Package 'Tplyr'

August 24, 2026

Title A Traceability Focused Grammar of Clinical Data Summary

Version 1.4.1

Description A traceability focused tool created to simplify the data manipulation necessary to create clinical summaries.

License MIT + file LICENSE

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

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

Encoding UTF-8

Depends R (>= 4.1.0)

Imports rlang (>= 0.4.6), assertthat (>= 0.2.1), magrittr (>= 1.5), dplyr (>= 1.0.0), purrr (>= 0.3.3), stringr (>= 1.4.0), tidyr (>= 1.0.2), tidyselect (>= 1.1.0), tibble (>= 3.0.1), lifecycle, forcats (>= 1.0.0)

Suggests testthat (>= 2.1.0), carrier (>= 0.2.0), haven (>= 2.2.0), knitr, mirai, rmarkdown, huxtable, tidyverse, readr, kableExtra, pharmaRTF, withr

VignetteBuilder knitr

RoxygenNote 7.3.2

RdMacros lifecycle

Config/testthat/edition 3

LazyData true

NeedsCompilation no

Author Eli Miller [aut] (ORCID: <https://orcid.org/0000-0002-2127-9456>), Mike Stackhouse [aut, cre] (ORCID: <https://orcid.org/0000-0001-6030-723X>), Ashley Tarasiewicz [aut], Nathan Kosiba [ctb] (ORCID: <https://orcid.org/0000-0001-5359-4234>), Sadchla Mascary [ctb], Andrew Bates [ctb],

Shiyu Chen [ctb],
Oleksii Mikryukov [ctb],
Atorus Research LLC [cph]

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

Repository CRAN

Date/Publication 2026-08-24 18:40:41 UTC

Contents

add_anti_join . . . 3
add_column_headers . . . 4
add_layer . . . 6
add_missing_subjects_row . . . 8
add_risk_diff . . . 8
add_total_row . . . 11
add_treat_grps . . . 12
add_variables . . . 13
append_metadata . . . 14
apply_conditional_format . . . 15
apply_formats . . . 16
apply_row_masks . . . 17
build . . . 18
collapse_row_labels . . . 19
f_str . . . 20
get_by . . . 23
get_data_labels . . . 23
get_desc_layer_formats . . . 24
get_metadata . . . 25
get_meta_result . . . 25
get_meta_subset . . . 26
get_numeric_data . . . 28
get_precision_by . . . 29
get_precision_on . . . 30
get_stats_data . . . 31
get_target_var . . . 32
get_tplyr_regex . . . 33
get_where.tplyr_layer . . . 34
group_count . . . 35
header_n . . . 37
keep_levels . . . 38
new_layer_template . . . 39
pop_data . . . 41
pop_treat_var . . . 42
replace_leading_whitespace . . . 42
set_custom_summaries . . . 43
set_denoms_by . . . 44
set_denom_ignore . . . 45

set_denom_where . . . 46
set_distinct_by . . . 47
set_format_strings . . . 48
set_indentation . . . 50
set_limit_data_by . . . 51
set_missing_count . . . 52
set_missing_subjects_row_label . . . 53
set_nest_count . . . 54
set_numeric_threshold . . . 54
set_order_count_method . . . 55
set_outer_sort_position . . . 58
set_precision_data . . . 59
set_stats_as_columns . . . 60
set_total_row_label . . . 61
str_extract_fmt_group . . . 62
str_indent_wrap . . . 63
Tplyr . . . 64
tplyr_adae . . . 66
tplyr_adas . . . 67
tplyr_adlb . . . 67
tplyr_adpe . . . 68
tplyr_adsl . . . 68
tplyr_layer . . . 69
tplyr_meta . . . 70
tplyr_table . . . 71
treat_var . . . 72

Index **74**

`add_anti_join` *Add an anti-join onto a tplyr_meta object*

Description

An anti-join allows a tplyr_meta object to refer to data that should be extracted from a separate dataset, like the population data of a Tplyr table, that is unavailable in the target dataset. The primary use case for this is the presentation of missing subjects, which in a Tplyr table is presented using the function `add_missing_subjects_row()`. The missing subjects themselves are not present in the target data, and are thus only available in the population data. The `add_anti_join()` function allows you to provide the meta information relevant to the population data, and then specify the `on` variable that should be used to join with the target dataset and find the values present in the population data that are missing from the target data.

Usage

```
add_anti_join(meta, join_meta, on)
```

Arguments

meta	A tplyr_meta object referring to the target data
join_meta	A tplyr_meta object referring to the population data
on	A list of quosures containing symbols - most likely set to USUBJID.

Value

A tplyr_meta object

Examples

```
tm <- tplyr_meta(
  rlang::quos(TRT01A, SEX, ETHNIC, RACE),
  rlang::quos(TRT01A == "Placebo", TRT01A == "SEX", ETHNIC == "HISPANIC OR LATINO")
)

tm %>%
  add_anti_join(
    tplyr_meta(
      rlang::quos(TRT01A, ETHNIC),
      rlang::quos(TRT01A == "Placebo", ETHNIC == "HISPANIC OR LATINO")
    ),
    on = rlang::quos(USUBJID)
  )
```

---

add_column_headers *Attach column headers to a Tplyr output*

Description

When working with 'huxtable' tables, column headers can be controlled as if they are rows in the data frame. add_column_headers eases the process of introducing these headers.

Usage

```
add_column_headers(.data, s, header_n = NULL)
```

Arguments

.data	The data.frame/tibble on which the headers shall be attached
s	The text containing the intended header string
header_n	A header_n or generic data.frame to use for binding count values. This is required if you are using the token replacement.

Details

Headers are created by providing a single string. Columns are specified by delimitting each header with a 'l' symbol. Instead of specifying the destination of each header, `add_column_headers` assumes that you have organized the columns of your data frame before hand. This means that after you use `Tplyr::build()`, if you'd like to reorganize the default column order (which is simply alphabetical), simply pass the build output to a `dplyr::select` or `dplyr::relocate` statement before passing into `add_column_headers`.

Spanning headers are also supported. A spanning header is an overarching header that sits across multiple columns. Spanning headers are introduced to `add_column_header` by providing the spanner text (i.e. the text that you'd like to sit in the top row), and then the spanned text (the bottom row) within curly brackets ('{}). For example, take the iris dataset. We have the names:

`"Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"`

If we wanted to provide a header string for this dataset, with spanners to help with categorization of the variables, we could provide the following string:

`"Sepal {Length | Width} | Petal {Length | Width} | Species"`

Value

A data.frame with the processed header string elements attached as the top rows

Important note

Make sure you are aware of the order of your variables prior to passing in to `add_column_headers`. The only requirement is that the number of column match. The rest is up to you.

Development notes

There are a few features of `add_column_header` that are intended but not yet supported:

- Nested spanners are not yet supported. Only a spanning row and a bottom row can currently be created
- Different delimiters and indicators for a spanned group may be used in the future. The current choices were intuitive, but based on feedback it could be determined that less common characters may be necessary.

Token Replacement

This function has support for reading values from the header_n object in a Tplyr table and adding them in the column headers. Note: The order of the parameters passed in the token is important. They should be first the treatment variable then any `cols` variables in the order they were passed in the table construction.

Use a double asterisk "**" at the begining to start the token and another double asterisk to close it. You can separate column parameters in the token with a single underscore. For example, **group1_flag2_param3** will pull the count from the header_n binding for group1 in the `treat_var`, flag2 in the first `cols` argument, and param3 in the second `cols` argument.

You can pass fewer arguments in the token to get the sum of multiple columns. For example, **group1** would get the sum of the group1 treat_var, and all cols from the header_n.

Examples

```
# Load in pipe
library(magrittr)
library(dplyr)
header_string <- "Sepal {Length | Width} | Petal {Length | Width} | Species"

iris2 <- iris %>%
  mutate_all(as.character)

iris2 %>% add_column_headers(header_string)

# Example with counts
mtcars2 <- mtcars %>%
  mutate_all(as.character)

t <- tplyr_table(mtcars2, vs, cols = am) %>%
  add_layer(
    group_count(cyl)
  )

b_t <- build(t) %>%
  mutate_all(as.character)

count_string <- paste0(" | V N=**0** {auto N=**0_0** | man N=**0_1**} |",
                       " S N=**1** {auto N=**1_0** | man N=**1_1**} | | ")

add_column_headers(b_t, count_string, header_n(t))
```

`add_layer` *Attach a layer to a* `tplyr_table` *object*

Description

`add_layer` attaches a `tplyr_layer` to a `tplyr_table` object. This allows for a tidy style of programming (using `magrittr` piping, i.e. `%>%`) with a secondary advantage - the construction of the layer object may consist of a series of piped functions itself.

`Tplyr` encourages a user to view the construction of a table as a series of "layers". The construction of each of these layers are isolated and independent of one another - but each of these layers are children of the table itself. `add_layer` isolates the construction of an individual layer and allows the user to construct that layer and insert it back into the parent. The syntax for this is intuitive and allows for tidy piping. Simply pipe the current table object in, and write the code to construct your layer within the `layer` parameter.

`add_layers` is another approach to attaching layers to a `tplyr_table`. Instead of constructing the entire table at once, `add_layers` allows you to construct layers as different objects. These layers can then be attached into the `tplyr_table` all at once.

`add_layer` and `add_layers` both additionally allow you to name the layers as you attach them. This is helpful when using functions like `get_numeric_data` or `get_stats_data` when you can access information from a layer directly. `add_layer` has a name parameter, and layers can be named in `add_layers` by submitting the layer as a named argument.

Usage

```
add_layer(parent, layer, name = NULL)

add_layers(parent, ...)
```

Arguments

parent	A tplyr_table or tplyr_layer/tplyr_subgroup_layer object
layer	A layer construction function and associated modifier functions
name	A name to provide the layer in the table layers container
...	Layers to be added

Value

A tplyr_table or tplyr_layer/tplyr_subgroup_layer with a new layer inserted into the layer binding

See Also

[tplyr_table(), tplyr_layer(), group_count(), group_desc(), group_shift()]

Examples

```
# Load in pipe
library(magrittr)

## Single layer
t <- tplyr_table(mtcars, cyl) %>%
  add_layer(
    group_desc(target_var=mpg)
  )

## Single layer with name
t <- tplyr_table(mtcars, cyl) %>%
  add_layer(name='mpg',
    group_desc(target_var=mpg)
  )

# Using add_layers
t <- tplyr_table(mtcars, cyl)
l1 <- group_desc(t, target_var=mpg)
l2 <- group_count(t, target_var=cyl)

t <- add_layers(t, l1, 'cyl' = l2)
```

add_missing_subjects_row *Add a missing subject row into a count summary.*

Description

This function calculates the number of subjects missing from a particular group of results. The calculation is done by examining the total number of subjects potentially available from the Header N values within the result column, and finding the difference with the total number of subjects present in the result group. Note that for accurate results, the subject variable needs to be defined using the 'set_distinct_by()' function. As with other methods, this function instructs how distinct results should be identified.

Usage

```
add_missing_subjects_row(e, fmt = NULL, sort_value = NULL)
```

Arguments

e	A 'count_layer' object
fmt	An f_str object used to format the total row. If none is provided, display is based on the layer formatting.
sort_value	The value that will appear in the ordering column for total rows. This must be a numeric value.

Examples

```
tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl) %>%
      add_missing_subjects_row(f_str("xxxx", n))
  ) %>%
  build()
```

add_risk_diff *Add risk difference to a count layer*

Description

A very common requirement for summary tables is to calculate the risk difference between treatment groups. `add_risk_diff` allows you to do this. The underlying risk difference calculations are performed using the Base R function `prop.test` - so prior to using this function, be sure to familiarize yourself with its functionality.

Usage

```
add_risk_diff(layer, ..., args = list(), distinct = TRUE)
```

Arguments

layer	Layer upon which the risk difference will be attached
...	Comparison groups, provided as character vectors where the first group is the comparison, and the second is the reference
args	Arguments passed directly into prop.test
distinct	Logical - Use distinct counts (if available).

Details

add_risk_diff can only be attached to a count layer, so the count layer must be constructed first. add_risk_diff allows you to compare the difference between treatment group, so all comparisons should be based upon the values within the specified treat_var in your tplyr_table object.

