# Package 'pmdplyr'

May 30, 2020

**Type** Package

**Title** 'dplyr' Extension for Common Panel Data Maneuvers

**Version** 0.3.3

**Description** Using the 'dplyr' package as a base, adds a family of functions designed to make manipulating panel data easier. Allows the addition of indexing variables to a tibble to create a pibble, and the manipulation of data based on those indexing variables.

**License** MIT + file LICENSE

**URL** https://nickch-k.github.io/pmdplyr, https://github.com/NickCH-K/pmdplyr

**BugReports** https://github.com/NickCH-K/pmdplyr/issues

**Copyright** file COPYRIGHTS

**Encoding** UTF-8

**Depends** R (>= 3.4), dplyr (>= 1.0.0)

**Imports** lubridate, magrittr, pillar, tidyr, tibble, tidyselect, utils, vctrs, rlang (>= 0.4.0)

**Suggests** tsibble, plm, panelr, haven, utf8, sjlabelled, knitr, rmarkdown, testthat (>= 2.1.0), covr

**RoxygenNote** 7.1.0.9000

**VignetteBuilder** knitr

**LazyData** true

**NeedsCompilation** no

**Author** Nick Huntington-Klein [aut, cre] (<https://orcid.org/0000-0002-7352-3991>), Philip Khor [aut] (<https://orcid.org/0000-0002-8333-1256>)

**Maintainer** Nick Huntington-Klein <nhuntington-klein@fullerton.edu>

**Repository** CRAN

**Date/Publication** 2020-05-30 07:30:02 UTC

# R topics documented:



---

`as_pibble` *Coerce to a pibble panel data set object*

---

## Description

This function coerces a tibble, data.frame, or list to a pibble tibble by adding the `.i`, `.t`, and `.d` attributes to it.

## Usage

```
as_pibble(x, .i = NULL, .t = NULL, .d = 1, .uniqcheck = FALSE, ...)

## S3 method for class 'tbl_df'
as_pibble(x, .i = NULL, .t = NULL, .d = 1, .uniqcheck = FALSE, ...)

## S3 method for class 'grouped_df'
as_pibble(x, .i = NULL, .t = NULL, .d = 1, .uniqcheck = FALSE, ...)

## S3 method for class 'data.frame'
as_pibble(x, .i = NULL, .t = NULL, .d = 1, .uniqcheck = FALSE, ...)
```

```
## S3 method for class 'list'
as_pibble(x, .i = NULL, .t = NULL, .d = 1, .uniqcheck = FALSE, ...)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble or list |
| .i | Quoted or unquoted variable(s) that identify the individual cases. If this is omitted, pibble will assume the data set is a single time series. |
| .t | Quoted or unquoted variable indicating the time. pmdplyr accepts two kinds of time variables: numeric variables where a fixed distance .d will take you from one observation to the next, or, if .d=0, any standard variable type with an order. Consider using the time_variable() function to create the necessary variable if your data uses a Date variable for time. |
| .d | Number indicating the gap in t between one period and the next. For example, if .t indicates a single day but data is collected once a week, you might set .d=7. To ignore gap length and assume that "one period ago" is always the most recent prior observation in the data, set .d=0. By default, .d=1. |
| .uniqcheck | Logical parameter. Set to TRUE to perform a check of whether .i and .t uniquely identify observations, and present a message if not. By default this is set to FALSE and the warning message occurs only once per session. |
| ... | Other arguments passed on to individual methods. |

## Details

- .i, Quoted or unquoted variable(s) indicating the individual-level panel identifier
- .t, Quoted or unquoted variable indicating the time variable
- .d, a number indicating the gap

Note that pibble does not require that .i and .t uniquely identify the observations in your data, but it will give a warning message (a maximum of once per session, unless .uniqcheck=TRUE) if they do not.

## Examples

```
data(SPrail)
# I set .d=0 here to indicate that I don't care how large the gap
# between one period and the next is.
# If I want to use 'insert_date' for .t with a fixed gap between periods,
# I need to transform it into an integer first; see time_variable()
SP <- as_pibble(SPrail,
  .i = c(origin, destination),
  .t = insert_date,
  .d = 0
)
is_pibble(SP)
attr(SP, ".i")
attr(SP, ".t")
attr(SP, ".d")
```

```
data(Scorecard)
# Here, year is an integer, so I can use it with .d = 1 to
# indicate that one period is a change of one unit in year
# Conveniently, .d = 1 is the default
Scorecard <- as_pibble(Scorecard, .i = unitid, .t = year)
is_pibble(Scorecard)
```

---

`fixed_check` *Check for inconsistency in variables that should be fixed*

---

## Description

This function checks whether one set of variables is consistent within values of another set of variables. If they are, returns `TRUE`. If they aren't, it will return a list of data frames, one for each element of `.var`, consisting only of the observations and variables in which there are inconsistencies.

## Usage

```
fixed_check(.df, .var = NULL, .within = NULL)
```

## Arguments

| | |
|---|---|
| `.df` | Data frame, pibble, or tibble. |
| `.var` | Quoted or unquoted variable(s) in `.df` that are to be checked for consistency. If not specified, uses all variables in `.df` that are not in `.within`. |
| `.within` | Quotes or unquoted variable(s) that the `.var` variables should be consistent within. |

## Examples

```
# In the Scorecard data, it should be the case that
# state_abbr and inst_name never change within university.
# Let's see if that's true
data(Scorecard)
fixed_check(Scorecard, .var = c(state_abbr, inst_name), .within = unitid)
# it returns TRUE! We're good to go

# count_not_working has no reason to be constant within unitid,
# but let's see what happens if we run it through
fixed_check(Scorecard, .var = count_not_working, .within = unitid)
# It gives back a tibble with inconsistent obs!
```

`fixed_force` *Enforce consistency in variables*

## Description

This function forces values the variables in `.var` to take constant values within combinations of the variables in `.within`. `fixed_force()` will return a data frame with consistency enforced.

## Usage

```
fixed_force(
  .df,
  .var = NULL,
  .within = NULL,
  .resolve = mode_order,
  .flag = NA
)
```

## Arguments

| | |
|---|---|
| `.df` | Data frame, pibble, or tibble. |
| `.var` | Quoted or unquoted variable(s) in `.df` that should be consistent. If not specified, uses all variables in `.df` that are not in `.within`. |
| `.within` | Quotes or unquoted variable(s) that the `.var` variables should be consistent within. |
| `.resolve` | Function capable of being passed to `dplyr::summarize()` that will be used to resolve inconsistencies. Or, set to `'drop'` or any string to drop all inconsistent observations. By default, this will return the mode (ties use the first observed value). |
| `.flag` | String indicating the name of a new variable that flags any observations altered by `fixed_force()`. |

## Details

Inconsistencies will be resolved by the function `.resolve`. Or, set `.resolve` to `'drop'` (or any string, really) to drop all cases with inconsistency.

## Examples

