Package 'surveytidy'

May 19, 2026

Title Tidy 'dplyr'/'tidyr' Verbs for Survey Design Objects

Version 0.6.0

Description Provides 'dplyr' and 'tidyr' verbs, survey-aware recoding helpers, and row-wise statistics for survey design objects created with the 'surveycore' package. filter() uses domain estimation to preserve variance estimation validity; other verbs preserve design variables and metadata automatically. Also supports survey_collection objects for applying the same operation across a list of surveys.

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.3

Config/testthat/edition 3

Depends R (>= 4.3.0)

Imports cli (>= 3.6.0), dplyr (>= 1.2.0), haven (>= 2.5.0), rlang (>= 1.1.0), S7 (>= 0.1.0), surveycore (>= 0.8.2), tidyr (>= 1.3.0), tidyselect (>= 1.2.0), vctrs (>= 0.6.0), withr (>= 2.5.0)

URL https://jdenn0514.github.io/surveytidy/, https://github.com/JDenn0514/surveytidy

BugReports https://github.com/JDenn0514/surveytidy/issues

Suggests covr, mockery, pkgdown, testthat (>= 3.0.0), tibble

Config/Needs/website rmarkdown

Config/Needs/coverage covr

NeedsCompilation no

Author Jacob Dennen [aut, cre, cph] (ORCID: <https://orcid.org/0000-0003-3006-7364>)

Maintainer Jacob Dennen <jdenn0514@gmail.com>

Repository CRAN

Date/Publication 2026-05-19 07:50:02 UTC

Contents

`arrange` *Order rows using column values*

Description

`arrange()` orders the rows of a `survey_base` object by the values of selected columns.

Unlike most other verbs, `arrange()` largely ignores grouping — use `.by_group = TRUE` to sort by grouping variables first.

Usage

```
## S3 method for class 'survey_base'
arrange(.data, ..., .by_group = FALSE, .locale = NULL)

## S3 method for class 'survey_result'
arrange(.data, ..., .by_group = FALSE)

## S3 method for class 'survey_collection'
arrange(.data, ..., .by_group = FALSE, .locale = NULL, .if_missing_var = NULL)

arrange(.data, ..., .by_group = FALSE)
```

Arguments

.data	A survey_base object, or a survey_result object returned by a surveycore estimation function.
...	<data-masking> Variables, or functions of variables. Use dplyr::desc() to sort a variable in descending order.
.by_group	If TRUE, sorts first by the grouping variables set by group_by().
.locale	The locale to use for ordering strings. If NULL, uses the "C" locale. See stringi::locale() for available locales.
.if_missing_var	Per-call override of collection@if_missing_var. One of "error" or "skip", or NULL (the default) to inherit the collection's stored value. See surveycore::set_collection_if_miss

Details

Missing values:
Unlike base sort(), NA values are always sorted to the end, even when using dplyr::desc().

Domain column:
The domain column moves with the rows — row reordering does not affect which rows are in or out of the survey domain.

Value

An object of the same type as .data with the following properties:

- All rows appear in the output, usually in a different position.
- Columns are not modified.
- Groups are not modified.
- Survey design attributes are preserved.

Survey collections

When applied to a survey_collection, arrange() is dispatched to each member independently. Each member's rows are sorted in place; per-member domain columns travel with the sorted rows. The output survey_collection preserves the input's @id, @if_missing_var, and @groups. Use .if_missing_var to override the collection's stored missing-variable behavior for this call.

See Also

`filter()` for domain-aware row marking, `slice()` for physical row selection

Examples

```
library(surveytidy)
library(surveycore)

# create a survey design from the pew_npors_2025 example dataset
d <- as_survey(pew_npors_2025, weights = weight, strata = stratum)

# sort by age category ascending
arrange(d, agecat)

# sort by age category descending
arrange(d, dplyr::desc(agecat))

# sort by multiple variables
arrange(d, gender, dplyr::desc(agecat))

# sort by grouping variables first
d_grouped <- group_by(d, gender)
arrange(d_grouped, .by_group = TRUE, agecat)
```

---

bind_cols *Append columns to a survey design by position*

Description

`bind_cols()` appends columns from one or more plain data frames to a survey design object, matching by row position. This is equivalent to an implicit row-index `left_join()`. All rows are preserved; row count is unchanged.

When `x` is not a survey object, this function delegates to `dplyr::bind_cols()` transparently.

Usage

```
bind_cols(x, ..., .name_repair = "unique")
```

Arguments

`x`	A `survey_base` object, or any object accepted by `dplyr::bind_cols()`.
`...`	One or more plain data frames or named lists. When `x` is a survey object, none of the objects may be survey objects.
`.name_repair`	Forwarded to `dplyr::bind_cols()`.

Details

Design integrity:

None of the objects in ... may be a survey object. If any new column name matches a design variable in x, that column is dropped with a warning. All inputs in ... must have exactly the same number of rows as x.

Dispatch note:

`dplyr::bind_cols()` uses `vctrs::vec_cbind()` internally and does not dispatch via S3 on x. surveytidy provides its own `bind_cols()` that intercepts survey objects before delegating to dplyr.

Value

When x is a survey object: a survey design object of the same type as x with new columns appended to `@data`. `visible_vars` is updated if it was set. When x is not a survey object: the result of `dplyr::bind_cols()`.

See Also

Other joins: `bind_rows()`, `inner_join`, `left_join`, `right_join`, `semi_join`

Examples

```
library(surveytidy)

# create a small survey object
df <- data.frame(
  psu = paste0("psu_", 1:5),
  strata = "s1",
  fpc = 100,
  wt = 1,
  y1 = 1:5
)
d <- surveycore::as_survey(
  df,
  ids = psu,
  weights = wt,
  strata = strata,
  fpc = fpc,
  nest = TRUE
)

# append a new column by row position
extra <- data.frame(label = letters[1:5])
bind_cols(d, extra)
```

bind_rows *Stack surveys with bind_rows (errors unconditionally)*

Description

`bind_rows()` errors unconditionally when the first argument is a survey design object. Stacking two surveys changes the design — the combined object requires a new design specification (e.g., a new survey-wave stratum).

When the first argument is not a survey object, this function delegates to `dplyr::bind_rows()` transparently.

Usage

```
bind_rows(x, ..., .id = NULL)
```

Arguments

`x`	A `survey_base` object (always errors), or any object accepted by `dplyr::bind_rows()` (transparent delegation).
`...`	Additional arguments.
`.id`	Forwarded to `dplyr::bind_rows()`.

Details

Known limitation: If the survey object is passed as a **non-first** argument (e.g., `bind_rows(df, survey)`), this function delegates to `dplyr::bind_rows(df, survey)` which will fail with a dplyr/vctrs error rather than the survey-specific error. Always pass the survey object as the first argument to ensure the correct error is triggered.

Dispatch note:

`dplyr::bind_rows()` uses `vctrs::vec_rbind()` internally for recent dplyr versions and does not reliably dispatch via S3 on `x` for S7 objects. surveytidy provides its own `bind_rows()` that intercepts survey objects before delegating to dplyr (GAP-6 verified: S3 dispatch does not work; standalone function approach used instead).

Value

Never returns when `x` is a survey object — always throws an error. When `x` is not a survey object, returns the result of `dplyr::bind_rows()`.

See Also

Other joins: `bind_cols()`, `inner_join`, `left_join`, `right_join`, `semi_join`

Examples

```
# NOTE: do not load dplyr here — its bind_rows() would mask surveytidy's
# bind_rows() and bypass the survey-object check shown below.

# two raw data frames that together define a combined survey
df1 <- data.frame(wt = c(1, 1), y1 = c(1, 2))
df2 <- data.frame(wt = c(1, 1), y1 = c(3, 4))

# bind_rows() on plain data frames delegates to dplyr::bind_rows()
bind_rows(df1, df2)

# but bind_rows() on a survey object always errors — stacking two surveys
# would change the design, requiring a new design specification
d1 <- surveycore::as_survey(df1, weights = wt)

tryCatch(
  bind_rows(d1, df2),
  error = function(e) message(conditionMessage(e))
)

# the recommended workflow: extract raw data from each survey, bind, then
# re-specify the design on the combined data frame
combined <- bind_rows(
  surveycore::survey_data(d1),
  df2
)
surveycore::as_survey(combined, weights = wt)
```

---

case_when *A generalised vectorised if-else*

Description

case_when() is a survey-aware version of dplyr::case_when() that evaluates each formula case sequentially and uses the first match for each element to determine the output value.

Use case_when() when creating an entirely new vector. When partially updating an existing vector, replace_when() is a better choice — it retains the original value wherever no case matches and inherits existing value labels from the input automatically.

When any of .label, .value_labels, .factor, or .description are supplied, output label metadata is written to @metadata after mutate(). When none of these arguments are used, the output is identical to dplyr::case_when().

Usage

```
case_when(
  ...,
  .default = NULL,
```

```
  .unmatched = "default",
  .ptype = NULL,
  .size = NULL,
  .label = NULL,
  .value_labels = NULL,
  .factor = FALSE,
  .description = NULL
)
```

Arguments

`...`	<dynamic-dots> A sequence of two-sided formulas (`condition ~ value`). The left-hand side must be a logical vector. The right-hand side provides the replacement value. Cases are evaluated sequentially; the first matching case is used. `NULL` inputs are ignored.
`.default`	The value used when all LHS conditions return `FALSE` or `NA`. If `NULL` (the default), unmatched rows receive `NA`.
`.unmatched`	Handling of unmatched rows. `"default"` (the default) uses `.default`; `"error"` raises an error if any row is unmatched.
`.ptype`	An optional prototype declaring the desired output type. Overrides the common type of the RHS inputs.
`.size`	An optional size declaring the desired output length. Overrides the common size computed from the LHS inputs.
`.label`	`character(1)` or `NULL`. Variable label stored in `@metadata@variable_labels` after `mutate()`. Cannot be combined with `.factor = TRUE`.
`.value_labels`	Named vector or `NULL`. Value labels stored in `@metadata@value_labels`. Names are the label strings; values are the data values.
`.factor`	`logical(1)`. If `TRUE`, returns a factor. Levels are ordered by the RHS values in formula order, or by `.value_labels` names if supplied. Cannot be combined with `.label`.
`.description`	`character(1)` or `NULL`. Plain-language description of how the variable was created. Stored in `@metadata@transformations[[col]]$description` after `mutate()`.

Value

A vector, factor, or `haven_labelled` vector:

- No surveytidy args — same output as `dplyr::case_when()`.
- `.factor = TRUE` — a factor with levels in RHS formula order.
- `.label` or `.value_labels` supplied — a `haven_labelled` vector.

See Also

- `dplyr::case_when()` for the base implementation.
- `replace_when()` to partially update an existing vector; also inherits existing value labels from the input automatically.

- `if_else()` for the two-condition case.
- `recode_values()` for value-mapping with explicit `from`/`to` vectors.

Other recoding: `if_else()`, `na_if()`, `recode_values()`, `replace_values()`, `replace_when()`

Examples

```
library(surveycore)
library(surveytidy)

# create the survey design
ns_wave1_svy <- as_survey_nonprob(
  ns_wave1,
  weights = weight
)

# basic case_when — identical to dplyr::case_when()
new <- ns_wave1_svy |>
  mutate(
    age_pid = case_when(
      age < 30 & pid3 == 1 ~ "18-29 Democrats",
      age < 30 & pid3 == 2 ~ "18-29 Republicans",
      age < 30 & pid3 %in% c(3:4) ~ "18-29 Independents",
      .default = "Everyone else"
    )
  ) |>
  select(age, pid3, age_pid)

# by default, no metadata is attached
new
new@metadata

# attach a variable label via .label
new <- ns_wave1_svy |>
  mutate(
    age_pid = case_when(
      age < 30 & pid3 == 1 ~ "18-29 Democrats",
      age < 30 & pid3 == 2 ~ "18-29 Republicans",
      age < 30 & pid3 %in% c(3:4) ~ "18-29 Independents",
      .default = "Everyone else",
      .label = "Age and Partisanship"
    )
  ) |>
  select(age, pid3, age_pid)

new@metadata@variable_labels

# attach a plain-language description of the transformation
new <- ns_wave1_svy |>
  mutate(
    age_pid = case_when(
      age < 30 & pid3 == 1 ~ "18-29 Democrats",
      age < 30 & pid3 == 2 ~ "18-29 Republicans",
```

```
      age < 30 & pid3 %in% c(3:4) ~ "18-29 Independents",
      .default = "Everyone else",
      .label = "Age and Partisanship",
      .description = paste(
        "Young (< 30) Democrats, Republicans, and Independents",
        "were grouped by partisanship; everyone else was set to",
        "'Everyone else'."
      )
    )
  ) |>
  select(age, pid3, age_pid)

new@metadata@transformations

# attach value labels alongside numeric codes
new <- ns_wave1_svy |>
  mutate(
    age_pid = case_when(
      age < 30 & pid3 == 1 ~ 1,
      age < 30 & pid3 == 2 ~ 2,
      age < 30 & pid3 %in% c(3:4) ~ 3,
      .default = 4,
      .label = "Age and Partisanship",
      .value_labels = c(
        "18-29 Democrats" = 1,
        "18-29 Republicans" = 2,
        "18-29 Independents" = 3,
        "Everyone else" = 4
      )
    )
  ) |>
  select(age, pid3, gender, age_pid)

new@metadata@value_labels

# return a factor with levels in formula order
new <- ns_wave1_svy |>
  mutate(
    age_pid = case_when(
      age < 30 & pid3 == 1 ~ "18-29 Democrats",
      age < 30 & pid3 == 2 ~ "18-29 Republicans",
      age < 30 & pid3 %in% c(3:4) ~ "18-29 Independents",
      .default = "Everyone else",
      .factor = TRUE
    )
  ) |>
  select(age, pid3, age_pid)

new
```

distinct *Remove duplicate rows from a survey design object*

Description

`distinct()` **physically removes duplicate rows** from a survey design object, always issuing `surveycore_warning_physical_subset`. Unlike `dplyr::distinct()`, all columns in `@data` are retained regardless of which columns are specified in `...` — design variables must never be lost from the survey object.

