = c("omit", "last", "first"),
  .prev_path = character()
)
```

Arguments

`tt`	`TableTree` (or related class). A `TableTree` object representing a populated table.
`path`	character. A vector path for a position within the structure of a `tabletree`. Each element represents a subsequent choice amongst the children of the previous choice.
`scorefun`	function. Scoring function, should accept the type of children directly under the position at `path` (either `VTableTree`, `VTableRow`, or `VTableNodeInfo`, which covers both) and return a numeric value to be sorted.
`decreasing`	logical(1). Should the the scores generated by `scorefun` be sorted in decreasing order. If unset (the default of `NA`), it is set to `TRUE` if the generated scores are numeric and `FALSE` if they are characters.
`na.pos`	character(1). What should be done with children (sub-trees/rows) with `NA` scores. Defaults to `"omit"`, which removes them, other allowed values are `"last"` and `"first"` which indicate where they should be placed in the order.
`.prev_path`	character. Internal detail, do not set manually.

Details

The path here can include the "wildcard" "*" as a step, which translates roughly to *any* node/branching element and means that each child at that step will be *separately* sorted based on scorefun and the remaining path entries. This can occur multiple times in a path.

Note that sorting needs a deeper understanding of table structure in rtables. Please consider reading related vignette (Sorting and Pruning) and explore table structure with useful functions like table_structure() and row_paths_summary(). It is also very important to understand the difference between "content" rows and "data" rows. The first one analyzes and describes the split variable generally and is generated with summarize_row_groups(), while the second one is commonly produced by calling one of the various analyze() instances.

Built-in score functions are cont_n_allcols() and cont_n_onecol(). They are both working with content rows (coming from summarize_row_groups()) while a custom score function needs to be used on DataRows. Here, some useful descriptor and accessor functions (coming from related vignette):

- cell_values() - Retrieves a named list of a TableRow or TableTree object's values.
- obj_name() - Retrieves the name of an object. Note this can differ from the label that is displayed (if any is) when printing.
- obj_label() - Retrieves the display label of an object. Note this can differ from the name that appears in the path.
- content_table() - Retrieves a TableTree object's content table (which contains its summary rows).
- tree_children() - Retrieves a TableTree object's direct children (either subtables, rows or possibly a mix thereof, though that should not happen in practice).

Value

A TableTree with the same structure as tt with the exception that the requested sorting has been done at path.

See Also

cont_n_allcols() and cont_n_onecol()

Examples

```
# Creating a table to sort

# Function that gives two statistics per table-tree "leaf"
more_analysis_fnc <- function(x) {
  in_rows(
    "median" = median(x),
    "mean" = mean(x),
    .formats = "xx.x"
  )
}

# Main layout of the table
```

```
raw_lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by(
    "RACE",
    split_fun = drop_and_remove_levels("WHITE") # dropping WHITE levels
  ) %>%
  summarize_row_groups() %>%
  split_rows_by("STRATA1") %>%
  summarize_row_groups() %>%
  analyze("AGE", afun = more_analysis_fnc)

# Creating the table and pruning empty and NAs
tbl <- build_table(raw_lyt, DM) %>%
  prune_table()

# Peek at the table structure to understand how it is built
table_structure(tbl)

#  Sorting only ASIAN sub-table, or, in other words, sorting STRATA elements for
# the ASIAN group/row-split. This uses content_table() accessor function as it
# is a "ContentRow". In this case, we also base our sorting only on the second column.
sort_at_path(tbl, c("ASIAN", "STRATA1"), cont_n_onecol(2))

# Custom scoring function that is working on "DataRow"s
scorefun <- function(tt) {
  # Here we could use browser()
  sum(unlist(row_values(tt))) # Different accessor function
}
# Sorting mean and median for all the AGE leaves!
sort_at_path(tbl, c("RACE", "*", "STRATA1", "*", "AGE"), scorefun)
```

`split_cols_by` *Declaring a column-split based on levels of a variable*

Description

Will generate children for each subset of a categorical variable

Usage

```
split_cols_by(
  lyt,
  var,
  labels_var = var,
  split_label = var,
  split_fun = NULL,
  format = NULL,
  nested = TRUE,
```

```
  child_labels = c("default", "visible", "hidden"),
  extra_args = list(),
  ref_group = NULL
)
```

Arguments

lyt	layout object pre-data used for tabulation
var	string, variable name
labels_var	string, name of variable containing labels to be displayed for the values of var
split_label	string. Label string to be associated with the table generated by the split. Not to be confused with labels assigned to each child (which are based on the data and type of split during tabulation).
split_fun	function/NULL. custom splitting function See custom_split_funs
format	FormatSpec. Format associated with this split. Formats can be declared via strings ("xx.x") or function. In cases such as analyze calls, they can character vectors or lists of functions.
nested	boolean. Should this layout instruction be applied within the existing layout structure *if possible* (TRUE, the default) or as a new top-level element ('FALSE). Ignored if it would nest a split underneath analyses, which is not allowed.
child_labels	string. One of "default", "visible", "hidden". What should the display behavior be for the labels (i.e. label rows) of the children of this split. Defaults to "default" which flags the label row as visible only if the child has 0 content rows.
extra_args	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.
ref_group	character(1) or NULL. Level of var which should be considered ref_group/reference

Value

A PreDataTableLayouts object suitable for passing to further layouting functions, and to build_table.

Custom Splitting Function Details

User-defined custom split functions can perform any type of computation on the incoming data provided that they meet the contract for generating 'splits' of the incoming data 'based on' the split object.

Split functions are functions that accept:

df data.frame of incoming data to be split

spl a Split object. this is largely an internal detail custom functions will not need to worry about, but obj_name(spl), for example, will give the name of the split as it will appear in paths in the resulting table

vals Any pre-calculated values. If given non-null values, the values returned should match these. Should be NULL in most cases and can likely be ignored

labels Any pre-calculated value labels. Same as above for `values`

trim If `TRUE`, resulting splits that are empty should be removed

(Optional) .spl_context a data.frame describing previously performed splits which collectively arrived at `df`

The function must then output a `named list` with the following elements:

values The vector of all values corresponding to the splits of `df`

datasplit a list of data.frames representing the groupings of the actual observations from `df`.

labels a character vector giving a string label for each value listed in the `values` element above

(Optional) extras If present, extra arguments are to be passed to summary and analysis functions whenever they are executed on the corresponding element of `datasplit` or a subset thereof

One way to generate custom splitting functions is to wrap existing split functions and modify either the incoming data before they are called or their outputs.

Author(s)

Gabriel Becker

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  analyze(c("AGE", "BMRKR2"))

tbl <- build_table(lyt, ex_adsl)
tbl

# Let's look at the splits in more detail

lyt1 <- basic_table() %>% split_cols_by("ARM")
lyt1

# add an analysis (summary)
lyt2 <- lyt1 %>%
  analyze(c("AGE", "COUNTRY"),
    afun = list_wrap_x(summary),
    format = "xx.xx"
  )
lyt2

tbl2 <- build_table(lyt2, DM)
tbl2

# By default sequentially adding layouts results in nesting
library(dplyr)
```

```
DM_MF <- DM %>%
  filter(SEX %in% c("M", "F")) %>%
  mutate(SEX = droplevels(SEX))

lyt3 <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by("SEX") %>%
  analyze(c("AGE", "COUNTRY"),
    afun = list_wrap_x(summary),
    format = "xx.xx"
  )
lyt3

tbl3 <- build_table(lyt3, DM_MF)
tbl3

# nested=TRUE vs not
lyt4 <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("SEX", split_fun = drop_split_levels) %>%
  split_rows_by("RACE", split_fun = drop_split_levels) %>%
  analyze("AGE")
lyt4

tbl4 <- build_table(lyt4, DM)
tbl4

lyt5 <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("SEX", split_fun = drop_split_levels) %>%
  analyze("AGE") %>%
  split_rows_by("RACE", nested = FALSE, split_fun = drop_split_levels) %>%
  analyze("AGE")
lyt5

tbl5 <- build_table(lyt5, DM)
tbl5
```

`split_cols_by_cuts` *Split on static or dynamic cuts of the data*

Description

Create columns (or row splits) based on values (such as quartiles) of `var`.