```
data(Scorecard)
# The variables pred_degree_awarded_ipeds and state_abbr should be constant within unitid
# However, sometimes colleges change what they offer.
# For the purpose of my analysis, though,
# I want to treat any changers as whatever they are most often (the mode).
# So let's enforce that with fixed_force
Scorecard <- fixed_force(Scorecard,
```

```
  .var = c(pred_degree_awarded_ipeds, state_abbr),
  .within = unitid, .flag = "changed"
)
# Did we catch any changers?
table(Scorecard$changed)
# We did!
```

---

`id_variable` *Create a single panel ID variable out of several*

---

## Description

The pmdplyr library accepts the use of multiple ID variables. However, you may wish to combine these into a single variable, or renumber the single variable you already have for some reason.

## Usage

```
id_variable(..., .method = "number", .minwidth = FALSE)
```

## Arguments

| | |
|---|---|
| `...` | variables (vectors) that, together, make up the ID variables in the data and uniquely identifies the individual. Note that `id_variable()` will not check whether you've selected an appropriate set of variables; try running `as_pibble()` after getting your ID and time variables. |
| `.method` | Can be `'number'`, `'random'`, or `'character'`, as described below. |
| `.minwidth` | If `.method = 'character'`, omits the additional spacing that makes the ID variable fixed-width and ensures uniqueness. WARNING: This option saves space but may in rare cases cause two individuals to have the same ID. Defaults to `FALSE`. |

## Details

By default, id_variable() will create a unique numeric identifier out of your ID variables, sequential following the order in the original data (`.method='number'`). However, you may want to remove the ordering and assign IDs randomly (`.method='random'`), or preserve all the original information and create a single fixed-width character ID variable that contains all the original information (`.method='character'`).

## Examples

```
data(SPrail)
# I want to identify observations at the route (origin-destination)/year level
# Let's make it a character variable so we can tell at a glance what route we're talking
SPrail <- SPrail %>%
  dplyr::mutate(route_id = id_variable(origin, destination, .method = "character"))
```

`inexact_join` *Join two data frames inexactly*

## Description

These functions are modifications of the standard `dplyr join` functions, except that it allows a variable of an ordered type (like date or numeric) in `x` to be matched in inexact ways to variables in `y`.

## Usage

```
inexact_inner_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  var = NULL,
  jvar = NULL,
  method,
  exact = TRUE
)

inexact_left_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  var = NULL,
  jvar = NULL,
  method,
  exact = TRUE
)

inexact_right_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  var = NULL,
  jvar = NULL,
  method,
```

```
  exact = TRUE
)

inexact_full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  var = NULL,
  jvar = NULL,
  method,
  exact = TRUE
)

inexact_semi_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  ...,
  var = NULL,
  jvar = NULL,
  method,
  exact = TRUE
)

inexact_nest_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  keep = FALSE,
  name = NULL,
  ...,
  var = NULL,
  jvar = NULL,
  method,
  exact = TRUE
)

inexact_anti_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  ...,
```

```
  var = NULL,
  jvar = NULL,
  method,
  exact = TRUE
)
```

## Arguments

| | |
|---|---|
| `x, y, by, copy, suffix, keep, name, ...` | Arguments to be passed to the relevant `join` function. |
| `var` | Quoted or unquoted variable from the `x` data frame which is to be indirectly matched. |
| `jvar` | Quoted or unquoted variable(s) from the `y` data frame which are to be indirectly matched. These cannot be variable names also in `x` or `var`. |
| `method` | The approach to be taken in performing the indirect matching. |
| `exact` | A logical, where `TRUE` indicates that exact matches are acceptable. For example, if `method = 'last'`, `x` contains `var = 2`, and `y` contains `jvar = 1` and `jvar = 2`, then `exact = TRUE` will match with the `jvar = 2` observation, and `exact = FALSE` will match with the `jvar = 1` observation. If `jvar` contains two variables and you want them treated differently, set to `c(TRUE,FALSE)` or `c(FALSE,TRUE)`. |

## Details

This allows matching, for example, if one data set contains data from multiple days in the week, while the other data set is weekly. Another example might be matching an observation in one data set to the *most recent* previous observation in the other.

The available methods for matching are:

- `method = "last"` matches `var` to the closest value of `jvar` that is *lower*.
- `method = "next"` matches `var` to the closest value of `jvar` that is *higher*.
- `method = "closest"` matches `var` to the closest value of `jvar`, above or below. If equidistant between two values, picks the lower of the two.
- `method = "between"` requires two variables in `jvar` which constitute the beginning and end of a range, and matches `var` to the range it is in. Make sure that the ranges are non-overlapping within the joining variables, or else you will get strange results (specifically, it should join to the earliest-starting range). If the end of one range is the exact start of another, `exact = c(TRUE,FALSE)` or `exact = c(FALSE,TRUE)` is recommended to avoid overlaps. Defaults to `exact = c(TRUE,FALSE)`.

Note that if, given the method, `var` finds no proper match, it will be merged with any `is.na(jvar[1])` values.

## Examples

```
data(Scorecard)
# We also have this data on the December unemployment rate for US college grads nationally
```

```
# but only every other year
unemp_data <- data.frame(
  unemp_year = c(2006, 2008, 2010, 2012, 2014, 2016, 2018),
  unemp = c(.017, .036, .048, .040, .028, .025, .020)
)
# I want to match the most recent unemployment data I have to each college
Scorecard <- Scorecard %>%
  inexact_left_join(unemp_data,
    method = "last",
    var = year,
    jvar = unemp_year
  )

# Or perhaps I want to find the most recent lagged value (i.e. no exact matches, only recent ones)
data(Scorecard)
Scorecard <- Scorecard %>%
  inexact_left_join(unemp_data,
    method = "last",
    var = year,
    jvar = unemp_year,
    exact = FALSE
  )

