Package ‘dtrackr’

July 5, 2022

Title Track your Data Pipelines

Version 0.2.4

Description Track and
document 'dplyr' data pipelines. As you filter, mutate, and join your
way through a data set, 'dtrackr' seamlessly keeps track of your data
flow and makes publication ready documentation of a data pipeline simple.

License MIT + file LICENSE

Language en-GB

Imports dplyr, glue, htmltools, magrittr, rlang, rsvg, stringr,
tibble, tidyr, utils, V8, fs, purrr, base64enc

Suggests here, knitr, magick, rmarkdown, staplr, tidyverse, devtools,
testthat (>= 2.1.0), rstudioapi

VignetteBuilder knitr

Encoding UTF-8

LazyData true

RoxygenNote 7.1.2

Depends R (>= 2.10)

NeedsCompilation no

Author Robert Challen [aut, cre] (<`https://orcid.org/0000-0002-5504-7768`>)

Maintainer Robert Challen <`rc538@exeter.ac.uk`>

Repository CRAN

Date/Publication 2022-07-05 21:00:09 UTC

R topics documented:

`add_count` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
add_count(
  .data,
  ...,
  wt = NULL,
  sort = FALSE,
  name = NULL,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`wt`	<`data-masking`> Frequency weights. Can be `NULL` or a variable: • If `NULL` (the default), counts the number of rows in each group. • If a variable, computes `sum(wt)` for each group.
`sort`	If `TRUE`, will show the largest groups at the top.
`name`	The name of the new column in the output. If omitted, it will default to `n`. If there's already a column called `n`, it will error, and require you to specify the name.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::add_count()

`add_tally` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
add_tally(
  .data,
  ...,
  wt = NULL,
  sort = FALSE,
  name = NULL,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`wt`	<`data-masking`> Frequency weights. Can be `NULL` or a variable: • If `NULL` (the default), counts the number of rows in each group. • If a variable, computes `sum(wt)` for each group.
`sort`	If `TRUE`, will show the largest groups at the top.
`name`	The name of the new column in the output. If omitted, it will default to `n`. If there's already a column called `n`, it will error, and require you to specify the name.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::add_tally()

`anti_join.trackr_df` *Anti join*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::anti_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
anti_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} not matched"),
  .headline = "Semi join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector

	to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`...`	Other parameters passed onto methods.
`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::anti_join()

`arrange.trackr_df` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
## S3 method for class 'trackr_df'
arrange(
  .data,
  ...,
  .by_group = FALSE,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.by_group`	If `TRUE`, will sort first by grouping variable. Applies to grouped data frames only.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::arrange()

`bind_rows` *Union of two or more data sets*

Description

This merges the history of 2 dataframes and binds the rows. It calculates the total number of resulting rows as .count.out in other terms it performs exactly the same operation as dplyr::bind_rows. See `dplyr::bind_rows()`.

Usage

```
bind_rows(
  ...,
  .id = NULL,
  .messages = "{.count.out} in union",
  .headline = "Union"
)
```

Arguments

`...`	the data frames to bind

`.id`	Data frame identifier. When `.id` is supplied, a new column of identifiers is created to link each row to its original data frame. The labels are taken from the named arguments to `bind_rows()`. When a list of data frames is supplied, the labels are taken from the names of the list. If no names are found a numeric sequence is used instead.
`.messages`	• a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	• a glue spec. The glue code can use any global variable, or {.count.out}

Value

the logical union of the dataframes with the history graph updated.

See Also

dplyr::bind_rows()

Examples

```
library(dplyr)
bind_rows( iris %>% comment("one"), iris %>% comment("two") ) %>% history()
```

`capture_exclusions` *Start capturing exclusions on a tracked dataframe.*

Description

Start capturing exclusions on a tracked dataframe.

Usage

```
capture_exclusions(.data, .capture = TRUE)
```

Arguments

`.data`	• a tracked dataframe
`.capture`	• Should we capture exclusions (things removed from the data set). This is useful for debugging data issues but comes at a significant cost. Defaults to the value of `getOption("dtrackr.exclusions")` or `FALSE`.

Value

the .data dataframe with the exclusions flag set (or cleared if `.capture=FALSE`).

Examples

```
library(dplyr)
tmp = iris %>% track() %>% capture_exclusions()
tmp %>% filter(Species!="versicolor") %>% history()
```

`comment` *Add a generic comment to the dtrackr history graph*

Description

A comment can be any kind fo note and is added once for every current grouping as defined by the `.message` field. It can be made context specific by including variables such as {.count} and {.total} in `.message` which refer to the grouped and ungrouped counts at this current stage of the pipeline for example. It can also pull in any global variable.

Usage

```
comment(
  .data,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = (.type == "exclusion"),
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`.messages`	• a character vector of glue specifications. A glue specification can refer to any grouping variables of .data, or any variables defined in the calling environment, the {.total} of all rows, the {.count} variable which is the count in each group and {.strata} a description of the group
`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment, or the {.total} variable which is nrow(.data)and {.strata}
`.type`	• one of "info","...,"exclusion": used to define formatting
`.asOffshoot`	• do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the same .data dataframe with the history graph updated with the comment

Examples

```
library(dplyr)
iris %>% track() %>% comment("hello {.total} rows") %>% history()
```

`count_subgroup` *Add a subgroup count to the dtrackr history graph*

Description

A frequent use case for more detailed description is to have a subgroup count within a flowchart. This works best for factor subgroup columns but other data will be converted to a factor automatically. The count of the items in each subgroup is added as a new stage in the flowchart.

Usage

```
count_subgroup(
  .data,
  .subgroup,
  ...,
  .messages = .defaultCountSubgroup(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = FALSE,
  .tag = NULL,
  .maxsubgroups = .defaultMaxSupportedGroupings()
)
```

Arguments

`.data`	• a dataframe which may be grouped
`.subgroup`	• a column with a small number of levels (e.g.)
`...`	• additional parameters will be passed to factor(subgroup,...) to control levels, ordering, etc.
`.messages`	• a character vector of glue specifications. A glue specification can refer to anything from the calling environment and .name for the subgroup name, .count for the subgroup count, .subtotal for the current grouping count and .total for the whole count
`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`.type`	• one of "info","exclusion": used to define formatting
`.asOffshoot`	• do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.
`.maxsubgroups`	• the maximum number of discrete values allowed is configurable with `options("dtrackr.max_suppo` The default is 16.

Value

the same .data dataframe with the history graph updated with a subgroup count as a new stage

Examples

```
library(dplyr)
ILPD %>% track() %>% group_by(Case_or_Control) %>% count_subgroup(Gender) %>% history()
```

`distinct.trackr_df` *Distinct values of data*

Description

Distinct acts in the same way as in `dplyr::distinct`. Prior to the operation the size of the group is calculated {.count.in} and after the operation the output size {.count.out} The group {.strata} is also available (if grouped) for reporting See `dplyr::distinct()`.

Usage

```
## S3 method for class 'trackr_df'
distinct(
  .data,
  .f,
  ...,
  .keep = FALSE,
  .messages = "removing {.count.in-.count.out} duplicates",
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`.data`	- a dataframe which may be grouped
`.f`	a function as expected by dplyr::group_modify e.g. function(d,g,...) ...do something with d and return a dataframe...
`...`	additional parameters for .f.
`.keep`	- are the grouping variables kept in d, or split out to g (the default)
`.messages`	- a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.headline`	- a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.tag`	- if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe with distinct values and history graph updated.

See Also

dplyr::distinct()

Examples

```
library(dplyr)
tmp = bind_rows(iris %>% track(), iris %>% track() %>% filter(Petal.Length > 5))
tmp %>% group_by(Species) %>% distinct() %>% history()
```

`dot2svg` *Convert Graphviz dot content to a SVG*

Description

Convert a graphviz dot digraph as string to SVG as string

Usage

```
dot2svg(dot)
```

Arguments

`dot`	• a graphviz dot string

Value

the SVG as a string

Examples

```
dot2svg("digraph { A->B }")
```

`excluded` *Get the dtrackr excluded data record*

Description

Get the dtrackr excluded data record

Usage

```
excluded(.data, simplify = TRUE)
```

Arguments

`.data`	• a dataframe which may be grouped
`simplify`	• return a single summary dataframe of all exclusions.

Value

a new dataframe of the excluded data up to this point in the workflow. This dataframe is by default flattened, but if `.simplify=FALSE` has a nested structure containing records excluded at each part of the pipeline.

Examples

```
library(dplyr)
tmp = iris %>% track() %>% capture_exclusions()
tmp %>% exclude_all(
   Petal.Length > 5.8 ~ "{.excluded} long ones",
   Petal.Length < 1.3 ~ "{.excluded} short ones",
   .stage = "petal length exclusion"
) %>% excluded()
```

`exclude_all` *Exclude all items matching one or more criteria*

Description

Apply a set of filters and summarise the actions of the filter to the dtrackr history graph Because of the ... filter specification, all parameters MUST BE NAMED. The filters work in an additive manner, i.e. the results excluding all things that match any of the criteria. If na.rm = TRUE they also remove anything that cannot be evaluated by a criteria.

Usage

```
exclude_all(
  .data,
  ...,
  .headline = .defaultHeadline(),
  na.rm = FALSE,
  .type = "exclusion",
  .asOffshoot = TRUE,
  .stage = ""
)
```

Arguments

`.data`	- a dataframe which may be grouped
`...`	- a dplyr filter specification as a formula where the RHS is a glue specification, defining the message. This can refer to grouping variables variables from the environment and {.excluded} and {.matched} or {.missing} (excluded = matched+missing), {.count} and {.total} - group and overall counts respectively, e.g. "excluding {.matched} items and {.missing} with missing values".