Usage

```
split_cols_by_cuts(
  lyt,
```

```
  var,
  cuts,
  cutlabels = NULL,
  split_label = var,
  nested = TRUE,
  cumulative = FALSE
)

split_rows_by_cuts(
  lyt,
  var,
  cuts,
  cutlabels = NULL,
  split_label = var,
  format = NULL,
  na_str = NA_character_,
  nested = TRUE,
  cumulative = FALSE,
  label_pos = "hidden",
  section_div = NA_character_
)

split_cols_by_cutfun(
  lyt,
  var,
  cutfun = qtile_cuts,
  cutlabelfun = function(x) NULL,
  split_label = var,
  nested = TRUE,
  extra_args = list(),
  cumulative = FALSE
)

split_cols_by_quartiles(
  lyt,
  var,
  split_label = var,
  nested = TRUE,
  extra_args = list(),
  cumulative = FALSE
)

split_rows_by_quartiles(
  lyt,
  var,
  split_label = var,
  format = NULL,
  na_str = NA_character_,
```

```
  nested = TRUE,
  child_labels = c("default", "visible", "hidden"),
  extra_args = list(),
  cumulative = FALSE,
  indent_mod = 0L,
  label_pos = "hidden",
  section_div = NA_character_
)

split_rows_by_cutfun(
  lyt,
  var,
  cutfun = qtile_cuts,
  cutlabelfun = function(x) NULL,
  split_label = var,
  format = NULL,
  na_str = NA_character_,
  nested = TRUE,
  child_labels = c("default", "visible", "hidden"),
  extra_args = list(),
  cumulative = FALSE,
  indent_mod = 0L,
  label_pos = "hidden",
  section_div = NA_character_
)
```

Arguments

lyt	layout object pre-data used for tabulation
var	string, variable name
cuts	numeric. Cuts to use
cutlabels	character (or NULL). Labels for the cuts
split_label	string. Label string to be associated with the table generated by the split. Not to be confused with labels assigned to each child (which are based on the data and type of split during tabulation).
nested	boolean. Should this layout instruction be applied within the existing layout structure *if possible* (TRUE, the default) or as a new top-level element ('FALSE). Ignored if it would nest a split underneath analyses, which is not allowed.
cumulative	logical. Should the cuts be treated as cumulative. Defaults to FALSE
format	FormatSpec. Format associated with this split. Formats can be declared via strings ("xx.x") or function. In cases such as analyze calls, they can character vectors or lists of functions.
na_str	character(1). String that should be displayed when the value of x is missing. Defaults to "NA".
label_pos	character(1). Location the variable label should be displayed, Accepts "hidden" (default for non-analyze row splits), "visible", "topleft", and - for analyze

splits only - "default". For analyze calls, "default" indicates that the variable should be visible if and only if multiple variables are analyzed at the same level of nesting.

section_div — character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.

cutfun — function. Function which accepts the full vector of var values and returns cut points to be passed to cut.

cutlabelfun — function. Function which returns either labels for the cuts or NULL when passed the return value of cutfun

extra_args — list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.

child_labels — string. One of "default", "visible", "hidden". What should the display behavior be for the labels (i.e. label rows) of the children of this split. Defaults to "default" which flags the label row as visible only if the child has 0 content rows.

indent_mod — numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.

Details

For dynamic cuts, the cut is transformed into a static cut by build_table *based on the full dataset*, before proceeding. Thus even when nested within another split in column/row space, the resulting split will reflect the overall values (e.g., quartiles) in the dataset, NOT the values for subset it is nested under.

Value

A PreDataTableLayouts object suitable for passing to further layouting functions, and to build_table.

Author(s)

Gabriel Becker

Examples

```
library(dplyr)

# split_cols_by_cuts
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by_cuts("AGE",
    split_label = "Age",
    cuts = c(0, 25, 35, 1000),
    cutlabels = c("young", "medium", "old")
```

```
  ) %>%
  analyze(c("BMRKR2", "STRATA2")) %>%
  append_topleft("counts")

tbl <- build_table(lyt, ex_adsl)
tbl

# split_rows_by_cuts
lyt2 <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by_cuts("AGE",
    split_label = "Age",
    cuts = c(0, 25, 35, 1000),
    cutlabels = c("young", "medium", "old")
  ) %>%
  analyze(c("BMRKR2", "STRATA2")) %>%
  append_topleft("counts")


tbl2 <- build_table(lyt2, ex_adsl)
tbl2

# split_cols_by_quartiles

lyt3 <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by_quartiles("AGE", split_label = "Age") %>%
  analyze(c("BMRKR2", "STRATA2")) %>%
  append_topleft("counts")

tbl3 <- build_table(lyt3, ex_adsl)
tbl3

# split_rows_by_quartiles
lyt4 <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by("ARM") %>%
  split_rows_by_quartiles("AGE", split_label = "Age") %>%
  analyze("BMRKR2") %>%
  append_topleft(c("Age Quartiles", " Counts BMRKR2"))

tbl4 <- build_table(lyt4, ex_adsl)
tbl4

# split_cols_by_cutfun
cutfun <- function(x) {
  cutpoints <- c(
    min(x),
    mean(x),
    max(x)
  )

  names(cutpoints) <- c("", "Younger", "Older")
  cutpoints
```

```
}

lyt5 <- basic_table() %>%
  split_cols_by_cutfun("AGE", cutfun = cutfun) %>%
  analyze("SEX")

tbl5 <- build_table(lyt5, ex_adsl)
tbl5

# split_rows_by_cutfun
lyt6 <- basic_table() %>%
  split_cols_by("SEX") %>%
  split_rows_by_cutfun("AGE", cutfun = cutfun) %>%
  analyze("BMRKR2")

tbl6 <- build_table(lyt6, ex_adsl)
tbl6
```

`split_cols_by_multivar`

Associate Multiple Variables with Columns

Description

In some cases, the variable to be ultimately analyzed is most naturally defined on a column, not a row basis. When we need columns to reflect different variables entirely, rather than different levels of a single variable, we use `split_cols_by_multivar`

Usage

```
split_cols_by_multivar(
  lyt,
  vars,
  split_fun = NULL,
  varlabels = vars,
  varnames = NULL,
  nested = TRUE,
  extra_args = list()
)
```

Arguments

`lyt`	layout object pre-data used for tabulation
`vars`	character vector. Multiple variable names.
`split_fun`	function/NULL. custom splitting function See `custom_split_funs`
`varlabels`	character vector. Labels for `vars`

varnames	character vector. Names for vars which will appear in pathing. When vars are all unique this will be the variable names. If not, these will be variable names with suffixes as necessary to enforce uniqueness.
nested	boolean. Should this layout instruction be applied within the existing layout structure *if possible* (TRUE, the default) or as a new top-level element ('FALSE). Ignored if it would nest a split underneath analyses, which is not allowed.
extra_args	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.

Value

A PreDataTableLayouts object suitable for passing to further layouting functions, and to build_table.