# Another way to do the same thing would be to specify the range of unemp_years I want exactly
data(Scorecard)
unemp_data$unemp_year2 <- unemp_data$unemp_year + 2
Scorecard <- Scorecard %>%
  inexact_left_join(unemp_data,
    method = "between",
    var = year,
    jvar = c(unemp_year, unemp_year2)
  )
```

---

`is_pibble` *Check whether an object has been declared as panel data*

---

## Description

Checks whether a data set (`data.frame` or `tibble`) has been assigned panel identifiers in the `pmdplyr` format. If so, returns those identifiers.

## Usage

```
is_pibble(.df, .silent = FALSE)
```

## Arguments

| | |
|---|---|
| `.df` | Data frame or tibble |
| `.silent` | Set to TRUE to suppress output reporting what the panel identifiers are. Defaults to FALSE |

## Examples

```
data(Scorecard)
Scorecard <- as_pibble(Scorecard, .i = "unitid", .t = "year")
is_pibble(Scorecard)
```

---

`join.tbl_pb` *Join two pibbles together*

---

## Description

These are generic functions that dispatch to individual pibble methods. pibble structure from `x` will be maintained. pibble structure from `y` will be lost. See join for complete documentation.

## Usage

```
## S3 method for class 'tbl_pb'
left_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

## S3 method for class 'tbl_pb'
inner_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

## S3 method for class 'tbl_pb'
right_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

## S3 method for class 'tbl_pb'
full_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

## S3 method for class 'tbl_pb'
semi_join(x, y, by = NULL, copy = FALSE, ...)

## S3 method for class 'tbl_pb'
nest_join(x, y, by = NULL, copy = FALSE, keep = FALSE, name = NULL, ...)

## S3 method for class 'tbl_pb'
anti_join(x, y, by = NULL, copy = FALSE, ...)
```

## Arguments

| | |
|---|---|
| `x` | A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details. |
| `y` | A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details. |
| `by` | A character vector of variables to join by.<br>If `NULL`, the default, *_join() will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. |

| | |
|---|---|
| | To join by different variables on x and y, use a named vector. For example, by = c("a" = "b") will match x$a to y$b.<br>To join by multiple variables, use a vector with length > 1. For example, by = c("a","b") will match x$a to y$a and x$b to y$b. Use a named vector to match different variables in x and y. For example, by = c("a" = "b","c" = "d") will match x$a to y$b and x$c to y$d.<br>To perform a cross-join, generating all combinations of x and y, use by = character(). |
| copy | If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it. |
| suffix | If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2. |
| ... | Other parameters passed onto methods. |
| keep | Should the join keys from both x and y be preserved in the output? Only applies to nest_join(), left_join(), right_join(), and full_join(). |
| name | The name of the list column nesting joins create. If NULL the name of y is used. |

---

mode_order *Calculate the mode, and use original order to break ties*

---

## Description

mode_order() calculates the mode of a vector, mostly used as the default .resolve option in fixed_force()

## Usage

```
mode_order(x)
```

## Arguments

| | |
|---|---|
| x | Vector to calculate the mode of. |

## Details

In the case of ties, the first-ordered value in the vector wins.

## Examples

```
x <- c(1, 2, 2, NA, 5, 3, 4)
mode_order(x)

# Ties are broken by order
x <- c(2, 2, 1, 1)
mode_order(x)
```

mutate_cascade *Perform mutate one time period at a time ('Cascading mutate')*

## Description

This function is a wrapper for `dplyr::mutate()` which performs `mutate` one time period at a time, allowing each period's calculation to complete before moving on to the next. This allows changes in one period to 'cascade down' to later periods. This is (number of time periods) slower than regular `mutate()` and, generally, is only used for mutations where an existing variable is being defined in terms of its own `lag()` or `tlag()`. This is similar in concept to (and also slower than) `cumsum` but is much more flexible, and works with data that has multiple observations per individual-period using `tlag()`. For example, this could be used to calculate the current value of a savings account given a variable with each period's deposits, withdrawals, and interest, or could calculate the cumulative number of credits a student has taken across all classes.

## Usage

```
mutate_cascade(
  .df,
  ...,
  .skip = TRUE,
  .backwards = FALSE,
  .group_i = TRUE,
  .i = NULL,
  .t = NULL,
  .d = NA,
  .uniqcheck = FALSE,
  .setpanel = TRUE
)
```

## Arguments

| | |
|---|---|
| `.df` | Data frame or tibble. |
| `...` | Specification to be passed to `mutate()`. |
| `.skip` | Set to `TRUE` to skip the first period present in the data (or present within each group for grouped data) when applying `mutate()`. Since most uses of `mutate_cascade()` will involve a `lag()` or `tlag()`, this avoids creating an `NA` in the first period that then cascades down. By default this is TRUE. If you set this to FALSE you should probably have some method for avoiding a first-period `NA` in your `...` entry, perhaps using the `default` option in `dplyr::lag` or the `.default` option in `tlag`. |
| `.backwards` | Set to `TRUE` to run `mutate_cascade()` from the last period to the first, rather than from the first to the last. |
| `.group_i` | By default, if `.i` is specified or found in the data, `mutate_cascade` will group the data by `.i`, ignoring any grouping already implemented (although the original grouping structure will be returned at the end). Set `.group_i = FALSE` to avoid this. |

| | |
|---|---|
| .i | Quoted or unquoted variables that identify the individual cases. Note that setting any one of .i, .t, or .d will override all three already applied to the data, and will return data that is as_pibble()d with all three, unless .setpanel=FALSE. |
| .t | Quoted or unquoted variables indicating the time. pmdplyr accepts two kinds of time variables: numeric variables where a fixed distance .d will take you from one observation to the next, or, if .d=0, any standard variable type with an order. Consider using the time_variable() function to create the necessary variable if your data uses a Date variable for time. |
| .d | Number indicating the gap in .t between one period and the next. For example, if .t indicates a single day but data is collected once a week, you might set .d=7. To ignore gap length and assume that "one period ago" is always the most recent prior observation in the data, set .d=0. The default .d = NA here will become .d = 1 if either .i or .t are declared. |
| .uniqcheck | Logical parameter. Set to TRUE to always check whether .i and .t uniquely identify observations in the data. By default this is set to FALSE and the check is only performed once per session, and only if at least one of .i, .t, or .d is set. |
| .setpanel | Logical parameter. TRUE by default, and so if .i, .t, and/or .d are declared, will return a pibble set in that way. |

## Details

To apply mutate_cascade() to non-panel data and without any grouping (perhaps to mimic standard Stata replace functionality), add a variable to your data indicating the order you'd like mutate performed in (perhaps using dplyr::row_number()) and .t to that new variable.

## Examples