Comparisons are specified by providing two-element character vectors. You can provide as many of these groups as you want. You can also use groups that have been constructed using add_treat_grps or add_total_group. The first element provided will be considered the 'reference' group (i.e. the left side of the comparison), and the second group will be considered the 'comparison'. So if you'd like to see the risk difference of 'T1 - Placebo', you would specify this as c('T1', 'Placebo').

Tplyr forms your two-way table in the background, and then runs prop.test appropriately. Similar to way that the display of layers are specified, the exact values and format of how you'd like the risk difference display are set using set_format_strings. This controls both the values and the format of how the risk difference is displayed. Risk difference formats are set within set_format_strings by using the name 'riskdiff'.

You have 5 variables to choose from in your data presentation:

comp Probability of the left hand side group (i.e. comparison)

ref Probability of the right hand side group (i.e. reference)

dif Difference of comparison - reference

low Lower end of the confidence interval (default is 95%, override with the args paramter)

high Upper end of the confidence interval (default is 95%, override with the args paramter)

Use these variable names when forming your f_str objects. The default presentation, if no string format is specified, will be:

f_str('xx.xxx (xx.xxx, xx.xxx)', dif, low, high)

Note - within Tplyr, you can account for negatives by allowing an extra space within your integer side settings. This will help with your alignment.

If columns are specified on a Tplyr table, risk difference comparisons still only take place between groups within the treat_var variable - but they are instead calculated treating the cols variables as by variables. Just like the tplyr layers themselves, the risk difference will then be transposed and display each risk difference as separate variables by each of the cols variables.

If distinct is TRUE (the default), all calculations will take place on the distinct counts, if they are available. Otherwise, non-distinct counts will be used.

One final note - prop.test may throw quite a few warnings. This is natural, because it alerts you when there's not enough data for the approximations to be correct. This may be unnerving coming from a SAS programming world, but this is R is trying to alert you that the values provided don't have enough data to truly be statistically accurate.

Examples

```
library(magrittr)

## Two group comparisons with default options applied
t <- tplyr_table(mtcars, gear)

# Basic risk diff for two groups, using defaults
l1 <- group_count(t, carb) %>%
  # Compare 3 vs. 4, 3 vs. 5
  add_risk_diff(
    c('3', '4'),
    c('3', '5')
  )

# Build and show output
add_layers(t, l1) %>% build()

## Specify custom formats and display variables
t <- tplyr_table(mtcars, gear)

# Create the layer with custom formatting
l2 <- group_count(t, carb) %>%
  # Compare 3 vs. 4, 3 vs. 5
  add_risk_diff(
    c('3', '4'),
    c('3', '5')
  ) %>%
  set_format_strings(
    'n_counts' = f_str('xx (xx.x)', n, pct),
   'riskdiff' = f_str('xx.xxx, xx.xxx, xx.xxx, xx.xxx, xx.xxx', comp, ref, dif, low, high)
  )

# Build and show output
add_layers(t, l2) %>% build()

## Passing arguments to prop.test
t <- tplyr_table(mtcars, gear)

# Create the layer with args option
l3 <- group_count(t, carb) %>%
  # Compare 3 vs. 4, 4 vs. 5
  add_risk_diff(
    c('3', '4'),
    c('3', '5'),
    args = list(conf.level = 0.9, correct=FALSE, alternative='less')
  )
```

```
# Build and show output
add_layers(t, l3) %>% build()
```

`add_total_row` *Add a Total row into a count summary.*

Description

Adding a total row creates an additional observation in the count summary that presents the total counts (i.e. the n's that are summarized). The format of the total row will be formatted in the same way as the other count strings.

Usage

```
add_total_row(e, fmt = NULL, count_missings = TRUE, sort_value = NULL)
```

Arguments

`e`	A `count_layer` object
`fmt`	An f_str object used to format the total row. If none is provided, display is based on the layer formatting.
`count_missings`	Whether or not to ignore the named arguments passed in 'set_count_missing()' when calculating counts total row. This is useful if you need to exclude/include the missing counts in your total row. Defaults to TRUE meaning total row will not ignore any values.
`sort_value`	The value that will appear in the ordering column for total rows. This must be a numeric value.

Details

Totals are calculated using all grouping variables, including treat_var and cols from the table level. If by variables are included, the grouping of the total and the application of denominators becomes ambiguous. You will be warned specifically if a percent is included in the format. To rectify this, use `set_denoms_by()`, and the grouping of `add_total_row()` will be updated accordingly.

Note that when using `add_total_row()` with `set_pop_data()`, you should call `add_total_row()` AFTER calling `set_pop_data()`, otherwise there is potential for unexpected behaivior with treatment groups.

Examples

```
# Load in Pipe
library(magrittr)

tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl) %>%
```

```
   add_total_row(f_str("xxxx", n))
) %>%
build()
```

---

add_treat_grps *Combine existing treatment groups for summary*

Description

Summary tables often present individual treatment groups, but may additionally have a "Treatment vs. Placebo" or "Total" group added to show grouped summary statistics or counts. This set of functions offers an interface to add these groups at a table level and be consumed by subsequent layers.

Usage

```
add_treat_grps(table, ...)

add_total_group(table, group_name = "Total")

treat_grps(table)
```

Arguments

table	A tplyr_table object
...	A named vector where names will become the new treatment group names, and values will be used to construct those treatment groups
group_name	The treatment group name used for the constructed 'Total' group

Details

add_treat_grps allows you to specify specific groupings. This is done by supplying named arguments, where the name becomes the new treatment group's name, and those treatment groups are made up of the argument's values.

add_total_group is a simple wrapper around add_treat_grps. Instead of producing custom groupings, it produces a "Total" group by the supplied name, which defaults to "Total". This "Total" group is made up of all existing treatment groups within the population dataset.

Note that when using add_treat_grps or add_total_row() with set_pop_data(), you should call add_total_row() AFTER calling set_pop_data(), otherwise there is potential for unexpected behaivior with treatment groups.

The function treat_grps allows you to see the custom treatment groups available in your tplyr_table object

Value

The modified table object

Examples

```
tab <- tplyr_table(iris, Species)

# A custom group
add_treat_grps(tab, "Not Setosa" = c("versicolor", "virginica"))

# Add a total group
add_total_group(tab)

treat_grps(tab)
# Returns:
# $`Not Setosa`
#[1] "versicolor" "virginica"
#
#$Total
#[1] "setosa"     "versicolor" "virginica"
```

`add_variables` *Add variables to a tplyr_meta object*

Description

Add additional variable names to a `tplyr_meta()` object.

Usage

```
add_variables(meta, names)

add_filters(meta, filters)
```

Arguments

`meta`	A tplyr_meta object
`names`	A list of names, providing variable names of interest. Provide as a list of quosures using `rlang::quos()`
`filters`	A list of symbols, providing variable names of interest. Provide as a list of quosures using ‘rlang::quos()‘

Value

tplyr_meta object

Examples

```
m <- tplyr_meta()
m <- add_variables(m, rlang::quos(a, b, c))
m <- add_filters(m, rlang::quos(a==1, b==2, c==3))
m
```

append_metadata *Append the Tplyr table metadata dataframe*

Description

append_metadata() allows a user to extend the Tplyr metadata data frame with user provided data. In some tables, Tplyr may be able to provided most of the data, but a user may have to extend the table with other summaries, statistics, etc. This function allows the user to extend the tplyr_table's metadata with their own metadata content using custom data frames created using the tplyr_meta object.

Usage

```
append_metadata(t, meta)
```

Arguments

t	A tplyr_table object
meta	A dataframe fitting the specifications of the details section of this function

Details

As this is an advanced feature of Tplyr, ownership is on the user to make sure the metadata data frame is assembled properly. The only restrictions applied by append_metadata() are that meta must have a column named row_id, and the values in row_id cannot be duplicates of any row_id value already present in the Tplyr metadata dataframe. tplyr_meta() objects align with constructed dataframes using the row_id and output dataset column name. As such, tplyr_meta() objects should be inserted into a data frame using a list column.

Value

A tplyr_table object

Examples

```
t <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_desc(wt)
  )

t %>%
  build(metadata=TRUE)

m <- tibble::tibble(
  row_id = c('x1_1'),
  var1_3 = list(tplyr_meta(rlang::quos(a, b, c), rlang::quos(a==1, b==2, c==3)))
)

append_metadata(t, m)
```

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

Description

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

Usage

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

Arguments

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

Value

A character vector

Examples

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

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

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

`apply_formats` *Apply Format Strings outside of a Tplyr table*

Description

The `f_str` object in Tplyr is used to drive formatting of the outputs strings within a Tplyr table. This function allows a user to use the same interface to apply formatted string on any data frame within a `dplyr::mutate()` context.

Usage

```
apply_formats(format_string, ..., empty = c(.overall = ""))
```

Arguments

`format_string`	The desired display format. X's indicate digits. On the left, the number of x's indicates the integer length. On the right, the number of x's controls decimal precision and rounding. Variables are inferred by any separation of the 'x' values other than a decimal.
`...`	The variables to be formatted using the format specified in `format_string`. These must be numeric variables.
`empty`	The string to display when the numeric data is not available. Use a single element character vector, with the element named '.overall' to instead replace the whole string.

Details

Note that auto-precision is not currently supported within `apply_formats()`

Value

Character vector of formatted values

Examples

```
library(dplyr)

mtcars %>%
  head() %>%
  mutate(
    fmt_example = apply_formats('xxx (xx.x)', hp, wt)
  )
```

`apply_row_masks` *Replace repeating row label variables with blanks in preparation for display.*

Description

Depending on the display package being used, row label values may need to be blanked out if they are repeating. This gives the data frame supporting the table the appearance of the grouping variables being grouped together in blocks. `apply_row_masks` does this work by blanking out the value of any row_label variable where the current value is equal to the value before it. Note - `apply_row_masks` assumes that the data frame has already be sorted and therefore should only be applied once the data frame is in its final sort sequence.

Usage

```
apply_row_masks(dat, row_breaks = FALSE, ...)
```

Arguments

`dat`	Data.frame / tibble to mask repeating row_labels
`row_breaks`	Boolean - set to TRUE to insert row breaks
`...`	Variable used to determine where row-breaks should be inserted. Breaks will be inserted when this group of variables changes values. This is determined by dataset order, so sorting should be done prior to using `apply_row_masks`. If left empty, `ord_layer_index` will be used.

Details

Additionally, `apply_row_masks` can add row breaks for you between each layer. Row breaks are inserted as blank rows. This relies on the "break by" variables (submitted via `...`) constructed in `build` still being attached to the dataset. An additional order variable is attached named `ord_break`, but the output dataset is sorted to properly insert the row breaks between layers.

Value

tibble with blanked out rows where values are repeating

build *Trigger the execution of the* tplyr_table

Description

The functions used to assemble a tplyr_table object and each of the layers do not trigger the processing of any data. Rather, a lazy execution style is used to allow you to construct your table and then explicitly state when the data processing should happen. build triggers this event.

Usage

```
build(x, metadata = FALSE)
```

Arguments

x	A tplyr_table object
metadata	Trigger to build metadata. Defaults to FALSE

Details

When the build command is executed, all of the data processing commences. Any pre-processing necessary within the table environment takes place first. Next, each of the layers begins executing. Once the layers complete executing, the output of each layer is stacked into the resulting data frame.

Once this process is complete, any post-processing necessary within the table environment takes place, and the final output can be delivered. Metadata and traceability information are kept within each of the layer environments, which allows an investigation into the source of the resulting datapoints. For example, numeric data from any summaries performed is maintained and accessible within a layer using get_numeric_data.

The ‘metadata‘ option of build will trigger the construction of traceability metadata for the constructed data frame. Essentially, for every "result" that Tplyr produces, Tplyr can also generate the steps necessary to obtain the source data which produced that result from the input. For more information, see vignette("metadata").