For subpopulation analyses, use `filter()` instead — it marks rows out-of-domain without removing them, preserving valid variance estimation.

Usage

```
## S3 method for class 'survey_base'
distinct(.data, ..., .keep_all = FALSE)

## S3 method for class 'survey_collection'
distinct(.data, ..., .keep_all = FALSE, .if_missing_var = NULL)

distinct(.data, ..., .keep_all = FALSE)
```

Arguments

`.data`	A `survey_base` object.
`...`	<`data-masking`> Optional columns used to determine uniqueness. If empty, all non-design columns are used. Note: `.keep_all` is always `TRUE` regardless of what is specified here.
`.keep_all`	Accepted for interface compatibility; **has no effect**. The survey implementation always retains all columns in `@data`.
`.if_missing_var`	Per-call override of `collection@if_missing_var`. One of `"error"` or `"skip"`, or `NULL` (the default) to inherit the collection's stored value. See `surveycore::set_collection_if_mis`

Details

Column retention:

`distinct()` always behaves as if `.keep_all = TRUE`. Specifying columns in `...` controls which columns determine uniqueness — it does **not** control which columns appear in the result. This is a deliberate divergence from `dplyr::distinct(df, x, y)` which by default drops all columns except `x` and `y`.

Default deduplication (empty `...`):

When `...` is empty, deduplication uses all non-design columns. Design variables (strata, PSU, weights, FPC) are excluded from the uniqueness check — deduplicating on them would produce meaningless or survey-corrupting results.

Design variable warning:

If ... includes a design variable, `surveytidy_warning_distinct_design_var` is issued before the operation. The operation still proceeds after the warning — the user is assumed to know what they are doing.

Value

An object of the same class as `.data` with the following properties:

- Rows physically reduced to distinct subset (fewer rows possible).
- All columns in `@data` are retained (`.keep_all = TRUE` always).
- `@variables$visible_vars` is unchanged — distinct is a pure row operation.
- `@metadata` is unchanged.
- `@groups` is unchanged.
- Always issues `surveycore_warning_physical_subset`.

Survey collections

When applied to a `survey_collection`, `distinct()` is dispatched to each member independently — there is no cross-survey deduplication. Two members that share a literally identical row will both retain that row in their post-`distinct()` results. This is the V9 contract from the survey-collection spec; collections deliberately avoid the `bind_rows()` analogy here because cross-survey deduplication has no coherent variance interpretation across designs.

Each member's `distinct.survey_base` issues `surveycore_warning_physical_subset` independently — N firings on an N-member collection. Capture with `withCallingHandlers()`.

See Also

`filter()` for domain-aware row marking (preferred for subpopulation analyses)

Other row operations: `drop_na`, `slice`

Examples

```
library(surveytidy)
library(surveycore)

# create a survey design from the pew_npors_2025 example dataset
d <- as_survey(pew_npors_2025, weights = weight, strata = stratum)

# deduplicate on all non-design columns (issues physical-subset warning)
distinct(d)

# deduplicate by one column (all other columns still retained)
distinct(d, cregion)
```

drop_na *Mark rows with missing values as out-of-domain*

Description

drop_na() marks rows where specified columns contain NA as out-of-domain, without removing them. If no columns are specified, any NA in any column marks the row out-of-domain.

This is the domain-aware equivalent of tidyr's drop_na(): rather than physically dropping rows, it applies filter() with !is.na() conditions, preserving all rows for correct variance estimation.

Usage

```
## S3 method for class 'survey_base'
drop_na(data, ...)

## S3 method for class 'survey_result'
drop_na(data, ...)

## S3 method for class 'survey_collection'
drop_na(data, ...)

drop_na(data, ...)
```

Arguments

data	A survey_base object, or a survey_result object returned by a surveycore estimation function.
...	<tidy-select> Columns to inspect for NA. If empty, all columns are checked.

Details

Chaining:

Successive drop_na() calls AND their conditions together, and they accumulate with filter() calls too. These are equivalent:

```
drop_na(d, bpxsy1) |> filter(ridageyr >= 18)
filter(d, !is.na(bpxsy1), ridageyr >= 18)
```

Value

An object of the same type as data with the following properties:

- Rows are not added or removed.
- Rows where selected columns contain NA are marked out-of-domain.
- Columns and survey design attributes are unchanged.

Survey collections

When applied to a survey_collection, drop_na() is dispatched to each member independently with the same .... Per-member empty-domain warnings fire as usual. The collection's stored @if_missing_var controls behavior when a tidyselect-named column is absent from one or more members; detection mode is class-catch (the tidyselect error is caught at dispatch time).

Unlike other collection verbs, drop_na() does not accept a per-call .if_missing_var argument: tidyr's drop_na() generic calls rlang::check_dots_unnamed() before S3 dispatch, which rejects any named ... argument. Use surveycore::set_collection_if_missing_var() to change the collection's stored behavior instead.

See Also

filter() for domain-aware row marking

Other row operations: distinct, slice

Examples

```
library(tidyr)

# create a survey object from the bundled NPORS dataset
d <- surveycore::as_survey(
  surveycore::pew_npors_2025,
  weights = weight,
  strata = stratum
)

# mark rows with NA in votegen_post as out-of-domain
drop_na(d, votegen_post)

# mark rows with NA in either social media column
drop_na(d, smuse_fb, smuse_yt)

# no columns specified — any NA in any column marks the row out-of-domain
drop_na(d)
```

---

filter *Keep or drop rows using domain estimation*

Description

filter() and filter_out() mark rows as in or out of the survey domain without removing them. Unlike a standard data frame filter, all rows are always retained — only their domain status changes. Estimation functions restrict analysis to in-domain rows while using the full design for variance estimation.

filter() marks rows **matching** the condition as in-domain. filter_out() marks rows **matching** the condition as out-of-domain — it is the complement of filter(), and reads more naturally when the intent is exclusion.

Usage

```
## S3 method for class 'survey_base'
filter(.data, ..., .by = NULL, .preserve = FALSE)

## S3 method for class 'survey_result'
filter(.data, ...)

## S3 method for class 'survey_collection'
filter(.data, ..., .by = NULL, .preserve = FALSE, .if_missing_var = NULL)

## S3 method for class 'survey_base'
filter_out(.data, ..., .by = NULL, .preserve = FALSE)

## S3 method for class 'survey_collection'
filter_out(.data, ..., .by = NULL, .preserve = FALSE, .if_missing_var = NULL)

filter(.data, ..., .by = NULL, .preserve = FALSE)
```

Arguments

.data	A survey_base object, or a survey_result object returned by a surveycore estimation function.
...	<data-masking> Logical conditions evaluated against the survey data. Multiple conditions are combined with &. NA results are treated as FALSE.
.by	Not supported for survey objects. Use group_by() to add grouping.
.preserve	Ignored. Included for compatibility with the dplyr generic.
.if_missing_var	Per-call override of collection@if_missing_var. One of "error" or "skip", or NULL (the default) to inherit the collection's stored value. See surveycore::set_collection_if_miss

Details

Chaining:

Multiple calls accumulate via AND: a row must satisfy every condition to remain in-domain. These are equivalent:

```
filter(d, ridageyr >= 18, riagendr == 2)
filter(d, ridageyr >= 18) |> filter(riagendr == 2)
```

Missing values:

Unlike base [, both functions treat NA as FALSE: rows where the condition evaluates to NA are treated as out-of-domain.

Useful filter functions:

- Comparisons: ==, >, >=, <, <=, !=
- Logical: &, |, !, xor()
- Missing values: is.na()

- Range: dplyr::between(), dplyr::near()
- Multi-column: dplyr::if_any(), dplyr::if_all()

Inspecting the domain:

The domain status of each row is stored in the ..surveycore_domain.. column of @data. TRUE means in-domain; FALSE means out-of-domain.

Value

An object of the same type as .data with the following properties:

- All rows appear in the output.
- Domain status of each row may be updated.
- Columns are not modified.
- Groups are not modified.
- Survey design attributes are preserved.

Survey collections

When applied to a survey_collection, filter() is dispatched to each member independently. Each member's domain column is updated per filter.survey_base's contract; the per-member empty-domain warning (surveycore_warning_empty_domain) fires N times on an N-member collection if every member's filter is empty. The output survey_collection preserves the input's @id, @if_missing_var, and @groups. Use .if_missing_var to override the collection's stored missing-variable behavior for this call.

.by is rejected at the collection layer with surveytidy_error_collection_by_unsupported. Set grouping with group_by() on the collection instead.

See Also

filter_out() for excluding rows, subset() for physical row removal

Examples

```
library(surveytidy)
library(surveycore)

# create a survey design from the pew_npors_2025 example dataset
d <- as_survey(pew_npors_2025, weights = weight, strata = stratum)

# keep adults 50 and older
filter(d, agecat >= 3)

# multiple conditions are AND-ed together
filter(d, agecat >= 3, gender == 2)

# filter_out() excludes matching rows — complement of filter()
filter_out(d, agecat == 1)
```

```
# chained calls accumulate (these are equivalent)
filter(d, agecat >= 3, gender == 2)
d |>
  filter(agecat >= 3) |>
  filter(gender == 2)

# multi-column conditions with if_any() and if_all()
filter(d, dplyr::if_any(c(smuse_fb, smuse_yt), ~ !is.na(.x)))
filter(d, dplyr::if_all(c(smuse_fb, smuse_yt), ~ !is.na(.x)))
```

`glimpse.survey_collection` *Get a glimpse of a survey design object*

Description

Print a transposed summary of the survey object's columns — column names run down the left, data types and values run across. Respects `select()`: if columns have been selected, only those columns are shown; design variables are hidden from the display.

Usage

```
## S3 method for class 'survey_collection'
glimpse(x, width = NULL, ..., .by_survey = FALSE)

glimpse(x, width = NULL, ...)

## S3 method for class 'survey_base'
glimpse(x, width = NULL, ...)
```

Arguments

`x`	A `survey_base` object.
`width`	Width of the output. Defaults to the console width.
`...`	Passed to `dplyr::glimpse()`.
`.by_survey`	If `TRUE`, render a separate labelled glimpse block per member, prefixed by the member name. Default `FALSE` renders a single bound tibble with the source survey id prepended as `coll@id` (default `.survey`).

Value

`x` invisibly.

`x` invisibly.

Survey collections

Default mode binds every member's `@data` into a single tibble (via `dplyr::bind_rows()` with `.id = coll@id`) and glimpses the result. If any member's `@data` already contains a column named `coll@id`, the surveytidy_error_collection_glimpse_id_collision error is raised BEFORE binding — symmetric with surveycore's `surveycore_error_collection_id_collision` for the construction-time case. Resolve by renaming the colliding column or setting a different `coll@id` via `surveycore::set_collection_id()`.

Internal column rename: when the member `@data` contains surveycore::SURVEYCORE_DOMAIN_COL (`..surveycore_domain..`), the column is renamed to `.in_domain` for the rendered output. Per-member `@data` is untouched.

Type-coercion footer: when `bind_rows()` coerces conflicting types across members (e.g., <chr> vs <dbl>), a footer enumerates the affected columns. Truncates after 5 columns; line width capped at 80 characters. No opt-out — the footer renders only when conflicts exist.

See Also

`select()` to control which columns are visible

Other selecting: `pull.survey_collection()`, `relocate`, `select`

Examples

```
library(surveytidy)
library(surveycore)

# create a survey design from the pew_npors_2025 example dataset
d <- as_survey(pew_npors_2025, weights = weight, strata = stratum)

# glimpse all columns
glimpse(d)

# after select(), shows only the selected columns
d |>
  select(gender, agecat, partysum) |>
  glimpse()
```

---

group_by *Group and ungroup a survey design object*

Description

`group_by()` stores grouping columns on the survey object for use in grouped operations like `mutate()`. `ungroup()` removes the grouping. `group_vars()` returns the current grouping column names.

Unlike dplyr, groups are not attached to the underlying data frame — they are stored on the survey object itself and applied when needed by verbs that support grouping.