```
if(interactive()){
data(Scorecard)
# I'd like to build a decaying function that remembers previous earnings but at a declining rate
# Let's only use nonmissing earnings
# And let's say we're only interested in four-year colleges in Colorado
# (mutate_cascade + tlag can be very slow so we're working with a smaller sample)
Scorecard <- Scorecard %>%
  dplyr::filter(
    !is.na(earnings_med),
    pred_degree_awarded_ipeds == 3,
    state_abbr == "CO"
  ) %>%
  # And declare the panel structure
  as_pibble(.i = unitid, .t = year)
Scorecard <- Scorecard %>%
  # Almost all instances involve a variable being set to a function of a lag of itself
  # we don't want to overwrite so let's make another
  # Note that earnings_med is an integer -
  # but we're about to make non-integer decay function, so call it a double!
  dplyr::mutate(decay_earnings = as.double(earnings_med)) %>%
  # Now we can cascade
```

```
  mutate_cascade(
    decay_earnings = decay_earnings +
      .5 * tlag(decay_earnings, .quick = TRUE)
  )
  }
```

---

`mutate_subset` *Propagate a calculation performed on a subset of data to the rest of the data*

---

## Description

This function performs `dplyr::summarize` on a `.filtered` subset of data. Then it applies the result to all observations (or all observations in the group, if applied to grouped data), filling in columns of the data with the summarize results, as though `dplyr::mutate` had been run.

## Usage

```
mutate_subset(
  .df,
  ...,
  .filter,
  .group_i = TRUE,
  .i = NULL,
  .t = NULL,
  .d = NA,
  .uniqcheck = FALSE,
  .setpanel = TRUE
)
```

## Arguments

| | |
|---|---|
| `.df` | Data frame or tibble. |
| `...` | Specification to be passed to `dplyr::summarize()`. |
| `.filter` | Unquoted logical condition for which observations `dplyr::summarize()` operations are to be run on. |
| `.group_i` | By default, if `.i` is specified or found in the data, `mutate_cascade` will group the data by `.i`, overwriting any grouping already implemented. Set `.group_i = FALSE` to avoid this. |
| `.i` | Quoted or unquoted variables that identify the individual cases. Note that setting any one of `.i`, `.t`, or `.d` will override all three already applied to the data, and will return data that is `as_pibble()`d with all three, unless `.setpanel=FALSE`. |
| `.t` | Quoted or unquoted variable indicating the time. `pmdplyr` accepts two kinds of time variables: numeric variables where a fixed distance `.d` will take you from one observation to the next, or, if `.d=0`, any standard variable type with an order. Consider using the `time_variable()` function to create the necessary variable if your data uses a `Date` variable for time. |

| | |
|---|---|
| `.d` | Number indicating the gap in `.t` between one period and the next. For example, if `.t` indicates a single day but data is collected once a week, you might set `.d=7`. To ignore gap length and assume that "one period ago" is always the most recent prior observation in the data, set `.d=0`. The default `.d = NA` here will become `.d = 1` if either `.i` or `.t` are declared. |
| `.uniqcheck` | Logical parameter. Set to TRUE to always check whether `.i` and `.t` uniquely identify observations in the data. By default this is set to FALSE and the check is only performed once per session, and only if at least one of `.i`, `.t`, or `.d` is set. |
| `.setpanel` | Logical parameter. `TRUE` by default, and so if `.i`, `.t`, and/or `.d` are declared, will return a `pibble` set in that way. |

## Details

One application of this is to partially widen data. For example, if your analysis uses childhood height as a control variable in all years, `mutate_subset()` could be used to easily generate a `height_age10` variable from a `height` variable.

## Examples

```
data(SPrail)
# In preparation for fitting a choice model for how people choose ticket type,
# I'd like to know the price of a "Promo" ticket for a given route
# So that I can compare each other type of ticket price to that type
SPrail <- SPrail %>%
  mutate_subset(
    promo_price = mean(price, na.rm = TRUE),
    .filter = fare == "Promo",
    .i = c(origin, destination)
  )
```

---

`panel_calculations` *Perform standard panel-data calculations*

---

## Description

These functions perform the standard between and within transformations on panel data.

## Usage

```
within_i(
  .var,
  .df = get(".", envir = parent.frame()),
  .fcn = function(x) mean(x, na.rm = TRUE),
  .i = NULL,
  .t = NULL,
  .uniqcheck = FALSE
```

```
)

between_i(
  .var,
  .df = get(".", envir = parent.frame()),
  .fcn = function(x) mean(x, na.rm = TRUE),
  .i = NULL,
  .t = NULL,
  .uniqcheck = FALSE
)
```

## Arguments

| | |
|---|---|
| .var | Vector to be transformed |
| .df | Data frame, pibble, or tibble (usually the data frame or tibble that contains .var) which contains the panel structure variables either listed in .i and .t, or earlier declared with as_pibble(). If tlag is called inside of a dplyr verb, this can be omitted and the data will be picked up automatically. |
| .fcn | The function to be passed to dplyr::summarize(). x -.fcn(x) within .i is the within tranformation. .fcn(x) within .i minus .fcn overall is the between transformation. This will almost always be the default .fcn = function(x) mean(x,na.rm=TRUE). |
| .i | Quoted or unquoted variable(s) that identify the individual cases. Note that setting any one of .i, .t, or .d will override all three already applied to the data, and will return data that is as_pibble()d with all three, unless .setpanel=FALSE. |
| .t | Quoted or unquoted variable with the single variable name indicating the time. pmdplyr accepts two kinds of time variables: numeric variables where a fixed distance .d will take you from one observation to the next, or, if .d=0, any standard variable type with an order. Consider using the time_variable() function to create the necessary variable if your data uses a Date variable for time. |
| .uniqcheck | Logical parameter. Set to TRUE to always check whether .i and .t uniquely identify observations in the data. By default this is set to FALSE and the check is only performed once per session, and only if at least one of .i, .t, or .d is set. |

## Details

These functions do not take a .d argument because it is irrelevant here.

## Examples

```
data(SPrail)
# Calculate within- and between-route variation in price and add it to the data
SPrail <- SPrail %>%
  dplyr::mutate(
    within_route = within_i(price, .i = c(origin, destination)),
```

```
  between_route = between_i(price, .i = c(origin, destination))
)
```

---

panel_convert *Convert between panel data types*

---

## Description

This function takes panel data objects declared using pmdplyr (pibble/tbl_pb), tsibble (tsibble/tbl_ts), plm (pdata.frame), and panelr (panel_data) and converts to one of the other three formats for use with functions in those packages.

## Usage