Value

An executed tplyr_table

See Also

tplyr_table, tplyr_layer, add_layer, add_layers, layer_constructors

Examples

```
# Load in Pipe
library(magrittr)

tplyr_table(iris, Species) %>%
  add_layer(
    group_desc(Sepal.Length, by = "Sepal Length")
  ) %>%
  add_layer(
    group_desc(Sepal.Width, by = "Sepal Width")
  ) %>%
  build()
```

---

collapse_row_labels *Collapse row labels into a single column*

Description

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

Usage

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

Arguments

x	Input data frame
...	Row labels to be collapsed
indent	Indentation string to be used, which is multiplied at each indentation level
target_col	The desired name of the output column containing collapsed row labels

Value

data.frame with row labels collapsed into a single column

Examples

```
x <- tibble::tribble(
~row_label1, ~row_label2, ~row_label3, ~row_label4, ~var1,
  "A",         "C",         "G",         "M",        1L,
  "A",         "C",         "G",         "N",        2L,
  "A",         "C",         "H",         "O",        3L,
  "A",         "D",         "H",         "P",        4L,
  "A",         "D",         "I",         "Q",        5L,
  "A",         "D",         "I",         "R",        6L,
```

```
  "B",        "E",        "J",        "S",        7L,
  "B",        "E",        "J",        "T",        8L,
  "B",        "E",        "K",        "U",        9L,
  "B",        "F",        "K",        "V",        10L,
  "B",        "F",        "L",        "W",        11L
)

collapse_row_labels(x, row_label1, row_label2, row_label3, row_label4)

collapse_row_labels(x, row_label1, row_label2, row_label3)

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

`f_str` *Create a* `f_str` *object*

Description

`f_str` objects are intended to be used within the function `set_format_strings`. The `f_str` object carries information that powers a significant amount of layer processing. The `format_string` parameter is capable of controlling the display of a data point and decimal precision. The variables provided in `...` control which data points are used to populate the string formatted output.

Usage

```
f_str(format_string, ..., empty = c(.overall = ""))
```

Arguments

`format_string`	The desired display format. X's indicate digits. On the left, the number of x's indicates the integer length. On the right, the number of x's controls decimal precision and rounding. Variables are inferred by any separation of the 'x' values other than a decimal.
`...`	The variables to be formatted using the format specified in `format_string`.
`empty`	The string to display when the numeric data is not available. For desc layers, an unnamed character vector will populate within the provided format string, set to the same width as the fitted numbers. Use a single element character vector, with the element named '.overall' to instead replace the whole string.

Details

Format strings are one of the most powerful components of 'Tplyr'. Traditionally, converting numeric values into strings for presentation can consume a good deal of time. Values and decimals need to align between rows, rounding before trimming is sometimes forgotten - it can become a tedious mess that is realistically not an important part of the analysis being performed. 'Tplyr' makes this process as simple as we can, while still allowing flexibility to the user.

Tplyr provides both manual and automatic decimal precision formatting. The display of the numbers in the resulting data frame is controlled by the `format_string` parameter. For manual precision, just like dummy values may be presented on your mocks, integer and decimal precision is specified by the user providing a string of 'x's for how you'd like your numbers formatted. If you'd like 2 integers with 3 decimal places, you specify your string as 'xx.xxx'. 'Tplyr' does the work to get the numbers in the right place.

To take this a step further, automatic decimal precision can also be obtained based on the collected precision within the data. When creating tables where results vary by some parameter, different results may call for different degrees of precision. To use automatic precision, use a single 'a' on either the integer and decimal side. If you'd like to use increased precision (i.e. you'd like mean to be collected precision +1), use 'a+1'. So if you'd like both integer and and decimal precision to be based on the data as collected, you can use a format like 'a.a' - or for collected+1 decimal precision, 'a.a+1'. You can mix and match this with manual formats as well, making format strings such as 'xx.a+1'.

If you want two numbers on the same line, you provide two sets of x's. For example, if you're presenting a value like "mean (sd)" - you could provide the string 'xx.xx (xx.xxx)', or perhaps 'a.a+1 (a.a+2). Note that you're able to provide different integer lengths and different decimal precision for the two values. Each format string is independent and relates only to the format specified.

As described above, when using 'x' or 'a', any other character within the format string will stay stationary. So for example, if your format string is 'xx (xxx.x)', your number may format as '12 (34.5)'. So the left side parenthesis stays fixed. In some displays, you may want the parenthesis to 'hug' your number. Following this example, when allotting 3 spaces for the integer within parentheses, the parentehsis should shift to the right, making the numbers appear '12 (34.5)'. Using `f_str()` you can achieve this by using a capital 'X' or 'A'. For this example, the format string would be 'xx (XXX.x)'.

There are a two rules when using 'parenthesis hugging':

- Capital letters should only be used on the integer side of a number
- A character must precede the capital letter, otherwise there's no character to 'hug'

The other parameters of the `f_str` call specify what values should fill the x's. `f_str` objects are used slightly differently between different layers. When declaring a format string within a count layer, `f_str()` expects to see the values `n` or `distinct_n` for event or distinct counts, `pct` or `distinct_pct` for event or distinct percentages, or `total` or `distinct_total` for denominator calculations. Note that in an `f_str()` for a count layer 'A' or 'a' are based on n counts, and therefore don't make sense to use in percentages. But in descriptive statistic layers, `f_str` parameters refer to the names of the summaries being performed, either by built in defaults, or custom summaries declared using `set_custom_summaries()`. See `set_format_strings()` for some more notes about layers specific implementation.

An `f_str()` may also be used outside of a Tplyr table. The function `apply_formats()` allows you to apply an `f_str` within the context of `dplyr::mutate()` or more generally a vectorized function.

Value

A `f_str` object

Valid `f_str()` Variables by Layer Type

Valid variables allowed within the `...` parameter of `f_str()` differ by layer type.

- Count layers
  - `n`
  - `pct`
  - `total`
  - `distinct_n`
  - `distinct_pct`
  - `distinct_total`
- Shift layers
  - `n`
  - `pct`
  - `total`
- Desc layers
  - `n`
  - `mean`
  - `sd`
  - `median`
  - `var`
  - `min`
  - `max`
  - `iqr`
  - `q1`
  - `q3`
  - `missing`
  - Custom summaries created by `set_custom_summaries()`

Examples

```
f_str("xx.x (xx.x)", mean, sd)

f_str("a.a+1 (a.a+2)", mean, sd)

f_str("xx.a (xx.a+1)", mean, sd)

f_str("xx.x, xx.x, xx.x", q1, median, q3)

f_str("xx (XXX.x%)", n, pct)

f_str("a.a+1 (A.a+2)", mean, sd)
```

get_by *Set or return by layer binding*

Description

Set or return by layer binding

Usage

```
get_by(layer)

set_by(layer, by)
```

Arguments

layer	A tplyr_layer object
by	A string, a variable name, or a list of variable names supplied using dplyr::vars.

Value

For get_by, the by binding of the supplied layer. For set_by the modified layer environment.

Examples

```
# Load in pipe
library(magrittr)
iris$Species2 <- iris$Species
lay <- tplyr_table(iris, Species) %>%
  group_count(Species) %>%
  set_by(vars(Species2, Sepal.Width))
```

get_data_labels *Get Data Labels*

Description

Get labels for data sets included in Tplyr.

Usage

```
get_data_labels(data)
```

Arguments

data	A Tplyr data set.

Value

A data.frame with columns ‘name‘ and ‘label‘ containing the names and labels of each column.

`get_desc_layer_formats` *Get or set the default format strings for descriptive statistics layers*

Description

Tplyr provides you with the ability to set table-wide defaults of format strings. You may wish to reuse the same format strings across numerous layers. `set_desc_layer_formats` and `set_count_layer_formats` allow you to apply your desired format strings within the entire scope of the table.

Usage

```
get_desc_layer_formats(obj)

set_desc_layer_formats(obj, ...)

get_count_layer_formats(obj)

set_count_layer_formats(obj, ...)

get_shift_layer_formats(obj)

set_shift_layer_formats(obj, ...)
```

Arguments

`obj`	A tplyr_table object
`...`	formats to pass forward

Details

For descriptive statistic layers, you can also use `set_format_strings` and `set_desc_layer_formats` together within a table, but not within the same layer. In the absence of specified format strings, first the table will be checked for any available defaults, and otherwise the `tplyr.desc_layer_default_formats` option will be used. `set_format_strings` will always take precedence over either. Defaults cannot be combined between `set_format_strings`, `set_desc_layer_formats`, and the `tplyr.desc_layer_default_formats` because the order of presentation of results is controlled by the format strings, so relying on combinations of these setting would not be intuitive.

For count layers, you can override the `n_counts` or `riskdiff` format strings separately, and the narrowest scope available will be used from layer, to table, to default options.

get_metadata *Get the metadata dataframe from a tplyr_table*

Description

Pull out the metadata dataframe from a tplyr_table to work with it directly

Usage

```
get_metadata(t)
```

Arguments

t	A Tplyr table with metadata built

Value

Tplyr metadata dataframe

Examples

```
t <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_desc(wt)
  )

t %>%
  build(metadata=TRUE)

get_metadata(t)
```

get_meta_result *Extract the result metadata of a Tplyr table*

Description

Given a row_id value and a result column, this function will return the tplyr_meta object associated with that 'cell'.

Usage

```
get_meta_result(x, row_id, column, ...)
```

Arguments

x	A built Tplyr table or a dataframe
row_id	The row_id value of the desired cell, provided as a character string
column	The result column of interest, provided as a character string
...	additional arguments

Details

If a Tplyr table is built with the `metadata=TRUE` option specified, then metadata is assembled behind the scenes to provide traceability on each result cell derived. The functions `get_meta_result()` and `get_meta_subset()` allow you to access that metadata by using an ID provided in the row_id column and the column name of the result you'd like to access. The purpose is of the row_id variable instead of a simple row index is to provide a sort resistant reference of the originating column, so the output Tplyr table can be sorted in any order but the metadata are still easily accessible.

The `tplyr_meta` object provided a list with two elements - names and filters. The metadata contain every column from the target data.frame of the Tplyr table that factored into the specified result cell, and the filters contains all the necessary filters to subset to data summarized to create the specified result cell. `get_meta_subset()` additionally provides a parameter to specify any additional columns you would like to include in the returned subset data frame.

Value

A tplyr_meta object

Examples

```
t <- tplyr_table(mtcars, cyl) %>%
  add_layer(
    group_desc(hp)
  )

dat <- t %>% build(metadata = TRUE)

get_meta_result(t, 'd1_1', 'var1_4')

m <- t$metadata
dat <- t$target

get_meta_result(t, 'd1_1', 'var1_4')
```

---

get_meta_subset *Extract the subset of data based on result metadata*

Description

Given a row_id value and a result column, this function will return the subset of data referenced by the tplyr_meta object associated with that 'cell', which provides traceability to tie a result to its source.

Usage

```
get_meta_subset(x, row_id, column, add_cols = vars(USUBJID), ...)

## S3 method for class 'data.frame'
get_meta_subset(
  x,
  row_id,
  column,
  add_cols = vars(USUBJID),
  target = NULL,
  pop_data = NULL,
  ...
)

## S3 method for class 'tplyr_table'
get_meta_subset(x, row_id, column, add_cols = vars(USUBJID), ...)
```

Arguments

x	A built Tplyr table or a dataframe
row_id	The row_id value of the desired cell, provided as a character string
column	The result column of interest, provided as a character string
add_cols	Additional columns to include in subset data.frame output
...	additional arguments
target	A data frame to be subset (if not pulled from a Tplyr table)
pop_data	A data frame to be subset through an anti-join (if not pulled from a Tplyr table)

Details

If a Tplyr table is built with the `metadata=TRUE` option specified, then metadata is assembled behind the scenes to provide traceability on each result cell derived. The functions `get_meta_result()` and `get_meta_subset()` allow you to access that metadata by using an ID provided in the row_id column and the column name of the result you'd like to access. The purpose is of the row_id variable instead of a simple row index is to provide a sort resistant reference of the originating column, so the output Tplyr table can be sorted in any order but the metadata are still easily accessible.

The `tplyr_meta` object provided a list with two elements - names and filters. The metadata contain every column from the target data.frame of the Tplyr table that factored into the specified result cell, and the filters contains all the necessary filters to subset to data summarized to create the specified result cell. `get_meta_subset()` additionally provides a parameter to specify any additional columns you would like to include in the returned subset data frame.

Value

A data.frame

Examples

```
t <- tplyr_table(mtcars, cyl) %>%
  add_layer(
    group_desc(hp)
  )


dat <- t %>% build(metadata = TRUE)

get_meta_subset(t, 'd1_1', 'var1_4', add_cols = dplyr::vars(carb))

m <- t$metadata
dat <- t$target

get_meta_subset(t, 'd1_1', 'var1_4', add_cols = dplyr::vars(carb), target = target)
```

---

get_numeric_data *Retrieve the numeric data from a tplyr objects*

Description

`get_numeric_data` provides access to the un-formatted numeric data for each of the layers within a `tplyr_table`, with options to allow you to extract distinct layers and filter as desired.

Usage