`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`na.rm`	• (default FALSE) if the filter cannot be evaluated for a row count that row as missing and either exclude it (TRUE) or don't exclude it (FALSE)
`.type`	• default "exclusion": used to define formatting
`.asOffshoot`	• do you want this comment to be an offshoot of the main flow (default = TRUE).
`.stage`	• a name for this step in the pathway

Value

the filtered .data dataframe with the history graph updated with the summary of excluded items as a new offshoot stage

Examples

```
library(dplyr)
iris %>% track() %>% capture_exclusions() %>% exclude_all(
      Petal.Length > 5 ~ "{.excluded} long ones”,
      Petal.Length < 2 ~ "{.excluded} short ones”
) %>% history()
```

---

filter.trackr_df *Filtering data*

Description

Filter acts in the same way as DPLYR. Prior to the operation the size of the group is calculated {.count.in} and after the operation the output size {.count.out}. The group {.strata} is also available (if grouped) for reporting. See `dplyr::filter()`.

Usage

```
## S3 method for class 'trackr_df'
filter(
  .data,
  ...,
  .preserve = FALSE,
  .messages = "excluded {.excluded} items”,
  .headline = .defaultHeadline(),
  .type = "exclusion”,
  .asOffshoot = (.type == "exclusion”),
  .stage = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	the filter criteria
`.preserve`	Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is.
`.messages`	• a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.headline`	• a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.type`	• the format type of the action - typically an exclusion
`.asOffshoot`	• if the type is exclusion, asOffshoot places the information box outside of the main flow, as an exclusion.
`.stage`	• a name for this step in the pathway
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the filtered .data dataframe with history graph updated

See Also

dplyr::filter()

Examples

```
library(dplyr)
tmp = iris %>% track() %>% group_by(Species)
tmp %>% filter(Petal.Length > 5) %>% history()
```

`flowchart` *Flowchart output*

Description

Generate a flowchart of the history of the dataframe, with all the transformations as stages in the flowchart.

Usage

```
flowchart(
  .data,
  filename = NULL,
  size = std_size$half,
  maxWidth = size$width,
  maxHeight = size$height,
  rot = size$rot,
  formats = c("dot", "png", "pdf", "svg"),
  defaultToHTML = TRUE,
  ...
)
```

Arguments

`.data`	• the tracked dataframes
`filename`	• a filename (without extension) which will be where the formatted flowcharts are saved
`size`	• a list of length and width in inches e.g. a std_size
`maxWidth`	• a width in inches is size is not defined
`maxHeight`	• a height in inches if size is not defined
`rot`	• an angle of rotation for the saved file if size is not defined
`formats`	• some of "pdf","dot","svg","png","ps"
`defaultToHTML`	• if the correct output format is not easy to determine from the context, default providing HTML or to embedding the PNG
`...`	• other params passed onto p_get_as_dot, notable ones are fill, fontsize, colour, size, maxWidth and maxHeight

Value

the nature of the flowchart output depends on the context in which the function is called. It will be some form of browse-able html output if called from an interactive session or a PNG/PDG link if in knitr and knitting latex or word type outputs,

Examples

```
library(dplyr)
tmp = iris %>% track() %>% comment(.tag = "step1") %>% filter(Species!="versicolor")
tmp %>% group_by(Species) %>% comment(.tag="step2") %>% flowchart()
```

`full_join.trackr_df` *Full join*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::full_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Full join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

`suffix`	If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
`...`	Other parameters passed onto methods.
`keep`	Should the join keys from both `x` and `y` be preserved in the output?
`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::full_join()

`group_by.trackr_df` *Stratifying your analysis*

Description

Grouping a data set acts in the normal way. When tracking a dataframe sometimes a `group_by()` operation will create a lot of groups. This happens for example if you are doing a `group_by()`, `summarise()` step that is aggregating data on a fine scale, e.g. by day in a timeseries. This is generally a terrible idea when tracking a dataframe as the resulting flowchart will have many many branches. `dtrackr` will detect this issue and pause tracking the dataframe with a warning. It is up to the user to the `resume()` tracking when the large number of groups have been resolved e.g. using a `dplyr::ungroup()`. This limit is configurable with `options("dtrackr.max_supported_groupings"=XX)`. The default is 16. See `dplyr::group_by()`.

Usage

```
## S3 method for class 'trackr_df'
group_by(
  .data,
  ...,
  .add = FALSE,
  .drop = dplyr::group_by_drop_default(.data),
  .messages = "stratify by {.cols}",
  .headline = NULL,
  .tag = NULL,
  .maxgroups = .defaultMaxSupportedGroupings()
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr column expressions.
`.add`	When `FALSE`, the default, `group_by()` will override existing groups. To add to the existing groups, use `.add = TRUE`. This argument was previously called `add`, but that prevented creating a new grouping variable called `add`, and conflicts with our naming conventions.
`.drop`	Drop groups formed by factor levels that don't appear in the data? The default is `TRUE` except when `.data` has been previously grouped with `.drop = FALSE`. See `group_by_drop_default()` for details.
`.messages`	• a set of glue specs. The glue code can use any global variable, or {.cols} which is the columns that are being grouped by.
`.headline`	• a headline glue spec. The glue code can use any global variable, or {.cols}.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.
`.maxgroups`	• the maximum number of subgroups allowed before the tracking is paused.

Value

the .data but grouped.

See Also

dplyr::group_by()

Examples

```
library(dplyr)
tmp = iris %>% track() %>% group_by(Species, .messages="stratify by {.cols}")
tmp %>% comment("{.strata}") %>% history()
```

`group_modify.trackr_df`

Group-wise modification of data and complex operations

Description

Group modifying a data set acts in the normal way. The internal mechanics of the modify function are opaque to the history. This means these can be used to wrap any unsupported operation without losing the history (e.g. `df %>% track() %>% group_modify(function(d,...) { d %>% unsupported_operation() })`) Prior to the operation the size of the group is calculated {.count.in} and after the operation the output size {.count.out} The group {.strata} is also available (if grouped) for reporting See `dplyr::group_modify()`.

Usage

```
## S3 method for class 'trackr_df'
group_modify(
  .data,
  .f,
  ...,
  .keep = FALSE,
  .messages = NULL,
  .headline = .defaultHeadline(),
  .type = "modify",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`.f`	a function as expected by dplyr::group_modify e.g. function(d,g,...) ...do something with d and return a dataframe...
`...`	additional parameters for .f.
`.keep`	• are the grouping variables kept in d, or split out to g (the default)
`.messages`	• a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.headline`	• a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.type`	• default "modify": used to define formatting
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the transformed .data dataframe with the history graph updated.

See Also

dplyr::group_modify()

Examples

```
library(dplyr)
tmp = iris %>% track() %>% group_by(Species)
tmp %>% group_modify(
      function(d,g,...) { return(tibble::tibble(x=runif(10))) },
      .messages="{.count.in} in, {.count.out} out"
) %>% history()
```

`history` *Get the dtrackr history graph*

Description

This provides the raw history graph and is not really intended for mainstream use. The internal structure of the graph is explained below. print and plot S3 methods exist for the dtrackr history graph.

Usage

```
history(.data)
```

Arguments

`.data`	• a dataframe which may be grouped

Value

the history graph. This is a list, of class trackr_graph, containing the following named items:

- excluded - the data items that have been excluded thus far as a nested dataframe
- tags - a dataframe of tag-value pairs containing the summary of the data at named points in the data flow (see `tagged()`)
- nodes - a dataframe of the nodes of the flow chart
- edges - an edgelist (as a dataframe) of the relationships between the nodes in the flow chart
- head - the current most recent nodes added into the graph as a dataframe.

The format of this data may grow over time but these fields are unlikely to be changed.

Examples

```
library(dplyr)
graph = iris %>% track() %>% comment("A comment") %>% history()
ls(graph)
```

`ILPD` *Indian Liver Patient Dataset*

Description

This data set contains 416 liver patient records and 167 non liver patient records. The data set was collected from north east of Andhra Pradesh, India. Selector is a class label used to divide into groups(liver patient or not). This data set contains 441 male patient records and 142 female patient records.

Usage

```
ILPD
```

Format

A data frame with 583 rows and 11 variables:

Age
Gender
Total_Bilirubin
Direct_Bilirubin
Alkaline_Phosphatase
Alamine_Aminotransferase
Aspartate_Aminotransferase
Total_Protein
Albumin
Albumin_Globulin_Ratio
Case_or_Control Selector field used to split the data into two sets (labeled by the experts)

Details

1. Bendi Venkata Ramana, Prof. M. S. Prasad Babu and Prof. N. B. Venkateswarlu, A Critical Comparative Study of Liver Patients from USA and INDIA: An Exploratory Analysis, International Journal of Computer Science Issues, ISSN :1694-0784, May 2012.
2. Bendi Venkata Ramana, Prof. M. S. Prasad Babu and Prof. N. B. Venkateswarlu, A Critical Study of Selected Classification Algorithms for Liver Disease Diagnosis, International Journal of Database Management Systems (IJDMS), Vol.3, No.2, ISSN : 0975-5705, PP 101-114, May 2011.
3. Dua, D. and Graff, C. (2019). UCI Machine Learning Repository `http://archive.ics.uci.edu/ml/`. Irvine, CA: University of California, School of Information and Computer Science.

Source

`http://archive.ics.uci.edu/ml/datasets/ILPD+(Indian+Liver+Patient+Dataset)`

`include_any` *Include any items matching a criteria*

Description

Apply a set of inclusion criteria and dplyr::summarise the actions of the filter to the dtrackr history graph Because of the ... filter specification, all parameters MUST BE NAMED. The criteria work in an alternative manner, i.e. the results include anything that match any of the criteria. If na.rm = TRUE they also keep anything that cannot be evaluated by a criteria - that may be true.

Usage

```
include_any(
  .data,
  ...,
  .headline = .defaultHeadline(),
  na.rm = TRUE,
  .type = "inclusion",
  .asOffshoot = FALSE
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	• a dplyr filter specification as a formula where the RHS is a glue specification, defining the message. This can refer to grouping variables, variables from the environment and {.included} and {.matched} or {.missing} (included = matched+missing), {.count} and {.total} - group and overall counts respectively, e.g. "excluding {.matched} items and {.missing} with missing values".
`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`na.rm`	• (default FALSE) if the filter cannot be evaluated for a row count that row as missing and either exclude it (TRUE) or don't exclude it (FALSE)
`.type`	• default "exclusion": used to define formatting
`.asOffshoot`	• do you want this comment to be an offshoot of the main flow (default = TRUE).