Usage

```
## S3 method for class 'survey_base'
group_by(.data, ..., .add = FALSE, .drop = dplyr::group_by_drop_default(.data))

## S3 method for class 'survey_base'
ungroup(x, ...)

## S3 method for class 'survey_collection'
group_by(.data, ..., .add = FALSE, .drop = TRUE, .if_missing_var = NULL)

## S3 method for class 'survey_collection'
ungroup(x, ..., .if_missing_var = NULL)

group_by(.data, ..., .add = FALSE, .drop = group_by_drop_default(.data))
```

Arguments

.data	A survey_base object.
...	<data-masking> For group_by(): columns to group by. Computed expressions (e.g., cut(ridageyr, breaks = c(0, 18, 65, Inf))) are supported. For ungroup(): columns to remove from the current grouping. Omit to remove all groups.
.add	When FALSE (default), replaces existing groups. Use .add = TRUE to add to the current grouping instead.
.drop	Accepted for compatibility with the dplyr interface; has no effect on survey design objects.
x	A survey_base object (for ungroup() and group_vars()).
.if_missing_var	Per-call override of collection@if_missing_var. One of "error" or "skip", or NULL (the default) to inherit the collection's stored value. See surveycore::set_collection_if_miss

Details

Grouped operations:

After calling group_by(), mutate() computes within groups. Future estimation functions will also use grouping to perform stratified analysis.

Adding to existing groups:

By default, group_by() replaces existing groups. Use .add = TRUE to append to the current grouping instead.

Rowwise mode and group_by():

group_by(.add = FALSE) (the default) exits rowwise mode — it clears @variables$rowwise and @variables$rowwise_id_cols. group_by(.add = TRUE) when the design is rowwise promotes the rowwise id columns to @groups, appends the new groups, then clears rowwise mode — mirroring dplyr's behaviour exactly.

Partial ungroup:
`ungroup()` with no arguments removes all groups and exits rowwise mode. With column arguments, it removes only the specified columns from the grouping — rowwise mode is **not** affected.

Value

An object of the same type as the input with the following properties:

- Rows, columns, and survey design attributes are unchanged.
- For `group_by()`: grouping columns are set or updated; rowwise keys are cleared.
- For `ungroup()`: all or specified grouping columns are removed; rowwise keys are cleared on full ungroup only.
- For `group_vars()`: a character vector of current grouping column names.

Survey collections

When applied to a `survey_collection`, `group_by()` is dispatched to each member independently. Every member's `@groups` is updated, and the rebuilt collection's `@groups` is synchronised from the members. If a grouping column is missing on some members, `.if_missing_var` controls whether those members are skipped or the call errors.

Survey collections (ungroup)

`ungroup()` clears `@groups` on every member and on the collection. The dispatcher's step-5 sync lifts the cleared per-member `@groups` to the rebuilt collection. `@id` and `@if_missing_var` are preserved.

See Also

Other grouping: `is_grouped()`, `is_rowwise()`, `rowwise`

Examples

```
library(surveytidy)
library(surveycore)
# create a survey design from the pew_npors_2025 example dataset
d <- as_survey(pew_npors_2025, weights = weight, strata = stratum)

# group by a single column
group_by(d, gender)

# grouped mutate — within-group mean centring
d |>
  group_by(gender) |>
  mutate(econ_centred = econ1mod - mean(econ1mod, na.rm = TRUE))

# add a second grouping variable with .add = TRUE
d |>
  group_by(gender) |>
  group_by(cregion, .add = TRUE)
```

```
# remove all groups
d |>
  group_by(gender) |>
  ungroup()

# partial ungroup — remove only gender, keep cregion
d |>
  group_by(gender, cregion) |>
  ungroup(gender)

# get current grouping column names
d |>
  group_by(gender, cregion) |>
  group_vars()
```

`if_else` *Vectorised if-else*

Description

`if_else()` is a survey-aware version of `dplyr::if_else()` that applies a binary condition element-wise: `true` values are used where `condition` is `TRUE`, `false` values where it is `FALSE`, and `missing` values where it is `NA`.

Compared to base `ifelse()`, this function is stricter about types: `true`, `false`, and `missing` must be compatible and will be cast to their common type.

When any of `.label`, `.value_labels`, or `.description` are supplied, output label metadata is written to `@metadata` after `mutate()`. When none of these arguments are used, the output is identical to `dplyr::if_else()`.

For more than two conditions, see `case_when()`.

Usage

```
if_else(
  condition,
  true,
  false,
  missing = NULL,
  ...,
  ptype = NULL,
  .label = NULL,
  .value_labels = NULL,
  .description = NULL
)
```

Arguments

condition	A logical vector.
true, false	Vectors to use for TRUE and FALSE values of condition. Both are recycled to the size of condition and cast to their common type.
missing	If not NULL, used as the value for NA values of condition. Follows the same size and type rules as true and false.
...	These dots are for future extensions and must be empty.
ptype	An optional prototype declaring the desired output type. Overrides the common type of true, false, and missing.
.label	character(1) or NULL. Variable label stored in @metadata@variable_labels after mutate().
.value_labels	Named vector or NULL. Value labels stored in @metadata@value_labels. Names are the label strings; values are the data values.
.description	character(1) or NULL. Plain-language description of how the variable was created. Stored in @metadata@transformations[[col]]$description after mutate().

Value

A vector the same size as condition and the common type of true, false, and missing. If .label or .value_labels are supplied, returns a haven_labelled vector; otherwise returns the same type as the common type of the inputs.

See Also

- dplyr::if_else() for the base implementation.
- case_when() for more than two conditions.
- na_if() to replace specific values with NA.

Other recoding: case_when(), na_if(), recode_values(), replace_values(), replace_when()

Examples

```
library(surveycore)
library(surveytidy)

# create the survey design
ns_wave1_svy <- as_survey_nonprob(ns_wave1, weights = weight)

# basic if_else — identical to dplyr::if_else()
new <- ns_wave1_svy |>
  mutate(senior = if_else(age >= 65, "Senior (65+)", "Non-senior")) |>
  select(age, senior)

# by default, no metadata is attached
new
new@metadata
```

```
# use missing = to specify the output value when condition is NA
new <- ns_wave1_svy |>
  mutate(
    dem = if_else(
      pid3 == 1,
      "Democrat",
      "Non-Democrat",
      missing = "Unknown"
    )
  ) |>
  select(pid3, dem)

new

# attach a variable label via .label
new <- ns_wave1_svy |>
  mutate(
    senior = if_else(
      age >= 65,
      "Senior (65+)",
      "Non-senior",
      .label = "Senior citizen (age 65+)"
    )
  ) |>
  select(age, senior)

new@metadata@variable_labels

# use integer codes and document them with value labels
new <- ns_wave1_svy |>
  mutate(
    senior = if_else(
      age >= 65,
      true = 1L,
      false = 0L,
      .label = "Senior citizen (age 65+)",
      .value_labels = c("Senior (65+)" = 1, "Non-senior" = 0)
    )
  ) |>
  select(age, senior)

new@metadata@value_labels

# attach a plain-language description of the transformation
new <- ns_wave1_svy |>
  mutate(
    senior = if_else(
      age >= 65,
      "Senior (65+)",
      "Non-senior",
      .label = "Senior citizen (age 65+)",
      .description = paste(
        "age >= 65 coded as 'Senior (65+)';",
```

```
        "everyone else as 'Non-senior'."
      )
    )
  ) |>
  select(age, senior)

new@metadata@transformations
```

`inner_join` *Domain-aware inner join for survey designs*

Description

`inner_join()` has two modes controlled by `.domain_aware` (default `TRUE`):

Domain-aware mode (`.domain_aware = TRUE`, default): Unmatched rows are marked `FALSE` in the domain column (exactly like `filter()` or `semi_join()`), and y's columns are added to all rows (with `NA` for unmatched rows). All rows remain in `@data`. Row count is unchanged. This is the survey-correct default.

Physical mode (`.domain_aware = FALSE`): Unmatched rows are physically removed, exactly like base R `inner_join`. Emits `surveycore_warning_physical_subset`. Errors for `survey_twophase` designs.

Usage

```
## S3 method for class 'survey_collection'
inner_join(x, y, ..., .if_missing_var = NULL)

inner_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL
)
```

Arguments

`x`	A `survey_base` object.
`y`	A plain data frame. Must not be a survey object.
`...`	Additional arguments forwarded to the underlying dplyr function.
`.if_missing_var`	Per-call override of `collection@if_missing_var`. One of `"error"` or `"skip"`, or `NULL` (the default) to inherit the collection's stored value. See `surveycore::set_collection_if_miss`

by	A character vector of column names or a dplyr::join_by() specification. NULL uses all common column names.
copy	Forwarded to the underlying dplyr function.
suffix	A character vector of length 2. Forwarded to the underlying dplyr function for handling shared column names.
keep	Forwarded to the underlying dplyr function.

Details

Choosing a mode:

The domain-aware default preserves variance estimation validity. The nrow() behaviour (count stays the same) is consistent with filter() and semi_join() precedents in surveytidy.

Physical mode (.domain_aware = FALSE) is appropriate only when you explicitly want to reduce the design to a specific subpopulation. For replicate designs (BRR, jackknife), physical row removal can corrupt half-sample or pairing structure, producing numerically wrong variance estimates. Domain-aware mode is recommended for replicate designs.

Duplicate keys:

Duplicate keys in y that would expand the row count are an error in both modes. Deduplicate y with dplyr::distinct() before joining.

The .domain_aware argument (survey-specific extension):

The surveytidy method adds one argument not present in the dplyr generic: .domain_aware = TRUE (default) performs domain-aware joining; set .domain_aware = FALSE for physical row removal (emits surveycore_warning_physical_subset; errors for survey_twophase).

Value

A survey design object of the same type as x.

- Domain-aware mode (.domain_aware = TRUE): row count unchanged; ..surveycore_domain.. updated; new columns from y appended.
- Physical mode (.domain_aware = FALSE): row count reduced to matched rows; new columns from y appended.

Survey collections

When called on a surveycore::survey_collection, inner_join() errors unconditionally with class surveytidy_error_collection_verb_unsupported. The semantics for joining a plain data frame onto a multi-survey container are still being designed. Apply the join inside a per-survey pipeline before constructing the collection.

See Also

Other joins: bind_cols(), bind_rows(), left_join, right_join, semi_join

Examples

```
# create a small survey object
df <- data.frame(
  psu = paste0("psu_", 1:5),
  strata = "s1",
  fpc = 100,
  wt = 1,
  y1 = 1:5
)
d <- surveycore::as_survey(
  df,
  ids = psu,
  weights = wt,
  strata = strata,
  fpc = fpc,
  nest = TRUE
)
lookup <- data.frame(y1 = 1:3, label = letters[1:3])

# domain-aware: marks rows 4 and 5 as out-of-domain
inner_join(d, lookup, by = "y1")

# physical: removes rows 4 and 5
inner_join(d, lookup, by = "y1", .domain_aware = FALSE)
```

---

is_grouped *Test whether a survey design has active grouping*

Description

Returns TRUE if the design has one or more grouping columns set via group_by(). Returns FALSE for ungrouped or rowwise (but not grouped) designs.

Usage

```
is_grouped(design)
```

Arguments

design	A survey_base object.

Value

A scalar logical.

See Also

Other grouping: group_by, is_rowwise(), rowwise

Examples

```
# create a survey object from the bundled NPORS dataset
d <- surveycore::as_survey(
  surveycore::pew_npors_2025,
  weights = weight,
  strata = stratum
)

# only group_by() makes is_grouped() TRUE; rowwise() does not count
is_grouped(d)
is_grouped(group_by(d, gender))
is_grouped(rowwise(d))
```

`is_rowwise` *Test whether a survey design is in rowwise mode*

Description

Returns `TRUE` if the design was created (or passed through) `rowwise()`. Use this predicate in estimation functions to detect and handle (or disallow) rowwise mode.

Usage

```
is_rowwise(design)
```

Arguments

`design`	A `survey_base` object.

Value

A scalar logical.

See Also

Other grouping: `group_by`, `is_grouped()`, `rowwise`

Examples

```
# create a survey object from the bundled NPORS dataset
d <- surveycore::as_survey(
  surveycore::pew_npors_2025,
  weights = weight,
  strata = stratum
)

# FALSE for a freshly-built design; TRUE after rowwise()
is_rowwise(d)
```

```
is_rowwise(rowwise(d))
```

`left_join` *Add columns from a data frame to a survey design*

Description

`left_join()` adds columns from a plain data frame `y` to a survey design object `x`, matching on keys defined by `by`. All rows of `x` are preserved (left join semantics). Rows with no match in `y` receive `NA` for the new columns.

Usage

```
## S3 method for class 'survey_collection'
left_join(x, y, ..., .if_missing_var = NULL)

left_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL
)
```

Arguments

`x`	A `survey_base` object.
`y`	A plain data frame with lookup columns. Must not be a survey object. Must not have column names matching any design variable in `x` (those are dropped with a warning).
`...`	Additional arguments forwarded to `dplyr::left_join()`.
`.if_missing_var`	Per-call override of `collection@if_missing_var`. One of `"error"` or `"skip"`, or `NULL` (the default) to inherit the collection's stored value. See `surveycore::set_collection_if_miss`
`by`	A character vector of column names or a `dplyr::join_by()` specification. `NULL` uses all common column names.
`copy`	Forwarded to `dplyr::left_join()`.
`suffix`	A character vector of length 2 appended to deduplicate column names shared between `x` and `y`. Forwarded to `dplyr::left_join()`.
`keep`	Forwarded to `dplyr::left_join()`.