Author(s)

Gabriel Becker

See Also

analyze_colvars()

Examples

```
library(dplyr)
ANL <- DM %>% mutate(value = rnorm(n()), pctdiff = runif(n()))

## toy example where we take the mean of the first variable and the
## count of >.5 for the second.
colfuns <- list(
  function(x) in_rows(mean = mean(x), .formats = "xx.x"),
  function(x) in_rows("# x > 5" = sum(x > .5), .formats = "xx")
)

lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by_multivar(c("value", "pctdiff")) %>%
  split_rows_by("RACE",
    split_label = "ethnicity",
    split_fun = drop_split_levels
  ) %>%
  summarize_row_groups() %>%
  analyze_colvars(afun = colfuns)
lyt

tbl <- build_table(lyt, ANL)
tbl
```

split_funcs *Split functions*

Description

Split functions

Usage

```
remove_split_levels(excl)

keep_split_levels(only, reorder = TRUE)

drop_split_levels(df, spl, vals = NULL, labels = NULL, trim = FALSE)

drop_and_remove_levels(excl)

reorder_split_levels(neworder, newlabels = neworder, drlevels = TRUE)

trim_levels_in_group(innervar, drop_outlevs = TRUE)
```

Arguments

excl	character. Levels to be excluded (they will not be reflected in the resulting table structure regardless of presence in the data).
only	character. Levels to retain (all others will be dropped).
reorder	logical(1). Should the order of only be used as the order of the children of the split. defaults to TRUE
df	dataset (data.frame or tibble)
spl	A Split object defining a partitioning or analysis/tabulation of the data.
vals	ANY. For internal use only.
labels	character. Labels to use for the remaining levels instead of the existing ones.
trim	logical(1). Should splits corresponding with 0 observations be kept when tabulating.
neworder	character. New order or factor levels.
newlabels	character. Labels for (new order of) factor levels
drlevels	logical(1). Should levels in the data which do not appear in neworder be dropped. Defaults to TRUE
innervar	character(1). Variable whose factor levels should be trimmed (e.g., empty levels dropped) *separately within each grouping defined at this point in the structure*
drop_outlevs	logical(1). Should empty levels in the variable being split on (i.e. the 'outer' variable, not innervar) be dropped? Defaults to TRUE

Value

a closure suitable for use as a splitting function (`splfun`) when creating a table layout

Custom Splitting Function Details

User-defined custom split functions can perform any type of computation on the incoming data provided that they meet the contract for generating 'splits' of the incoming data 'based on' the split object.

Split functions are functions that accept:

df data.frame of incoming data to be split

spl a Split object. this is largely an internal detail custom functions will not need to worry about, but `obj_name(spl)`, for example, will give the name of the split as it will appear in paths in the resulting table

vals Any pre-calculated values. If given non-null values, the values returned should match these. Should be NULL in most cases and can likely be ignored

labels Any pre-calculated value labels. Same as above for `values`

trim If `TRUE`, resulting splits that are empty should be removed

(Optional) .spl_context a data.frame describing previously performed splits which collectively arrived at `df`

The function must then output a `named list` with the following elements:

values The vector of all values corresponding to the splits of `df`

datasplit a list of data.frames representing the groupings of the actual observations from `df`.

labels a character vector giving a string label for each value listed in the `values` element above

(Optional) extras If present, extra arguments are to be passed to summary and analysis functions whenever they are executed on the corresponding element of `datasplit` or a subset thereof

One way to generate custom splitting functions is to wrap existing split functions and modify either the incoming data before they are called or their outputs.

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("COUNTRY",
    split_fun = remove_split_levels(c(
      "USA", "CAN",
      "CHE", "BRA"
    ))
  ) %>%
  analyze("AGE")

tbl <- build_table(lyt, DM)
tbl

lyt <- basic_table() %>%
```

```
  split_cols_by("ARM") %>%
  split_rows_by("COUNTRY",
    split_fun = keep_split_levels(c("USA", "CAN", "BRA"))
  ) %>%
  analyze("AGE")

tbl <- build_table(lyt, DM)
tbl
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("SEX", split_fun = drop_split_levels) %>%
  analyze("AGE")

tbl <- build_table(lyt, DM)
tbl
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("SEX", split_fun = drop_and_remove_levels(c("M", "U"))) %>%
  analyze("AGE")

tbl <- build_table(lyt, DM)
tbl
```

`split_rows_by` *Add Rows according to levels of a variable*

Description

Add Rows according to levels of a variable

Usage

```
split_rows_by(
  lyt,
  var,
  labels_var = var,
  split_label = var,
  split_fun = NULL,
  format = NULL,
  na_str = NA_character_,
  nested = TRUE,
  child_labels = c("default", "visible", "hidden"),
  label_pos = "hidden",
  indent_mod = 0L,
  page_by = FALSE,
  page_prefix = split_label,
  section_div = NA_character_
)
```

Arguments

lyt	layout object pre-data used for tabulation
var	string, variable name
labels_var	string, name of variable containing labels to be displayed for the values of var
split_label	string. Label string to be associated with the table generated by the split. Not to be confused with labels assigned to each child (which are based on the data and type of split during tabulation).
split_fun	function/NULL. custom splitting function See custom_split_funs
format	FormatSpec. Format associated with this split. Formats can be declared via strings ("xx.x") or function. In cases such as analyze calls, they can character vectors or lists of functions.
na_str	character(1). String that should be displayed when the value of x is missing. Defaults to "NA".
nested	boolean. Should this layout instruction be applied within the existing layout structure *if possible* (TRUE, the default) or as a new top-level element ('FALSE). Ignored if it would nest a split underneath analyses, which is not allowed.
child_labels	string. One of "default", "visible", "hidden". What should the display behavior be for the labels (i.e. label rows) of the children of this split. Defaults to "default" which flags the label row as visible only if the child has 0 content rows.
label_pos	character(1). Location the variable label should be displayed, Accepts "hidden" (default for non-analyze row splits), "visible", "topleft", and - for analyze splits only - "default". For analyze calls, "default" indicates that the variable should be visible if and only if multiple variables are analyzed at the same level of nesting.
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
page_by	logical(1). Should pagination be forced between different children resulting form this split. An error will rise if the selected split does not contain at least one value that is not NA.
page_prefix	character(1). Prefix, to be appended with the split value, when forcing pagination between the children of this split/table
section_div	character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.

Value

A PreDataTableLayouts object suitable for passing to further layouting functions, and to build_table.

Custom Splitting Function Details

User-defined custom split functions can perform any type of computation on the incoming data provided that they meet the contract for generating 'splits' of the incoming data 'based on' the split object.

Split functions are functions that accept:

df data.frame of incoming data to be split

spl a Split object. this is largely an internal detail custom functions will not need to worry about, but `obj_name(spl)`, for example, will give the name of the split as it will appear in paths in the resulting table

vals Any pre-calculated values. If given non-null values, the values returned should match these. Should be NULL in most cases and can likely be ignored

labels Any pre-calculated value labels. Same as above for `values`

trim If `TRUE`, resulting splits that are empty should be removed

(Optional) .spl_context a data.frame describing previously performed splits which collectively arrived at `df`

The function must then output a `named list` with the following elements:

values The vector of all values corresponding to the splits of `df`

datasplit a list of data.frames representing the groupings of the actual observations from `df`.

labels a character vector giving a string label for each value listed in the `values` element above

(Optional) extras If present, extra arguments are to be passed to summary and analysis functions whenever they are executed on the corresponding element of `datasplit` or a subset thereof

One way to generate custom splitting functions is to wrap existing split functions and modify either the incoming data before they are called or their outputs.

Note

If `var` is a factor with empty unobserved levels and `labels_var` is specified, it must also be a factor with the same number of levels as `var`. Currently the error that occurs when this is not the case is not very informative, but that will change in the future.

Author(s)

Gabriel Becker

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("RACE", split_fun = drop_split_levels) %>%
  analyze("AGE", mean, var_labels = "Age", format = "xx.xx")

tbl <- build_table(lyt, DM)
tbl
```

```
lyt2 <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("RACE") %>%
  analyze("AGE", mean, var_labels = "Age", format = "xx.xx")

tbl2 <- build_table(lyt2, DM)
tbl2

lyt3 <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by("SEX") %>%
  summarize_row_groups(label_fstr = "Overall (N)") %>%
  split_rows_by("RACE",
    split_label = "Ethnicity", labels_var = "ethn_lab",
    split_fun = drop_split_levels
  ) %>%
  summarize_row_groups("RACE", label_fstr = "%s (n)") %>%
  analyze("AGE", var_labels = "Age", afun = mean, format = "xx.xx")

lyt3

library(dplyr)
DM2 <- DM %>%
  filter(SEX %in% c("M", "F")) %>%
  mutate(
    SEX = droplevels(SEX),
    gender_lab = c(
      "F" = "Female", "M" = "Male",
      "U" = "Unknown",
      "UNDIFFERENTIATED" = "Undifferentiated"
    )[SEX],
    ethn_lab = c(
      "ASIAN" = "Asian",
      "BLACK OR AFRICAN AMERICAN" = "Black or African American",
      "WHITE" = "White",
      "AMERICAN INDIAN OR ALASKA NATIVE" = "American Indian or Alaska Native",
      "MULTIPLE" = "Multiple",
      "NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER" =
        "Native Hawaiian or Other Pacific Islander",
      "OTHER" = "Other", "UNKNOWN" = "Unknown"
    )[RACE]
  )

tbl3 <- build_table(lyt3, DM2)
tbl3
```