Value

the filtered .data dataframe with the history graph updated with the summary of included items as a new stage

Examples

```
library(dplyr)
iris %>% track() %>% include_any(
      Petal.Length > 5 ~ "{.included} long ones",
      Petal.Length < 2 ~ "{.included} short ones"
) %>% history()
```

`inner_join.trackr_df` *Inner joins*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::inner_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
inner_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Inner join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector

	to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`suffix`	If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
`...`	Other parameters passed onto methods.
`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::inner_join()

`landscape` *Convert page size from portrait to landscape*

Description

Convert page size from portrait to landscape

Usage

```
landscape(size)
```

Arguments

`size`	• list of width and height in inches, e.g. a std_size

Value

a landscape size

`left_join.trackr_df` *Left join*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::left_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
left_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Left join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

`suffix`	If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
`...`	Other parameters passed onto methods.
`keep`	Should the join keys from both `x` and `y` be preserved in the output?
`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::left_join()

`mutate.trackr_df` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
## S3 method for class 'trackr_df'
mutate(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::mutate()

`pause` *Pause tracking the dataframe*

Description

Pause tracking the dataframe

Usage

```
pause(.data)
```

Arguments

`.data`	• a tracked dataframe

Value

the .data dataframe with history graph tracking paused

Examples

```
library(dplyr)
iris %>% track() %>% pause() %>% history()
```

`pivot_longer.trackr_df` *Reshaping data using* `tidyr::pivot_longer`

Description

A drop in replacement for tidyr::pivot_longer which optionally takes a message and headline to store in the history graph. See `tidyr::pivot_longer()`.

Usage

```
## S3 method for class 'trackr_df'
pivot_longer(
  data,
  cols,
  names_to = "name",
  names_prefix = NULL,
  names_sep = NULL,
  names_pattern = NULL,
  names_ptypes = list(),
  names_transform = list(),
  names_repair = "check_unique",
  values_to = "value",
  values_drop_na = FALSE,
  values_ptypes = list(),
  values_transform = list(),
  ...,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`data`	A data frame to pivot.
`cols`	<`tidy-select`> Columns to pivot into longer format.
`names_to`	A character vector specifying the new column or columns to create from the information stored in the column names of `data` specified by `cols`. • If length 0, or if `NULL` is supplied, no columns will be created. • If length 1, a single column will be created which will contain the column names specified by `cols`. • If length >1, multiple columns will be created. In this case, one of `names_sep` or `names_pattern` must be supplied to specify how the column names should be split. There are also two additional character values you can take advantage of: – `NA` will discard the corresponding component of the column name. – `".value"` indicates that the corresponding component of the column name defines the name of the output column containing the cell values, overriding `values_to` entirely.
`names_prefix`	A regular expression used to remove matching text from the start of each variable name.
`names_sep`	If `names_to` contains multiple values, these arguments control how the column name is broken up. `names_sep` takes the same specification as `separate()`, and can either be a numeric vector (specifying positions to break on), or a single string (specifying a regular expression to split on).

	`names_pattern` takes the same specification as `extract()`, a regular expression containing matching groups (`()`). If these arguments do not give you enough control, use `pivot_longer_spec()` to create a spec object and process manually as needed.
`names_pattern`	If `names_to` contains multiple values, these arguments control how the column name is broken up. `names_sep` takes the same specification as `separate()`, and can either be a numeric vector (specifying positions to break on), or a single string (specifying a regular expression to split on). `names_pattern` takes the same specification as `extract()`, a regular expression containing matching groups (`()`). If these arguments do not give you enough control, use `pivot_longer_spec()` to create a spec object and process manually as needed.
`names_ptypes`	Optionally, a list of column name-prototype pairs. Alternatively, a single empty prototype can be supplied, which will be applied to all columns. A prototype (or ptype for short) is a zero-length vector (like `integer()` or `numeric()`) that defines the type, class, and attributes of a vector. Use these arguments if you want to confirm that the created columns are the types that you expect. Note that if you want to change (instead of confirm) the types of specific columns, you should use `names_transform` or `values_transform` instead. For backwards compatibility reasons, supplying `list()` is interpreted as being identical to `NULL` rather than as using a list prototype on all columns. Expect this to change in the future.
`names_transform`	Optionally, a list of column name-function pairs. Alternatively, a single function can be supplied, which will be applied to all columns. Use these arguments if you need to change the types of specific columns. For example, `names_transform = list(week = as.integer)` would convert a character variable called `week` to an integer. If not specified, the type of the columns generated from `names_to` will be character, and the type of the variables generated from `values_to` will be the common type of the input columns used to generate them.
`names_repair`	What happens if the output has invalid column names? The default, `"check_unique"` is to error if the columns are duplicated. Use `"minimal"` to allow duplicates in the output, or `"unique"` to de-duplicated by adding numeric suffixes. See `vctrs::vec_as_names()` for more options.
`values_to`	A string specifying the name of the column to create from the data stored in cell values. If `names_to` is a character containing the special `.value` sentinel, this value will be ignored, and the name of the value column will be derived from part of the existing column names.
`values_drop_na`	If `TRUE`, will drop rows that contain only `NA`s in the `value_to` column. This effectively converts explicit missing values to implicit missing values, and should generally be used only when missing values in `data` were created by its structure.
`values_ptypes`	Optionally, a list of column name-prototype pairs. Alternatively, a single empty prototype can be supplied, which will be applied to all columns. A prototype

	(or ptype for short) is a zero-length vector (like `integer()` or `numeric()`) that defines the type, class, and attributes of a vector. Use these arguments if you want to confirm that the created columns are the types that you expect. Note that if you want to change (instead of confirm) the types of specific columns, you should use `names_transform` or `values_transform` instead. For backwards compatibility reasons, supplying `list()` is interpreted as being identical to `NULL` rather than as using a list prototype on all columns. Expect this to change in the future.
`values_transform`	Optionally, a list of column name-function pairs. Alternatively, a single function can be supplied, which will be applied to all columns. Use these arguments if you need to change the types of specific columns. For example, `names_transform = list(week = as.integer)` would convert a character variable called `week` to an integer. If not specified, the type of the columns generated from `names_to` will be character, and the type of the variables generated from `values_to` will be the common type of the input columns used to generate them.
`...`	Additional arguments passed on to methods.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the result of the tidyr::pivot_wider but with a history graph updated.

See Also

tidyr::pivot_longer()

`pivot_wider.trackr_df` *Reshaping data using* `tidyr::pivot_wider`

Description

A drop in replacement for `tidyr::pivot_wider` which optionally takes a message and headline to store in the history graph. See `tidyr::pivot_wider()`.

Usage

```
## S3 method for class 'trackr_df'
pivot_wider(
  data,
  id_cols = NULL,
  names_from = as.symbol("name"),
  names_prefix = "",
  names_sep = "_",
  names_glue = NULL,
  names_sort = FALSE,
  names_repair = "check_unique",
  values_from = as.symbol("value"),
  values_fill = NULL,
  values_fn = NULL,
  ...,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`data`	A data frame to pivot.
`id_cols`	<`tidy-select`> A set of columns that uniquely identifies each observation. Defaults to all columns in `data` except for the columns specified in `names_from` and `values_from`. Typically used when you have redundant variables, i.e. variables whose values are perfectly correlated with existing variables.
`names_from`	<`tidy-select`> A pair of arguments describing which column (or columns) to get the name of the output column (`names_from`), and which column (or columns) to get the cell values from (`values_from`). If `values_from` contains multiple values, the value will be added to the front of the output column.
`names_prefix`	String added to the start of every variable name. This is particularly useful if `names_from` is a numeric vector and you want to create syntactic variable names.
`names_sep`	If `names_from` or `values_from` contains multiple variables, this will be used to join their values together into a single string to use as a column name.
`names_glue`	Instead of `names_sep` and `names_prefix`, you can supply a glue specification that uses the `names_from` columns (and special `.value`) to create custom column names.
`names_sort`	Should the column names be sorted? If `FALSE`, the default, column names are ordered by first appearance.
`names_repair`	What happens if the output has invalid column names? The default, `"check_unique"` is to error if the columns are duplicated. Use `"minimal"` to allow duplicates in the output, or `"unique"` to de-duplicated by adding numeric suffixes. See `vctrs::vec_as_names()` for more options.