```
get_numeric_data(x, layer = NULL, where = TRUE, ...)
```

Arguments

`x`	A tplyr_table or tplyr_layer object
`layer`	Layer name or index to select out specifically
`where`	Subset criteria passed to dplyr::filter
`...`	Additional arguments to pass forward

Details

When used on a `tplyr_table` object, this method will aggregate the numeric data from all Tplyr layers. The data will be returned to the user in a list of data frames. If the data has already been processed (i.e. `build` has been run), the numeric data is already available and will be returned without reprocessing. Otherwise, the numeric portion of the layer will be processed.

Using the layer and where parameters, data for a specific layer can be extracted and subset. This is most clear when layers are given text names instead of using a layer index, but a numeric index works as well.

Value

Numeric data from the Tplyr layer

Examples

```
# Load in pipe
library(magrittr)

t <- tplyr_table(mtcars, gear) %>%
 add_layer(name='drat',
           group_desc(drat)
 ) %>%
 add_layer(name='cyl',
           group_count(cyl)
 )

 # Return a list of the numeric data frames
 get_numeric_data(t)

 # Get the data from a specific layer
 get_numeric_data(t, layer='drat')
 get_numeric_data(t, layer=1)

 # Choose multiple layers by name or index
 get_numeric_data(t, layer=c('cyl', 'drat'))
 get_numeric_data(t, layer=c(2, 1))

 # Get the data and filter it
 get_numeric_data(t, layer='drat', where = gear==3)
```

`get_precision_by` *Set or return precision_by layer binding*

Description

The precision_by variables are used to collect the integer and decimal precision when auto-precision is used. These by variables are used to group the input data and identify the maximum precision available within the dataset for each by group. The precision_by variables must be a subset of the by variables

Usage

```
get_precision_by(layer)

set_precision_by(layer, precision_by)
```

Arguments

`layer`	A `tplyr_layer` object
`precision_by`	A string, a variable name, or a list of variable names supplied using `dplyr::vars`.

Value

For `get_precision_by`, the precision_by binding of the supplied layer. For `set_precision_by` the modified layer environment.

Examples

```
# Load in pipe
library(magrittr)
lay <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_desc(mpg, by=vars(carb, am)) %>%
    set_precision_by(carb)
  )
```

`get_precision_on` *Set or return precision_on layer binding*

Description

The precision_on variable is the variable used to establish numeric precision. This variable must be included in the list of `target_var` variables.

Usage

```
get_precision_on(layer)

set_precision_on(layer, precision_on)
```

Arguments

`layer`	A `tplyr_layer` object
`precision_on`	A string, a variable name, or a list of variable names supplied using `dplyr::vars`.

Value

For `get_precision_on`, the precision_on binding of the supplied layer. For `set_precision_on` the modified layer environment.

Examples

```
# Load in pipe
library(magrittr)
lay <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_desc(vars(mpg, disp), by=vars(carb, am)) %>%
    set_precision_on(disp)
  )
```

`get_stats_data` *Get statistics data*

Description

Like the layer numeric data, Tplyr also stores the numeric data produced from statistics like risk difference. This helper function gives you access to obtain that data from the environment

Usage

```
get_stats_data(x, layer = NULL, statistic = NULL, where = TRUE, ...)
```

Arguments

`x`	A tplyr_table or tplyr_layer object
`layer`	Layer name or index to select out specifically
`statistic`	Statistic name or index to select
`where`	Subset criteria passed to dplyr::filter
`...`	Additional arguments passed to dispatch

Details

When used on a `tplyr_table` object, this method will aggregate the numeric data from all Tplyr layers and calculate all statistics. The data will be returned to the user in a list of data frames. If the data has already been processed (i.e. `build` has been run), the numeric data is already available and the statistic data will simply be returned. Otherwise, the numeric portion of the layer will be processed.

Using the layer, where, and statistic parameters, data for a specific layer statistic can be extracted and subset, allowing you to directly access data of interest. This is most clear when layers are given text names instead of using a layer index, but a numeric index works as well. If just a statistic is specified, that statistic will be collected and returned in a list of data frames, allowing you to grab, for example, just the risk difference statistics across all layers.

Value

The statistics data of the supplied layer

Examples

```
library(magrittr)

t <- tplyr_table(mtcars, gear) %>%
  add_layer(name='drat',
            group_desc(drat)
  ) %>%
  add_layer(name="cyl",
            group_count(cyl)
```

```
 ) %>%
 add_layer(name="am",
           group_count(am) %>%
             add_risk_diff(c('4', '3'))
 ) %>%
 add_layer(name="carb",
           group_count(carb) %>%
             add_risk_diff(c('4', '3'))
 )

# Returns a list of lists, containing stats data from each layer
get_stats_data(t)

# Returns just the riskdiff statistics from each layer - NULL
# for layers without riskdiff
get_stats_data(t, statistic="riskdiff")

# Return the statistic data for just the "am" layer - a list
get_stats_data(t, layer="am")
get_stats_data(t, layer=3)

# Return the statistic data for just the "am" and "cyl", layer - a
# list of lists
get_stats_data(t, layer=c("am", "cyl"))
get_stats_data(t, layer=c(3, 2))

# Return just the statistic data for "am" and "cyl" - a list
get_stats_data(t, layer=c("am", "cyl"), statistic="riskdiff")
get_stats_data(t, layer=c(3, 2), statistic="riskdiff")


# Return the riskdiff for the "am" layer - a data frame
get_stats_data(t, layer="am", statistic="riskdiff")

# Return and filter the riskdiff for the am layer - a data frame
get_stats_data(t, layer="am", statistic="riskdiff", where = summary_var==1)
```

`get_target_var` *Set or return treat_var binding*

Description

Set or return treat_var binding

Usage

```
get_target_var(layer)

set_target_var(layer, target_var)
```

Arguments

layer	A tplyr_layer object
target_var	A symbol to perform the analysis on

Value

For treat_var, the treatment variable binding of the layer object. For set_treat_var, the modified layer environment.

Examples

```
# Load in pipe
library(magrittr)
iris$Species2 <- iris$Species
lay <- tplyr_table(iris, Species) %>%
  group_count(Species) %>%
  set_target_var(Species2)
```

---

get_tplyr_regex *Retrieve one of Tplyr's regular expressions*

Description

This function allows you to extract important regular expressions used inside Tplyr.

Usage

```
get_tplyr_regex(rx = c("format_string", "format_group", "number_group"))
```

Arguments

rx	A character string with either the value 'format_string' or 'format_group'

Details

There are three important regular expressions used within Tplyr. The format_string expression is the expression to parse format strings. This is what is used to make sense out of strings like 'xx (XX.x%)' or 'a+1 (A.a+2)' by inferring what the user is specifying about number formatting.

The 'format_group' regex is the opposite of this, and when given a string of numbers, such as ' 5 (34%) [9]' will return the separate segments of numbers broken into their format groups, which in this example would be ' 5', '(34%)', and '[9]'. Lastly, the 'number_group' regex has a similar application to the 'format_group' regex, but targets only numbers

Value

A regular expression object

Examples

```
get_tplyr_regex('format_string')

get_tplyr_regex('format_group')

get_tplyr_regex('number_group')
```

`get_where.tplyr_layer` *Set or return where binding for layer or table*

Description

Set or return where binding for layer or table

Usage

```
## S3 method for class 'tplyr_layer'
get_where(obj)

## S3 method for class 'tplyr_layer'
set_where(obj, where)

get_where(obj)

## S3 method for class 'tplyr_table'
get_where(obj)

set_where(obj, where)

## S3 method for class 'tplyr_table'
set_where(obj, where)

set_pop_where(obj, where)

get_pop_where(obj)
```

Arguments

`obj`	A `tplyr_layer` or `tplyr_table` object.
`where`	An expression (i.e. syntax) to be used to subset the data. Supply as programming logic (i.e. x < 5 & y == 10)

Value

For `where`, the where binding of the supplied object. For `set_where`, the modified object

Examples

```
# Load in pipe
library(magrittr)

iris$Species2 <- iris$Species
lay <- tplyr_table(iris, Species) %>%
  group_count(Species) %>%
  set_where(Petal.Length > 3) %>%
  # Set logic for pop_data as well
  set_pop_where(Petal.Length > 3)
```

`group_count` *Create a* `count`*,* `desc`*, or* `shift` *layer for discrete count based summaries, descriptive statistics summaries, or shift count summaries*

Description

This family of functions specifies the type of summary that is to be performed within a layer. `count` layers are used to create summary counts of some discrete variable. `desc` layers create summary statistics, and `shift` layers summaries the counts of different changes in states. See the "details" section below for more information.

Usage

```
group_count(parent, target_var, by = vars(), where = TRUE, ...)

group_desc(parent, target_var, by = vars(), where = TRUE, ...)

group_shift(parent, target_var, by = vars(), where = TRUE, ...)
```

Arguments

`parent`	Required. The parent environment of the layer. This must be the `tplyr_table` object that the layer is contained within.
`target_var`	Symbol. Required, The variable name(s) on which the summary is to be performed. Must be a variable within the target dataset. Enter unquoted - i.e. target_var = AEBODSYS. You may also provide multiple variables with `vars`.
`by`	A string, a variable name, or a list of variable names supplied using `vars`
`where`	Call. Filter logic used to subset the target data when performing a summary.
`...`	Additional arguments to pass forward

Details

Count Layers Count layers allow you to create summaries based on counting values with a variable. Additionally, this layer allows you to create n (%) summaries where you're also summarizing the proportion of instances a value occurs compared to some denominator. Count layers are also capable of producing counts of nested relationships. For example, if you want to produce counts of an overall outside group, and then the subgroup counts within that group, you can specify the target variable as vars(OutsideVariable, InsideVariable). This allows you to do tables like Adverse Events where you want to see the Preferred Terms within Body Systems, all in one layer. Further control over denominators is available using the function `set_denoms_by` and distinct counts can be set using `set_distinct_by`

Descriptive Statistics Layers Descriptive statistics layers perform summaries on continuous variables. There are a number of summaries built into Tplyr already that you can perform, including n, mean, median, standard deviation, variance, min, max, inter-quartile range, Q1, Q3, and missing value counts. From these available summaries, the default presentation of a descriptive statistic layer will output 'n', 'Mean (SD)', 'Median', 'Q1, Q3', 'Min, Max', and 'Missing'. You can change these summaries using `set_format_strings`, and you can also add your own summaries using `set_custom_summaries`. This allows you to implement any additional summary statistics you want presented.

Shift Layers A shift layer displays an endpoint's 'shift' throughout the duration of the study. It is an abstraction over the count layer, however we have provided an interface that is more efficient and intuitive. Targets are passed as named symbols using `dplyr::vars`. Generally the baseline is passed with the name 'row' and the shift is passed with the name 'column'. Both counts (n) and percentages (pct) are supported and can be specified with the `set_format_strings` function. To allow for flexibility when defining percentages, you can define the denominator using the `set_denoms_by` function. This function takes variable names and uses those to determine the denominator for the counts.

Value

An `tplyr_layer` environment that is a child of the specified parent. The environment contains the object as listed below.

A `tplyr_layer` object

See Also

[add_layer, add_layers, tplyr_table, tplyr_layer]

Examples

```
# Load in pipe
library(magrittr)

t <- tplyr_table(iris, Species) %>%
  add_layer(
    group_desc(target_var=Sepal.Width)
  )

t <- tplyr_table(iris, Species) %>%
```

```
  add_layer(
    group_desc(target_var=Sepal.Width)
  )

t <- tplyr_table(mtcars, am) %>%
  add_layer(
    group_shift(vars(row=gear, column=carb), by=cyl)
  )
```

---

header_n *Return or set header_n binding*

Description

The 'header_n()' functions can be used to automatically pull the header_n derivations from the table or change them for future use.

Usage