---

split_rows_by_multivar

Associate Multiple Variables with Rows

Description

When we need rows to reflect different variables rather than different levels of a single variable, we use `split_rows_by_multivar`.

Usage

```
split_rows_by_multivar(
  lyt,
  vars,
  split_fun = NULL,
  split_label = "",
  varlabels = vars,
  format = NULL,
  na_str = NA_character_,
  nested = TRUE,
  child_labels = c("default", "visible", "hidden"),
  indent_mod = 0L,
  section_div = NA_character_,
  extra_args = list()
)
```

Arguments

`lyt`	layout object pre-data used for tabulation
`vars`	character vector. Multiple variable names.
`split_fun`	function/NULL. custom splitting function See `custom_split_funs`
`split_label`	string. Label string to be associated with the table generated by the split. Not to be confused with labels assigned to each child (which are based on the data and type of split during tabulation).
`varlabels`	character vector. Labels for `vars`
`format`	`FormatSpec`. Format associated with this split. Formats can be declared via strings (`"xx.x"`) or function. In cases such as `analyze` calls, they can character vectors or lists of functions.
`na_str`	character(1). String that should be displayed when the value of `x` is missing. Defaults to `"NA"`.
`nested`	boolean. Should this layout instruction be applied within the existing layout structure *if possible* (`TRUE`, the default) or as a new top-level element ('FALSE). Ignored if it would nest a split underneath analyses, which is not allowed.
`child_labels`	string. One of `"default"`, `"visible"`, `"hidden"`. What should the display behavior be for the labels (i.e. label rows) of the children of this split. Defaults to `"default"` which flags the label row as visible only if the child has 0 content rows.
`indent_mod`	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.

section_div — character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.

extra_args — list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.

Value

A PreDataTableLayouts object suitable for passing to further layouting functions, and to build_table.

See Also

split_rows_by() for typical row splitting, and split_cols_by_multivar() to perform the same type of split on a column basis.

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by_multivar(c("SEX", "STRATA1")) %>%
  summarize_row_groups() %>%
  analyze(c("AGE", "SEX"))

tbl <- build_table(lyt, DM)
tbl
```

---

spl_context *.spl_context within analysis and split functions*

Description

.spl_context is an optional parameter for any of rtables' special functions, them being afun (analysis function in analyze), cfun (content or label function in summarize_row_groups), or split_fun (e.g. for split_rows_by).

Details

The .spl_context data.frame gives information about the subsets of data corresponding to the splits within-which the current analyze action is nested. Taken together, these correspond to the path that the resulting (set of) rows the analysis function is creating, although the information is in a slightly different form. Each split (which correspond to groups of rows in the resulting table), as well as the initial 'root' "split", is represented via the following columns:

split The name of the split (often the variable being split in the simple case)

value The string representation of the value at that split

full_parent_df a dataframe containing the full data (i.e. across all columns) corresponding to the path defined by the combination of `split` and `value` of this row *and all rows above this row*

all_cols_n the number of observations corresponding to this row grouping (union of all columns)

***(row-split and analyze contexts only)* <1 column for each column in the table structure** These list columns (named the same as `names(col_exprs(tab))`) contain logical vectors corresponding to the subset of this row's `full_parent_df` corresponding to that column

cur_col_id Identifier of the current column. This may be an internal name, constructed by pasting the column path together

cur_col_subset List column containing logical vectors indicating the subset of that row's `full_parent_df` for the column currently being created by the analysis function

cur_col_expr List of current column expression. This may be used to filter `.alt_df_row` or any external data by column. Filtering `.alt_df_row` by columns produces `.alt_df`.

cur_col_n integer column containing the observation counts for that split

cur_col_split Current column split names. This is recovered from the current column path

cur_col_split_val Current column split values. This is recovered from the current column path

note Within analysis functions that accept `.spl_context`*, the* `all_cols_n` *and* `cur_col_n` *columns of the dataframe will contain the 'true' observation counts corresponding to the row-group and row-group x column subsets of the data. These numbers will not, and currently cannot, reflect alternate column observation counts provided by the* `alt_counts_df`, `col_counts` *or* `col_total` *arguments to* `build_table`

`spl_context_to_disp_path` *Translate spl_context to Path for display in error messages*

Description

Translate spl_context to Path for display in error messages

Usage

```
spl_context_to_disp_path(ctx)
```

Arguments

`ctx`	data.frame. The `spl_context` data.frame where the error occurred

Value

A character string containing a description of the row path corresponding to the `ctx`

spl_variable *Variable Associated With a Split*

Description

This function is intended for use when writing custom splitting logic. In cases where the split is associated with a single variable, the name of that variable will be returned. At time of writing this includes splits generated via the split_rows_by, split_cols_by, split_rows_by_cuts, split_cols_by_cuts, split_rows_by_cutfun, and split_cols_by_cutfun layout directives.

Usage

```
spl_variable(spl)

## S4 method for signature 'VarLevelSplit'
spl_variable(spl)

## S4 method for signature 'VarDynCutSplit'
spl_variable(spl)

## S4 method for signature 'VarStaticCutSplit'
spl_variable(spl)

## S4 method for signature 'Split'
spl_variable(spl)
```

Arguments

spl	Split. The split object

Value

for splits with a single variable associated with them, the split, for others, an error is raised.

See Also

make_split_fun

summarize_rows *summarize_rows*

Description

summarize_rows is deprecated in favor of make_row_df.

Usage

```
summarize_rows(obj)
```

Arguments

obj	VTableTree.

Value

A data.frame summarizing the rows in obj.

---

summarize_row_groups *Add a content row of summary counts*

Description

Add a content row of summary counts

Usage