`values_from`	<`tidy-select`> A pair of arguments describing which column (or columns) to get the name of the output column (`names_from`), and which column (or columns) to get the cell values from (`values_from`). If `values_from` contains multiple values, the value will be added to the front of the output column.
`values_fill`	Optionally, a (scalar) value that specifies what each `value` should be filled in with when missing. This can be a named list if you want to apply different fill values to different value columns.
`values_fn`	Optionally, a function applied to the value in each cell in the output. You will typically use this when the combination of `id_cols` and `names_from` columns does not uniquely identify an observation. This can be a named list if you want to apply different aggregations to different `values_from` columns.
`...`	Additional arguments passed on to methods.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the data dataframe result of the tidyr::pivot_wider function but with a history graph updated with a `.message` if requested.

See Also

tidyr::pivot_wider()

`plot.trackr_graph` *Plots a history graph as html*

Description

Plots a history graph as html

Usage

```
## S3 method for class 'trackr_graph'
plot(x, fill = "lightgrey", fontsize = "8", colour = "black", ...)
```

Arguments

`x`	a dtrackr history graph (e.g. output from `history()`)
`fill`	• the default node fill colour
`fontsize`	• the default font size
`colour`	• the default font colour
`...`	not used

Value

HTML displayed

Examples

```
library(dplyr)
iris %>% comment("hello {.total} rows") %>% history() %>% plot()
```

`print.trackr_graph` *Print a history graph to the console*

Description

Print a history graph to the console

Usage

```
## S3 method for class 'trackr_graph'
print(x, ...)
```

Arguments

`x`	a dtrackr history graph (e.g. output from `p_get()`)
`...`	not used

Value

nothing

Examples

```
library(dplyr)
iris %>% comment("hello {.total} rows") %>% history() %>% print()
```

`p_add_count` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
p_add_count(
  .data,
  ...,
  wt = NULL,
  sort = FALSE,
  name = NULL,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`wt`	<`data-masking`> Frequency weights. Can be `NULL` or a variable: • If `NULL` (the default), counts the number of rows in each group. • If a variable, computes `sum(wt)` for each group.
`sort`	If `TRUE`, will show the largest groups at the top.
`name`	The name of the new column in the output. If omitted, it will default to `n`. If there's already a column called `n`, it will error, and require you to specify the name.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::add_count()

`p_add_tally` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
p_add_tally(
  .data,
  ...,
  wt = NULL,
  sort = FALSE,
  name = NULL,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`wt`	<`data-masking`> Frequency weights. Can be `NULL` or a variable: • If `NULL` (the default), counts the number of rows in each group. • If a variable, computes `sum(wt)` for each group.
`sort`	If `TRUE`, will show the largest groups at the top.
`name`	The name of the new column in the output. If omitted, it will default to `n`. If there's already a column called `n`, it will error, and require you to specify the name.

`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::add_tally()

`p_anti_join` *Anti join*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::anti_join()` for more details on the underlying functions.

Usage

```
p_anti_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} not matched"),
  .headline = "Semi join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.

`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`...`	Other parameters passed onto methods.
`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::anti_join()

`p_arrange` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
p_arrange(
  .data,
  ...,
  .by_group = FALSE,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.by_group`	If `TRUE`, will sort first by grouping variable. Applies to grouped data frames only.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::arrange()

`p_bind_rows` *Union of two or more data sets*

Description

This merges the history of 2 dataframes and binds the rows. It calculates the total number of resulting rows as .count.out in other terms it performs exactly the same operation as dplyr::bind_rows. See `dplyr::bind_rows()`.

Usage

```
p_bind_rows(
  ...,
  .id = NULL,
  .messages = "{.count.out} in union",
  .headline = "Union"
)
```

Arguments

`...`	the data frames to bind
`.id`	Data frame identifier. When `.id` is supplied, a new column of identifiers is created to link each row to its original data frame. The labels are taken from the named arguments to `bind_rows()`. When a list of data frames is supplied, the labels are taken from the names of the list. If no names are found a numeric sequence is used instead.
`.messages`	• a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	• a glue spec. The glue code can use any global variable, or {.count.out}

Value

the logical union of the dataframes with the history graph updated.

See Also

dplyr::bind_rows()

Examples

```
library(dplyr)
bind_rows( iris %>% comment("one"), iris %>% comment("two") ) %>% history()
```

`p_capture_exclusions` *Start capturing exclusions on a tracked dataframe.*

Description

Start capturing exclusions on a tracked dataframe.

Usage

```
p_capture_exclusions(.data, .capture = TRUE)
```

Arguments

`.data`	• a tracked dataframe
`.capture`	• Should we capture exclusions (things removed from the data set). This is useful for debugging data issues but comes at a significant cost. Defaults to the value of `getOption("dtrackr.exclusions")` or `FALSE`.

Value

the .data dataframe with the exclusions flag set (or cleared if `.capture=FALSE`).

Examples

```
library(dplyr)
tmp = iris %>% track() %>% capture_exclusions()
tmp %>% filter(Species!="versicolor") %>% history()
```

`p_clear` *Clear the dtrackr history graph*

Description

Clear the dtrackr history graph

Usage

```
p_clear(.data)
```

Arguments

`.data`	• a dataframe which may be grouped

Value

the .data dataframe with the history graph removed

Examples

```
library(dplyr)
mtcars %>% track() %>% comment("A comment") %>% p_clear() %>% history()
```

`p_comment` *Add a generic comment to the dtrackr history graph*

Description

A comment can be any kind fo note and is added once for every current grouping as defined by the `.message` field. It can be made context specific by including variables such as {.count} and {.total} in `.message` which refer to the grouped and ungrouped counts at this current stage of the pipeline for example. It can also pull in any global variable.

Usage

```
p_comment(
  .data,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = (.type == "exclusion"),
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`.messages`	• a character vector of glue specifications. A glue specification can refer to any grouping variables of .data, or any variables defined in the calling environment, the {.total} of all rows, the {.count} variable which is the count in each group and {.strata} a description of the group
`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment, or the {.total} variable which is nrow(.data)and {.strata}
`.type`	• one of "info","...,"exclusion": used to define formatting
`.asOffshoot`	• do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the same .data dataframe with the history graph updated with the comment

Examples

```
library(dplyr)
iris %>% track() %>% comment("hello {.total} rows") %>% history()
```

`p_copy` *Copy the dtrackr history graph from one df to another*

Description

Copy the dtrackr history graph from one df to another

Usage

```
p_copy(.data, from)
```

Arguments

`.data`	• a dataframe which may be grouped
`from`	• the dataframe to copy the history graph from

Value

the .data dataframe with the history graph of "from"

Examples

```
library(dplyr)
mtcars %>% p_copy(iris %>% comment("A comment")) %>% history()
```

`p_count_if` *Simple count_if dplyr summary function*

Description

Simple count_if dplyr summary function

Usage

```
p_count_if(..., na.rm = TRUE)
```

Arguments

`...`	• expression to be evaluated
`na.rm`	• ignore NA values?

Value

a count of the number of times the expression evaluated to true, in the current context

Examples

```
library(dplyr)
tmp = iris %>% dplyr::group_by(Species)
tmp %>% dplyr::summarise(long_ones = p_count_if(Petal.Length > 4))
```

`p_count_subgroup` *Add a subgroup count to the dtrackr history graph*

Description

A frequent use case for more detailed description is to have a subgroup count within a flowchart. This works best for factor subgroup columns but other data will be converted to a factor automatically. The count of the items in each subgroup is added as a new stage in the flowchart.

Usage

```
p_count_subgroup(
  .data,
  .subgroup,
  ...,
  .messages = .defaultCountSubgroup(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = FALSE,
  .tag = NULL,
  .maxsubgroups = .defaultMaxSupportedGroupings()
)
```

Arguments

`.data`	• a dataframe which may be grouped
`.subgroup`	• a column with a small number of levels (e.g.)
`...`	• additional parameters will be passed to factor(subgroup,...) to control levels, ordering, etc.
`.messages`	• a character vector of glue specifications. A glue specification can refer to anything from the calling environment and .name for the subgroup name, .count for the subgroup count, .subtotal for the current grouping count and .total for the whole count
`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`.type`	• one of "info","exclusion": used to define formatting
`.asOffshoot`	• do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.
`.maxsubgroups`	• the maximum number of discrete values allowed is configurable with `options("dtrackr.max_supp` The default is 16.

Value

the same .data dataframe with the history graph updated with a subgroup count as a new stage

Examples

```
library(dplyr)
ILPD %>% track() %>% group_by(Case_or_Control) %>% count_subgroup(Gender) %>% history()
```

`p_distinct` *Distinct values of data*

Description

Distinct acts in the same way as in `dplyr::distinct`. Prior to the operation the size of the group is calculated {.count.in} and after the operation the output size {.count.out} The group {.strata} is also available (if grouped) for reporting See `dplyr::distinct()`.