```
panel_convert(data, to, ...)
```

## Arguments

| | |
|---|---|
| data | Data frame - a pibble, tsibble, pdata.frame, or panel_data object. |
| to | Character variable set to "pmdplyr","pibble","tbl_pb","tsibble","tbl_ts","plm","pdata.fram or "panel_data" indicating the type/package to be converted to. |
| ... | Additional arguments to be sent to, respectively, as_pibble(), tsibble::as_tsibble(), plm::pdata.frame(), or panelr::panel_data(). |

## Details

Any grouping will be lost. You must have the relevant package installed to convert to the type for that package. Conversions from pdata.frame will be improved if sjlabelled is also installed.

When using panel_convert, be aware of the requirements that each type has:

| Feature/Requirement | pibble | tsibble | pdata.frame | panel_data |
|---|---|---|---|---|
| ID | .i | key | index[1] | id |
| Time | .t | index | index[2] | wave |
| Gap control | .d | regular | No | No |
| ID must exist | No | No | Yes | Yes |
| Time must exist | No | Yes | Yes | Yes[1] |
| Only one ID variable[2] | No | No | Yes | Yes |
| Unique identification | No | Yes | No[3] | No[3] |

[1] pdata.frame does not require that time be provided, but if not provided will create it based on original ordering of the data. The pdata.frame option to set index equal to an integer for a balanced panel and have it figure out the rest by itself is not supported.

[2] Use pmdplyr::id_variable() to generate a single ID variable from multiple if one is required.

[3] pdata.frame and panel_data do not require that ID and time uniquely identify the observations on declaring the data, but functions in these packages may not work correctly without unique

identification.

In addition to the above, be aware that the different packages have different requirements on which variable classes can be Time variables. `pmdplyr::time_variable()` can build an integer variable that will work in all packages.

You may run into some trouble if your data contains variables by the names `panel_convert_id`, `panel_convert_time`, `pibble_d`, or `panel_convert_regular`.

## Examples

```
# Only run examples if the relevant packages are installed
pkgs <- utils::installed.packages()

data(Scorecard)

# The example will turn a pibble to everything else
# But starting with another type will of course work!
S_pibble <- as_pibble(Scorecard, .i = unitid, .t = year)

# Get a tsibble
if ("tsibble" %in% pkgs) {
  head(panel_convert(S_pibble, to = "tsibble"))
}

# Now for pdata.frame
if ("plm" %in% pkgs) {
  head(panel_convert(S_pibble, to = "plm"))
}

# And finally panel_data
if ("panelr" %in% pkgs) {
  head(panel_convert(S_pibble, to = "panelr"))
}
```

---

`panel_fill` *Fill in gaps in panel data*

---

## Description

This function creates new observations to fill in any gaps in panel data. For example, if individual 1 has an observation in periods t = 1 and t = 3 but no others, this function will create an observation for t = 2. By default, the t = 2 observation will be identical to the t = 1 observation except for the time variable, but this can be adjusted. This function returns data sorted by `.i` and `.t`.

## Usage

```
panel_fill(
  .df,
```

```
  .set_NA = FALSE,
  .min = NA,
  .max = NA,
  .backwards = FALSE,
  .group_i = TRUE,
  .flag = NA,
  .i = NULL,
  .t = NULL,
  .d = 1,
  .uniqcheck = FALSE,
  .setpanel = TRUE
)
```

## Arguments

| | |
|---|---|
| .df | Tibble or data frame which either has the .t and .d (and perhaps .i) attributes included by as_pibble(), or the appropriate panel structure is declared in the function. |
| .set_NA | Should values in newly-created observations be set to adjacent values or to NA? Set to TRUE to set all new values to NA except for .i and .t. To make only specific variables NA, list them as a character vector. Defaults to FALSE; all values are filled in using the most recently available data. |
| .min | Sets the first time period in the data for each individual to be .min, and fills in gaps between period .min and the actual start of the data. Copies data from the first period present in the data for each individual (if grouped). Handy for creating balanced panels. |
| .max | Sets the last time period in the data for each individual to be .max, and fills in gaps between period .max and the actual start of the data. Copies data from the flast period present in the data for each individual (if grouped). Handy for creating balanced panels. |
| .backwards | By default, values of newly-created observations are copied from the most recently available period. Set .backwards = TRUE to instead copy values from the closest *following* period. |
| .group_i | By default, panel_fill() will fill in gaps within values of .i. If .i is missing, it won't do that. If .i is in the data and you still don't want panel_fill() to run within .i, set .group_i = FALSE. |
| .flag | The name of a new variable indicating which observations are newly created by panel_fill(). |
| .i | Quoted or unquoted variables that identify the individual cases. Note that setting any one of .i, .t, or .d will override all three already applied to the data, and will return data that is as_pibble()d with all three, unless .setpanel=FALSE. |
| .t | Quoted or unquoted variable indicating the time. pmdplyr accepts two kinds of time variables: numeric variables where a fixed distance .d will take you from one observation to the next, or, if .d=0, any standard variable type with an order. Consider using the time_variable() function to create the necessary variable if your data uses a Date variable for time. |

| | |
|---|---|
| .d | Number indicating the gap in .t between one period and the next. For example, if .t indicates a single day but data is collected once a week, you might set .d=7. To ignore gap length and assume that "one period ago" is always the most recent prior observation in the data, set .d=0. By default, .d=1. |
| .uniqcheck | Logical parameter. Set to TRUE to always check whether .i and .t uniquely identify observations in the data. By default this is set to FALSE and the check is only performed once per session, and only if at least one of .i, .t, or .d is set. |
| .setpanel | Logical parameter. TRUE by default, and so if .i, .t, and/or .d are declared, will return a pibble set in that way. |

## Details

Note that, in the case where there is more than one observation for a given individual/time period (or just time period if .group_i = FALSE), panel_fill() will create copies of *every observation* in the appropriate individual/time period for filling-in purposes. So if there are four t = 1 observations and nothing in t = 2, panel_fill() will create four new observations with t = 2, copying the original four in t = 1.

By default, the panel_fill() operation is grouped by .i, although it will return the data in the original grouping structure. Leave .i blank, or, if .i is already in the data from as_pibble, set .group_i=FALSE to run the function ungrouped, or with the existing group structure.

This function requires .t and .d to be declared in the function or already established in the data by as_pibble(). Also, this requires a cardinal .t. It must not be the case that .d=0.

## Examples