Details

Design integrity:

y must be a plain data frame, not a survey object. If y has column names that match any design variable in x (weights, strata, PSU, FPC, replicate weights, or the domain column), those columns are dropped from y with a warning before joining. Join keys in by are excluded from this check.

Row count:

`left_join()` errors if y has duplicate keys that would expand x beyond its original row count. Duplicate respondent rows corrupt variance estimation. Deduplicate y with `dplyr::distinct()` before joining.

Metadata:

New columns from y receive no variable labels in `@metadata`. If a column in `x@data` is suffix-renamed because y has a non-design column with the same name, the corresponding `@metadata@variable_labels` key is updated to the new suffixed name.

Value

A survey design object of the same type as x with new columns from y appended to `@data`. `visible_vars` is updated if it was set.

Survey collections

When called on a `surveycore::survey_collection`, `left_join()` errors unconditionally with class `surveytidy_error_collection_verb_unsupported`. The semantics for joining a plain data frame onto a multi-survey container are still being designed. Apply the join inside a per-survey pipeline before constructing the collection.

See Also

Other joins: `bind_cols()`, `bind_rows()`, `inner_join`, `right_join`, `semi_join`

Examples

```
# create a small survey object
df <- data.frame(
  psu = paste0("psu_", 1:5),
  strata = "s1",
  fpc = 100,
  wt = 1,
  y1 = 1:5
)
d <- surveycore::as_survey(
  df,
  ids = psu,
  weights = wt,
  strata = strata,
  fpc = fpc,
  nest = TRUE
)
```

```
# add a lookup column from a plain data frame
lookup <- data.frame(y1 = 1:5, label = letters[1:5])
left_join(d, lookup, by = "y1")
```

`make_binary` *Convert a dichotomous variable to a numeric 0/1 indicator*

Description

`make_binary()` converts a variable that can be collapsed to exactly two levels (via `make_dicho()`) into an integer vector of 0s and 1s. By default, the first level maps to `1L` and the second to `0L`. Use `flip_values = TRUE` to reverse the mapping.

When called inside `mutate()`, metadata is recorded in `@metadata@transformations[[col]]`.

Usage

```
make_binary(
  x,
  flip_values = FALSE,
  .exclude = NULL,
  .label = NULL,
  .description = NULL
)
```

Arguments

`x`	Vector. Same types as `make_factor()`. Must yield exactly 2 levels (after `.exclude`) or error.
`flip_values`	`logical(1)`. If `TRUE`, map the first level to `0L` and the second to `1L`. Default maps first level to `1L`.
`.exclude`	`character` or `NULL`. Passed directly to `make_dicho()`. Level names to set to `NA` before encoding.
`.label`	`character(1)` or `NULL`. Variable label override. Falls back to `attr(x, "label")` then the column name.
`.description`	`character(1)` or `NULL`. Transformation description.

Value

An integer vector with values `0L`, `1L`, or `NA_integer_`.

See Also

Other transformation: `make_dicho()`, `make_factor()`, `make_flip()`, `make_rev()`, `row_means()`, `row_sums()`

Examples

```
# build a 2-level factor with one NA
x <- factor(
  c("Agree", "Disagree", "Agree", NA),
  levels = c("Agree", "Disagree")
)

# encode as a 0/1 integer indicator
make_binary(x)
```

`make_dicho` *Collapse a multi-level factor to two levels*

Description

`make_dicho()` converts a variable to a two-level factor by stripping the first qualifier word from each level label and grouping the resulting stems. For example, a 4-level Likert scale with labels `c("Strongly agree", "Agree", "Disagree", "Strongly disagree")` collapses to `c("Agree", "Disagree")` by removing the qualifier "Strongly".

When called inside `mutate()`, metadata is recorded in `@metadata@transformations[[col]]`.

Usage

```
make_dicho(
  x,
  flip_levels = FALSE,
  .exclude = NULL,
  .label = NULL,
  .description = NULL
)
```

Arguments

`x`	Vector. Same types as `make_factor()`.
`flip_levels`	`logical(1)`. If `TRUE`, reverse the order of the two output levels.
`.exclude`	`character` or `NULL`. Level name(s) to set to `NA` before collapsing. Intended for middle categories and "don't know"/"refused".
`.label`	`character(1)` or `NULL`. Variable label override. Falls back to `attr(x, "label")` then the column name.
`.description`	`character(1)` or `NULL`. Transformation description.

Value

A 2-level R factor.

See Also

Other transformation: `make_binary()`, `make_factor()`, `make_flip()`, `make_rev()`, `row_means()`, `row_sums()`

Examples

```
# build a 4-level Likert factor
x <- factor(
  c(
    "Always agree",
    "Sometimes agree",
    "Sometimes disagree",
    "Always disagree"
  ),
  levels = c(
    "Always agree",
    "Sometimes agree",
    "Sometimes disagree",
    "Always disagree"
  )
)

# collapse to 2 levels by stripping the qualifier word
make_dicho(x)
```

`make_factor` *Convert a vector to a factor using value labels*

Description

`make_factor()` converts a labelled numeric, factor, or character vector to an R factor. For labelled numeric input (e.g., from **haven** or with a `"labels"` attribute), factor levels are derived from the value labels. For factor input, levels are preserved. For character input, levels are set alphabetically.

When called inside `mutate()`, metadata is recorded in `@metadata@transformations[[col]]`.

Usage

```
make_factor(
  x,
  ordered = FALSE,
  drop_levels = TRUE,
  force = FALSE,
  na.rm = FALSE,
  .label = NULL,
  .description = NULL
)
```

Arguments

x	Vector to convert. Must be a labelled numeric, plain numeric with a "labels" attribute, R factor, or character vector.
ordered	logical(1). If TRUE, returns an ordered factor.
drop_levels	logical(1). If TRUE (the default), removes levels with no observed values in x.
force	logical(1). If TRUE, coerce a numeric x without value labels via as.factor(), issuing a surveytidy_warning_make_factor_forced warning. If FALSE (the default), error instead.
na.rm	logical(1). If TRUE, values in attr(x, "na_values") and attr(x, "na_range") are converted to NA before building factor levels, so they do not produce factor levels. Ignored for factor and character input.
.label	character(1) or NULL. Variable label override. If NULL, inherits from attr(x, "label"); if that is also NULL, falls back to the column name.
.description	character(1) or NULL. Transformation description stored in surveytidy_recode.

Value

An R factor (ordered if ordered = TRUE).

See Also

Other transformation: make_binary(), make_dicho(), make_flip(), make_rev(), row_means(), row_sums()

Examples

```
# attach value labels to a numeric vector and convert to a factor
x <- c(1, 2, 1, 2)
attr(x, "labels") <- c("Yes" = 1, "No" = 2)
make_factor(x)
```

---

make_flip *Flip the semantic valence of a variable*

Description

make_flip() reverses the label string associations of a numeric variable without changing its values. This is used to flip the polarity of a survey item for composite scoring - for example, converting "I like the color blue" to "I dislike the color blue" without changing the underlying numeric codes.

Unlike make_rev(), which changes numeric values and keeps label strings in place, make_flip() keeps values unchanged and reverses which label strings are attached to which values.

A new variable label is **required** because flipping always changes the semantic meaning of the variable.

When called inside mutate(), metadata is recorded in @metadata@transformations[[col]].

Usage

```
make_flip(x, label, .description = NULL)
```

Arguments

x	A numeric vector. typeof(x) must be "double" or "integer".
label	character(1). **Required.** New variable label describing the flipped semantic meaning.
.description	character(1) or NULL. Transformation description.

Value

A numeric vector (same typeof() as x). Values are unchanged.

See Also

Other transformation: make_binary(), make_dicho(), make_factor(), make_rev(), row_means(), row_sums()

Examples

```
# build a labelled numeric vector with a 4-level Likert scale
x <- c(1, 2, 3, 4)
attr(x, "labels") <- c(
  "Strongly agree" = 1,
  "Agree" = 2,
  "Disagree" = 3,
  "Strongly disagree" = 4
)

# flip the semantic meaning while keeping numeric values unchanged
make_flip(x, "I dislike the color blue")
```

---

make_rev *Reverse the numeric values of a scale variable*

Description

make_rev() reverses the direction of a numeric scale variable using the formula min(x) + max(x) - x. This preserves the scale range: a 1-4 scale reversed stays a 1-4 scale; a 2-5 scale reversed stays a 2-5 scale.

Value labels are remapped: each label's numeric value becomes min + max - old_value, so the label string stays tied to its original concept at its new position.

When called inside mutate(), metadata is recorded in @metadata@transformations[[col]].

Usage

```
make_rev(x, .label = NULL, .description = NULL)
```

Arguments

x	A numeric vector. typeof(x) must be "double" or "integer".
.label	character(1) or NULL. Variable label override. If NULL, inherits from attr(x, "label"); if that is also NULL, falls back to the column name.
.description	character(1) or NULL. Transformation description.

Value

A numeric vector (same typeof() as x) with reversed values.

See Also

Other transformation: make_binary(), make_dicho(), make_factor(), make_flip(), row_means(), row_sums()

Examples

```
# reverse a 1-4 numeric scale: 1 swaps with 4, 2 swaps with 3
x <- c(1, 2, 3, 4)
make_rev(x)
```

---

mutate *Create, modify, and delete columns of a survey design object*

Description

mutate() adds new columns or modifies existing ones while preserving the survey design structure required for valid variance estimation. It delegates column computation to dplyr::mutate() on the underlying data.

Use NULL as a value to delete a column. Design variables (weights, strata, PSUs) cannot be deleted this way — they are always preserved.

Usage

```
## S3 method for class 'survey_base'
mutate(
  .data,
  ...,
  .by = NULL,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL
```

```
)

## S3 method for class 'survey_result'
mutate(.data, ...)

## S3 method for class 'survey_collection'
mutate(
  .data,
  ...,
  .by = NULL,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL,
  .if_missing_var = NULL
)

mutate(.data, ...)
```

Arguments

`.data`	A `survey_base` object, or a `survey_result` object returned by a surveycore estimation function.
`...`	<`data-masking`> Name-value pairs. The name gives the output column name; the value is an expression evaluated against the survey data. Use `NULL` to delete a non-design column.
`.by`	Not used directly. Set grouping with `group_by()` instead. When `@groups` is non-empty and `.by` is `NULL` (the default), the active groups are applied automatically.
`.keep`	Which columns to retain. One of `"all"` (default), `"used"`, `"unused"`, or `"none"`. Design variables are always re-attached regardless of this argument.
`.before, .after`	<`tidy-select`> Optionally position new columns before or after an existing one.
`.if_missing_var`	Per-call override of `collection@if_missing_var`. One of `"error"` or `"skip"`, or `NULL` (the default) to inherit the collection's stored value. See `surveycore::set_collection_if_miss`

Details

Design variable modification:

If the left-hand side of a mutation names a design variable (e.g., `mutate(d, wt = wt * 2)`), a `surveytidy_warning_mutate_design_var` warning is issued. Detection is name-based: `across()` calls that happen to modify design variables will **not** trigger the warning.

`.keep` **and design variables:**

Design variables (weights, strata, PSUs, FPC, replicate weights, and the domain column) are always preserved in the output, regardless of `.keep`. This ensures variance estimation remains valid even when `.keep = "none"`.

Grouped mutate:

Grouping set by `group_by()` is respected automatically — leave `.by = NULL` (the default) and mutate expressions will compute within groups. The `.by` argument is not used directly.

Useful mutate functions:

- Arithmetic: `+`, `-`, `*`, `/`, `^`, `%%`, `%/%`
- Rounding: `round()`, `floor()`, `ceiling()`, `trunc()`
- Ranking: `dplyr::dense_rank()`, `dplyr::min_rank()`, `dplyr::row_number()`
- Cumulative: `cumsum()`, `cummax()`, `cummin()`, `dplyr::cummean()`
- Conditional: `dplyr::if_else()`, `dplyr::case_when()`, `dplyr::case_match()`
- Missing values: `dplyr::na_if()`, `dplyr::coalesce()`

Value

An object of the same type as `.data` with the following properties:

- Rows are not added or removed.
- Columns are retained, modified, or removed per `...` and `.keep`.
- Design variables (weights, strata, PSUs) are always present.
- Groups and survey design attributes are preserved.

Survey collections

When applied to a `survey_collection`, `mutate()` is dispatched to each member independently. Per-member warnings (e.g., `surveytidy_warning_mutate_weight_col` when modifying the weight column) fire once per member in which they apply — an N-member collection that all modify the weight column will surface N warnings.

If members have non-uniform rowwise state (some are rowwise, some are not), `mutate()` emits `surveytidy_warning_collection_rowwise_mixed` once before dispatch as a soft-invariant diagnostic. Dispatch still proceeds; per-member rowwise/non-rowwise semantics apply for the call. To resolve, call `rowwise()` or `ungroup()` on the entire collection first.

`.by` is rejected at the collection layer with `surveytidy_error_collection_by_unsupported`. Set grouping with `group_by()` on the collection instead.