Usage

```
p_distinct(
  .data,
  .f,
  ...,
  .keep = FALSE,
  .messages = "removing {.count.in-.count.out} duplicates",
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`.f`	a function as expected by dplyr::group_modify e.g. function(d,g,...) ...do something with d and return a dataframe...
`...`	additional parameters for .f.
`.keep`	• are the grouping variables kept in d, or split out to g (the default)
`.messages`	• a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.headline`	• a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe with distinct values and history graph updated.

See Also

dplyr::distinct()

Examples

```
library(dplyr)
tmp = bind_rows(iris %>% track(), iris %>% track() %>% filter(Petal.Length > 5))
tmp %>% group_by(Species) %>% distinct() %>% history()
```

`p_excluded` *Get the dtrackr excluded data record*

Description

Get the dtrackr excluded data record

Usage

```
p_excluded(.data, simplify = TRUE)
```

Arguments

`.data`	• a dataframe which may be grouped
`simplify`	• return a single summary dataframe of all exclusions.

Value

a new dataframe of the excluded data up to this point in the workflow. This dataframe is by default flattened, but if `.simplify=FALSE` has a nested structure containing records excluded at each part of the pipeline.

Examples

```
library(dplyr)
tmp = iris %>% track() %>% capture_exclusions()
tmp %>% exclude_all(
   Petal.Length > 5.8 ~ "{.excluded} long ones",
   Petal.Length < 1.3 ~ "{.excluded} short ones",
   .stage = "petal length exclusion"
) %>% excluded()
```

`p_exclude_all` *Exclude all items matching one or more criteria*

Description

Apply a set of filters and summarise the actions of the filter to the dtrackr history graph Because of the ... filter specification, all parameters MUST BE NAMED. The filters work in an additive manner, i.e. the results excluding all things that match any of the criteria. If na.rm = TRUE they also remove anything that cannot be evaluated by a criteria.

Usage

```
p_exclude_all(
  .data,
  ...,
  .headline = .defaultHeadline(),
  na.rm = FALSE,
  .type = "exclusion",
  .asOffshoot = TRUE,
  .stage = ""
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	• a dplyr filter specification as a formula where the RHS is a glue specification, defining the message. This can refer to grouping variables variables from the environment and {.excluded} and {.matched} or {.missing} (excluded = matched+missing), {.count} and {.total} - group and overall counts respectively, e.g. "excluding {.matched} items and {.missing} with missing values".
`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`na.rm`	• (default FALSE) if the filter cannot be evaluated for a row count that row as missing and either exclude it (TRUE) or don't exclude it (FALSE)
`.type`	• default "exclusion": used to define formatting
`.asOffshoot`	• do you want this comment to be an offshoot of the main flow (default = TRUE).
`.stage`	• a name for this step in the pathway

Value

the filtered .data dataframe with the history graph updated with the summary of excluded items as a new offshoot stage

Examples

```
library(dplyr)
iris %>% track() %>% capture_exclusions() %>% exclude_all(
      Petal.Length > 5 ~ "{.excluded} long ones",
      Petal.Length < 2 ~ "{.excluded} short ones"
) %>% history()
```

`p_filter` *Filtering data*

Description

Filter acts in the same way as DPLYR. Prior to the operation the size of the group is calculated {.count.in} and after the operation the output size {.count.out}. The group {.strata} is also available (if grouped) for reporting. See `dplyr::filter()`.

Usage

```
p_filter(
  .data,
  ...,
  .preserve = FALSE,
  .messages = "excluded {.excluded} items",
  .headline = .defaultHeadline(),
  .type = "exclusion",
  .asOffshoot = (.type == "exclusion"),
  .stage = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	the filter criteria
`.preserve`	Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is.
`.messages`	• a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.headline`	• a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.type`	• the format type of the action - typically an exclusion
`.asOffshoot`	• if the type is exclusion, asOffshoot places the information box outside of the main flow, as an exclusion.
`.stage`	• a name for this step in the pathway
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the filtered .data dataframe with history graph updated

See Also

dplyr::filter()

Examples

```
library(dplyr)
tmp = iris %>% track() %>% group_by(Species)
tmp %>% filter(Petal.Length > 5) %>% history()
```

`p_flowchart` *Flowchart output*

Description

Generate a flowchart of the history of the dataframe, with all the transformations as stages in the flowchart.

Usage

```
p_flowchart(
  .data,
  filename = NULL,
  size = std_size$half,
  maxWidth = size$width,
  maxHeight = size$height,
  rot = size$rot,
  formats = c("dot", "png", "pdf", "svg"),
  defaultToHTML = TRUE,
  ...
)
```

Arguments

`.data`	• the tracked dataframes
`filename`	• a filename (without extension) which will be where the formatted flowcharts are saved
`size`	• a list of length and width in inches e.g. a std_size
`maxWidth`	• a width in inches is size is not defined
`maxHeight`	• a height in inches if size is not defined
`rot`	• an angle of rotation for the saved file if size is not defined
`formats`	• some of "pdf","dot","svg","png","ps"

`defaultToHTML`	- if the correct output format is not easy to determine from the context, default providing HTML or to embedding the PNG
`...`	- other params passed onto p_get_as_dot, notable ones are fill, fontsize, colour, size, maxWidth and maxHeight

Value

the nature of the flowchart output depends on the context in which the function is called. It will be some form of browse-able html output if called from an interactive session or a PNG/PDG link if in knitr and knitting latex or word type outputs,

Examples

```
library(dplyr)
tmp = iris %>% track() %>% comment(.tag = "step1") %>% filter(Species!="versicolor")
tmp %>% group_by(Species) %>% comment(.tag="step2") %>% flowchart()
```

`p_full_join` *Full join*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::full_join()` for more details on the underlying functions.

Usage

```
p_full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Full join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.

`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`suffix`	If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
`...`	Other parameters passed onto methods.
`keep`	Should the join keys from both `x` and `y` be preserved in the output?
`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::full_join()

`p_get` *Get the dtrackr history graph*

Description

This provides the raw history graph and is not really intended for mainstream use. The internal structure of the graph is explained below. print and plot S3 methods exist for the dtrackr history graph.

Usage

```
p_get(.data)
```

Arguments

`.data`	• a dataframe which may be grouped

Value

the history graph. This is a list, of class trackr_graph, containing the following named items:

- excluded - the data items that have been excluded thus far as a nested dataframe
- tags - a dataframe of tag-value pairs containing the summary of the data at named points in the data flow (see `tagged()`)
- nodes - a dataframe of the nodes of the flow chart
- edges - an edgelist (as a dataframe) of the relationships between the nodes in the flow chart
- head - the current most recent nodes added into the graph as a dataframe.

The format of this data may grow over time but these fields are unlikely to be changed.

Examples

```
library(dplyr)
graph = iris %>% track() %>% comment("A comment") %>% history()
ls(graph)
```

`p_get_as_dot` *DOT output*

Description

(advance usage) outputs a dtrackr history graph as a DOT string for rendering with `Graphviz`

Usage

```
p_get_as_dot(.data, fill = "lightgrey", fontsize = "8", colour = "black", ...)
```

Arguments

`.data`	• the tracked dataframe
`fill`	• the default node fill colour
`fontsize`	• the default font size
`colour`	• the default font colour
`...`	• not used

Value

a representation of the history graph in `Graphviz` dot format.

Examples

```
library(dplyr)
tmp = iris %>% track() %>% comment(.tag = "step1") %>% filter(Species!="versicolor")
dot = tmp %>% group_by(Species) %>% comment(.tag="step2") %>% p_get_as_dot()
cat(dot)
```

`p_group_by` *Stratifying your analysis*

Description

Grouping a data set acts in the normal way. When tracking a dataframe sometimes a `group_by()` operation will create a lot of groups. This happens for example if you are doing a `group_by()`, `summarise()` step that is aggregating data on a fine scale, e.g. by day in a timeseries. This is generally a terrible idea when tracking a dataframe as the resulting flowchart will have many many branches. `dtrackr` will detect this issue and pause tracking the dataframe with a warning. It is up to the user to the `resume()` tracking when the large number of groups have been resolved e.g. using a `dplyr::ungroup()`. This limit is configurable with `options("dtrackr.max_supported_groupings"=XX)`. The default is 16. See `dplyr::group_by()`.

Usage

```
p_group_by(
  .data,
  ...,
  .add = FALSE,
  .drop = dplyr::group_by_drop_default(.data),
  .messages = "stratify by {.cols}",
  .headline = NULL,
  .tag = NULL,
  .maxgroups = .defaultMaxSupportedGroupings()
)
```

Arguments

`.data`	- a dataframe which may be grouped
`...`	a set of dplyr column expressions.
`.add`	When `FALSE`, the default, `group_by()` will override existing groups. To add to the existing groups, use `.add = TRUE`. This argument was previously called `add`, but that prevented creating a new grouping variable called `add`, and conflicts with our naming conventions.
`.drop`	Drop groups formed by factor levels that don't appear in the data? The default is `TRUE` except when `.data` has been previously grouped with `.drop = FALSE`. See `group_by_drop_default()` for details.
`.messages`	- a set of glue specs. The glue code can use any global variable, or {.cols} which is the columns that are being grouped by.

`.headline`	- a headline glue spec. The glue code can use any global variable, or {.cols}.
`.tag`	- if you want the summary data from this step in the future then give it a name with .tag.
`.maxgroups`	- the maximum number of subgroups allowed before the tracking is paused.

Value

the .data but grouped.

See Also

dplyr::group_by()

Examples

```
library(dplyr)
tmp = iris %>% track() %>% group_by(Species, .messages="stratify by {.cols}")
tmp %>% comment("{.strata}") %>% history()
```

`p_group_modify` *Group-wise modification of data and complex operations*

Description

Group modifying a data set acts in the normal way. The internal mechanics of the modify function are opaque to the history. This means these can be used to wrap any unsupported operation without losing the history (e.g. `df %>% track() %>% group_modify(function(d,...) { d %>% unsupported_operation() })`) Prior to the operation the size of the group is calculated {.count.in} and after the operation the output size {.count.out} The group {.strata} is also available (if grouped) for reporting See `dplyr::group_modify()`.