```
header_n(table)

header_n(x) <- value

set_header_n(table, value)
```

Arguments

table	A tplyr_table object
x	A tplyr_table object
value	A data.frame with columns with the treatment variable, column variabes, and a variable with counts named 'n'.
header_n	A data.frame with columns with the treatment variable, column variabes, and a variable with counts named 'n'.

Details

The 'header_n' object is created by Tplyr when a table is built and intended to be used by the 'add_column_headers()' function when displaying table level population totals. These methods are intended to be used for calling the population totals calculated by Tplyr, and to overwrite them if a user chooses to.

If you have a need to change the header Ns that appear in your table headers, say you know you are working with a subset of the data that doesn't represent the totals, you can replace the data used with 'set_header_n()'.

Value

For tplyr_header_n the header_n binding of the tplyr_table object. For tplyr_header_n<- and set_tplyr_header_n the modified object.

Examples

```
tab <- tplyr_table(mtcars, gear)

header_n(tab) <- data.frame(
  gear = c(3, 4, 5),
  n = c(10, 15, 45)
)
```

`keep_levels` *Select levels to keep in a count layer*

Description

In certain cases you only want a layer to include certain values of a factor. The 'keep_levels()' function allows you to pass character values to be included in the layer. The others are ignored. **NOTE: Denominator calculation is unaffected by this function, see the examples on how to include this logic in your percentages'**

Usage

```
keep_levels(e, ...)
```

Arguments

`e`	A `count_layer` object
`...`	Character values to count int he layer

Value

The modified Tplyr layer object

Examples

```
library(dplyr)
mtcars <- mtcars %>%
  mutate_all(as.character)

t <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl) %>%
      keep_levels("4", "8") %>%
      set_denom_where(cyl %in% c("4", "8"))
 ) %>%
 build()
```

new_layer_template *Create, view, extract, remove, and use Tplyr layer templates*

Description

There are several scenarios where a layer template may be useful. Some tables, like demographics tables, may have many layers that will all essentially look the same. Categorical variables will have the same count layer settings, and continuous variables will have the same desc layer settings. A template allows a user to build those settings once per layer, then reference the template when the Tplyr table is actually built.

Usage

```
new_layer_template(name, template)

remove_layer_template(name)

get_layer_template(name)

get_layer_templates()

use_template(name, ..., add_params = NULL)
```

Arguments

name	Template name
template	Template layer syntax, starting with a layer constructor group_count|desc|shift. This function should be called with an ellipsis argument (i.e. group_count(...)).
...	Arguments passed directly into a layer constructor, matching the target, by, and where parameters.
add_params	Additional parameters passed into layer modifier functions. These arguments are specified in a template within curly brackets such as {param}. Supply as a named list, where the element name is the parameter.

Details

This suite of functions allows a user to create and use layer templates. Layer templates allow a user to pre-build and reuse an entire layer configuration, from the layer constructor down to all modifying functions. Furthermore, users can specify parameters they may want to be interchangeable. Additionally, layer templates are extensible, so a template can be use and then further extended with additional layer modifying functions.

Layers are created using new_layer_template(). To use a layer, use the function use_template() in place of group_count|desc|shift(). If you want to view a specific template, use get_layer_template(). If you want to view all templates, use get_layer_templates(). And to remove a layer template use remove_layer_template(). Layer templates themselves are stored in the option tplyr.layer_templates, but a user should not access this directly and instead use the Tplyr supplied functions.

When providing the template layer syntax, the layer must start with a layer constructor. These are one of the function `group_count()`, `group_desc()`, or `group_shift()`. Instead of passing arguments into these function, templates are specified using an ellipsis in the constructor, i.e. `group_count(...)`. This is required, as after the template is built a user supplies these arguments via `use_template()`

`use_template()` takes the `group_count|desc|shift()` arguments by default. If a user specified additional arguments in the template, these are provided in a list throught the argument `add_params`. Provide these arguments exactly as you would in a normal layer. When creating the template, these parameters can be specified by using curly brackets. See the examples for details.

Examples

```
op <- options()

new_layer_template(
  "example_template",
  group_count(...) %>%
    set_format_strings(f_str('xx (xx%)', n, pct))
)

get_layer_templates()

get_layer_template("example_template")

tplyr_table(mtcars, vs) %>%
  add_layer(
    use_template("example_template", gear)
  ) %>%
  build()

remove_layer_template("example_template")

new_layer_template(
  "example_template",
  group_count(...) %>%
    set_format_strings(f_str('xx (xx%)', n, pct)) %>%
    set_order_count_method({sort_meth}) %>%
    set_ordering_cols({sort_cols})
)

get_layer_template("example_template")

tplyr_table(mtcars, vs) %>%
  add_layer(
    use_template("example_template", gear, add_params =
                   list(
                     sort_meth = "bycount",
                     sort_cols = `1`
                   ))
  ) %>%
  build()
```

```
remove_layer_template("example_template")

options(op)
```

---

pop_data *Return or set population data bindings*

Description

The population data is used to gather information that may not be available from the target dataset. For example, missing treatment groups, population N counts, and proper N counts for denominators will be provided through the population dataset. The population dataset defaults to the target dataset unless otherwise specified using `set_pop_data`.

Usage

```
pop_data(table)

pop_data(x) <- value

set_pop_data(table, pop_data)
```

Arguments

`table`	A `tplyr_table` object
`x`	A `tplyr_table` object
`value`	A data.frame with population level information
`pop_data`	A data.frame with population level information

Value

For `tplyr_pop_data` the pop_data binding of the `tplyr_table` object. For `tplyr_pop_data<-` nothing is returned, the pop_data binding is set silently. For `set_tplyr_pop_data` the modified object.

Examples

```
tab <- tplyr_table(iris, Species)

pop_data(tab) <- mtcars

tab <- tplyr_table(iris, Species) %>%
  set_pop_data(mtcars)
```

pop_treat_var *Return or set pop_treat_var binding*

Description

The treatment variable used in the target data may be different than the variable within the population dataset. `set_pop_treat_var` allows you to change this.

Usage

```
pop_treat_var(table)

set_pop_treat_var(table, pop_treat_var)
```

Arguments

`table`	A `tplyr_table` object
`pop_treat_var`	Variable containing treatment group assignments within the `pop_data` binding. Supply unquoted.

Value

For `tplyr_pop_treat_var` the pop_treat_var binding of the `tplyr_table` object. For `set_tplyr_pop_treat_var` the modified object.

Examples

```
tab <- tplyr_table(iris, Species)

pop_data(tab) <- mtcars
set_pop_treat_var(tab, mpg)
```

replace_leading_whitespace *Reformat strings with leading whitespace for HTML*

Description

Reformat strings with leading whitespace for HTML

Usage

```
replace_leading_whitespace(x, tab_width = 4)
```

Arguments

x	Target string
tab_width	Number of spaces to compensate for tabs

Value

String with   replaced for leading whitespace

Examples

```
x <- c(" Hello there", "  Goodbye Friend ",  "\tNice to meet you",
"  \t What are you up to? \t \t ")
replace_leading_whitespace(x)

replace_leading_whitespace(x, tab=2)
```

---

set_custom_summaries *Set custom summaries to be performed within a descriptive statistics layer*

Description

This function allows a user to define custom summaries to be performed in a call to `dplyr::summarize()`. A custom summary by the same name as a default summary will override the default. This allows the user to override the default behavior of summaries built into 'Tplyr', while also adding new desired summary functions.

Usage

```
set_custom_summaries(e, ...)
```

Arguments

e	`desc` layer on which the summaries should be bound
...	Named parameters containing syntax to be used in a call to `dplyr::summarize()`

Details

When programming the logic of the summary function, use the variable name `.var` to within your summary functions. This allows you apply the summary function to each variable when multiple target variables are declared.

An important, yet not immediately obvious, part of using `set_custom_summaries` is to understand the link between the named parameters you set in `set_custom_summaries` and the names called in `f_str` objects within `set_format_strings`. In `f_str`, after you supply the string format you'd like your numbers to take, you specify the summaries that fill those strings.

When you go to set your format strings, the name you use to declare a summary in `set_custom_summaries` is the same name that you use in your `f_str` call. This is necessary because `set_format_strings` needs some means of putting two summaries in the same value, and setting a row label for the summary being performed.

Review the examples to see this put into practice. Note the relationship between the name created in `set_custom_summaries` and the name used in `set_format_strings` within the `f_str` call

Value

Binds a variable `custom_summaries` to the specified layer

Examples

```
#Load in pipe
library(magrittr)

tplyr_table(iris, Species) %>%
  add_layer(
    group_desc(Sepal.Length, by = "Sepal Length") %>%
      set_custom_summaries(
        geometric_mean = exp(sum(log(.var[.var > 0]),
                                 na.rm=TRUE) / length(.var))
      ) %>%
      set_format_strings(
        'Geometric Mean' = f_str('xx.xx', geometric_mean)
      )
  ) %>%
  build()
```

---

set_denoms_by *Set variables used in pct denominator calculation*

Description

This function is used when calculating pct in count or shift layers. The percentages default to the treatment variable and any column variables but can be calculated on any variables passed to target_var, treat_var, by, or cols.

Usage

```
set_denoms_by(e, ...)
```

Arguments

`e`	A count/shift layer object
`...`	Unquoted variable names

Value

The modified layer object

Examples

```
library(magrittr)

# Default has matrix of treatment group, additional columns,
# and by variables sum to 1
tplyr_table(mtcars, am) %>%
  add_layer(
    group_shift(vars(row=gear, column=carb), by=cyl) %>%
      set_format_strings(f_str("xxx (xx.xx%)", n, pct))
  ) %>%
  build()

tplyr_table(mtcars, am) %>%
  add_layer(
    group_shift(vars(row=gear, column=carb), by=cyl) %>%
      set_format_strings(f_str("xxx (xx.xx%)", n, pct)) %>%
      set_denoms_by(cyl, gear) # Row % sums to 1
  ) %>%
  build()

tplyr_table(mtcars, am) %>%
  add_layer(
    group_shift(vars(row=gear, column=carb), by=cyl) %>%
      set_format_strings(f_str("xxx (xx.xx%)", n, pct)) %>%
      set_denoms_by(cyl, gear, am) # % within treatment group sums to 1
  ) %>%
  build()
```

`set_denom_ignore` *Set values the denominator calculation will ignore*

Description

‘r lifecycle::badge("defunct")‘

This is generally used for missing values. Values like "", NA, "NA" are common ways missing values are presented in a data frame. In certain cases, percentages do not use "missing" values in the denominator. This function notes different values as "missing" and excludes them from the denominators.

Usage

```
set_denom_ignore(e, ...)
```

Arguments

e	A count_layer object
...	Values to exclude from the percentage calculation. If you use 'set_missing_counts()' this should be the name of the parameters instead of the values, see the example below.

Value

The modified layer object

Examples

```
library(magrittr)
mtcars2 <- mtcars
mtcars2[mtcars$cyl == 6, "cyl"] <- NA
mtcars2[mtcars$cyl == 8, "cyl"] <- "Not Found"

tplyr_table(mtcars2, gear) %>%
  add_layer(
    group_count(cyl) %>%
      set_missing_count(f_str("xx ", n), Missing = c(NA, "Not Found"))
      # This function is currently deprecated. It was replaced with an
      # argument in set_missing_count
      # set_denom_ignore("Missing")
  ) %>%
  build()
```

---

set_denom_where *Set Logic for denominator subsetting*

Description

By default, denominators in count layers are subset based on the layer level where logic. In some cases this might not be correct. This functions allows the user to override this behavior and pass custom logic that will be used to subset the target dataset when calculating denominators for the layer.

Usage

```
set_denom_where(e, denom_where)
```

Arguments

e	A count_layer/shift_layer object
denom_where	An expression (i.e. syntax) to be used to subset the target dataset for calculating layer denominators. Supply as programming logic (i.e. x < 5 & y == 10). To remove the layer where parameter subsetting for the total row and thus the percentage denominators, pass 'TRUE' to this function.

Value

The modified Tplyr layer object

Examples

```
library(magrittr)
t10 <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl, where = cyl != 6) %>%
    set_denom_where(TRUE)
    # The denominators will be based on all of the values, including 6
  ) %>%
 build()
```

---

set_distinct_by *Set counts to be distinct by some grouping variable.*

Description

In some situations, count summaries may want to see distinct counts by a variable like subject. For example, the number of subjects in a population who had a particular adverse event. `set_distinct_by` allows you to set the by variables used to determine a distinct count.

Usage

```
set_distinct_by(e, distinct_by)
```

Arguments

`e`	A `count_layer/shift_layer` object
`distinct_by`	Variable(s) to get the distinct data.

Details

When a `distinct_by` value is set, distinct counts will be used by default. If you wish to combine distinct and not distinct counts, you can choose which to display in your `f_str()` objects using `n`, `pct`, `distinct_n`, and `distinct_pct`. Additionally, denominators may be presented using `total` and `distinct_total`

Value

The layer object with

Examples

```
#Load in pipe
library(magrittr)

tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl) %>%
      set_distinct_by(carb)
  ) %>%
  build()
```

`set_format_strings` *Set the format strings and associated summaries to be performed in a layer*

Description

'Tplyr' gives you extensive control over how strings are presented. `set_format_strings` allows you to apply these string formats to your layer. This behaves slightly differently between layers.

Usage

```
set_format_strings(e, ...)