```
# Examples are too slow to run - this function is slow!
if (interactive()) {
  data(Scorecard)
 # Notice that, in the Scorecard data, the gap between one year and the next is not always constant
  table((Scorecard %>% dplyr::arrange(year) %>%
    dplyr::group_by(unitid) %>%
    dplyr::mutate(diff = year - dplyr::lag(year)))$diff)
 # And also that not all universities show up for the first or last times in the same year
  year_range <- Scorecard %>%
    dplyr::group_by(unitid) %>%
    dplyr::summarize(first_year = min(year), last_year = max(year))
  table(year_range$first_year)
  table(year_range$last_year)
  rm(year_range)

  # We can deal with the inconsistent-gaps problem by creating new obs to fill in
 # this version will fill in the new obs with the most recently observed data, and flag them
  Scorecard_filled <- panel_fill(Scorecard,
    .i = unitid,
    .t = year,
    .flag = "new"
  )
```

```
  # Or maybe we want those observations in there but don't want to treat them as real data
  # so instead of filling them in, just leave all the data in the new obs blank
  # (note this sets EVERYTHING not in .i or .t to NA - if you only want some variables NA,
  # make .set_NA a character vector of those variable names)
  Scorecard_filled <- panel_fill(Scorecard,
    .i = unitid,
    .t = year,
    .flag = "new",
    .set_NA = TRUE
  )

  # Perhaps we want a perfectly balanced panel. So let's set .max and .min to the start and end
  # of the data, and it will fill in everything.
  Scorecard_filled <- panel_fill(Scorecard,
    .i = unitid, .t = year, .flag = "new",
    .min = min(Scorecard$year), .max = max(Scorecard$year)
  )
  # how many obs of each college? Should be identical, and equal to the number of years there are
  table(table(Scorecard_filled$unitid))
  length(unique(Scorecard_filled$year))
}
```

---

panel_locf *Fill in missing (or other) values of a panel data set using known data*

---

## Description

This function looks for a list of values (usually, just NA) in a variable .var and overwrites those values with the most recent (or next-coming) values that are not from that list ("last observation carried forward").

## Usage

```
panel_locf(
  .var,
  .df = get(".", envir = parent.frame()),
  .fill = NA,
  .backwards = FALSE,
  .resolve = "error",
  .group_i = TRUE,
  .i = NULL,
  .t = NULL,
  .d = 1,
  .uniqcheck = FALSE
)
```

## Arguments

| | |
|---|---|
| .var | Vector to be modified. |
| .df | Data frame, pibble, or tibble (usually the one containing .var) that contains the panel structure variables either listed in .i and .t, or earlier declared with as_pibble(). If tlag is called inside of a dplyr verb, this can be omitted and the data will be picked up automatically. |
| .fill | Vector of values to be overwritten. Just NA by default. |
| .backwards | By default, values of newly-created observations are copied from the most recently available period. Set .backwards = TRUE to instead copy values from the closest *following* period. |
| .resolve | If there is more than one observation per individual/period, and the value of .var is identical for all of them, that's no problem. But what should panel_locf() do if they're not identical? Set .resolve = 'error' (or, really, any string) to throw an error in this circumstance. Or, set .resolve to a function that can be used within dplyr::summarize() to select a single value per individual/period. For example, .resolve = function(x) mean(x) to get the mean value of all observations present for that individual/period. .resolve will also be used to fill in values if some values in a given individual/period are to be overwritten and others aren't. Using a function will be quicker than .resolve = 'error', so if you're certain there's no issue, you can speed up execution by setting, say, .resolve = dplyr::first. |
| .group_i | By default, if .i is specified or found in the data, panel_locf() will group the data by .i, ignoring any grouping already implemented. Set .group_i = FALSE to avoid this. |
| .i | Quoted or unquoted variables that identify the individual cases. Note that setting any one of .i, .t, or .d will override all three already applied to the data, and will return data that is as_pibble()d with all three, unless .setpanel=FALSE. |
| .t | Quoted or unquoted variable indicating the time. pmdplyr accepts two kinds of time variables: numeric variables where a fixed distance .d will take you from one observation to the next, or, if .d=0, any standard variable type with an order. Consider using the time_variable() function to create the necessary variable if your data uses a Date variable for time. |
| .d | Number indicating the gap in .t between one period and the next. For example, if .t indicates a single day but data is collected once a week, you might set .d=7. To ignore gap length and assume that "one period ago" is always the most recent prior observation in the data, set .d=0. By default, .d=1. |
| .uniqcheck | Logical parameter. Set to TRUE to always check whether .i and .t uniquely identify observations in the data. By default this is set to FALSE and the check is only performed once per session, and only if at least one of .i, .t, or .d is set. |

## Details

panel_locf() is unusual among last-observation-carried-forward functions (like zoo::na.locf()) in that it is usable even if observations are not uniquely identified by .t (and .i, if defined).

## Examples

```
# The SPrail data has some missing price values.
# Let's fill them in!
# Note .d=0 tells it to ignore how big the gaps are
# between one period and the next, just look for the most recent insert_date
# .resolve tells it what value to pick if there are multiple
# observed prices for that route/insert_date
# (.resolve is not necessary if .i and .t uniquely identify obs,
# or if .var is either NA or constant within them)
# Also note - this will fill in using CURRENT-period
# data first (if available) before looking for lagged data.
data(SPrail)
sum(is.na(SPrail$price))
SPrail <- SPrail %>%
  dplyr::mutate(price = panel_locf(price,
    .i = c(origin, destination), .t = insert_date, .d = 0,
    .resolve = function(x) mean(x, na.rm = TRUE)
  ))

# The spec is a little easier with data like Scorecard where
# .i and .t uniquely identify observations
# so .resolve isn't needed.
data(Scorecard)
sum(is.na(Scorecard$earnings_med))
Scorecard <- Scorecard %>%
  # Let's speed this up by just doing four-year colleges in Colorado
  dplyr::filter(
    pred_degree_awarded_ipeds == 3,
    state_abbr == "CO"
  ) %>%
  # Now let's fill in NAs and also in case there are any erroneous 0s
  dplyr::mutate(earnings_med = panel_locf(earnings_med,
    .fill = c(NA, 0),
    .i = unitid, .t = year
  ))
# Note that there are still some missings - these are missings that come before the first
# non-missing value in that unitid, so there's nothing to pull from.
sum(is.na(Scorecard$earnings_med))
```

---

`pibble` *Create a pibble panel data set object*

---

## Description

This function declares a pibble tibble with the attributes `.i`, `.t`, and `.d`.

## Usage