See Also

`rename()` to rename columns, `select()` to drop columns

Other modification: `rename`

Examples

```
library(surveytidy)
library(surveycore)
# create a survey design from the pew_npors_2025 example dataset
d <- as_survey(pew_npors_2025, weights = weight, strata = stratum)

# add a new column
```

```
mutate(d, college_grad = educcat == 1)

# conditional recoding
mutate(
  d,
  college = dplyr::if_else(educcat == 1, "college+", "non-college")
)

# grouped mutate — within-group mean centring
d |>
  group_by(gender) |>
  mutate(econ_centred = econ1mod - mean(econ1mod, na.rm = TRUE))

# .keep = "none" keeps only new columns plus design vars (always preserved)
mutate(
  d,
  college = dplyr::if_else(educcat == 1, "college+", "non-college"),
  .keep = "none"
)
```

`na_if` *Convert values to* `NA`

Description

`na_if()` is a survey-aware version of `dplyr::na_if()` that converts values equal to `y` to `NA`. It is useful for replacing sentinel values (e.g., `999` for "don't know") with proper missing values.

Unlike `dplyr::na_if()`, which accepts only a scalar `y`, this version accepts a vector `y` and replaces all matching values in a single call.

When `x` carries value labels, `na_if()` automatically inherits those labels. By default (`.update_labels = TRUE`), the label entries for the NA'd values are removed from the output; set `.update_labels = FALSE` to retain them (useful when you want to document what was set to missing).

Usage

```
na_if(x, y, .update_labels = TRUE, .description = NULL)
```

Arguments

`x`	Vector to modify.
`y`	Value or vector of values to replace with `NA`. `y` is cast to the type of `x` before comparison. When `y` has more than one element, each value is replaced sequentially.
`.update_labels`	`logical(1)`. If `TRUE` (the default) and `x` carries value labels, label entries for values in `y` are removed from the output's value labels. Set to `FALSE` to retain all inherited labels even for values that were set to `NA`.

`.description` character(1) or NULL. Plain-language description of how the variable was created. Stored in `@metadata@transformations[[col]]$description` after `mutate()`.

Value

A modified version of `x` where values equal to `y` are replaced with `NA`. If `x` carries value labels, returns a `haven_labelled` vector with updated (or retained) labels; otherwise returns the same type as `x`.

See Also

- `dplyr::na_if()` for the base implementation.
- `dplyr::coalesce()` to replace NAs with the first non-missing value.
- `replace_values()` for replacing specific values with a new value rather than `NA`.
- `replace_when()` for condition-based in-place replacement.

Other recoding: `case_when()`, `if_else()`, `recode_values()`, `replace_values()`, `replace_when()`

Examples

```
library(surveycore)
library(surveytidy)

# create the survey design
ns_wave1_svy <- as_survey_nonprob(ns_wave1, weights = weight)

# basic na_if — replace "Something else" (pid3 == 4) with NA
new <- ns_wave1_svy |>
  mutate(pid3_clean = na_if(pid3, 4)) |>
  select(pid3, pid3_clean)

new

# replace multiple values at once — Independent (3) and "Something else" (4)
new <- ns_wave1_svy |>
  mutate(pid3_2party = na_if(pid3, c(3, 4))) |>
  select(pid3, pid3_2party)

new

# .update_labels = TRUE (default) drops label entries for NA'd values
new <- ns_wave1_svy |>
  mutate(pid3_clean = na_if(pid3, 4, .update_labels = TRUE)) |>
  select(pid3, pid3_clean)

# "Something else" (4) is removed from pid3_clean's value labels
new@metadata@value_labels$pid3_clean

# .update_labels = FALSE retains label entries even for NA'd values
new <- ns_wave1_svy |>
```

```
  mutate(pid3_clean = na_if(pid3, 4, .update_labels = FALSE)) |>
  select(pid3, pid3_clean)

# "Something else" (4) is still in pid3_clean's value labels
new@metadata@value_labels$pid3_clean

# attach a plain-language description of the transformation
new <- ns_wave1_svy |>
  mutate(
    pid3_clean = na_if(
      pid3,
      4,
      .description = "Set 'Something else' (pid3 == 4) to NA."
    )
  ) |>
  select(pid3, pid3_clean)

new@metadata@transformations
```

`pull.survey_collection` *Extract a column from a survey design object*

Description

Pull a single column out of a survey design object as a plain vector. This is a terminal operation — the result is not a survey object and cannot be piped back into survey verbs.

Usage

```
## S3 method for class 'survey_collection'
pull(.data, var = -1, name = NULL, ..., .if_missing_var = NULL)

pull(.data, var = -1, name = NULL, ...)

## S3 method for class 'survey_base'
pull(.data, var = -1, name = NULL, ...)
```

Arguments

`.data`	A `survey_base` object.
`var`	<`data-masking`> The column to extract. Accepts a bare name, a positive integer (counting from the left), or a negative integer (counting from the right). Defaults to the last column.
`name`	<`data-masking`> An optional column whose values are used as names for the returned vector.
`...`	Passed to `dplyr::pull()`.

`.if_missing_var`
: Per-call override of `collection@if_missing_var`. One of `"error"` or `"skip"`, or `NULL` (the default) to inherit the collection's stored value. See `surveycore::set_collection_if_miss`

Value

A vector the same length as the number of rows in `.data`.

Survey collections

When applied to a `survey_collection`, `pull()` extracts the column from each member and combines the per-member vectors via `vctrs::vec_c()`. Detection of missing columns uses class-catch only — both `var` and `name` flow through a single `tryCatch` handler that re-raises `vctrs_error_subscript_oob` / `rlang_error_data_pronoun_not_found` as `surveytidy_error_collection_verb_failed`.

Naming options:

- `name = NULL` (default) — unnamed combined vector.
- `name = coll@id` — by-survey naming sentinel: each combined element is named by its source survey. The sentinel string is whatever `coll@id` resolves to (default `.survey`; user-set values like `"wave"` work identically).
- `name = "<other_column>"` — passes through to `dplyr::pull`'s `name` arg unchanged (per-row names from another column inside each member), then combined across surveys via the same `vctrs::vec_c()` path as the values.

If `vctrs::vec_c()` raises `vctrs_error_incompatible_type` (e.g., one member has the column as numeric and another as character), the error is re-raised as `surveytidy_error_collection_pull_incompatible_types` with `parent = cnd` and the column name and conflicting surveys. No auto-coercion — `pull` returns a single vector and silent coercion would mask the kind of data-type bug users almost certainly want surfaced. (`glimpse.survey_collection` auto-coerces with a footer; the divergence is intentional — `glimpse` is diagnostic, `pull` is computational.)

Domain inclusion

Inherits the contract of `pull()` for `survey_base`: the returned vector includes both in-domain and out-of-domain values. `pull.survey_base` calls `dplyr::pull(@data, ...)` directly without filtering on the domain column, so the combined vector mixes both kinds of rows. The user has no per-element marker for domain membership — this is a known limitation of `pull` at the per-survey verb level (not the collection layer). Use a per-member `filter()` or `tibble::tibble()` before pulling if domain filtering is required.

See Also

`select()` to keep columns in the survey object

Other selecting: `glimpse.survey_collection()`, `relocate`, `select`

Examples

```
library(surveytidy)
library(surveycore)

# create a survey design from the pew_npors_2025 example dataset
d <- as_survey(pew_npors_2025, weights = weight, strata = stratum)

# extract a column by name
pull(d, agecat)

# named vector — values of agecat named by respid
pull(d, agecat, name = respid)
```

`recode_values` *Recode values using an explicit mapping*

Description

`recode_values()` replaces each value of `x` with a corresponding new value. The mapping can be supplied in any of three ways:

- **Formula interface** — pass `old_value ~ new_value` formulas in `...`: `recode_values(score, 1 ~ "SD", 2 ~ "D", 3 ~ "N", 4 ~ "A", 5 ~ "SA")`.
- **Lookup-table interface** — pass parallel `from` and `to` vectors.
- **Label-driven interface** — set `.use_labels = TRUE` to build the map from `attr(x, "labels")` (values become `from`, label strings become `to`).

Values not found in the map are either kept unchanged (`.unmatched = "default"`, the default) or trigger an error (`.unmatched = "error"`).

Unlike `replace_values()`, which updates only specific matching values and retains everything else, `recode_values()` is intended for full remapping: every possible value in `x` typically has a corresponding entry in the map.

When any of `.label`, `.value_labels`, `.factor`, or `.description` are supplied, output label metadata is written to `@metadata` after `mutate()`. When none of these arguments are used, the output is identical to `dplyr::recode_values()`.

Usage

```
recode_values(
  x,
  ...,
  from = NULL,
  to = NULL,
  default = NULL,
  .unmatched = "default",
  ptype = NULL,
```

```
  .label = NULL,
  .value_labels = NULL,
  .factor = FALSE,
  .use_labels = FALSE,
  .description = NULL
)
```

Arguments

x	Vector to recode.
...	old_value ~ new_value formulas describing the recoding map. Equivalent to supplying parallel from/to vectors. When ... is non-empty, from and .use_labels = TRUE must not be used.
from	Vector (or list of vectors, for many-to-one mapping) of old values. Required unless formulas are supplied in ... or .use_labels = TRUE. Must be the same type as x.
to	Vector of new values corresponding to from. Must be the same length as from.
default	Value for entries in x not found in from. NULL (the default) keeps unmatched values unchanged. Ignored when .unmatched = "error".
.unmatched	"default" (the default) or "error". When "error", any value in x not present in from triggers a surveytidy_error_recode_unmatched_values error.
ptype	An optional prototype declaring the desired output type.
.label	character(1) or NULL. Variable label stored in @metadata@variable_labels after mutate(). Cannot be combined with .factor = TRUE.
.value_labels	Named vector or NULL. Value labels stored in @metadata@value_labels. Names are the label strings; values are the data values.
.factor	logical(1). If TRUE, returns a factor. Levels are taken from .value_labels names if supplied, otherwise from to in lookup mode or from the right-hand sides of the ... formulas in formula mode. Cannot be combined with .label.
.use_labels	logical(1). If TRUE, reads attr(x, "labels") to build the from/to map automatically: values become from, label strings become to. x must carry value labels; errors if not. Cannot be combined with formulas in ....
.description	character(1) or NULL. Plain-language description of how the variable was created. Stored in @metadata@transformations[[col]]$description after mutate().

Value

A vector, factor, or haven_labelled vector:

- No surveytidy args — same output as dplyr::recode_values().
- .factor = TRUE — a factor with levels in to order.
- .label or .value_labels supplied — a haven_labelled vector.

See Also

- `dplyr::recode_values()` for the base implementation.
- `replace_values()` for partial replacement (updates only matching values, retains existing value labels from x).
- `case_when()` for condition-based remapping.

Other recoding: `case_when()`, `if_else()`, `na_if()`, `replace_values()`, `replace_when()`

Examples

```
library(surveycore)
library(surveytidy)

# create the survey design
ns_wave1_svy <- as_survey_nonprob(ns_wave1, weights = weight)

# formula interface — recode pid3 using `old ~ new` formulas in `...`
new <- ns_wave1_svy |>
  mutate(
    party = recode_values(
      pid3,
      1 ~ "Democrat",
      2 ~ "Republican",
      3 ~ "Independent",
      4 ~ "Other"
    )
  ) |>
  select(pid3, party)

new

# formula interface with default for unmatched values
new <- ns_wave1_svy |>
  mutate(
    dem = recode_values(pid3, 1 ~ "Democrat", default = "Non-Democrat")
  ) |>
  select(pid3, dem)

new

# explicit from/to mapping — recode numeric codes to character labels
new <- ns_wave1_svy |>
  mutate(
    party = recode_values(
      pid3,
      from = c(1, 2, 3, 4),
      to = c("Democrat", "Republican", "Independent", "Other")
    )
  ) |>
  select(pid3, party)

new
```

```
# use default to catch unmatched values
new <- ns_wave1_svy |>
  mutate(
    dem = recode_values(
      pid3,
      from = c(1),
      to = c("Democrat"),
      default = "Non-Democrat"
    )
  ) |>
  select(pid3, dem)

new

# .use_labels = TRUE builds the from/to map from existing value labels
new <- ns_wave1_svy |>
  mutate(party = recode_values(pid3, .use_labels = TRUE)) |>
  select(pid3, party)

new

# attach a variable label via .label
new <- ns_wave1_svy |>
  mutate(
    party = recode_values(
      pid3,
      from = c(1, 2, 3, 4),
      to = c("Democrat", "Republican", "Independent", "Other"),
      .label = "Party identification"
    )
  ) |>
  select(pid3, party)

new@metadata@variable_labels

# collapse 4 categories to 3 and document via .value_labels
new <- ns_wave1_svy |>
  mutate(
    party = recode_values(
      pid3,
      from = c(1, 2, 3, 4),
      to = c(1, 2, 3, 3),
      .label = "Party ID (3 categories)",
      .value_labels = c(
        "Democrat" = 1,
        "Republican" = 2,
        "Independent/Other" = 3
      )
    )
  ) |>
  select(pid3, party)
```

```
new@metadata@value_labels

# return a factor with levels in `to` order
new <- ns_wave1_svy |>
  mutate(
    party = recode_values(
      pid3,
      from = c(1, 2, 3, 4),
      to = c("Democrat", "Republican", "Independent", "Other"),
      .factor = TRUE
    )
  ) |>
  select(pid3, party)

new

# attach a plain-language description of the transformation
new <- ns_wave1_svy |>
  mutate(
    party = recode_values(
      pid3,
      from = c(1, 2, 3, 4),
      to = c("Democrat", "Republican", "Independent", "Other"),
      .label = "Party identification",
      .description = paste(
        "pid3 recoded: 1->Democrat, 2->Republican,",
        "3->Independent, 4->Other."
      )
    )
  ) |>
  select(pid3, party)

new@metadata@transformations
```

`relocate` *Change column order in a survey design object*

Description

`relocate()` moves columns to a new position using the same tidyselect mini-language as `select()`. Design variables (weights, strata, PSUs) are not moved — only analysis columns change position.