## S3 method for class 'desc_layer'
set_format_strings(
  e,
  ...,
  cap = getOption("tplyr.precision_cap"),
  max_int = getOption("tplyr.max_precision")["int"],
  max_dec = getOption("tplyr.max_precision")["dec"]
)

## S3 method for class 'count_layer'
set_format_strings(e, ...)
```

Arguments

`e`	Layer on which to bind format strings
`...`	Named parameters containing calls to `f_str` to set the format strings
`cap`	A named character vector containing an 'int' element for the cap on integer precision, and a 'dec' element for the cap on decimal precision. This caps the length allotted to the 'a' of an auto-precision format, so any '+' modifier is applied on top of the capped value.

`max_int` The overall maximum integer length allowed in the output. This is applied after auto-precision and any '+' modifier have been resolved, so an 'a+2' format will never be allotted more than `max_int` integer places. Only applies to auto-precision format groups. Defaults to the 'int' element of the `tplyr.max_precision` option.

`max_dec` The overall maximum decimal length allowed in the output. This is applied after auto-precision and any '+' modifier have been resolved, so an 'a+2' format will never display more than `max_dec` decimal places. Only applies to auto-precision format groups. Defaults to the 'dec' element of the `tplyr.max_precision` option.

Details

Format strings are one of the most powerful components of 'Tplyr'. Traditionally, converting numeric values into strings for presentation can consume a good deal of time. Values and decimals need to align between rows, rounding before trimming is sometimes forgotten - it can become a tedious mess that, in the grand scheme of things, is not an important part of the analysis being performed. 'Tplyr' makes this process as simple as we can, while still allowing flexibility to the user.

In a count layer, you can simply provide a single `f_str` object to specify how you want your n's, percentages, and denominators formatted. If you are additionally supplying a statistic, like risk difference using `add_risk_diff`, you specify the count formats using the name 'n_counts'. The risk difference formats would then be specified using the name 'riskdiff'. In a descriptive statistic layer, `set_format_strings` allows you to do a couple more things:

- By naming parameters with character strings, those character strings become a row label in the resulting data frame
- The actual summaries that are performed come from the variable names used within the `f_str` calls
- Using multiple summaries (declared by your `f_str` calls), multiple summary values can appear within the same line. For example, to present "Mean (SD)" like displays.
- Format strings in the desc layer also allow you to configure how empty values should be presented. In the `f_str` call, use the `empty` parameter to specify how missing values should present. A single element character vector should be provided. If the vector is unnamed, that value will be used in the format string and fill the space similar to how the numbers will display. Meaning - if your empty string is 'NA' and your format string is 'xx (xxx)', the empty values will populate as 'NA (NA)'. If you name the character vector in the 'empty' parameter '.overall', like `empty = c(.overall='')`, then that exact string will fill the value instead. For example, providing 'NA' will instead create the formatted string as 'NA' exactly.

See the `f_str` documentation for more details about how this implementation works.

Value

The layer environment with the format string binding added

tplyr_layer object with formats attached

Returns the modified layer object.

Examples

```
# Load in pipe
library(magrittr)

# In a count layer
tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl) %>%
      set_format_strings(f_str('xx (xx%)', n, pct))
  ) %>%
  build()

# In a descriptive statistics layer
tplyr_table(mtcars, gear) %>%
  add_layer(
    group_desc(mpg) %>%
      set_format_strings(
        "n"         = f_str("xx", n),
        "Mean (SD)" = f_str("xx.x", mean, empty='NA'),
        "SD"        = f_str("xx.xx", sd),
        "Median"    = f_str("xx.x", median),
        "Q1, Q3"    = f_str("xx, xx", q1, q3, empty=c(.overall='NA')),
        "Min, Max"  = f_str("xx, xx", min, max),
        "Missing"   = f_str("xx", missing)
      )
  ) %>%
  build()

# In a shift layer
tplyr_table(mtcars, am) %>%
  add_layer(
    group_shift(vars(row=gear, column=carb), by=cyl) %>%
    set_format_strings(f_str("xxx (xx.xx%)", n, pct))
  ) %>%
  build()
```

`set_indentation` *Set the option to prefix the row_labels in the inner count_layer*

Description

When a count layer uses nesting (i.e. triggered by `set_nest_count`), the `indentation` argument's value will be used as a prefix for the inner layer's records

Usage

```
set_indentation(e, indentation)
```

Arguments

e	A count_layer object
indentation	A character to prefix the row labels in an inner count layer

Value

The modified count_layer environment

---

set_limit_data_by *Set variables to limit reported data values only to those that exist rather than fully completing all possible levels*

Description

This function allows you to select a combination of by variables or potentially target variables for which you only want to display values present in the data. By default, Tplyr will create a cartesian combination of potential values of the data. For example, if you have 2 by variables present, then each potential combination of those by variables will have a row present in the final table. `set_limit_data_by()` allows you to choose the by variables whose combination you wish to limit to values physically present in the available data.

Usage

```
set_limit_data_by(e, ...)
```

Arguments

e	A tplyr_layer
...	Subset of variables within by or target variables

Value

a tplyr_table

Examples

```
tplyr_table(tplyr_adpe, TRT01A) %>%
  add_layer(
    group_desc(AVAL, by = vars(PECAT, PARAM, AVISIT))
  ) %>%
  build()

tplyr_table(tplyr_adpe, TRT01A) %>%
  add_layer(
    group_desc(AVAL, by = vars(PECAT, PARAM, AVISIT)) %>%
      set_limit_data_by(PARAM, AVISIT)
  ) %>%
  build()
```

```
tplyr_table(tplyr_adpe, TRT01A) %>%
  add_layer(
    group_count(AVALC, by = vars(PECAT, PARAM, AVISIT)) %>%
      set_limit_data_by(PARAM, AVISIT)
  ) %>%
  build()

tplyr_table(tplyr_adpe, TRT01A) %>%
  add_layer(
    group_count(AVALC, by = vars(PECAT, PARAM, AVISIT)) %>%
      set_limit_data_by(PECAT, PARAM, AVISIT)
  ) %>%
  build()
```

`set_missing_count` *Set the display for missing strings*

Description

Controls how missing counts are handled and displayed in the layer

Usage

```
set_missing_count(e, fmt = NULL, sort_value = NULL, denom_ignore = FALSE, ...)
```

Arguments

`e`	A `count_layer` object
`fmt`	An f_str object to change the display of the missing counts
`sort_value`	A numeric value that will be used in the ordering column. This should be numeric. If it is not supplied the ordering column will be the maximum value of what appears in the table plus one.
`denom_ignore`	A boolean. Specifies Whether or not to include the missing counts specified within the ... parameter within denominators. If set to TRUE, the values specified within ... will be ignored.
`...`	Parameters used to note which values to describe as missing. Generally NA and "Missing" would be used here. Parameters can be named character vectors where the names become the row label.

Value

The modified layer

Examples

```
library(magrittr)
library(dplyr)
  mtcars2 <- mtcars %>%
mutate_all(as.character)
mtcars2[mtcars$cyl == 6, "cyl"] <- NA

tplyr_table(mtcars2, gear) %>%
  add_layer(
    group_count(cyl) %>%
      set_missing_count(f_str("xx ", n), Missing = NA)
  ) %>%
  build()
```

`set_missing_subjects_row_label` *Set the label for the missing subjects row*

Description

Set the label for the missing subjects row

Usage

```
set_missing_subjects_row_label(e, missing_subjects_row_label)
```

Arguments

`e`	A `count_layer` object
`missing_subjects_row_label`	A character to label the total row

Value

The modified `count_layer` object

Examples

```
t <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl) %>%
      add_missing_subjects_row() %>%
      set_missing_subjects_row_label("Missing")
  )
build(t)
```

set_nest_count *Set the option to nest count layers*

Description

If set to TRUE, the second variable specified in `target_var` will be nested inside of the first variable. This allows you to create displays like those commonly used in adverse event tables, where one column holds both the labels of the outer categorical variable and the inside event variable (i.e. AEBODSYS and AEDECOD).

Usage

```
set_nest_count(e, nest_count)
```

Arguments

`e`	A `count_layer` object
`nest_count`	A logical value to set the nest option

Value

The modified layer

set_numeric_threshold *Set a numeric cutoff*

Description

[Experimental]

In certain tables, it may be necessary to only include rows that meet numeric conditions. Rows that are less than a certain cutoff can be suppressed from the output. This function allows you to pass a cutoff, a cutoff stat(n, distinct_n, pct, or distinct_pct) to supress values that are lesser than the cutoff.

Usage

```
set_numeric_threshold(e, numeric_cutoff, stat, column = NULL)
```

Arguments

`e`	A `count_layer` object
`numeric_cutoff`	A numeric value where only values greater than or equal to will be displayed.
`stat`	The statistic to use when filtering out rows. Either 'n', 'distinct_n', or 'pct' are allowable
`column`	If only a particular column should be used to cutoff values, it can be supplied here as a character value.

Value

The modified Tplyr layer object

Examples

```
mtcars %>%
tplyr_table(gear) %>%
  add_layer(
    group_count(cyl) %>%
      set_numeric_threshold(10, "n") %>%
      add_total_row() %>%
      set_order_count_method("bycount")
  )
```

`set_order_count_method`

Set the ordering logic for the count layer

Description

The sorting of a table can greatly vary depending on the situation at hand. For count layers, when creating tables like adverse event summaries, you may wish to order the table by descending occurrence within a particular treatment group. But in other situations, such as AEs of special interest, or subject disposition, there may be a specific order you wish to display values. Tplyr offers solutions to each of these situations.

Instead of allowing you to specify a custom sort order, Tplyr instead provides you with order variables that can be used to sort your table after the data are summarized. Tplyr has a default order in which the table will be returned, but the order variables will always persist. This allows you to use powerful sorting functions like `arrange` to get your desired order, and in double programming situations, helps your validator understand the how you achieved a particular sort order and where discrepancies may be coming from.

When creating order variables for a layer, for each 'by' variable Tplyr will search for a <VAR>N version of that variable (i.e. VISIT <-> VISITN, PARAM <-> PARAMN). If available, this variable will be used for sorting. If not available, Tplyr will created a new ordered factor version of that variable to use in alphanumeric sorting. This allows the user to control a custom sorting order by leaving an existing <VAR>N variable in your dataset if it exists, or create one based on the order in which you wish to sort - no custom functions in Tplyr required.

Ordering of results is where things start to differ. Different situations call for different methods. Descriptive statistics layers keep it simple - the order in which you input your formats using `set_format_strings` is the order in which the results will appear (with an order variable added). For count layers, Tplyr offers three solutions: If there is a <VAR>N version of your target variable, use that. If not, if the target variable is a factor, use the factor orders. Finally, you can use a specific data point from your results columns. The result column can often have multiple data points, between the n counts, percent, distinct n, and distinct percent. Tplyr allows you to choose which of these values will be used when creating the order columns for a specified result column (i.e. based on the `treat_var` and `cols` arguments). See the 'Sorting a Table' section for more information.

Shift layers sort very similarly to count layers, but to order your row shift variable, use an ordered factor.

Usage

```
set_order_count_method(e, order_count_method, break_ties = NULL)

set_ordering_cols(e, ...)