```
summarize_row_groups(
  lyt,
  var = "",
  label_fstr = "%s",
  format = "xx (xx.x%)",
  na_str = "-",
  cfun = NULL,
  indent_mod = 0L,
  extra_args = list()
)
```

Arguments

lyt	layout object pre-data used for tabulation
var	string, variable name
label_fstr	string. An sprintf style format string containing. For non-comparison splits, it can contain up to one "%s" which takes the current split value and generates the row/column label. Comparison-based splits it can contain up to two "%s".
format	FormatSpec. Format associated with this split. Formats can be declared via strings ("xx.x") or function. In cases such as analyze calls, they can character vectors or lists of functions.
na_str	character(1). String that should be displayed when the value of x is missing. Defaults to "NA".

cfun	list/function/NULL. tabulation function(s) for creating content rows. Must accept x or df as first parameter. Must accept labelstr as the second argument. Can optionally accept all optional arguments accepted by analysis functions. See analyze.
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
extra_args	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.

Details

If format expects 1 value (i.e. it is specified as a format string and xx appears for two values (i.e. xx appears twice in the format string) or is specified as a function, then both raw and percent of column total counts are calculated. If format is a format string where xx appears only one time, only raw counts are used.

cfun must accept x or df as its first argument. For the df argument cfun will receive the subset data.frame corresponding with the row- and column-splitting for the cell being calculated. Must accept labelstr as the second parameter, which accepts the label of the level of the parent split currently being summarized. Can additionally take any optional argument supported by analysis functions. (see analyze).

In addition, if complex custom functions are needed, we suggest checking the available additional_fun_params that apply here as for afun.

Value

A PreDataTableLayouts object suitable for passing to further layouting functions, and to build_table.

Author(s)

Gabriel Becker

Examples

```
DM2 <- subset(DM, COUNTRY %in% c("USA", "CAN", "CHN"))

lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("COUNTRY", split_fun = drop_split_levels) %>%
  summarize_row_groups(label_fstr = "%s (n)") %>%
  analyze("AGE", afun = list_wrap_x(summary), format = "xx.xx")
lyt

tbl <- build_table(lyt, DM2)
tbl
```

```
row_paths_summary(tbl) # summary count is a content table


## use a cfun and extra_args to customize summarization
## behavior
sfun <- function(x, labelstr, trim) {
  in_rows(
    c(mean(x, trim = trim), trim),
    .formats = "xx.x (xx.x%)",
    .labels = sprintf(
      "%s (Trimmed mean and trim %%)",
      labelstr
    )
  )
}

lyt2 <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by("ARM") %>%
  split_rows_by("COUNTRY", split_fun = drop_split_levels) %>%
  summarize_row_groups("AGE",
    cfun = sfun,
    extra_args = list(trim = .2)
  ) %>%
  analyze("AGE", afun = list_wrap_x(summary), format = "xx.xx") %>%
  append_topleft(c("Country", "  Age"))

tbl2 <- build_table(lyt2, DM2)
tbl2
```

`table_shell` *Table shells*

Description

A table shell is a rendering of the table which maintains the structure, but does not display the values, rather displaying the formatting instructions for each cell.

Usage

```
table_shell(
  tt,
  widths = NULL,
  col_gap = 3,
  hsep = default_hsep(),
  tf_wrap = FALSE,
  max_width = NULL
)
```

```
table_shell_str(
  tt,
  widths = NULL,
  col_gap = 3,
  hsep = default_hsep(),
  tf_wrap = FALSE,
  max_width = NULL
)
```

Arguments

tt	TableTree (or related class). A TableTree object representing a populated table.
widths	numeric (or NULL). (proposed) widths for the columns of x. The expected length of this numeric vector can be retrieved with ncol() + 1 as the column of row names must also be considered.
col_gap	numeric(1). Space (in characters) between columns
hsep	character(1). Characters to repeat to create header/body separator line. If NULL, the object value will be used. If " ", an empty separator will be printed. Check default_hsep() for more information.
tf_wrap	logical(1). Should the texts for title, subtitle, and footnotes be wrapped?
max_width	integer(1), character(1) or NULL. Width that title and footer (including footnotes) materials should be word-wrapped to. If NULL, it is set to the current print width of the session (getOption("width")). If set to "auto", the width of the table (plus any table inset) is used. Ignored completely if tf_wrap is FALSE.

Value

for table_shell_str the string representing the table shell, for table_shell, NULL, as the function is called for the side effect of printing the shell to the console

Examples

```
library(dplyr)

iris2 <- iris %>%
  group_by(Species) %>%
  mutate(group = as.factor(rep_len(c("a", "b"), length.out = n()))) %>%
  ungroup()

lyt <- basic_table() %>%
  split_cols_by("Species") %>%
  split_cols_by("group") %>%
 analyze(c("Sepal.Length", "Petal.Width"), afun = list_wrap_x(summary), format = "xx.xx")

tbl <- build_table(lyt, iris2)
table_shell(tbl)
```

table_structure *Summarize Table*

Description

Summarize Table

Usage

```
table_structure(x, detail = c("subtable", "row"))
```

Arguments

x	a table object
detail	either row or subtable

Value

currently no return value. Called for the side-effect of printing a row- or subtable-structure summary of x.

Examples

```
library(dplyr)

iris2 <- iris %>%
  group_by(Species) %>%
  mutate(group = as.factor(rep_len(c("a", "b"), length.out = n()))) %>%
  ungroup()

lyt <- basic_table() %>%
  split_cols_by("Species") %>%
  split_cols_by("group") %>%
  analyze(c("Sepal.Length", "Petal.Width"),
    afun = list_wrap_x(summary),
    format = "xx.xx"
  )

tbl <- build_table(lyt, iris2)
tbl

row_paths(tbl)

table_structure(tbl)

table_structure(tbl, detail = "row")
```

top_left *Top Left Material (Experimental)*

Description

A TableTree object can have *top left material* which is a sequence of strings which are printed in the area of the table between the column header display and the label of the first row. These functions access and modify that material.

Usage

```
top_left(obj)

## S4 method for signature 'VTableTree'
top_left(obj)

## S4 method for signature 'InstantiatedColumnInfo'
top_left(obj)

## S4 method for signature 'PreDataTableLayouts'
top_left(obj)

top_left(obj) <- value

## S4 replacement method for signature 'VTableTree'
top_left(obj) <- value

## S4 replacement method for signature 'InstantiatedColumnInfo'
top_left(obj) <- value

## S4 replacement method for signature 'PreDataTableLayouts'
top_left(obj) <- value
```

Arguments

obj	ANY. The object for the accessor to access or modify
value	The new value

Value

A character vector representing the top-left material of obj (or obj after modification, in the case of the setter).

tostring *Convert an* rtable *object to a string*

Description

Transform a complex object into a string representation ready to be printed or written to a plain-text file

All objects that are printed to console pass by `toString`. This function allows fundamental formatting specifications for the final output, like column widths and relative wrapping (`width`), title and footer wrapping (`tf_wrap = TRUE` and `max_width`), or horizontal separator character (e.g. `hsep = "+"`).

Usage

```
## S4 method for signature 'VTableTree'
toString(
  x,
  widths = NULL,
  col_gap = 3,
  hsep = horizontal_sep(x),
  indent_size = 2,
  tf_wrap = FALSE,
  max_width = NULL
)
```

Arguments

`x`	ANY. Object to be prepared for rendering.
`widths`	numeric (or `NULL`). (proposed) widths for the columns of `x`. The expected length of this numeric vector can be retrieved with `ncol() + 1` as the column of row names must also be considered.
`col_gap`	numeric(1). Space (in characters) between columns
`hsep`	character(1). Characters to repeat to create header/body separator line. If `NULL`, the object value will be used. If `" "`, an empty separator will be printed. Check `default_hsep()` for more information.
`indent_size`	numeric(1). Number of spaces to use per indent level. Defaults to 2
`tf_wrap`	logical(1). Should the texts for title, subtitle, and footnotes be wrapped?
`max_width`	integer(1), character(1) or `NULL`. Width that title and footer (including footnotes) materials should be word-wrapped to. If `NULL`, it is set to the current print width of the session (`getOption("width")`). If set to `"auto"`, the width of the table (plus any table inset) is used. Ignored completely if `tf_wrap` is `FALSE`.

Details

Manual insertion of newlines is not supported when `tf_wrap` is on and will result in a warning and undefined wrapping behavior. Passing vectors of already split strings remains supported, however in this case each string is word-wrapped separately with the behavior described above.

Value

a string representation of x as it appears when printed.

See Also

wrap_string()

Examples