Usage

```
relocate(.data, ..., .before = NULL, .after = NULL)

## S3 method for class 'survey_base'
relocate(.data, ..., .before = NULL, .after = NULL)
```

```
## S3 method for class 'survey_collection'
relocate(.data, ..., .before = NULL, .after = NULL, .if_missing_var = NULL)
```

Arguments

.data	A survey_base object.
...	<tidy-select> Columns to move.
.before, .after	<tidy-select> A destination column. Columns in ... are placed immediately before or after it. Specify at most one of .before and .after.
.if_missing_var	Per-call override of collection@if_missing_var. One of "error" or "skip", or NULL (the default) to inherit the collection's stored value. See surveycore::set_collection_if_miss

Details

Design variable positions:

Design variables are always preserved at their current position in the underlying data. When you call relocate(), only non-design columns are affected by the reordering.

After select():

When select() has been called, relocate() reorders the visible columns (those shown when you print the object). This has no effect on the physical column order in the underlying data.

Value

An object of the same type as .data with the following properties:

- Rows are not modified.
- All columns are present; only their order changes.
- Design variables are not moved.
- Groups and survey design attributes are preserved.

Survey collections

When applied to a survey_collection, relocate() is dispatched to each member independently. Each member's relocate.survey_base reorders columns according to the user's tidyselect (and .before/.after), preserving design variables and @groups. Negative tidyselect like relocate(coll, -group, .before = wt) is permitted because relocate only reorders — it never removes columns. The select group-removal pre-flight does not apply.

See Also

select() to keep or drop columns, rename() to rename them

Other selecting: glimpse.survey_collection(), pull.survey_collection(), select

Examples

```
library(surveytidy)
library(surveycore)

# create a survey design from the pew_npors_2025 example dataset
d <- as_survey(pew_npors_2025, weights = weight, strata = stratum)

# move agecat before gender
relocate(d, agecat, .before = gender)

# move all social media columns to the front
relocate(d, tidyselect::starts_with("smuse_"))

# after select(), relocate reorders the visible columns
d |>
  select(gender, agecat, partysum) |>
  relocate(partysum, .before = gender)
```

---

rename *Rename columns of a survey design object*

Description

`rename()` and `rename_with()` change column names in the underlying data and automatically keep the survey design in sync. Variable labels, value labels, and other metadata follow the rename — no manual bookkeeping required.

Use `rename()` for `new_name = old_name` pairs; use `rename_with()` to apply a function across a selection of column names.

Renaming a design variable (weights, strata, PSUs) is fully supported: the design specification updates automatically and a `surveytidy_warning_rename_design_var` warning is issued to confirm the change.

Usage

```
rename(.data, ...)

## S3 method for class 'survey_base'
rename(.data, ...)

## S3 method for class 'survey_result'
rename(.data, ...)

## S3 method for class 'survey_base'
rename_with(.data, .fn, .cols = dplyr::everything(), ...)

## S3 method for class 'survey_result'
```

```
rename_with(.data, .fn, .cols = dplyr::everything(), ...)

## S3 method for class 'survey_collection'
rename(.data, ..., .if_missing_var = NULL)

## S3 method for class 'survey_collection'
rename_with(
  .data,
  .fn,
  .cols = dplyr::everything(),
  ...,
  .if_missing_var = NULL
)
```

Arguments

`.data`	A `survey_base` object, or a `survey_result` object returned by a surveycore estimation function.
`...`	<`tidy-select`> Use `new_name = old_name` pairs to rename columns. Any number of columns can be renamed in a single call.
`.fn`	A function (or formula/lambda) applied to selected column names. Must return a character vector of the same length as its input, with no duplicates and no conflicts with existing non-renamed column names.
`.cols`	<`tidy-select`> Columns whose names `.fn` will transform. Defaults to all columns.
`.if_missing_var`	Per-call override of `collection@if_missing_var`. One of `"error"` or `"skip"`, or `NULL` (the default) to inherit the collection's stored value. See `surveycore::set_collection_if_miss`

Details

What gets updated:

- **Column names in** `@data` — the rename takes effect immediately.
- **Design specification** — if a renamed column is a design variable (weights, strata, PSU, FPC, or replicate weights), `@variables` is updated to track the new name.
- **Metadata** — variable labels, value labels, question prefaces, notes, and transformation records in `@metadata` are re-keyed to the new name.
- `visible_vars` — any occurrence of the old name in `@variables$visible_vars` is replaced with the new name, so `select()` + `rename()` pipelines work correctly.
- **Groups** — if a renamed column is in the active grouping, `@groups` is updated to use the new name.

Renaming design variables:

Renaming a design variable (e.g., the weights column) is intentionally allowed. A `surveytidy_warning_rename_design_` warning is issued as a reminder that the design specification has been updated — not to indicate an error.

rename_with() function forms:

`.fn` can be any of:

- A bare function: `rename_with(d, toupper)`
- A formula: `rename_with(d, ~ toupper(.))`
- A lambda: `rename_with(d, \(x) paste0(x, "_v2"))`

Extra arguments to `.fn` can be passed via `...`:

```
rename_with(d, stringr::str_replace, .cols = tidyselect::starts_with("y"),
            pattern = "y", replacement = "outcome")
```

`.cols` uses tidy-select syntax. The default `dplyr::everything()` applies `.fn` to all columns including design variables — which will trigger a `surveytidy_warning_rename_design_var` warning for each renamed design variable.

Value

An object of the same type as `.data` with the following properties:

- Rows are not added or removed.
- Column order is preserved.
- Renamed columns are updated in `@data`, `@variables`, `@metadata`, and `@groups`.
- Survey design attributes are preserved.

Survey collections

When applied to a `survey_collection`, `rename()` is dispatched to each member independently. Each member's `rename.survey_base` updates `@data`, `@variables`, `@metadata`, and `@groups` atomically.

Before dispatching, `rename.survey_collection` resolves the rename map against each member's `@data` and raises `surveytidy_error_collection_rename_group_partial` if any column in `coll@groups` would be renamed on some members but not others — that would leave the collection with an inconsistent `@groups` invariant (G1) that no `.if_missing_var` policy can recover. For plain `rename` the rename map is universal, so this branch normally fires only as a defense-in-depth catch for regressions in the surveycore G1b validator.

Renaming a non-group design variable (weights, ids, strata, fpc) emits `surveytidy_warning_rename_design_var` once per member — N firings on an N-member collection. Capture with `withCallingHandlers()`.

When applied to a `survey_collection`, `rename_with()` is dispatched to each member independently. Each member resolves `.cols` against its own `@data`, so a `.cols` like `where(is.factor)` may select different columns on different members.

Before dispatching, `rename_with.survey_collection` resolves `.cols` per-member and raises `surveytidy_error_collection_rename_group_partial` if any column in `coll@groups` would be renamed on some members but not others. This is the genuine trigger for the partial-rename class — `.cols` resolving differently across a heterogeneous collection is the path the spec is designed to catch (see §IV.4 reachability note).

Per-member design-variable warnings fire once per affected member.

See Also

mutate() to add or modify column values, select() to drop columns

Other modification: mutate

Examples

```
library(surveytidy)
library(surveycore)

# create a survey design from the pew_npors_2025 example dataset
d <- as_survey(pew_npors_2025, weights = weight, strata = stratum)

# rename() ------------------------------------------------------------------

# rename an outcome column
rename(d, financial_situation = fin_sit)

# rename multiple columns at once
rename(d, region = cregion, education = educcat)

# rename a design variable — warns and updates the design specification
rename(d, survey_weight = weight)

# rename_with() -------------------------------------------------------------

# apply a function to all matching columns
rename_with(d, toupper, .cols = tidyselect::starts_with("econ"))

# use a formula
rename_with(d, ~ paste0(., "_v2"), .cols = tidyselect::starts_with("econ"))
```

---

replace_values *Partially update values using an explicit mapping*

Description

replace_values() replaces each value of x found in from with the corresponding value from to. Values not found in from retain their original value unchanged.

Use replace_values() when updating only specific values of an existing variable. When remapping the full range of values in x, recode_values() is a better choice.

replace_values() automatically inherits value labels and the variable label from x. Supply .label or .value_labels to override the inherited values.

When any of .label, .value_labels, or .description are supplied, or when x carries existing labels, output label metadata is written to @metadata after mutate(). When none apply, the output is the same type as x.

Usage

```
replace_values(
  x,
  ...,
  from = NULL,
  to = NULL,
  .label = NULL,
  .value_labels = NULL,
  .description = NULL
)
```

Arguments

x	Vector to partially update.
...	These dots are for future extensions and must be empty.
from	Vector of old values to replace. Must be the same type as x.
to	Vector of new values corresponding to from. Must be the same length as from.
.label	character(1) or NULL. Variable label stored in @metadata@variable_labels after mutate(). Overrides the label inherited from x.
.value_labels	Named vector or NULL. Value labels stored in @metadata@value_labels. Names are the label strings; values are the data values. Merged with any existing labels inherited from x; entries in .value_labels take precedence over inherited entries with the same name.
.description	character(1) or NULL. Plain-language description of how the variable was created. Stored in @metadata@transformations[[col]]$description after mutate().

Value

An updated version of x with the same type and size. If x carries labels or any surveytidy args are supplied, returns a haven_labelled vector; otherwise returns the same type as x.

See Also

- dplyr::replace_values() for the base implementation.
- recode_values() for full value remapping with explicit from/to vectors; does not inherit labels from x automatically.
- replace_when() for condition-based partial replacement.
- na_if() to replace specific values with NA.

Other recoding: case_when(), if_else(), na_if(), recode_values(), replace_when()

Examples

```
library(surveycore)
library(surveytidy)
```

```
# create the survey design
ns_wave1_svy <- as_survey_nonprob(ns_wave1, weights = weight)

# basic replace_values — replace pid3 == 4 ("Something else") with 3
new <- ns_wave1_svy |>
  mutate(pid3_clean = replace_values(pid3, from = 4, to = 3)) |>
  select(pid3, pid3_clean)

new

# value labels from pid3 carry over to pid3_clean automatically
new@metadata@value_labels

# override the inherited variable label via .label
new <- ns_wave1_svy |>
  mutate(
    pid3_clean = replace_values(
      pid3,
      from = 4,
      to = 3,
      .label = "Party ID (3 categories)"
    )
  ) |>
  select(pid3, pid3_clean)

new@metadata@variable_labels

# provide updated value labels that reflect the recoded categories
new <- ns_wave1_svy |>
  mutate(
    pid3_clean = replace_values(
      pid3,
      from = 4,
      to = 3,
      .label = "Party ID (3 categories)",
      .value_labels = c(
        "Democrat" = 1,
        "Republican" = 2,
        "Independent/Other" = 3
      )
    )
  ) |>
  select(pid3, pid3_clean)

new@metadata@value_labels

# attach a plain-language description of the transformation
new <- ns_wave1_svy |>
  mutate(
    pid3_clean = replace_values(
      pid3,
      from = 4,
      to = 3,
```

```
      .label = "Party ID (3 categories)",
      .description = paste(
        "'Something else' (pid3 == 4) replaced with",
        "value 3 (Independent)."
      )
    )
  ) |>
  select(pid3, pid3_clean)

new@metadata@transformations
```

`replace_when` *Partially update a vector using conditional formulas*

Description

`replace_when()` is a survey-aware version of `dplyr::replace_when()` that evaluates each formula case sequentially and replaces matching elements of `x` with the corresponding RHS value. Elements where no case matches retain their original value from `x`.

Use `replace_when()` when partially updating an existing vector. When creating an entirely new vector from conditions, `case_when()` is a better choice.

`replace_when()` automatically inherits value labels and the variable label from `x`. Supply `.label` or `.value_labels` to override the inherited values.

When any of `.label`, `.value_labels`, or `.description` are supplied, or when `x` carries existing labels, output label metadata is written to `@metadata` after `mutate()`. When none apply, the output is identical to `dplyr::replace_when()`.