set_result_order_var(e, result_order_var)
```

Arguments

e	A count_layer object
order_count_method	The logic determining how the rows in the final layer output will be indexed. Options are 'bycount', 'byfactor', and 'byvarn'.
break_ties	In certain cases, a 'bycount' sort will result in conflicts if the counts aren't unique. break_ties will add a decimal to the sorting column so resolve conflicts. A character value of 'asc' will add a decimal based on the alphabetical sorting. 'desc' will do the same but sort descending in case that is the intention.
...	Unquoted variables used to select the columns whose values will be extracted for ordering.
result_order_var	The numeric value the ordering will be done on. This can be either n, distinct_n, pct, or distinct_pct. Due to the evaluation of the layer you can add a value that isn't actually being evaluated, if this happens this will only error out in the ordering.

Value

Returns the modified layer object. The 'ord_' columns are added during the build process.

Sorting a Table

When a table is built, the output has several ordering(ord_) columns that are appended. The first represents the layer index. The index is determined by the order the layer was added to the table. Following are the indices for the by variables and the target variable. The by variables are ordered based on:

1. The 'by' variable is a factor in the target dataset
2. If the variable isn't a factor, but has a <VAR>N variable (i.e. VISIT -> VISITN, TRT -> TRTN)
3. If the variable is not a factor in the target dataset, it is coerced to one and ordered alphabetically.

The target variable is ordered depending on the type of layer. See more below.

Ordering a Count Layer

There are many ways to order a count layer depending on the preferences of the table programmer. `Tplyr` supports sorting by a descending amount in a column in the table, sorting by a <VAR>N variable, and sorting by a custom order. These can be set using the 'set_order_count_method' function.

Sorting by a numeric count A selected numeric value from a selected column will be indexed based on the descending numeric value. The numeric value extracted defaults to 'n' but can be changed with 'set_result_order_var'. The column selected for sorting defaults to the first value in the treatment group variable. If there were arguments passed to the 'cols' argument in the table those must be specified with 'set_ordering_columns'.

Sorting by a 'varn' variable If the treatment variable has a <VAR>N variable. It can be indexed to that variable.

Sorting by a factor(Default) If a factor is found for the target variable in the target dataset that is used to order, if no factor is found it is coerced to a factor and sorted alphabetically.

Sorting a nested count layer If two variables are targeted by a count layer, two methods can be passed to 'set_order_count'. If two are passed, the first is used to sort the blocks, the second is used to sort the "inside" of the blocks. If one method is passed, that will be used to sort both.

Ordering a Desc Layer

The order of a desc layer is mostly set during the object construction. The by variables are resolved and index with the same logic as the count layers. The target variable is ordered based on the format strings that were used when the layer was created.

Examples

```
library(dplyr)

# Default sorting by factor
t <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl)
  )
build(t)

# Sorting by <VAR>N
mtcars$cylN <- mtcars$cyl
t <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl) %>%
      set_order_count_method("byvarn")
  )

# Sorting by row count
t <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl) %>%
      set_order_count_method("bycount") %>%
      # Orders based on the 6 gear group
```

```
      set_ordering_cols(6)
  )

# Sorting by row count by percentages
t <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl) %>%
      set_order_count_method("bycount") %>%
      set_result_order_var(pct)
  )

# Sorting when you have column arguments in the table
t <- tplyr_table(mtcars, gear, cols = vs) %>%
  add_layer(
    group_count(cyl) %>%
      # Uses the fourth gear group and the 0 vs group in ordering
      set_ordering_cols(4, 0)
  )

# Using a custom factor to order
mtcars$cyl <- factor(mtcars$cyl, c(6, 4, 8))
t <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl) %>%
      # This is the default but can be used to change the setting if it is
      #set at the table level.
      set_order_count_method("byfactor")
  )
```

`set_outer_sort_position` *Set the value of a outer nested count layer to Inf or -Inf*

Description

Set the value of a outer nested count layer to Inf or -Inf

Usage

```
set_outer_sort_position(e, outer_sort_position)
```

Arguments

`e`	A `count_layer` object
`outer_sort_position`	Either 'asc' or 'desc'. If desc the final ordering helper will be set to Inf, if 'asc' the ordering helper is set to -Inf.

Value

The modified count layer.

`set_precision_data` *Set precision data*

Description

In some cases, there may be organizational standards surrounding decimal precision. For example, there may be a specific standard around the representation of precision relating to lab results. As such, `set_precision_data()` provides an interface to provide integer and decimal precision from an external data source.

Usage

```
set_precision_data(layer, prec, default = c("error", "auto"))
```

Arguments

`layer`	A `tplyr_layer` object
`prec`	A dataframe following the structure specified in the function details
`default`	Handling of unspecified by variable groupings. Defaults to 'error'. Set to 'auto' to automatically infer any missing groups.

Details

The ultimate behavior of this feature is just that of the existing auto precision method, except that the precision is specified in the provided precision dataset rather than inferred from the source data. At a minimum, the precision dataset must contain the integer variables `max_int` and `max_dec`. If by variables are provided, those variables must be available in the layer by variables.

When the table is built, by default Tplyr will error if the precision dataset is missing by variable groupings that exist in the target dataset. This can be overriden using the `default` parameter. If `default` is set to "auto", any missing values will be automatically inferred from the source data.

Examples

```
prec <- tibble::tribble(
  ~vs, ~max_int, ~max_dec,
  0,        1,        1,
  1,        2,        2
)

tplyr_table(mtcars, gear) %>%
  add_layer(
    group_desc(wt, by = vs) %>%
      set_format_strings(
        'Mean (SD)' = f_str('a.a+1 (a.a+2)', mean, sd)
```

```
    ) %>%
    set_precision_data(prec) %>%
    set_precision_on(wt)
) %>%
build()
```

`set_stats_as_columns` *Set descriptive statistics as columns*

Description

In many cases, treatment groups are represented as columns within a table. But some tables call for a transposed presentation, where the treatment groups displayed by row, and the descriptive statistics are represented as columns. `set_stats_as_columns()` allows Tplyr to output a built table using this transposed format and deviate away from the standard representation of treatment groups as columns.

Usage

```
set_stats_as_columns(e, stats_as_columns = TRUE)
```

Arguments

`e`	`desc_layer` on descriptive statistics summaries should be represented as columns
`stats_as_columns`	Boolean to set stats as columns

Details

This function leaves all specified by variables intact. The only switch that happens during the build process is that the provided descriptive statistics are transposed as columns and the treatment variable is left as rows. Column variables will remain represented as columns, and multiple target variables will also be respected properly.

Value

The input tplyr_layer

Examples

```
dat <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_desc(wt, by = vs) %>%
      set_format_strings(
        "n"        = f_str("xx", n),
        "sd"       = f_str("xx.x", sd, empty = c(.overall = "BLAH")),
        "Median"   = f_str("xx.x", median),
        "Q1, Q3"   = f_str("xx, xx", q1, q3),
```

```
      "Min, Max" = f_str("xx, xx", min, max),
      "Missing"  = f_str("xx", missing)
    ) %>%
    set_stats_as_columns()
) %>%
build()
```

`set_total_row_label` *Set the label for the total row*

Description

The row label for a total row defaults to "Total", however this can be overriden using this function.

Usage

```
set_total_row_label(e, total_row_label)
```

Arguments

`e`	A `count_layer` object
`total_row_label`	A character to label the total row

Value

The modified `count_layer` object

Examples

```
# Load in pipe
library(magrittr)

t <- tplyr_table(mtcars, gear) %>%
  add_layer(
    group_count(cyl) %>%
      add_total_row() %>%
      set_total_row_label("Total Cyl")
  )
build(t)
```

`str_extract_fmt_group` *Extract format group strings or numbers*

Description

These functions allow you to extract segments of information from within a result string by targetting specific format groups. `str_extract_fmt_group()` allows you to pull out the individual format group string, while `str_extract_num()` allows you to pull out that specific numeric result.

Usage

```
str_extract_fmt_group(string, format_group)

str_extract_num(string, format_group)
```

Arguments

`string`	A string of number results from which to extract format groups
`format_group`	An integer representing format group that should be extracted

Details

Format groups refer to individual segments of a string. For example, given the string ' 5 (34.4%) [9]', there are three separate format groups, which are ' 5', '(34.4%)', and '[9]'.

Value

A character vector

Examples

```
string <- c(" 0  (0.0%)", " 8  (9.3%)", "78 (90.7%)",  "-1 (-.56, .75) -523%, 56 | -34")

str_extract_fmt_group(string, 2)

str_extract_num(string, 2)
```

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

Description

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

Usage

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

Arguments

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

Details

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

This function takes steps to resolve these two issues, while trying to minimize additional overhead required to apply the wrapping of strings.

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

Value

A character vector with string wrapping applied

Examples

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

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

Tplyr *A grammar of summary data for clinical reports*

Description

‘r lifecycle::badge("experimental")‘

Details

'Tplyr' is a package dedicated to simplifying the data manipulation necessary to create clinical reports. Clinical data summaries can often be broken down into two factors - counting discrete variables (or counting shifts in state), and descriptive statistics around a continuous variable. Many of the reports that go into a clinical report are made up of these two scenarios. By abstracting this process away, 'Tplyr' allows you to rapidly build these tables without worrying about the underlying data manipulation.

'Tplyr' takes this process a few steps further by abstracting away most of the programming that goes into proper presentation, which is where a great deal of programming time is spent. For example, 'Tplyr' allows you to easily control:

String formatting Different reports warrant different presentation of your strings. Programming this can get tedious, as you typically want to make sure that your decimals properly align. 'Tplyr' abstracts this process away and provides you with a simple interface to specify how you want your data presented

Treatment groups Need a total column? Need to group summaries of multiple treatments? 'Tplyr' makes it simple to add additional treatment groups into your report

Denominators n (%) counts often vary based on the summary being performed. 'Tplyr' allows you to easily control what denominators are used based on a few common scenarios

Sorting Summarizing data is one thing, but ordering it for presentation. Tplyr automatically derives sorting variable to give you the data you need to order your table properly. This process is flexible so you can easily get what you want by leveraging your data or characteristics of R.

Another powerful aspect of 'Tplyr' are the objects themselves. 'Tplyr' does more than format your data. Metadata about your table is kept under the hood, and functions allow you to access information that you need. For example, 'Tplyr' allows you to calculate and access the raw numeric data of calculations as well, and easily pick out just the pieces of information that you need.

Lastly, 'Tplyr' was built to be flexible, yet intuitive. A common pitfall of building tools like this is over automation. By doing to much, you end up not doing enough. 'Tplyr' aims to hit the sweet spot in between. Additionally, we designed our function interfaces to be clean. Modifier functions offer you flexibility when you need it, but defaults can be set to keep the code concise. This allows you to quickly assemble your table, and easily make changes where necessary.

Author(s)

Maintainer: Mike Stackhouse <mike.stackhouse@atorusresearch.com> (ORCID)

Authors:

- Eli Miller <Eli.Miller@AtorusResearch.com> (ORCID)
- Ashley Tarasiewicz <Ashley.Tarasiewicz@atorusresearch.com>

Other contributors:

- Nathan Kosiba <Nathan.Kosiba@atorusresearch.com> (ORCID) [contributor]
- Sadchla Mascary <sadchla.mascary@atorusresearch.com> [contributor]
- Andrew Bates <andrew.bates@atorusresearch.com> [contributor]
- Shiyu Chen <shiyu.chen@atorusresearch.com> [contributor]
- Oleksii Mikryukov <alex.mikryukov@atorusresearch.com> [contributor]
- Atorus Research LLC [copyright holder]

See Also

Useful links:

- https://atorus-research.github.io/Tplyr/
- https://github.com/atorus-research/Tplyr
- Report bugs at https://github.com/atorus-research/Tplyr/issues

Examples

```
# Load in pipe
library(magrittr)

# Use just the defaults
tplyr_table(mtcars, gear) %>%
  add_layer(
    group_desc(mpg, by=cyl)
  ) %>%
  add_layer(
    group_count(carb, by=cyl)
  ) %>%
  build()

# Customize and modify
tplyr_table(mtcars, gear) %>%
  add_layer(
    group_desc(mpg, by=cyl) %>%
      set_format_strings(
        "n"         = f_str("xx", n),
        "Mean (SD)" = f_str("a.a+1 (a.a+2)", mean, sd, empty='NA'),
        "Median"    = f_str("a.a+1", median),
        "Q1, Q3"    = f_str("a, a", q1, q3, empty=c(.overall='NA')),
        "Min, Max"  = f_str("a, a", min, max),
        "Missing"   = f_str("xx", missing)
      )
  ) %>%
  add_layer(
    group_count(carb, by=cyl) %>%
```