Usage

```
p_group_modify(
  .data,
  .f,
  ...,
  .keep = FALSE,
  .messages = NULL,
  .headline = .defaultHeadline(),
  .type = "modify",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`.f`	a function as expected by dplyr::group_modify e.g. function(d,g,...) ...do something with d and return a dataframe...
`...`	additional parameters for .f.
`.keep`	• are the grouping variables kept in d, or split out to g (the default)
`.messages`	• a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.headline`	• a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.type`	• default "modify": used to define formatting
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the transformed .data dataframe with the history graph updated.

See Also

dplyr::group_modify()

Examples

```
library(dplyr)
tmp = iris %>% track() %>% group_by(Species)
tmp %>% group_modify(
      function(d,g,...) { return(tibble::tibble(x=runif(10))) },
      .messages="{.count.in} in, {.count.out} out"
) %>% history()
```

`p_include_any` *Include any items matching a criteria*

Description

Apply a set of inclusion criteria and dplyr::summarise the actions of the filter to the dtrackr history graph Because of the ... filter specification, all parameters MUST BE NAMED. The criteria work in an alternative manner, i.e. the results include anything that match any of the criteria. If na.rm = TRUE they also keep anything that cannot be evaluated by a criteria - that may be true.

Usage

```
p_include_any(
  .data,
  ...,
  .headline = .defaultHeadline(),
  na.rm = TRUE,
  .type = "inclusion",
  .asOffshoot = FALSE
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	• a dplyr filter specification as a formula where the RHS is a glue specification, defining the message. This can refer to grouping variables, variables from the environment and {.included} and {.matched} or {.missing} (included = matched+missing), {.count} and {.total} - group and overall counts respectively, e.g. "excluding {.matched} items and {.missing} with missing values".
`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`na.rm`	• (default FALSE) if the filter cannot be evaluated for a row count that row as missing and either exclude it (TRUE) or don't exclude it (FALSE)
`.type`	• default "exclusion": used to define formatting
`.asOffshoot`	• do you want this comment to be an offshoot of the main flow (default = TRUE).

Value

the filtered .data dataframe with the history graph updated with the summary of included items as a new stage

Examples

```
library(dplyr)
iris %>% track() %>% include_any(
      Petal.Length > 5 ~ "{.included} long ones",
      Petal.Length < 2 ~ "{.included} short ones"
) %>% history()
```

`p_inner_join` *Inner joins*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::inner_join()` for more details on the underlying functions.

Usage

```
p_inner_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Inner join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`suffix`	If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

`...`	Other parameters passed onto methods.
`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::inner_join()

`p_left_join` *Left join*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::left_join()` for more details on the underlying functions.

Usage

```
p_left_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Left join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.

`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`suffix`	If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
`...`	Other parameters passed onto methods.
`keep`	Should the join keys from both `x` and `y` be preserved in the output?
`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::left_join()

`p_mutate` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
p_mutate(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	- a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.messages`	- a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	- a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	- if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::mutate()

`p_pause` *Pause tracking the dataframe*

Description

Pause tracking the dataframe

Usage

```
p_pause(.data)
```

Arguments

`.data`	- a tracked dataframe

Value

the .data dataframe with history graph tracking paused

Examples

```
library(dplyr)
iris %>% track() %>% pause() %>% history()
```

`p_pivot_longer` *Reshaping data using* `tidyr::pivot_longer`

Description

A drop in replacement for tidyr::pivot_longer which optionally takes a message and headline to store in the history graph. See `tidyr::pivot_longer()`.

Usage

```
p_pivot_longer(
  data,
  cols,
  names_to = "name",
  names_prefix = NULL,
  names_sep = NULL,
  names_pattern = NULL,
  names_ptypes = list(),
  names_transform = list(),
  names_repair = "check_unique",
  values_to = "value",
  values_drop_na = FALSE,
  values_ptypes = list(),
  values_transform = list(),
  ...,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`data`	A data frame to pivot.
`cols`	<`tidy-select`> Columns to pivot into longer format.
`names_to`	A character vector specifying the new column or columns to create from the information stored in the column names of `data` specified by `cols`.

- If length 0, or if `NULL` is supplied, no columns will be created.
- If length 1, a single column will be created which will contain the column names specified by `cols`.
- If length >1, multiple columns will be created. In this case, one of `names_sep` or `names_pattern` must be supplied to specify how the column names should be split. There are also two additional character values you can take advantage of:
  - `NA` will discard the corresponding component of the column name.

	– `".value"` indicates that the corresponding component of the column name defines the name of the output column containing the cell values, overriding `values_to` entirely.
`names_prefix`	A regular expression used to remove matching text from the start of each variable name.
`names_sep`	If `names_to` contains multiple values, these arguments control how the column name is broken up. `names_sep` takes the same specification as `separate()`, and can either be a numeric vector (specifying positions to break on), or a single string (specifying a regular expression to split on). `names_pattern` takes the same specification as `extract()`, a regular expression containing matching groups (`()`). If these arguments do not give you enough control, use `pivot_longer_spec()` to create a spec object and process manually as needed.
`names_pattern`	If `names_to` contains multiple values, these arguments control how the column name is broken up. `names_sep` takes the same specification as `separate()`, and can either be a numeric vector (specifying positions to break on), or a single string (specifying a regular expression to split on). `names_pattern` takes the same specification as `extract()`, a regular expression containing matching groups (`()`). If these arguments do not give you enough control, use `pivot_longer_spec()` to create a spec object and process manually as needed.
`names_ptypes`	Optionally, a list of column name-prototype pairs. Alternatively, a single empty prototype can be supplied, which will be applied to all columns. A prototype (or ptype for short) is a zero-length vector (like `integer()` or `numeric()`) that defines the type, class, and attributes of a vector. Use these arguments if you want to confirm that the created columns are the types that you expect. Note that if you want to change (instead of confirm) the types of specific columns, you should use `names_transform` or `values_transform` instead. For backwards compatibility reasons, supplying `list()` is interpreted as being identical to `NULL` rather than as using a list prototype on all columns. Expect this to change in the future.
`names_transform`	Optionally, a list of column name-function pairs. Alternatively, a single function can be supplied, which will be applied to all columns. Use these arguments if you need to change the types of specific columns. For example, `names_transform = list(week = as.integer)` would convert a character variable called `week` to an integer. If not specified, the type of the columns generated from `names_to` will be character, and the type of the variables generated from `values_to` will be the common type of the input columns used to generate them.
`names_repair`	What happens if the output has invalid column names? The default, `"check_unique"` is to error if the columns are duplicated. Use `"minimal"` to allow duplicates in the output, or `"unique"` to de-duplicated by adding numeric suffixes. See `vctrs::vec_as_names()` for more options.

`values_to`	A string specifying the name of the column to create from the data stored in cell values. If `names_to` is a character containing the special `.value` sentinel, this value will be ignored, and the name of the value column will be derived from part of the existing column names.
`values_drop_na`	If `TRUE`, will drop rows that contain only `NA`s in the `value_to` column. This effectively converts explicit missing values to implicit missing values, and should generally be used only when missing values in `data` were created by its structure.
`values_ptypes`	Optionally, a list of column name-prototype pairs. Alternatively, a single empty prototype can be supplied, which will be applied to all columns. A prototype (or ptype for short) is a zero-length vector (like `integer()` or `numeric()`) that defines the type, class, and attributes of a vector. Use these arguments if you want to confirm that the created columns are the types that you expect. Note that if you want to change (instead of confirm) the types of specific columns, you should use `names_transform` or `values_transform` instead. For backwards compatibility reasons, supplying `list()` is interpreted as being identical to `NULL` rather than as using a list prototype on all columns. Expect this to change in the future.
`values_transform`	Optionally, a list of column name-function pairs. Alternatively, a single function can be supplied, which will be applied to all columns. Use these arguments if you need to change the types of specific columns. For example, `names_transform = list(week = as.integer)` would convert a character variable called `week` to an integer. If not specified, the type of the columns generated from `names_to` will be character, and the type of the variables generated from `values_to` will be the common type of the input columns used to generate them.
`...`	Additional arguments passed on to methods.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the result of the tidyr::pivot_wider but with a history graph updated.

See Also

tidyr::pivot_longer()

`p_pivot_wider` *Reshaping data using* `tidyr::pivot_wider`

Description

A drop in replacement for `tidyr::pivot_wider` which optionally takes a message and headline to store in the history graph. See `tidyr::pivot_wider()`.

Usage

```
p_pivot_wider(
  data,
  id_cols = NULL,
  names_from = as.symbol("name"),
  names_prefix = "",
  names_sep = "_",
  names_glue = NULL,
  names_sort = FALSE,
  names_repair = "check_unique",
  values_from = as.symbol("value"),
  values_fill = NULL,
  values_fn = NULL,
  ...,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`data`	A data frame to pivot.
`id_cols`	<`tidy-select`> A set of columns that uniquely identifies each observation. Defaults to all columns in `data` except for the columns specified in `names_from` and `values_from`. Typically used when you have redundant variables, i.e. variables whose values are perfectly correlated with existing variables.
`names_from`	<`tidy-select`> A pair of arguments describing which column (or columns) to get the name of the output column (`names_from`), and which column (or columns) to get the cell values from (`values_from`). If `values_from` contains multiple values, the value will be added to the front of the output column.
`names_prefix`	String added to the start of every variable name. This is particularly useful if `names_from` is a numeric vector and you want to create syntactic variable names.
`names_sep`	If `names_from` or `values_from` contains multiple variables, this will be used to join their values together into a single string to use as a column name.