Usage

```
replace_when(x, ..., .label = NULL, .value_labels = NULL, .description = NULL)
```

Arguments

`x`	A vector to partially update.
`...`	<dynamic-dots> A sequence of two-sided formulas (`condition ~ value`). The left-hand side must be a logical vector the same size as `x`. The right-hand side provides the replacement value, cast to the type of `x`. Cases are evaluated sequentially; the first matching case is used. `NULL` inputs are ignored.
`.label`	`character(1)` or `NULL`. Variable label stored in `@metadata@variable_labels` after `mutate()`. Overrides the label inherited from `x`.
`.value_labels`	Named vector or `NULL`. Value labels stored in `@metadata@value_labels`. Names are the label strings; values are the data values. Merged with any existing labels inherited from `x`; entries in `.value_labels` take precedence over inherited entries with the same name.
`.description`	`character(1)` or `NULL`. Plain-language description of how the variable was created. Stored in `@metadata@transformations[[col]]$description` after `mutate()`.

Value

An updated version of `x` with the same type and size. If `x` carries labels or any surveytidy args are supplied, returns a `haven_labelled` vector; otherwise returns the same type as `x`.

See Also

- `dplyr::replace_when()` for the base implementation.
- `case_when()` to create an entirely new vector from conditions.
- `replace_values()` for in-place replacement using an explicit `from`/`to` mapping rather than conditions.

Other recoding: `case_when()`, `if_else()`, `na_if()`, `recode_values()`, `replace_values()`

Examples

```
library(surveycore)
library(surveytidy)

# create the survey design
ns_wave1_svy <- as_survey_nonprob(ns_wave1, weights = weight)

# basic replace_when — replace pid3 == 4 ("Something else") with 3
new <- ns_wave1_svy |>
  mutate(pid3_clean = replace_when(pid3, pid3 == 4 ~ 3)) |>
  select(pid3, pid3_clean)

new

# value labels from pid3 carry over to pid3_clean automatically
new@metadata@value_labels

# override the inherited variable label via .label
new <- ns_wave1_svy |>
  mutate(
    pid3_clean = replace_when(
      pid3,
      pid3 == 4 ~ 3,
      .label = "Party ID (3 categories)"
    )
  ) |>
  select(pid3, pid3_clean)

new@metadata@variable_labels

# provide updated value labels reflecting the collapsed categories
new <- ns_wave1_svy |>
  mutate(
    pid3_clean = replace_when(
      pid3,
      pid3 == 4 ~ 3,
      .label = "Party ID (3 categories)",
      .value_labels = c(
```

```
        "Democrat" = 1,
        "Republican" = 2,
        "Independent/Other" = 3
      )
    )
  ) |>
  select(pid3, pid3_clean)

new@metadata@value_labels

# attach a plain-language description of the transformation
new <- ns_wave1_svy |>
  mutate(
    pid3_clean = replace_when(
      pid3,
      pid3 == 4 ~ 3,
      .label = "Party ID (3 categories)",
      .description = paste(
        "Recoded pid3: 'Something else' (4) merged into",
        "Independent (3)."
      )
    )
  ) |>
  select(pid3, pid3_clean)

new@metadata@transformations
```

`right_join` *Unsupported joins for survey designs*

Description

`right_join()` and `full_join()` error unconditionally for survey design objects because they can add rows from `y` that have no match in the survey. Those new rows would have `NA` for all design variables (weights, strata, PSU), producing an invalid design object.

Usage

```
## S3 method for class 'survey_collection'
right_join(x, y, ..., .if_missing_var = NULL)

## S3 method for class 'survey_collection'
full_join(x, y, ..., .if_missing_var = NULL)

right_join(
  x,
  y,
  by = NULL,
```

```
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL
)

full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL
)
```

Arguments

x	A survey_base object.
y	A data frame or survey object.
...	Additional arguments (ignored; the function always errors).
.if_missing_var	Per-call override of collection@if_missing_var. One of "error" or "skip", or NULL (the default) to inherit the collection's stored value. See surveycore::set_collection_if_miss
by	Ignored — the function always errors.
copy	Ignored — the function always errors.
suffix	Ignored — the function always errors.
keep	Ignored — the function always errors.

Details

Use left_join() to add lookup columns from y. Use filter() or semi_join() to restrict the survey domain.

Value

Never returns — always throws an error.

Survey collections

When called on a surveycore::survey_collection, right_join() errors unconditionally with class surveytidy_error_collection_verb_unsupported. The semantics for joining a plain data frame onto a multi-survey container are still being designed. Apply the join inside a per-survey pipeline before constructing the collection.

When called on a surveycore::survey_collection, full_join() errors unconditionally with class surveytidy_error_collection_verb_unsupported. The semantics for joining a plain data frame onto a multi-survey container are still being designed. Apply the join inside a per-survey pipeline before constructing the collection.

See Also

Other joins: bind_cols(), bind_rows(), inner_join, left_join, semi_join

Examples

```
# create a tiny survey object and a lookup table with an extra row
d <- surveycore::as_survey(
  data.frame(wt = c(1, 1), y1 = c(1, 2)),
  weights = wt
)
lookup <- data.frame(y1 = c(1, 2, 3), label = c("a", "b", "c"))

# right_join() and full_join() always error on a survey object — they would
# add rows with NA design variables, producing an invalid design
tryCatch(
  right_join(d, lookup, by = "y1"),
  error = function(e) message(conditionMessage(e))
)

tryCatch(
  full_join(d, lookup, by = "y1"),
  error = function(e) message(conditionMessage(e))
)

# the recommended alternative: use left_join() to add lookup columns
# without changing the row set
left_join(d, lookup, by = "y1")
```

---

rowwise *Compute row-wise on a survey design object*

Description

rowwise() enables row-by-row computation in mutate(). Each row is treated as an independent group, so expressions like mutate(d, row_max = max(dplyr::c_across(tidyselect::starts_with("y")))) compute the maximum across columns for each row independently.

Use ungroup() or group_by() to exit rowwise mode.

Usage

```
rowwise(data, ...)

## S3 method for class 'survey_base'
rowwise(data, ...)

## S3 method for class 'survey_collection'
rowwise(data, ..., .if_missing_var = NULL)
```

Arguments

data	A survey_base object.
...	<tidy-select> Optional id columns that identify each row (used with dplyr::c_across()). Commonly omitted.
.if_missing_var	Per-call override of collection@if_missing_var. One of "error" or "skip", or NULL (the default) to inherit the collection's stored value. See surveycore::set_collection_if_miss

Details

Storage:

Rowwise mode is stored in @variables$rowwise (logical TRUE) and @variables$rowwise_id_cols (character vector of id column names). @groups is **not** modified — rowwise mode is independent of grouping.

Exiting rowwise mode:

- ungroup(d) — exits rowwise mode and removes all groups.
- group_by(d, ...) — exits rowwise mode and sets new groups.
- group_by(d, ..., .add = TRUE) — promotes id columns to groups, then appends the new groups, then exits rowwise mode.

mutate() behaviour:

mutate() detects rowwise mode and routes internally through dplyr::rowwise(@data) before calling dplyr::mutate(). The rowwise_df class is stripped from @data after mutation so subsequent operations are not accidentally rowwise.

Value

data with @variables$rowwise = TRUE and @variables$rowwise_id_cols set. All other properties are unchanged.

Survey collections

When applied to a survey_collection, rowwise() is dispatched to each member independently — every member receives @variables$rowwise = TRUE and the same @variables$rowwise_id_cols. The collection has no rowwise marker; rowwise state lives entirely per-member. @groups, @id, and @if_missing_var on the collection are unchanged.

Construction-time uniformity is by-construction: every member is rowwise after the call. Mixed rowwise state across members is detected later by mutate() (see §IV.5 of the survey-collection spec) and warned about rather than blocked.

See Also

Other grouping: group_by, is_grouped(), is_rowwise()

Examples

```
# create a survey object from the bundled NPORS dataset
d <- surveycore::as_survey(
  surveycore::pew_npors_2025,
  weights = weight,
  strata = stratum
)

# row-wise max across several columns
d |>
  rowwise() |>
  mutate(
    row_max = max(dplyr::c_across(tidyselect::starts_with("econ")), na.rm = TRUE)
  )

# exit rowwise mode
d |>
  rowwise() |>
  ungroup()
```

`row_means` *Compute row-wise means across selected columns*

Description

`row_means()` computes the mean of each row across a tidyselect-selected set of numeric columns. It is designed for use inside `mutate()` on survey design objects. When called inside `mutate()`, the transformation is recorded in `@metadata@transformations[[col]]`.

Usage

```
row_means(.cols, na.rm = FALSE, .label = NULL, .description = NULL)
```

Arguments

`.cols`	<tidy-select> Columns to average across, evaluated via `dplyr::pick()`. Typical values: `c(a, b, c)`, `starts_with("y")`, `where(is.numeric)`. Must resolve to at least one column, and all selected columns must be numeric.
`na.rm`	`logical(1)`. If `TRUE`, `NA` values are excluded before computing the mean. If all values in a row are `NA` and `na.rm = TRUE`, the result is `NaN` (matching base R `rowMeans()` behavior). Default `FALSE`.
`.label`	`character(1)` or `NULL`. Variable label stored in `@metadata@variable_labels[[col]]` after `mutate()`. If `NULL`, falls back to the output column name from `dplyr::cur_column()`.
`.description`	`character(1)` or `NULL`. Plain-language description of the transformation stored in `@metadata@transformations[[col]]$description` after `mutate()`.

Value

A double vector of length equal to the number of rows in the current data context.

See Also

Other transformation: make_binary(), make_dicho(), make_factor(), make_flip(), make_rev(), row_sums()

Examples

```
# create a dummy survey object
d <- surveycore::as_survey(
  data.frame(
    y1 = c(1, 2, 3),
    y2 = c(4, 5, 6),
    wt = c(1, 1, 1)
  ),
  weights = wt
)

# use a vector of columns to create the score
mutate(d, score = row_means(c(y1, y2)))

# use tidy-select for columns and add a label
d |>
  mutate(
    score = row_means(
      tidyselect::starts_with("y"),
      na.rm = TRUE,
      .label = "Score"
    )
  )
```

---

row_sums *Compute row-wise sums across selected columns*

Description

row_sums() computes the sum of each row across a tidyselect-selected set of numeric columns. It is designed for use inside mutate() on survey design objects. When called inside mutate(), the transformation is recorded in @metadata@transformations[[col]].

Usage

```
row_sums(.cols, na.rm = FALSE, .label = NULL, .description = NULL)
```

Arguments

.cols	<tidy-select> Columns to sum across, evaluated via dplyr::pick(). Typical values: c(a, b, c), starts_with("y"), where(is.numeric). Must resolve to at least one column, and all selected columns must be numeric.
na.rm	logical(1). If TRUE, NA values are excluded before summing. If all values in a row are NA and na.rm = TRUE, the result is 0 (matching base R rowSums() behavior). Default FALSE.
.label	character(1) or NULL. Variable label stored in @metadata@variable_labels[[col]] after mutate(). If NULL, falls back to the output column name from dplyr::cur_column().
.description	character(1) or NULL. Plain-language description of the transformation stored in @metadata@transformations[[col]]$description after mutate().

Value

A double vector of length equal to the number of rows in the current data context.

See Also

Other transformation: make_binary(), make_dicho(), make_factor(), make_flip(), make_rev(), row_means()

Examples

```
# create a dummy survey object
d <- surveycore::as_survey(
  data.frame(
    y1 = c(1, 2, 3),
    y2 = c(4, 5, 6),
    wt = c(1, 1, 1)
  ),
  weights = wt
)

# use a vector of columns to create the total
mutate(d, total = row_sums(c(y1, y2)))

# use tidy-select for columns and add a label
d |>
  mutate(
    total = row_sums(
      tidyselect::starts_with("y"),
      na.rm = TRUE,
      .label = "Total"
    )
  )
```

`select` *Keep or drop columns using their names and types*

Description

`select()` keeps the named columns and drops all others, using the tidyselect mini-language to describe column sets. Design variables (weights, strata, PSU, FPC, replicate weights) are **always retained** even when not explicitly selected — they are required for variance estimation. After `select()`, `print()` shows only the columns you selected; design variables remain in the object but are hidden from display.

`select()` is irreversible: dropped columns are permanently removed from the survey object and cannot be recovered within the same pipeline.

Usage

```
select(.data, ...)

## S3 method for class 'survey_base'
select(.data, ...)

## S3 method for class 'survey_result'
select(.data, ...)

## S3 method for class 'survey_collection'
select(.data, ..., .if_missing_var = NULL)
```

Arguments

`.data`	A `survey_base` object, or a `survey_result` object returned by a surveycore estimation function.
`...`	<`tidy-select`> One or more unquoted column names or tidy-select expressions.
`.if_missing_var`	Per-call override of `collection@if_missing_var`. One of `"error"` or `"skip"`, or `NULL` (the default) to inherit the collection's stored value. See `surveycore::set_collection_if_miss`

Details

Design variable preservation:

Regardless of what you select, the following are always kept in the survey object: weights, strata, PSUs, FPC columns, replicate weights, and the domain column (if set by `filter()`). They are hidden from `print()` output but remain available for variance estimation.

Metadata:

Variable labels, value labels, and other metadata for dropped columns are removed. Metadata for retained columns is preserved.

Value

An object of the same type as `.data` with the following properties:

- Rows are not modified.
- Non-selected, non-design columns are permanently removed.
- Design variables are always retained.
- Survey design attributes are preserved.