```
pibble(..., .i = NULL, .t = NULL, .d = 1, .uniqcheck = FALSE)
```

## Arguments

| | |
|---|---|
| ... | A set of name-value pairs to make up the variables of a pibble. |
| .i | Quoted or unquoted variable(s) that identify the individual cases. If this is omitted, pibble will assume the data set is a single time series. |
| .t | Quoted or unquoted variable indicating the time. pmdplyr accepts two kinds of time variables: numeric variables where a fixed distance .d will take you from one observation to the next, or, if .d=0, any standard variable type with an order. Consider using the time_variable() function to create the necessary variable if your data uses a Date variable for time. |
| .d | Number indicating the gap in t between one period and the next. For example, if .t indicates a single day but data is collected once a week, you might set .d=7. To ignore gap length and assume that "one period ago" is always the most recent prior observation in the data, set .d=0. By default, .d=1. |
| .uniqcheck | Logical parameter. Set to TRUE to perform a check of whether .i and .t uniquely identify observations, and present a message if not. By default this is set to FALSE and the warning message occurs only once per session. |

## Details

- .i, Quoted or unquoted variable(s) indicating the individual-level panel identifier
- .t, Quoted or unquoted variable indicating the time variable
- .d, a number indicating the gap

The pibble() function is for the purpose of creating pibble objects from scratch. You probably want as_pibble.

Note that pibble does not require that .i and .t uniquely identify the observations in your data, but it will give a warning message (a maximum of once per session, unless .uniqcheck=TRUE) if they do not.

## Examples

```
# Creating a pibble from scratch
pd <- pibble(
  i = c(1, 1, 1, 2, 2, 2),
  t = c(1, 2, 3, 1, 2, 2),
  x = rnorm(6),
  .i = i,
  .t = t
)
is_pibble(pd)
# I set .d=0 here to indicate that I don't care how large the gap between one period and the next is
# If I want to use 'seconds' for t.
# See time_variable() to turn unruly variables into well-behaved integers, as well
pd2 <- pibble(
  i = c(1, 1, 1, 2, 2, 2),
  seconds = c(123, 456, 789, 103, 234, 238),
  .i = i,
  .t = seconds,
```

```
  .d = 0
)
is_pibble(pd2)
```

---

`pibble_methods` *pibble methods*

---

## Description

These are variants of existing functions that are designed to retain the `pibble` status of the object, as well as its `.i`, `.t`, and `.d` attributes.

## Usage

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

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

## S3 method for class 'tbl_pb'
group_by(.data, ...)

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

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

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

## S3 method for class 'tbl_pb'
summarize(.data, ...)

## S3 method for class 'tbl_pb'
summarise(.data, ...)

## S3 method for class 'tbl_pb'
transmute(.data, ...)
```

## Arguments

| | |
|---|---|
| `.data, x` | These functions take a `tbl_pb` (i.e. `pibble`) object as input |
| `.keep_all, ...` | Other parameters to be passed to the relevant functions |

## Details

Some functions that already preserve pibble status and so don't need special methods include:

dplyr::add_row(),tibble:add_column(),dplyr::arrange(),dplyr::bind_cols(),dplyr::filter(),dplyr::sam

as well as all scoped variants (_all, _if, _at) of dplyr functions.

dplyr::bind_rows() is currently not supported. If you use dplyr::bind_rows() you should pipe it to as_pibble().

Any function that takes two data frames/tibbles as inputs will retain the panel structure of the *first* argument.

If a function is not on the above list or elsewhere in this help file, then you may need to re-as_pibble your object after using the function.

---

pmdplyr pmdplyr *package*

---

## Description

Suite of tools extending the dplyr package to perform data manipulation. These tools are geared towards use in panel data and hierarchical data.

## Details

Unlike other suites dealing with panel data, all functions in pmdplyr are designed to work even when considering a set of variables that do not uniquely identify rows. This is handy when working with any kind of hierarchical data, or panel data where there are multiple observations per individual per time period, like student/term/class education data.

pmdplyr contains the following functions:

- between_i and within_i Standard between and within panel calculations.
- fixed_check Checks a list of variables for consistency within a panel structure.
- fixed_force Forces a list of variables to be constant within a panel structure.
- id_variable Takes a list of variables that make up an individual identifier and turns it into a single variable.
- time_variable Takes a time variable, or set of time variables, and turns them into a single well-behaved integer time variable of the kind required by most panel functions.
- inexact_join Wrapper for the dplyr join functions which allows for a variable to be matched inexactly, for example joining a time variable in x to the most recent previous value in y.
- safe_join Set of wrappers for the dplyr::join and pmdplyr::inexact_join functions which checks before merging whether each data set is uniquely identified as expected.
- pibble, as_pibble, and is_pibble Set the panel structure for a data set, or check if it is already set.

- panel_convert Converts between the panel data types pmdplyr::pibble, tsibble::tsibble, plm::pdata.frame, and panelr::panel_data.
- mutate_cascade A wrapper for dplyr mutate which runs one period at a time, allowing changes in one period to finalize before the next period is calculated.
- mutate_subset A wrapper for dplyr mutate that performs a calculation on a subset of data, and then applies the result to all the observations (within group).
- panel_fill Fills in gaps in the panel. Can also fill in at the beginning or end of the data to create a perfectly balanced panel.
- panel_locf A last-observation-carried-forward function for panels. Fills in NAs with recent nonmissing observations.
- tlag Lags a variable in time.

---

`safe_join` *Join two data frames safely*

---

## Description

This function is a wrapper for the standard dplyr join functions and the pmdplyr inexact_join functions.

## Usage

```
safe_join(x, y, expect = NULL, join = NULL, ...)
```

## Arguments

| | |
|---|---|
| x, y | The left and right data sets to join. |
| expect | Either "1:m" (or "x"), "m:1" (or "y"), or "1:1" (or c("x","y") or "xy") - the match you expect to perform. You can specify this as the kind of match you expect to be performing (one-to-many, many-to-one, or one-to-one), or as the data set(s) you expect to be uniquely identified by the joining variables ("x", "y", or c("x","y")/"xy"). Alternately, set to expect = "no m:m" if you don't care what join you're doing as long as it isn't many-to-many. |
| join | A join or inexact_join function to run if safe_join determines your join is safe. By default, simply returns TRUE instead of running the join. |
| ... | Other arguments to be passed to the function specified in join. If performing an inexact_join, put the var and jvar arguments in as quoted variables. |

## Details

When performing a join, we generally expect that one or both of the joined data sets is uniquely identified by the set of joining variables.