`names_glue`	Instead of `names_sep` and `names_prefix`, you can supply a glue specification that uses the `names_from` columns (and special `.value`) to create custom column names.
`names_sort`	Should the column names be sorted? If `FALSE`, the default, column names are ordered by first appearance.
`names_repair`	What happens if the output has invalid column names? The default, `"check_unique"` is to error if the columns are duplicated. Use `"minimal"` to allow duplicates in the output, or `"unique"` to de-duplicated by adding numeric suffixes. See `vctrs::vec_as_names()` for more options.
`values_from`	<`tidy-select`> A pair of arguments describing which column (or columns) to get the name of the output column (`names_from`), and which column (or columns) to get the cell values from (`values_from`). If `values_from` contains multiple values, the value will be added to the front of the output column.
`values_fill`	Optionally, a (scalar) value that specifies what each `value` should be filled in with when missing. This can be a named list if you want to apply different fill values to different value columns.
`values_fn`	Optionally, a function applied to the value in each cell in the output. You will typically use this when the combination of `id_cols` and `names_from` columns does not uniquely identify an observation. This can be a named list if you want to apply different aggregations to different `values_from` columns.
`...`	Additional arguments passed on to methods.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the data dataframe result of the tidyr::pivot_wider function but with a history graph updated with a `.message` if requested.

See Also

tidyr::pivot_wider()

`p_relocate` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
p_relocate(
  .data,
  ...,
  .before = NULL,
  .after = NULL,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.before`	<`tidy-select`> Destination of columns selected by `...`. Supplying neither will move columns to the left-hand side; specifying both is an error.
`.after`	<`tidy-select`> Destination of columns selected by `...`. Supplying neither will move columns to the left-hand side; specifying both is an error.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::relocate()

`p_rename` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
p_rename(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	- a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.messages`	- a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	- a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	- if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::rename()

`p_rename_with` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
p_rename_with(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	- a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.messages`	- a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	- a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	- if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::rename_with()

`p_resume` *Resume tracking the dataframe. This may reset the grouping of the tracked data*

Description

Resume tracking the dataframe. This may reset the grouping of the tracked data

Usage

```
p_resume(.data)
```

Arguments

`.data`	• a tracked dataframe

Value

the .data dataframe with history graph tracking resumed

Examples

```
library(dplyr)
iris %>% track() %>% pause() %>% resume() %>% history()
```

`p_right_join` *Right join*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::right_join()` for more details on the underlying functions.

Usage

```
p_right_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Right join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`suffix`	If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
`...`	Other parameters passed onto methods.
`keep`	Should the join keys from both `x` and `y` be preserved in the output?
`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::right_join()

`p_select` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
p_select(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::select()

`p_semi_join` *Semi join*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::semi_join()` for more details on the underlying functions.

Usage

```
p_semi_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in intersection"),
  .headline = "Semi join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`...`	Other parameters passed onto methods.
`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::semi_join()

`p_set` *Set the dtrackr history graph*

Description

This is unlikely to be useful to an end user and is called automatically by many of the other functions here. On the off chance you need to copy history metadata from one dataframe to another

Usage

```
p_set(.data, .graph)
```

Arguments

`.data`	• a dataframe which may be grouped
`.graph`	• a history graph list (consisting of nodes, edges, and head) see examples

Value

the .data dataframe with the history graph metadata set to the provided value

Examples

```
library(dplyr)
mtcars %>% p_set(iris %>% comment("A comment") %>% p_get()) %>% history()
```

`p_status` *Add a summary to the dtrackr history graph*

Description

In the middle of a pipeline you may wish to document something about the data that is more complex than the simple counts. `status` is essentially a `dplyr` summarisation step which is connected to a `glue` specification output, that is recorded in the data frame history. This means you can do an arbitrary summarisation and put the result into the flowchart.

Usage

```
p_status(
  .data,
  ...,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = FALSE,
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	• any normal dplyr::summarise specification, e.g. count=n() or av=mean(x) etc.
`.messages`	• a character vector of glue specifications. A glue specification can refer to the summary outputs, any grouping variables of .data, the {.strata}, or any variables defined in the calling environment
`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`.type`	• one of "info","exclusion": used to define formatting
`.asOffshoot`	• do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Details

Because of the ... summary specification parameters MUST BE NAMED.

Value

the same .data dataframe with the history metadata updated with the status inserted as a new stage

Examples

```
library(dplyr)
tmp = iris %>% track() %>% group_by(Species)
tmp %>% status(
      long = p_count_if(Petal.Length>5),
      short = p_count_if(Petal.Length<2),
      .messages="{Species}: {long} long ones & {short} short ones"
) %>% history()
```

`p_summarise` *Summarise a data set*

Description

Summarising a data set acts in the normal way. Any columns resulting form the summary can be added to the history graph In the history this joins any stratified branches and acts as a specific type of p_summary, allowing you to generate some summary statistics about the un-grouped data. See `dplyr::summarise()`.

Usage

```
p_summarise(
  .data,
  ...,
  .groups = NULL,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.groups`	• Experimental lifecycle Grouping structure of the result.
`.messages`	• a set of glue specs. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.headline`	• a headline glue spec. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe summarised with the history graph updated showing the summarise operation as a new stage

See Also

dplyr::summarise()

Examples

```
library(dplyr)
tmp = iris %>% group_by(Species)
tmp %>% summarise(avg = mean(Petal.Length), .messages="{avg} length") %>% history()
```

`p_tagged` *Retrieve tagged data in the history graph*

Description

Any counts at the individual stages that was stored with a `.tag` option in a pipeline step can be recovered here. The idea here is to provide a quick way to access a single value for the counts or other details tagged in a pipeline into a format that can be reported in text of a document. (e.g. for a results section). For more examples the consort statement vignette has some examples of use.

Usage

```
p_tagged(.data, .tag = NULL, .strata = NULL, .glue = NULL, ...)
```

Arguments

`.data`	the tracked dataframe.
`.tag`	(optional) the tag to retrieve.
`.strata`	(optional) filter the tagged data by the strata. set to "" to filter just the top level ungrouped data.
`.glue`	(optional) a glue specification which will be applied to the tagged content to generate a `.label` for the tagged content.
`...`	(optional) any other named parameters will be passed to `glue::glue` and can be used to generate a label.

Value

various things depending on what is requested.

By default a tibble with a `.tag` column and all associated summary values in a nested `.content` column.

If a `.strata` column is specified the results are filtered to just those that match a given `.strata` grouping (i.e. this will be the grouping label on the flowchart). Ungrouped content will have an empty "" as `.strata`

If `.tag` is specified the result will be for a single tag and `.content` will be automatically un-nested to give a single un-nested dataframe of the content captured at the `.tag` tagged step. This could be single or multiple rows depending on whether the original data was grouped at the point of tagging.

If both the `.tag` and `.glue` is specified a `.label` column will be computed from `.glue` and the tagged content. If the result of this is a single row then just the string value of `.label` is returned.

If just the `.glue` is specified, an un-nested dataframe with `.tag`,`.strata` and `.label` columns with a label for each tag in each strata.

If this seems complex then the best thing is to experiment until you get the output you want, leaving any `.glue` options until you think you know what you are doing. It made sense at the time.

Examples

```
library(dplyr)
tmp = iris %>% track() %>% comment(.tag = "step1")
tmp = tmp %>% filter(Species!="versicolor") %>% group_by(Species)
tmp %>% comment(.tag="step2") %>% tagged(.glue = "{.count}/{.total}")
```

`p_track` *Start tracking the dtrackr history graph*

Description

Start tracking the dtrackr history graph

Usage

```
p_track(
  .data,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`.messages`	• a character vector of glue specifications. A glue specification can refer to any grouping variables of .data, or any variables defined in the calling environment, the {.total} variable which is the count of all rows, the {.count} variable which is the count of rows in the current group and the {.strata} which describes the current group. Defaults to the value of `getOption("dtrackr.default_messag`
`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment, or the {.total} variable which is nrow(.data), or {.strata} a summary of the current group. Defaults to the value of getOption("dtrackr.default_headline").
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe with additional history graph metadata, to allow tracking.

Examples

```
library(dplyr)
iris %>% track() %>% history()
```

`p_transmute` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
p_transmute(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	- a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.messages`	- a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	- a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	- if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::transmute()

`p_ungroup` *Remove a stratification from a data set*

Description

Un-grouping a data set logically combines the different arms. In the history this joins any stratified branches and acts as a specific type of p_summary, allowing you to generate some summary statistics about the un-grouped data. See `dplyr::ungroup()`.

Usage

```
p_ungroup(
  x,
  ...,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`x`	• a dataframe which may be grouped (why not .data?)
`...`	• passed to dplyr::ungroup()
`.messages`	• a set of glue specs. The glue code can use any any global variable, or {.count}. the default is "total {.count} items"
`.headline`	• a headline glue spec. The glue code can use {.count} and {.strata}.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe but dplyr::ungrouped with the history graph updated showing the ungroup operation as a new stage.

See Also

dplyr::ungroup()

Examples

```
library(dplyr)
tmp = iris %>% group_by(Species) %>% comment("A test")
tmp %>% ungroup(.messages="{.count} items") %>% history()
```

`p_untrack` *Remove tracking from the dataframe*

Description

Remove tracking from the dataframe

Usage

```
p_untrack(.data)
```

Arguments

`.data`	- a tracked dataframe

Value

the .data dataframe with history graph metadata removed.