```
      add_risk_diff(
        c('5', '3'),
        c('4', '3')
      ) %>%
      set_format_strings(
        n_counts = f_str('xx (xx%)', n, pct),
        riskdiff = f_str('xx.xxx (xx.xxx, xx.xxx)', dif, low, high)
      ) %>%
      set_order_count_method("bycount") %>%
      set_ordering_cols('4') %>%
      set_result_order_var(pct)
  ) %>%
  build()

# A Shift Table
tplyr_table(mtcars, am) %>%
  add_layer(
    group_shift(vars(row=gear, column=carb), by=cyl) %>%
    set_format_strings(f_str("xxx (xx.xx%)", n, pct))
  ) %>%
  build()
```

`tplyr_adae` *ADAE Data*

Description

A subset of the PHUSE Test Data Factory ADAE data set.

Usage

```
tplyr_adae
```

Format

A data.frame with 276 rows and 55 columns.

Source

https://github.com/phuse-org/TestDataFactory

See Also

[get_data_labels()]

tplyr_adas *ADAS Data*

Description

A subset of the PHUSE Test Data Factory ADAS data set.

Usage

```
tplyr_adas
```

Format

A data.frame with 1,040 rows and 40 columns.

Source

https://github.com/phuse-org/TestDataFactory

See Also

[get_data_labels()]

tplyr_adlb *ADLB Data*

Description

A subset of the PHUSE Test Data Factory ADLB data set.

Usage

```
tplyr_adlb
```

Format

A data.frame with 311 rows and 46 columns.

Source

https://github.com/phuse-org/TestDataFactory

See Also

[get_data_labels()]

tplyr_adpe *ADPE Data*

Description

A mock-up dataset that is fit for testing data limiting

Usage

```
tplyr_adpe
```

Format

A data.frame with 21 rows and 8 columns.

tplyr_adsl *ADSL Data*

Description

A subset of the PHUSE Test Data Factory ADSL data set.

Usage

```
tplyr_adsl
```

Format

A data.frame with 254 rows and 49 columns.

Source

https://github.com/phuse-org/TestDataFactory

See Also

[get_data_labels()]

tplyr_layer *Create a* tplyr_layer *object*

Description

This object is the workhorse of the tplyr package. A tplyr_layer can be thought of as a block, or "layer" of a table. Summary tables typically consist of different sections that require different summaries. When programming these section, your code will create different layers that need to be stacked or merged together. A tplyr_layer is the container for those isolated building blocks.

When building the tplyr_table, each layer will execute independently. When all of the data processing has completed, the layers are brought together to construct the output.

tplyr_layer objects are not created directly, but are rather created using the layer constructor functions group_count, group_desc, and group_shift.

Usage

```
tplyr_layer(parent, target_var, by, where, type, ...)
```

Arguments

parent	tplyr_table or tplyr_layer. Required. The parent environment of the layer. This must be either the tplyr_table object that the layer is contained within, or another tplyr_layer object of which the layer is a subgroup.
target_var	Symbol. Required, The variable name on which the summary is to be performed. Must be a variable within the target dataset. Enter unquoted - i.e. target_var = AEBODSYS.
by	A string, a variable name, or a list of variable names supplied using dplyr::vars
where	Call. Filter logic used to subset the target data when performing a summary.
type	"count", "desc", or "shift". Required. The category of layer - either "counts" for categorical counts, "desc" for descriptive statistics, or "shift" for shift table counts
...	Additional arguments

Value

A tplyr_layer environment that is a child of the specified parent. The environment contains the object as listed below.

tplyr_layer Core Object Structure

type This is an attribute. A string indicating the layer type, which controls the summary that will be performed.

target_var A quosure of a name, which is the variable on which a summary will be performed.

`by` A list of quosures representing either text labels or variable names used in grouping. Variable names must exist within the target dataset Text strings submitted do not need to exist in the target dataset.

`cols` A list of quosures used to determine the variables that are used to display in columns.

`where` A quosure of a call that containers the filter logic used to subset the target dataset. This filtering is in addition to any subsetting done based on `where` criteria specified in `tplyr_table`

`layers` A list with class `tplyr_layer_container`. Initialized as empty, but serves as the container for any sublayers of the current layer. Used internally.

Different layer types will have some different bindings specific to that layer's needs.

See Also

tplyr_table

Examples

```
tab <- tplyr_table(iris, Sepal.Width)

l <- group_count(tab, by=vars('Label Text', Species),
                 target_var=Species, where= Sepal.Width < 5.5,
                 cols = Species)
```

---

tplyr_meta *Tplyr Metadata Object*

Description

If a Tplyr table is built with the 'metadata=TRUE' option specified, then metadata is assembled behind the scenes to provide traceability on each result cell derived. The functions 'get_meta_result()' and 'get_meta_subset()' allow you to access that metadata by using an ID provided in the row_id column and the column name of the result you'd like to access. The purpose is of the row_id variable instead of a simple row index is to provide a sort resistant reference of the originating column, so the output Tplyr table can be sorted in any order but the metadata are still easily accessible.

Usage

```
tplyr_meta(names = list(), filters = exprs())
```

Arguments

`names`	List of symbols
`filters`	List of expressions

Details

The ‘tplyr_meta‘ object provided a list with two elements - names and filters. The names contain every column from the target data.frame of the Tplyr table that factored into the specified result cell, and the filters contains all the necessary filters to subset the target data to create the specified result cell. ‘get_meta_subset()‘ additionally provides a parameter to specify any additional columns you would like to include in the returned subset data frame.

Value

tplyr_meta object

Examples

```
tplyr_meta(
   names = rlang::quos(x, y, z),
   filters = rlang::quos(x == 1, y==2, z==3)
 )
```

---

tplyr_table *Create a Tplyr table object*

Description

The `tplyr_table` object is the main container upon which a Tplyr table is constructed. Tplyr tables are made up of one or more layers. Each layer contains an instruction for a summary to be performed. The `tplyr_table` object contains those layers, and the general data, metadata, and logic necessary.

Usage

```
tplyr_table(target, treat_var, where = TRUE, cols = vars())
```

Arguments

`target`	Dataset upon which summaries will be performed
`treat_var`	Variable containing treatment group assignments. Supply unquoted.
`where`	A general subset to be applied to all layers. Supply as programming logic (i.e. x < 5 & y == 10)
`cols`	A grouping variable to summarize data by column (in addition to treat_var). Provide multiple column variables by using `vars`

Details

When a `tplyr_table` is created, it will contain the following bindings:

- target - The dataset upon which summaries will be performed
- pop_data - The data containing population information. This defaults to the target dataset
- cols - A categorical variable to present summaries grouped by column (in addition to treat_var)
- table_where - The `where` parameter provided, used to subset the target data
- treat_var - Variable used to distinguish treatment groups.
- header_n - Default header N values based on `treat_var`
- pop_treat_var - The treatment variable for `pop_data` (if different)
- layers - The container for individual layers of a `tplyr_table`
- treat_grps - Additional treatment groups to be added to the summary (i.e. Total)

`tplyr_table` allows you a basic interface to instantiate the object. Modifier functions are available to change individual parameters catered to your analysis. For example, to add a total group, you can use the `add_total_group`.

In future releases, we will provide vignettes to fully demonstrate these capabilities.

Value

A `tplyr_table` object

Examples

```
tab <- tplyr_table(iris, Species, where = Sepal.Length < 5.8)
```

`treat_var` *Return or set the treatment variable binding*

Description

Return or set the treatment variable binding

Usage

```
treat_var(table)

set_treat_var(table, treat_var)
```

Arguments

`table`	A `tplyr_table` object to set or return treatment variable the table is split by.
`treat_var`	Variable containing treatment group assignments. Supply unquoted.

Value

For `tplyr_treat_var` the treat_var binding of the `tplyr_table` object. For `set_tplyr_treat_var` the modified object.

Examples

```
tab <- tplyr_table(mtcars, cyl)

set_treat_var(tab, gear)
```

Index

∗ **Layer Construction Functions**
    group_count, 35
∗ **Layer Templates**
    new_layer_template, 39
∗ **Layer attachment**
    add_layer, 6
∗ **Layer construction functions**
    tplyr_layer, 69
∗ **Metadata additions**
    add_variables, 13
∗ **String extractors**
    str_extract_fmt_group, 62
∗ **datasets**
    tplyr_adae, 66
    tplyr_adas, 67
    tplyr_adlb, 67
    tplyr_adpe, 68
    tplyr_adsl, 68

add_anti_join, 3
add_column_headers, 4
add_filters *(*add_variables*)*, 13
add_layer, 6, *36*
add_layers, *36*
add_layers *(*add_layer*)*, 6
add_missing_subjects_row, 8
add_risk_diff, 8, *49*
add_total_group, *9*, *72*
add_total_group *(*add_treat_grps*)*, 12
add_total_row, 11
add_treat_grps, *9*, 12
add_variables, 13
append_metadata, 14
apply_conditional_format, 15
apply_formats, 16
apply_formats(), *21*
apply_row_masks, 17
arrange, *55*

build, 18

collapse_row_labels, 19

dplyr::mutate(), *21*

f_str, *9*, 20, *43*, *44*, *49*

get_by, 23
get_count_layer_formats
        *(*get_desc_layer_formats*)*, 24
get_data_labels, 23
get_desc_layer_formats, 24
get_layer_template
        *(*new_layer_template*)*, 39
get_layer_templates
        *(*new_layer_template*)*, 39
get_meta_result, 25
get_meta_subset, 26
get_metadata, 25
get_numeric_data, *6*, *18*, 28
get_pop_where *(*get_where.tplyr_layer*)*, 34
get_precision_by, 29
get_precision_on, 30
get_shift_layer_formats
        *(*get_desc_layer_formats*)*, 24
get_stats_data, *6*, 31
get_target_var, 32
get_tplyr_regex, 33
get_where *(*get_where.tplyr_layer*)*, 34
get_where.tplyr_layer, 34
group_count, 35, *69*
group_desc, *69*
group_desc *(*group_count*)*, 35
group_shift, *69*
group_shift *(*group_count*)*, 35

header_n, 37
header_n<- *(*header_n*)*, 37

keep_levels, 38

new_layer_template, 39

pop_data, 41
pop_data<- (pop_data), 41
pop_treat_var, 42
prop.test, 8–10

remove_layer_template
(new_layer_template), 39
replace_leading_whitespace, 42

set_by (get_by), 23
set_count_layer_formats
(get_desc_layer_formats), 24
set_custom_summaries, 36, 43
set_custom_summaries(), 21, 22
set_denom_ignore, 45
set_denom_where, 46
set_denoms_by, 36, 44
set_desc_layer_formats
(get_desc_layer_formats), 24
set_distinct_by, 36, 47
set_format_strings, 9, 36, 43, 44, 48, 55
set_format_strings(), 21
set_header_n (header_n), 37
set_indentation, 50
set_limit_data_by, 51
set_missing_count, 52
set_missing_subjects_row_label, 53
set_nest_count, 50, 54
set_numeric_threshold, 54
set_order_count_method, 55
set_ordering_cols
(set_order_count_method), 55
set_outer_sort_position, 58
set_pop_data (pop_data), 41
set_pop_treat_var (pop_treat_var), 42
set_pop_where (get_where.tplyr_layer),
34
set_precision_by (get_precision_by), 29
set_precision_data, 59
set_precision_on (get_precision_on), 30
set_result_order_var
(set_order_count_method), 55
set_shift_layer_formats
(get_desc_layer_formats), 24
set_stats_as_columns, 60
set_target_var (get_target_var), 32
set_total_row_label, 61
set_treat_var (treat_var), 72
set_where (get_where.tplyr_layer), 34
str_extract_fmt_group, 62
str_extract_num
(str_extract_fmt_group), 62
str_indent_wrap, 63

Tplyr, 64
Tplyr-package (Tplyr), 64
tplyr_adae, 66
tplyr_adas, 67
tplyr_adlb, 67
tplyr_adpe, 68
tplyr_adsl, 68
tplyr_layer, 36, 69
tplyr_meta, 70
tplyr_table, 36, 70, 71
treat_grps (add_treat_grps), 12
treat_var, 72

use_template (new_layer_template), 39

vars, 35, 71