Survey collections

When applied to a `survey_collection`, `select()` is dispatched to each member independently. Each member resolves its own tidyselect expression against its own `@data`, so members may end up with different visible columns when the selection is partial (e.g., `any_of()` against a heterogeneous collection). Per-member design variables are always retained.

Before dispatching, `select.survey_collection` resolves the selection against the first member's `@data` and raises `surveytidy_error_collection_select_group_removed` if any column in `coll@groups` would be removed. Group columns must remain in every member; silently dropping them would violate the surveycore class validator (G1b). Use `ungroup()` first if you intend to remove a group column.

`relocate.survey_collection` is **not** subject to this pre-flight — `dplyr::relocate` only reorders columns and never drops them.

See Also

`relocate()` to reorder columns, `rename()` to rename them, `mutate()` to add new ones

Other selecting: `glimpse.survey_collection()`, `pull.survey_collection()`, `relocate`

Examples

```
library(surveytidy)
library(surveycore)

# create a survey design from the pew_npors_2025 example dataset
d <- as_survey(pew_npors_2025, weights = weight, strata = stratum)

# select by name
select(d, gender, agecat)

# select by name pattern
select(d, tidyselect::starts_with("smuse_"))

# select by type
select(d, tidyselect::where(is.numeric))

# drop columns with !
select(d, !tidyselect::starts_with("smuse_"))
```

semi_join *Domain-aware semi- and anti-join for survey designs*

Description

`semi_join()` marks rows as in-domain when they have a match in `y`. `anti_join()` marks rows as in-domain when they do NOT have a match in `y`. Neither function removes rows or adds new columns — they are implemented as domain operations, exactly like `filter()`.

Usage

```
## S3 method for class 'survey_collection'
semi_join(x, y, ..., .if_missing_var = NULL)

## S3 method for class 'survey_collection'
anti_join(x, y, ..., .if_missing_var = NULL)

semi_join(x, y, by = NULL, copy = FALSE, ...)

anti_join(x, y, by = NULL, copy = FALSE, ...)
```

Arguments

`x`	A `survey_base` object.
`y`	A plain data frame. Must not be a survey object.
`...`	Additional arguments forwarded to the underlying dplyr function.
`.if_missing_var`	Per-call override of `collection@if_missing_var`. One of `"error"` or `"skip"`, or `NULL` (the default) to inherit the collection's stored value. See `surveycore::set_collection_if_miss`
`by`	A character vector of column names or a `dplyr::join_by()` specification. `NULL` uses all common column names.
`copy`	Forwarded to the underlying dplyr function.

Details

Domain awareness:

Unlike standard `dplyr::semi_join()` and `dplyr::anti_join()`, these implementations never physically remove rows. Instead, unmatched (or matched, for `anti_join`) rows are marked `FALSE` in the `..surveycore_domain..` column of `@data`, exactly as `filter()` does. This preserves variance estimation validity.

Chaining:

Multiple calls accumulate via AND: a row must satisfy every condition from every `filter()`, `semi_join()`, and `anti_join()` call to remain in-domain.

Duplicate keys in y:

Duplicate keys in `y` collapse to a single `TRUE` (for `semi_join`) or a single `FALSE` (for `anti_join`) per survey row. Row expansion is not possible with these functions.

@variables$domain sentinel:

A typed S3 sentinel of class `"surveytidy_join_domain"` is appended to `@variables$domain`. Phase 1 consumers can use `inherits(entry, "surveytidy_join_domain")` to distinguish join sentinels from quosures.

Value

A survey design object of the same type as `x` with the domain column (`..surveycore_domain..`) updated. Row count unchanged. No new columns added.

Survey collections

When called on a `surveycore::survey_collection`, `semi_join()` errors unconditionally with class `surveytidy_error_collection_verb_unsupported`. The semantics for joining a plain data frame onto a multi-survey container are still being designed. Apply the join inside a per-survey pipeline before constructing the collection.

When called on a `surveycore::survey_collection`, `anti_join()` errors unconditionally with class `surveytidy_error_collection_verb_unsupported`. The semantics for joining a plain data frame onto a multi-survey container are still being designed. Apply the join inside a per-survey pipeline before constructing the collection.

See Also

Other joins: `bind_cols()`, `bind_rows()`, `inner_join`, `left_join`, `right_join`

Examples

```
# create a small survey object
df <- data.frame(
  psu = paste0("psu_", 1:5),
  strata = "s1",
  fpc = 100,
  wt = 1,
  y1 = 1:5
)
d <- surveycore::as_survey(
  df,
  ids = psu,
  weights = wt,
  strata = strata,
  fpc = fpc,
  nest = TRUE
)
keepers <- data.frame(y1 = c(1, 3, 5))

# semi_join: rows matching keepers stay in-domain
semi_join(d, keepers, by = "y1")
```

```
# anti_join: rows matching keepers are marked out-of-domain
anti_join(d, keepers, by = "y1")
```

---

slice *Physically select rows of a survey design object*

Description

`slice()`, `slice_head()`, `slice_tail()`, `slice_min()`, `slice_max()`, and `slice_sample()` **physically remove rows** from a survey design object. For subpopulation analyses, use `filter()` instead — it marks rows as out-of-domain without removing them, preserving valid variance estimation.

All slice functions always issue `surveycore_warning_physical_subset` and error if the result would have 0 rows.

Usage

```
slice(.data, ..., .by = NULL, .preserve = FALSE)

## S3 method for class 'survey_base'
slice(.data, ...)

## S3 method for class 'survey_base'
slice_head(.data, ...)

## S3 method for class 'survey_base'
slice_tail(.data, ...)

## S3 method for class 'survey_base'
slice_min(.data, ...)

## S3 method for class 'survey_base'
slice_max(.data, ...)

## S3 method for class 'survey_base'
slice_sample(.data, ...)

## S3 method for class 'survey_result'
slice(.data, ...)

## S3 method for class 'survey_result'
slice_head(.data, ...)

## S3 method for class 'survey_result'
slice_tail(.data, ...)
```

```
## S3 method for class 'survey_result'
slice_min(.data, ...)

## S3 method for class 'survey_result'
slice_max(.data, ...)

## S3 method for class 'survey_result'
slice_sample(.data, ...)

## S3 method for class 'survey_collection'
slice(.data, ...)

## S3 method for class 'survey_collection'
slice_head(.data, ..., n = NULL, prop = NULL)

## S3 method for class 'survey_collection'
slice_tail(.data, ..., n = NULL, prop = NULL)

## S3 method for class 'survey_collection'
slice_min(
  .data,
  order_by,
  ...,
  n = NULL,
  prop = NULL,
  by = NULL,
  with_ties = TRUE,
  na_rm = FALSE,
  .if_missing_var = NULL
)

## S3 method for class 'survey_collection'
slice_max(
  .data,
  order_by,
  ...,
  n = NULL,
  prop = NULL,
  by = NULL,
  with_ties = TRUE,
  na_rm = FALSE,
  .if_missing_var = NULL
)

## S3 method for class 'survey_collection'
slice_sample(
  .data,
  ...,
```

```
  n = NULL,
  prop = NULL,
  by = NULL,
  weight_by = NULL,
  replace = FALSE,
  seed = NULL,
  .if_missing_var = NULL
)
```

Arguments

.data	A survey_base object, a survey_result object returned by a surveycore estimation function, or a survey_collection.
...	Passed to the corresponding dplyr::slice_*() function. For slice() only, the ... accepts a vector of row indices.
.by	Accepted for interface compatibility; not used by survey methods.
.preserve	Accepted for interface compatibility; not used by survey methods.
n	Number of rows to keep. See dplyr::slice_head().
prop	Fraction of rows to keep (between 0 and 1). See dplyr::slice_head().
order_by	<data-masking> Variable to order by, used by slice_min() and slice_max(). See dplyr::slice_min().
by	Per-call grouping override accepted by slice_min(), slice_max(), and slice_sample(). Not supported on survey_collection — passing a non-NULL value raises surveytidy_error_collection_by_unsupported. Use group_by() on the collection (or set coll@groups) instead.
with_ties	Should ties be kept together? Used by slice_min() and slice_max(). See dplyr::slice_min().
na_rm	Should missing values in order_by be removed before slicing? Used by slice_min() and slice_max(). See dplyr::slice_min().
.if_missing_var	Per-call override of collection@if_missing_var. One of "error" or "skip", or NULL (the default) to inherit the collection's stored value. See surveycore::set_collection_if_miss
weight_by	<data-masking> Sampling weights for slice_sample(). See dplyr::slice_sample(). Independent of the survey design weights — issues surveytidy_warning_slice_sample_weight_by as a reminder.
replace	Should sampling be performed with replacement? Used by slice_sample(). See dplyr::slice_sample().
seed	Used by slice_sample.survey_collection only. NULL (the default) leaves the ambient RNG state alone; an integer seed makes per-survey samples deterministic and order-independent (see "Survey collections" below).

Details

Physical subsetting:

Unlike filter(), slice functions actually remove rows. This changes the survey design — unless the design was explicitly built for the subset population, variance estimates may be incorrect.

`slice_sample()` **and survey weights:**

`slice_sample(weight_by = )` samples rows proportional to a column's values, independently of the survey design weights. A `surveytidy_warning_slice_sample_weight_by` warning is issued as a reminder. If you intend probability-proportional sampling, use the design weights directly.

Value

An object of the same type as `.data` with the following properties:

- A subset of rows is retained; unselected rows are permanently removed.
- Columns and survey design attributes are unchanged.
- Always issues `surveycore_warning_physical_subset`.

Survey collections

Slice variants are dispatched to each member independently. Each member's `slice_*.survey_base` call emits `surveycore_warning_physical_subset` — an N-member collection therefore surfaces N warnings.

Before dispatching, a verb-specific pre-flight raises `surveytidy_error_collection_slice_zero` when the supplied arguments would produce a 0-row result on every member (e.g., `n = 0`, literal `slice(integer(0))`). This stops dispatch before any member is touched, so users see a slice-specific message instead of a misleading per-member validator failure.

`slice`, `slice_head`, `slice_tail`, and `slice_sample` (when `weight_by = NULL`) reference no user columns — their signatures omit `.if_missing_var`. `slice_min`, `slice_max`, and `slice_sample` with a non-NULL `weight_by` do reference user columns; their signatures include `.if_missing_var`.

`slice_min`, `slice_max`, and `slice_sample` reject the per-call `by` argument with `surveytidy_error_collection_by_unsu` use `group_by()` on the collection (or `coll@groups`) instead.

`slice_sample.survey_collection` **reproducibility**

`slice_sample.survey_collection` adds a `seed = NULL` argument absent from `slice_sample.survey_base`.

- `seed = NULL` (default): no seed manipulation. Per-survey `slice_sample()` calls draw from the ambient RNG state in iteration order. Reproducibility requires a single upstream `set.seed()` AND a stable collection size and member order — adding or removing a survey changes the samples drawn from every subsequent survey.
- `seed = <integer>`: each per-survey call is wrapped with a deterministic per-survey seed derived as `strtoi(substr(rlang::hash(paste0(survey_name, "::", seed)), 1, 7), 16L)`. Per-survey samples are stable regardless of collection order, additions, or removals. The ambient `.Random.seed` is restored on exit.

For any analysis intended to be reproducible, pass an explicit integer `seed`.

See Also

`filter()` for domain-aware row marking (preferred for subpopulation analyses), `arrange()` for row sorting

Other row operations: `distinct`, `drop_na`

Examples

```
# create a survey object from the bundled NPORS dataset
d <- surveycore::as_survey(
  surveycore::pew_npors_2025,
  weights = weight,
  strata = stratum
)

# first 10 rows (issues a physical subset warning)
slice_head(d, n = 10)

# rows with the 5 lowest survey weights
slice_min(d, order_by = weight, n = 5)

# random sample of 50 rows
slice_sample(d, n = 50)
```

---

subset.survey_base *Physically remove rows from a survey design object*

Description

`subset()` physically removes rows from a `survey_base` object where `condition` evaluates to `FALSE`. **This changes the survey design.** Unless the design was explicitly built for the subset population, variance estimates will be incorrect.

For subpopulation analyses, use `filter()` instead. `filter()` marks rows as in or out of the domain without removing them, leaving the full design intact for variance estimation.

`subset()` always emits a `surveycore_warning_physical_subset` warning as a reminder of the statistical implications.

Usage

```
## S3 method for class 'survey_base'
subset(x, condition, ...)
```

Arguments

`x`	A `survey_base` object.
`condition`	A logical expression evaluated against the survey data. Rows where `condition` is `FALSE` or `NA` are removed.
`...`	Ignored. Included for compatibility with the base `subset()` generic.

Value

An object of the same type as `x` with only matching rows retained. Always issues `surveycore_warning_physical_subset`.

See Also

filter() for domain-aware row marking (preferred for subpopulation analyses)

Examples

```
library(surveytidy)
library(surveycore)

# create a survey design from the pew_npors_2025 example dataset
d <- as_survey(pew_npors_2025, weights = weight, strata = stratum)

# physical row removal — always issues a warning
subset(d, agecat >= 3)
```

Index