Examples

```
library(dplyr)
iris %>% track() %>% untrack() %>% class()
```

`relocate.trackr_df` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
## S3 method for class 'trackr_df'
relocate(
  .data,
  ...,
  .before = NULL,
  .after = NULL,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.before`	<`tidy-select`> Destination of columns selected by `...`. Supplying neither will move columns to the left-hand side; specifying both is an error.
`.after`	<`tidy-select`> Destination of columns selected by `...`. Supplying neither will move columns to the left-hand side; specifying both is an error.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::relocate()

`rename.trackr_df` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
## S3 method for class 'trackr_df'
rename(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	- a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.messages`	- a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	- a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	- if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::rename()

`rename_with.trackr_df` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
## S3 method for class 'trackr_df'
rename_with(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	- a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.messages`	- a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	- a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	- if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::rename_with()

`resume` *Resume tracking the dataframe. This may reset the grouping of the tracked data*

Description

Resume tracking the dataframe. This may reset the grouping of the tracked data

Usage

```
resume(.data)
```

Arguments

`.data`	• a tracked dataframe

Value

the .data dataframe with history graph tracking resumed

Examples

```
library(dplyr)
iris %>% track() %>% pause() %>% resume() %>% history()
```

`right_join.trackr_df` *Right join*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::right_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
right_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Right join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`suffix`	If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
`...`	Other parameters passed onto methods.
`keep`	Should the join keys from both `x` and `y` be preserved in the output?
`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::right_join()

`save_dot` *Save DOT content to a file*

Description

Convert a digraph in dot file to SVG and save it to an output file

Usage

```
save_dot(
  dot,
  filename,
  size = std_size$half,
  maxWidth = size$width,
  maxHeight = size$height,
  rot = size$rot,
  formats = c("dot", "png", "pdf", "svg")
)
```

Arguments

`dot`	• a graphviz dot string
`filename`	• the full path of the filename (minus extension for multiple formats)
`size`	• a list of length and width in inches e.g. a std_size
`maxWidth`	• a width in inches is size is not defined
`maxHeight`	• a height in inches if size is not defined
`rot`	• an angle of rotation for the saved file if size is not defined
`formats`	• some of "pdf","dot","svg","png","ps"

Value

a list with items `paths` with the absolute paths of the saved files, and svg as the SVG string of the rendered dot file.

Examples

```
dot2svg("digraph {A->B} ")
```

`select.trackr_df` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
## S3 method for class 'trackr_df'
select(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::select()

`semi_join.trackr_df` *Semi join*

Description

Mutating joins behave as dplyr joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::semi_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
semi_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in intersection"),
  .headline = "Semi join by {.keys}"
)
```

Arguments

`x`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly. To join by different variables on `x` and `y`, use a named vector. For example, `by = c("a" = "b")` will match `x$a` to `y$b`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`...`	Other parameters passed onto methods.

`.messages`	• a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	• a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::semi_join()

`status` *Add a summary to the dtrackr history graph*

Description

In the middle of a pipeline you may wish to document something about the data that is more complex than the simple counts. `status` is essentially a `dplyr` summarisation step which is connected to a `glue` specification output, that is recorded in the data frame history. This means you can do an arbitrary summarisation and put the result into the flowchart.

Usage

```
status(
  .data,
  ...,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = FALSE,
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	• any normal dplyr::summarise specification, e.g. count=n() or av=mean(x) etc.
`.messages`	• a character vector of glue specifications. A glue specification can refer to the summary outputs, any grouping variables of .data, the {.strata}, or any variables defined in the calling environment
`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment

`.type`	- one of "info","exclusion": used to define formatting
`.asOffshoot`	- do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	- if you want the summary data from this step in the future then give it a name with .tag.

Details

Because of the ... summary specification parameters MUST BE NAMED.

Value

the same .data dataframe with the history metadata updated with the status inserted as a new stage

Examples

```
library(dplyr)
tmp = iris %>% track() %>% group_by(Species)
tmp %>% status(
      long = p_count_if(Petal.Length>5),
      short = p_count_if(Petal.Length<2),
      .messages="{Species}: {long} long ones & {short} short ones"
) %>% history()
```

`std_size` *Standard paper sizes*

Description

A list of standard paper sizes

Usage

```
std_size
```

Format

An object of class `list` of length 8.

`summarise.trackr_df` *Summarise a data set*

Description

Summarising a data set acts in the normal way. Any columns resulting form the summary can be added to the history graph In the history this joins any stratified branches and acts as a specific type of p_summary, allowing you to generate some summary statistics about the un-grouped data. See `dplyr::summarise()`.

Usage

```
## S3 method for class 'trackr_df'
summarise(
  .data,
  ...,
  .groups = NULL,
  .messages = "",
  .headline = "",
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.groups`	• Experimental lifecycle Grouping structure of the result.
`.messages`	• a set of glue specs. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.headline`	• a headline glue spec. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe summarised with the history graph updated showing the summarise operation as a new stage

See Also

dplyr::summarise()

Examples

```
library(dplyr)
tmp = iris %>% group_by(Species)
tmp %>% summarise(avg = mean(Petal.Length), .messages="{avg} length") %>% history()
```

`tagged` *Retrieve tagged data in the history graph*

Description

Any counts at the individual stages that was stored with a `.tag` option in a pipeline step can be recovered here. The idea here is to provide a quick way to access a single value for the counts or other details tagged in a pipeline into a format that can be reported in text of a document. (e.g. for a results section). For more examples the consort statement vignette has some examples of use.

Usage

```
tagged(.data, .tag = NULL, .strata = NULL, .glue = NULL, ...)
```

Arguments

`.data`	the tracked dataframe.
`.tag`	(optional) the tag to retrieve.
`.strata`	(optional) filter the tagged data by the strata. set to "" to filter just the top level ungrouped data.
`.glue`	(optional) a glue specification which will be applied to the tagged content to generate a `.label` for the tagged content.
`...`	(optional) any other named parameters will be passed to `glue::glue` and can be used to generate a label.

Value

various things depending on what is requested.

By default a tibble with a `.tag` column and all associated summary values in a nested `.content` column.

If a `.strata` column is specified the results are filtered to just those that match a given `.strata` grouping (i.e. this will be the grouping label on the flowchart). Ungrouped content will have an empty "" as `.strata`

If `.tag` is specified the result will be for a single tag and `.content` will be automatically un-nested to give a single un-nested dataframe of the content captured at the `.tag` tagged step. This could be single or multiple rows depending on whether the original data was grouped at the point of tagging.

If both the `.tag` and `.glue` is specified a `.label` column will be computed from `.glue` and the tagged content. If the result of this is a single row then just the string value of `.label` is returned.

If just the `.glue` is specified, an un-nested dataframe with `.tag`,`.strata` and `.label` columns with a label for each tag in each strata.

If this seems complex then the best thing is to experiment until you get the output you want, leaving any `.glue` options until you think you know what you are doing. It made sense at the time.

Examples

```
library(dplyr)
tmp = iris %>% track() %>% comment(.tag = "step1")
tmp = tmp %>% filter(Species!="versicolor") %>% group_by(Species)
tmp %>% comment(.tag="step2") %>% tagged(.glue = "{.count}/{.total}")
```

`track` *Start tracking the dtrackr history graph*

Description

Start tracking the dtrackr history graph

Usage

```
track(
  .data,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`.data`	• a dataframe which may be grouped
`.messages`	• a character vector of glue specifications. A glue specification can refer to any grouping variables of .data, or any variables defined in the calling environment, the {.total} variable which is the count of all rows, the {.count} variable which is the count of rows in the current group and the {.strata} which describes the current group. Defaults to the value of `getOption("dtrackr.default_message`
`.headline`	• a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment, or the {.total} variable which is nrow(.data), or {.strata} a summary of the current group. Defaults to the value of getOption("dtrackr.default_headline").
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe with additional history graph metadata, to allow tracking.

Examples

```
library(dplyr)
iris %>% track() %>% history()
```

`transmute.trackr_df` *Standard dplyr modifying operations*

Description

Equivalent Dplyr functions for mutating, selecting and renaming a data set act in the normal way. mutates / selects / rename generally don't add anything in documentation so the default behaviour is to miss these out of the history. This can be overridden with the .messages, or .headline values in which case they behave just like a `comment()` See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details.

Usage

```
## S3 method for class 'trackr_df'
transmute(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	• a dataframe which may be grouped
`...`	a set of dplyr summary expressions.
`.messages`	• a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	• a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe after being modified by the dplyr equivalent function, but with the history graph updated with a new stage if the `.messages` field is not empty

See Also

dplyr::transmute()

`ungroup.trackr_df` *Remove a stratification from a data set*

Description

Un-grouping a data set logically combines the different arms. In the history this joins any stratified branches and acts as a specific type of p_summary, allowing you to generate some summary statistics about the un-grouped data. See `dplyr::ungroup()`.

Usage

```
## S3 method for class 'trackr_df'
ungroup(
  x,
  ...,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`x`	• a dataframe which may be grouped (why not .data?)
`...`	• passed to dplyr::ungroup()
`.messages`	• a set of glue specs. The glue code can use any any global variable, or {.count}. the default is "total {.count} items"
`.headline`	• a headline glue spec. The glue code can use {.count} and {.strata}.
`.tag`	• if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe but dplyr::ungrouped with the history graph updated showing the ungroup operation as a new stage.

See Also

dplyr::ungroup()

Examples

```
library(dplyr)
tmp = iris %>% group_by(Species) %>% comment("A test")
tmp %>% ungroup(.messages="{.count} items") %>% history()
```

`untrack` *Remove tracking from the dataframe*

Description

Remove tracking from the dataframe

Usage

```
untrack(.data)
```

Arguments

`.data`	• a tracked dataframe

Value

the .data dataframe with history graph metadata removed.

Examples

```
library(dplyr)
iris %>% track() %>% untrack() %>% class()
```

Index