```
library(dplyr)

iris2 <- iris %>%
  group_by(Species) %>%
  mutate(group = as.factor(rep_len(c("a", "b"), length.out = n()))) %>%
  ungroup()

lyt <- basic_table() %>%
  split_cols_by("Species") %>%
  split_cols_by("group") %>%
  analyze(c("Sepal.Length", "Petal.Width"), afun = list_wrap_x(summary), format = "xx.xx")

tbl <- build_table(lyt, iris2)

cat(toString(tbl, col_gap = 3))
```

---

tree_children *Retrieve or set the direct children of a Tree-style object*

Description

Retrieve or set the direct children of a Tree-style object

Usage

```
tree_children(x)

tree_children(x) <- value
```

Arguments

x	An object with a Tree structure
value	New list of children.

Value

List of direct children of x

trim_levels_in_facets *Trim Levels of Another Variable From Each Facet (Postprocessing split step)*

Description

Trim Levels of Another Variable From Each Facet (Postprocessing split step)

Usage

```
trim_levels_in_facets(innervar)
```

Arguments

innervar	character. The variable(s) to trim (remove unobserved levels) independently within each facet.

Value

a function suitable for use in the pre (list) argument of make_split_fun

See Also

make_split_fun

Other make_custom_split: add_combo_facet(), drop_facet_levels(), make_split_fun(), make_split_result()

trim_levels_to_map *Trim Levels to map*

Description

This split function constructor creates a split function which trims levels of a variable to reflect restrictions on the possible combinations of two or more variables which are split by (along the same axis) within a layout.

Usage

```
trim_levels_to_map(map = NULL)
```

Arguments

map	data.frame. A data.frame defining allowed combinations of variables. Any combination at the level of this split not present in the map will be removed from the data, both for the variable being split and those present in the data but not associated with this split or any parents of it.

Details

When splitting occurs, the map is subset to the values of all previously performed splits. The levels of the variable being split are then pruned to only those still present within this subset of the map representing the current hierarchical splitting context.

Splitting is then performed via the `keep_split_levels` split function.

Each resulting element of the partition is then further trimmed by pruning values of any remaining variables specified in the map to those values allowed under the combination of the previous and current split.

Value

a fun

See Also

`trim_levels_in_group()`

Examples

```
map <- data.frame(
  LBCAT = c("CHEMISTRY", "CHEMISTRY", "CHEMISTRY", "IMMUNOLOGY"),
  PARAMCD = c("ALT", "CRP", "CRP", "IGA"),
  ANRIND = c("LOW", "LOW", "HIGH", "HIGH"),
  stringsAsFactors = FALSE
)

lyt <- basic_table() %>%
  split_rows_by("LBCAT") %>%
  split_rows_by("PARAMCD", split_fun = trim_levels_to_map(map = map)) %>%
  analyze("ANRIND")
tbl <- build_table(lyt, ex_adlb)
```

---

trim_rows *Trim rows from a populated table without regard for table structure*

Description

Trim rows from a populated table without regard for table structure

Usage

```
trim_rows(tt, criteria = all_zero_or_na)
```

Arguments

`tt`	`TableTree` (or related class). A `TableTree` object representing a populated table.
`criteria`	function. Function which takes a `TableRow` object and returns `TRUE` if that row should be removed. Defaults to `all_zero_or_na`

Details

This function will be deprecated in the future in favor of the more elegant and versatile `prune_table()` function which can perform the same function as `trim_rows()` but is more powerful as it takes table structure into account.

Value

The table with rows that have only NA or 0 cell values removed

Note

Visible `LabelRows` are including in this trimming, which can lead to either all label rows being trimmed or label rows remaining when all data rows have been trimmed, depending on what `criteria` returns when called on a `LabelRow` object. To avoid this, use the structurally-aware `prune_table` machinery instead.

See Also

`prune_table()`

Examples

```
adsl <- ex_adsl
levels(adsl$SEX) <- c(levels(ex_adsl$SEX), "OTHER")

tbl_to_trim <- basic_table() %>%
  analyze("BMRKR1") %>%
  split_cols_by("ARM") %>%
  split_rows_by("SEX") %>%
  summarize_row_groups() %>%
  split_rows_by("STRATA1") %>%
  summarize_row_groups() %>%
  analyze("AGE") %>%
  build_table(adsl)

tbl_to_trim %>% trim_rows()

tbl_to_trim %>% trim_rows(all_zero)
```

`trim_zero_rows` *Trim Zero Rows*

Description

Trim Zero Rows

Usage

```
trim_zero_rows(tbl)
```

Arguments

tbl	table object

Value

an rtable object

---

tt_at_path *Get or set table elements at specified path*

Description

Get or set table elements at specified path

Usage

```
tt_at_path(tt, path, ...)

tt_at_path(tt, path, ...) <- value
```

Arguments

tt	TableTree (or related class). A TableTree object representing a populated table.
path	character. A vector path for a position within the structure of a tabletree. Each element represents a subsequent choice amongst the children of the previous choice.
...	unused.
value	The new value

Note

Setting NULL at a defined path removes the corresponding sub table.

Examples

```
# Accessing sub table.
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("SEX") %>%
  split_rows_by("BMRKR2") %>%
  analyze("AGE")
```

```
tbl <- build_table(lyt, ex_adsl) %>% prune_table()
sub_tbl <- tt_at_path(tbl, path = c("SEX", "F", "BMRKR2"))

# Removing sub table.
tbl2 <- tbl
tt_at_path(tbl2, path = c("SEX", "F")) <- NULL
tbl2

# Setting sub table.
lyt3 <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_rows_by("SEX") %>%
  analyze("BMRKR2")

tbl3 <- build_table(lyt3, ex_adsl) %>% prune_table()

tt_at_path(tbl3, path = c("SEX", "F", "BMRKR2")) <- sub_tbl
tbl3
```

---

tt_to_flextable *Create a* FlexTable *from an* rtables *table*

Description

Principally used for export (export_as_docx()), this function produces a flextable from an rtables table. If theme = NULL, rtables-like style will be used. Otherwise, theme_docx_default() will produce a .docx-friendly table.

Usage

```
tt_to_flextable(
  tt,
  theme = theme_docx_default(tt),
  border = flextable::fp_border_default(width = 0.5),
  indent_size = NULL,
  titles_as_header = TRUE,
  footers_as_text = FALSE,
  counts_in_newline = FALSE,
  paginate = FALSE,
  lpp = NULL,
  cpp = NULL,
  ...,
  colwidths = propose_column_widths(matrix_form(tt, indent_rownames = TRUE)),
  tf_wrap = !is.null(cpp),
  max_width = cpp,
  total_width = 10
)
```

```
theme_docx_default(
  tt = NULL,
  font = "Arial",
  font_size = 9,
  bold = c("header", "content_rows", "label_rows"),
  bold_manual = NULL,
  border = flextable::fp_border_default(width = 0.5)
)
```

Arguments

tt	TableTree (or related class). A TableTree object representing a populated table.
theme	function(1). Defaults to theme_docx_default(tt). It expects a a theme function that is designed internally as a function of a flextable object and changes its layout and style. If set to NULL, it will produce a table similar to rtables default.
border	officer border object. Defaults to officer::fp_border(width = 0.5).
indent_size	integer(1). If NULL, the default indent size of the table (see matrix_form() indent_size) is used. To work with docx, any size is multiplied by 2 mm (5.67 pt) as default.
titles_as_header	logical(1). Defaults to TRUE for tt_to_flextable(), so the table is self-contained as it makes additional header rows for main_title() string and subtitles() character vector (one per element). FALSE is suggested for export_as_docx(). This adds titles and subtitles as a text paragraph above the table. Same style is applied.
footers_as_text	logical(1). Defaults to FALSE for tt_to_flextable(), so the table is self-contained with the flextable definition of footnotes. TRUE is used for export_as_docx() to add the footers as a new paragraph after the table. Same style is applied, but with a smaller font.
counts_in_newline	logical(1). Defaults to FALSE. In rtables text printing (formatters::toString()), the column counts, i.e. (N=xx), is always on a new line. We noticed that for docx exports could be necessary to have it on the same line.
paginate	logical(1). If you need .docx export and you use export_as_docx, we suggest relying on word pagination system. Cooperation between the two mechanisms is not guaranteed. This option splits tt in different "pages" as multiple flextables. Defaults to FALSE.
lpp	numeric. Maximum lines per page including (re)printed header and context rows
cpp	numeric(1) or NULL. Width (in characters) of the pages for horizontal pagination. NA (the default) indicates cpp should be inferred from the page size; NULL indicates no horizontal pagination should be done regardless of page size.
...	Passed on to methods or tabulation functions.

colwidths	numeric vector. Column widths for use with vertical pagination.
tf_wrap	logical(1). Should the texts for title, subtitle, and footnotes be wrapped?
max_width	integer(1), character(1) or NULL. Width that title and footer (including footnotes) materials should be word-wrapped to. If NULL, it is set to the current print width of the session (getOption("width")). If set to "auto", the width of the table (plus any table inset) is used. Ignored completely if tf_wrap is FALSE.
total_width	numeric(1). Total width in inches for the resulting flextable(s). Defaults to 10.
font	character(1). Defaults to "Arial". If the font is not vailable, flextable default is used.
font_size	integer(1). Positive integerish value that defaults to 9.
bold	character vector. It can be any combination of c("header", "content_rows", "label_rows"). The first one renders all column names bold (not topleft content). Second and third option use make_row_df() to render content or/and label rows as bold.
bold_manual	named list. List of indexes lists. See example for needed structure. Accepted groupings/names are c("header", "body").

Value

a flextable object.

Functions

- theme_docx_default(): main theme function for export_as_docx()

See Also

export_as_docx()

export_as_docx()

Examples

```
analysisfun <- function(x, ...) {
  in_rows(
    row1 = 5,
    row2 = c(1, 2),
    .row_footnotes = list(row1 = "row 1 - row footnote"),
    .cell_footnotes = list(row2 = "row 2 - cell footnote")
  )
}

lyt <- basic_table(
  title = "Title says Whaaaat", subtitles = "Oh, ok.",
  main_footer = "ha HA! Footer!"
) %>%
  split_cols_by("ARM") %>%
  analyze("AGE", afun = analysisfun)
```

```
tbl <- build_table(lyt, ex_adsl)
# rtables style
tt_to_flextable(tbl, theme = NULL)

tt_to_flextable(tbl, theme = theme_docx_default(tbl, font_size = 7))

# Custom theme
special_bold <- list(
  "header" = list("i" = 1, "j" = c(1, 3)),
  "body" = list("i" = c(1, 2), "j" = 1)
)
custom_theme <- theme_docx_default(tbl,
  font_size = 10,
  font = "Brush Script MT",
  border = flextable::fp_border_default(color = "pink", width = 2),
  bold = NULL,
  bold_manual = special_bold
)
tt_to_flextable(tbl,
  border = flextable::fp_border_default(color = "pink", width = 2),
  theme = custom_theme
)
```

`update_ref_indexing` *Update footnote indexes on a built table*

Description

Re-indexes footnotes within a built table

Usage

```
update_ref_indexing(tt)
```

Arguments

`tt`	`TableTree` (or related class). A `TableTree` object representing a populated table.

Details

After adding or removing referential footnotes manually, or after subsetting a table, the reference indexes (i.e. the number associated with specific footnotes) may be incorrect. This function recalculates these based on the full table.

Note

In the future this should not generally need to be called manually.

validate_table_struct *Validate and Assert valid table structure (Experimental).*

Description

Validate and Assert valid table structure (Experimental).

Usage

```
validate_table_struct(tt)

assert_valid_table(tt, warn_only = FALSE)
```

Arguments

tt	TableTree A TableTree (rtables-built table) is considered degenerate if 1. it contains no subtables or data rows (content rows do not count) 2. it contains a subtable which is degenerate by the criterion above validate_table_struct assesses whether tt has a valid (non-degenerate) structure. assert_valid_table asserts a table must have a valid structure, and throws an informative error (the default) or warning (if warn_only is TRUE) if the table is degenerate (has invalid structure or contains one or more invalid substructures
warn_only	logical(1). Should a warning be thrown instead of an error? Defaults to FALSE

Value

for validate_table_struct a logical value indicating valid structure; assert_valid_table is called for its side-effect of throwing an error or warning for degenerate tables.

Note

This function is experimental and the exact text of the warning/error is subject to change in future releases.

Examples

```
validate_table_struct(rtable("hahaha"))
## Not run:
assert_valid_table(rtable("oops"))

## End(Not run)
```

value_formats *Value Formats*

Description

Returns a matrix of formats for the cells in a table

Usage

```
value_formats(obj, default = obj_format(obj))

## S4 method for signature 'ANY'
value_formats(obj, default = obj_format(obj))

## S4 method for signature 'TableRow'
value_formats(obj, default = obj_format(obj))

## S4 method for signature 'LabelRow'
value_formats(obj, default = obj_format(obj))

## S4 method for signature 'VTableTree'
value_formats(obj, default = obj_format(obj))
```

Arguments

obj	A table or row object.
default	FormatSpec.

Value

Matrix (storage mode list) containing the effective format for each cell position in the table (including 'virtual' cells implied by label rows, whose formats are always NULL)

Examples

```
lyt <- basic_table() %>%
  split_rows_by("RACE", split_fun = keep_split_levels(c("ASIAN", "WHITE"))) %>%
  analyze("AGE")

tbl <- build_table(lyt, DM)
value_formats(tbl)
```

VarLevelSplit-class *Split on levels within a variable*

Description

Split on levels within a variable

Usage

```
VarLevelSplit(
  var,
  split_label,
  labels_var = NULL,
  cfun = NULL,
  cformat = NULL,
  cna_str = NA_character_,
  split_fun = NULL,
  split_format = NULL,
  split_na_str = NA_character_,
  valorder = NULL,
  split_name = var,
  child_labels = c("default", "visible", "hidden"),
  extra_args = list(),
  indent_mod = 0L,
  label_pos = c("topleft", "hidden", "visible"),
  cindent_mod = 0L,
  cvar = "",
  cextra_args = list(),
  page_prefix = NA_character_,
  section_div = NA_character_
)

VarLevWBaselineSplit(
  var,
  ref_group,
  labels_var = var,
  split_label,
  split_fun = NULL,
  label_fstr = "%s - %s",
  cfun = NULL,
  cformat = NULL,
  cna_str = NA_character_,
  cvar = "",
  split_format = NULL,
  split_na_str = NA_character_,
  valorder = NULL,
  split_name = var,
```

```
  extra_args = list()
)
```

Arguments

var	string, variable name
split_label	string. Label string to be associated with the table generated by the split. Not to be confused with labels assigned to each child (which are based on the data and type of split during tabulation).
labels_var	string, name of variable containing labels to be displayed for the values of var
cfun	list/function/NULL. tabulation function(s) for creating content rows. Must accept x or df as first parameter. Must accept labelstr as the second argument. Can optionally accept all optional arguments accepted by analysis functions. See analyze.
cformat	format spec. Format for content rows
cna_str	character. NA string for use with cformat for content table.
split_fun	function/NULL. custom splitting function See custom_split_funs
split_format	FormatSpec. Default format associated with the split being created.
split_na_str	character. NA string vector for use with split_format.
valorder	character vector. Order that the split children should appear in resulting table.
split_name	string. Name associated with this split (for pathing, etc)
child_labels	string. One of "default", "visible", "hidden". What should the display behavior be for the labels (i.e. label rows) of the children of this split. Defaults to "default" which flags the label row as visible only if the child has 0 content rows.
extra_args	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
label_pos	character(1). Location the variable label should be displayed, Accepts "hidden" (default for non-analyze row splits), "visible", "topleft", and - for analyze splits only - "default". For analyze calls, "default" indicates that the variable should be visible if and only if multiple variables are analyzed at the same level of nesting.
cindent_mod	numeric(1). The indent modifier for the content tables generated by this split.
cvar	character(1). The variable, if any, which the content function should accept. Defaults to NA.
cextra_args	list. Extra arguments to be passed to the content function when tabulating row group summaries.
page_prefix	character(1). Prefix, to be appended with the split value, when forcing pagination between the children of this split/table

section_div	character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.
ref_group	character. Value of var to be taken as the ref_group/control to be compared against.
label_fstr	string. An sprintf style format string containing. For non-comparison splits, it can contain up to one "%s" which takes the current split value and generates the row/column label. Comparison-based splits it can contain up to two "%s".

Value

a VarLevelSplit object.

Author(s)

Gabriel Becker

VarStaticCutSplit-class

Splits for cutting by values of a numeric variable

Description

Splits for cutting by values of a numeric variable

Create static cut or static cumulative cut split

Usage

```
make_static_cut_split(
  var,
  split_label,
  cuts,
  cutlabels = NULL,
  cfun = NULL,
  cformat = NULL,
  cna_str = NA_character_,
  split_format = NULL,
  split_na_str = NA_character_,
  split_name = var,
  child_labels = c("default", "visible", "hidden"),
  extra_args = list(),
  indent_mod = 0L,
  cindent_mod = 0L,
  cvar = "",
  cextra_args = list(),
  label_pos = "visible",
```

```
  cumulative = FALSE,
  page_prefix = NA_character_,
  section_div = NA_character_
)

VarDynCutSplit(
  var,
  split_label,
  cutfun,
  cutlabelfun = function(x) NULL,
  cfun = NULL,
  cformat = NULL,
  cna_str = NA_character_,
  split_format = NULL,
  split_na_str = NA_character_,
  split_name = var,
  child_labels = c("default", "visible", "hidden"),
  extra_args = list(),
  cumulative = FALSE,
  indent_mod = 0L,
  cindent_mod = 0L,
  cvar = "",
  cextra_args = list(),
  label_pos = "visible",
  page_prefix = NA_character_,
  section_div = NA_character_
)
```

Arguments

var	string, variable name
split_label	string. Label string to be associated with the table generated by the split. Not to be confused with labels assigned to each child (which are based on the data and type of split during tabulation).
cuts	numeric. Cuts to use
cutlabels	character (or NULL). Labels for the cuts
cfun	list/function/NULL. tabulation function(s) for creating content rows. Must accept x or df as first parameter. Must accept labelstr as the second argument. Can optionally accept all optional arguments accepted by analysis functions. See analyze.
cformat	format spec. Format for content rows
cna_str	character. NA string for use with cformat for content table.
split_format	FormatSpec. Default format associated with the split being created.
split_na_str	character. NA string vector for use with split_format.
split_name	string. Name associated with this split (for pathing, etc)

child_labels	string. One of "default", "visible", "hidden". What should the display behavior be for the labels (i.e. label rows) of the children of this split. Defaults to "default" which flags the label row as visible only if the child has 0 content rows.
extra_args	list. Extra arguments to be passed to the tabulation function. Element position in the list corresponds to the children of this split. Named elements in the child-specific lists are ignored if they do not match a formal argument of the tabulation function.
indent_mod	numeric. Modifier for the default indent position for the structure created by this function(subtable, content table, or row) *and all of that structure's children*. Defaults to 0, which corresponds to the unmodified default behavior.
cindent_mod	numeric(1). The indent modifier for the content tables generated by this split.
cvar	character(1). The variable, if any, which the content function should accept. Defaults to NA.
cextra_args	list. Extra arguments to be passed to the content function when tabulating row group summaries.
label_pos	character(1). Location the variable label should be displayed, Accepts "hidden" (default for non-analyze row splits), "visible", "topleft", and - for analyze splits only - "default". For analyze calls, "default" indicates that the variable should be visible if and only if multiple variables are analyzed at the same level of nesting.
cumulative	logical. Should the cuts be treated as cumulative. Defaults to FALSE
page_prefix	character(1). Prefix, to be appended with the split value, when forcing pagination between the children of this split/table
section_div	character(1). String which should be repeated as a section divider after each group defined by this split instruction, or NA_character_ (the default) for no section divider.
cutfun	function. Function which accepts the *full vector* of var values and returns cut points to be used (via cut) when splitting data during tabulation
cutlabelfun	function. Function which returns either labels for the cuts or NULL when passed the return value of cutfun

Value

a VarStaticCutSplit, CumulativeCutSplit object for make_static_cut_split, or a VarDynCutSplit object for VarDynCutSplit()

---

vars_in_layout *List Variables required by a pre-data table layout*

Description

List Variables required by a pre-data table layout

Usage

```
vars_in_layout(lyt)

## S4 method for signature 'PreDataTableLayouts'
vars_in_layout(lyt)

## S4 method for signature 'PreDataAxisLayout'
vars_in_layout(lyt)

## S4 method for signature 'SplitVector'
vars_in_layout(lyt)

## S4 method for signature 'Split'
vars_in_layout(lyt)

## S4 method for signature 'CompoundSplit'
vars_in_layout(lyt)

## S4 method for signature 'ManualSplit'
vars_in_layout(lyt)
```

Arguments

`lyt`	The Layout (or a component thereof)

Details

This will walk the layout declaration and return a vector of the names of the unique variables that are used in any of the following ways:

- Variable being split on (directly or via cuts)
- Element of a Multi-variable column split
- Content variable
- Value-label variable

Value

A character vector containing the unique variables explicitly used in the layout (see Notes).

Note

This function will not detect dependencies implicit in analysis or summary functions which accept `x` or `df` and then rely on the existence of particular variables not being split on/ analyzed.

The order these variable names appear within the return vector is undefined and should not be relied upon.

Examples

```
lyt <- basic_table() %>%
  split_cols_by("ARM") %>%
  split_cols_by("SEX") %>%
  summarize_row_groups(label_fstr = "Overall (N)") %>%
  split_rows_by("RACE",
    split_label = "Ethnicity", labels_var = "ethn_lab",
    split_fun = drop_split_levels
  ) %>%
  summarize_row_groups("RACE", label_fstr = "%s (n)") %>%
  analyze("AGE", var_labels = "Age", afun = mean, format = "xx.xx")

vars_in_layout(lyt)
```

Viewer *Display an rtable object in the Viewer pane in RStudio or in a browser*

Description

The table will be displayed using the bootstrap styling for tables.

Usage

```
Viewer(x, y = NULL, ...)
```

Arguments

x	object of class rtable or shiny.tag (defined in htmltools package)
y	optional second argument of same type as x
...	arguments passed to as_html

Value

not meaningful. Called for the side effect of opening a browser or viewer pane.

Examples

```
if (interactive()) {
  sl5 <- factor(iris$Sepal.Length > 5,
    levels = c(TRUE, FALSE),
    labels = c("S.L > 5", "S.L <= 5")
  )

  df <- cbind(iris, sl5 = sl5)
```

```
  lyt <- basic_table() %>%
    split_cols_by("sl5") %>%
    analyze("Sepal.Length")

  tbl <- build_table(lyt, df)

  Viewer(tbl)
  Viewer(tbl, tbl)


  tbl2 <- htmltools::tags$div(
    class = "table-responsive",
    as_html(tbl, class_table = "table")
  )

  Viewer(tbl, tbl2)
}
```

Index