If this is not true, the results of the join will often not be what you expect. Unfortunately, join does not warn you that you may have just done something strange.

This issue is especially likely to arise with panel data, where you may have multiple different data sets at different observation levels.

`safe_join` forces you to specify which of your data sets you think are uniquely identified by the joining variables. If you are wrong, it will return an error. If you are right, it will pass you on to your preferred `join` function, given in `join`. If `join` is not specified, it will just return `TRUE`.

## Examples

```
# left is panel data and i does not uniquely identify observations
left <- data.frame(
  i = c(1, 1, 2, 2),
  t = c(1, 2, 1, 2),
  a = 1:4
)
# right is individual-level data uniquely identified by i
right <- data.frame(
  i = c(1, 2),
  b = 1:2
)

# I think that I can do a one-to-one merge on i
# Forgetting that left is identified by i and t together
# So, this produces an error
## Not run:
safe_join(left, right, expect = "1:1", join = left_join)

## End(Not run)

# If I realize I'm doing a many-to-one merge, that is correct,
# so safe_join will perform it for us
safe_join(left, right, expect = "m:1", join = left_join)
```

---

# Scorecard *Earnings and Loan Repayment in US Four-Year Colleges*

---

## Description

From the College Scorecard, this data set contains by-college-by-year data on how students who attended those colleges are doing.

## Usage

```
Scorecard
```

## Format

A data frame with 48,445 rows and 8 variables:

**unitid** College identifiers.

**inst_name** Name of the college or university.

**state_abbr** Two-letter abbreviation for the state the college is in.

**pred_degree_awarded_ipeds** Predominant degree awarded. 1 = less-than-two-year, 2 = two-year, 3 = four-year+

**year** Year in which outcomes are measured.

**earnings_med** Median earnings among students (a) who received federal financial aid, (b) who began as undergraduates at the institution ten years prior, (c) with positive yearly earnings.

**count_not_working** Number of students who are (a) not working (not necessarily unemployed), (b) received federal financial aid, and (c) who began as undergraduates at the institution ten years prior.

**count_working** Number of students who are (a) working, (b) who received federal financial aid, and (c) who began as undergraduates at the institution ten years prior.

**repay_rate** Proportion of students who (a) received federal loans as an undergraduate at this institution, (b) entered repayment seven years ago, (c) are not in default, (d) have paid off all accrued interest, and (e) are still making progress on payment. Only available 2013-2016.

## Details

This data is not just limited to four-year colleges and includes a very wide variety of institutions.

Note that the labor market (earnings, working) and repayment rate data do not refer to the same cohort of students, but rather are matched on the year in which outcomes are recorded. Labor market data refers to cohorts beginning college as undergraduates ten years prior, repayment rate data refers to cohorts entering repayment seven years prior.

Data was downloaded using the Urban Institute's `educationdata` package.

## Source

Education Data Portal (Version 0.4.0 - Beta), Urban Institute, Center on Education Data and Policy, accessed June 28, 2019. https://educationdata.urban.org/documentation/, Scorecard.

---

`setops` *Set operations*

---

## Description

These functions overwrite the set functions provided in base to make them generic to be used to join pibbles. See setops for details.

## Usage

```
## S3 method for class 'tbl_pb'
intersect(x, y, ...)

## S3 method for class 'tbl_pb'
union(x, y, ...)
```

```
## S3 method for class 'tbl_pb'
union_all(x, y, ...)

## S3 method for class 'tbl_pb'
setdiff(x, y, ...)
```

## Arguments

| | |
|---|---|
| x | objects to perform set function on (ignoring order) |
| y | objects to perform set function on (ignoring order) |
| ... | other arguments passed on to methods |

---

`SPrail` *2,000 Spanish train trips*

---

## Description

This data set is a random subsample of a much larger database of trips taken on the Spanish High Speed Train Service (Renfe AVE).

## Usage

```
SPrail
```

## Format

A data frame with 2,000 rows and 9 variables:

**insert_date** Date and time when ticket was paid for.

**origin** Origin City

**destination** Destination City

**start_date** Date and time for train departure.

**end_date** Date and time for train arrival.

**train_type** Train service name.

**price** Price of ticket in Euros.

**train_class** Class of ticket: tourist, business, etc.. Variable in Spanish.

**fare** Type of ticket fare.

## Details

All dates and times are European Central Time.

The larger data set from which `SPrail` was sampled was compiled and released under GPL-2 public license by Pedro Muñoz and David Cañones.

## Source

https://www.kaggle.com/thegurusteam/spanish-high-speed-rail-system-ticket-pricing

---

`time_variable` *Create a single integer time period index variable*

---

## Description

This function takes either multiple time variables, or a single Date-class variable, and creates a single integer time variable easily usable with functions in `pmdplyr` and other packages like `plm` and `panelr`.

## Usage

```
time_variable(
  ...,
  .method = "present",
  .datepos = NA,
  .start = 1,
  .skip = NA,
  .breaks = NA,
  .turnover = NA,
  .turnover_start = NA
)
```

## Arguments

| | |
|---|---|
| `...` | variables (vectors) to be used to generate the time variable, in order of increasing specificity. So if you have a variable each for year, month, and day (with the names year, month, and day), you would use `year,month,day` (if a data set containing those variables has been attached using `with` or `dplyr`) or `data$year,data$month,data$day` (if not). |
| `.method` | The approach that will be taken to create your variable. See below for the options. By default, this is `.method = "present"`. |
| `.datepos` | A numeric vector containing the character/digit positions, in order, of the YY or YYYY year (or year/month in YYMM or YYYYMM format, or year/month/day in YYMMDD or YYYYMMDD) for the `.method="year"`, `.method="month"`, or `.method="day"` options, respectively. Give it only the data it needs - if you give `.method="year"` YYMM information, it will assume you're giving it YYYY and mess up. For example, if dates are stored as a character variable in the format '2013-07-21' and you want the year and month, you might specify `.datepos=c(1:4,6:7)`. If two-digit year is given, `.datepos` uses the `lubridate` package to determine century. |
| `.start` | A numeric variable indicating the day of the week/month that begins a new week/month, if `.method="week"` or `.method="month"` is used. By default, 1, where for `.method=week` 1 is Monday, 7 Sunday. If used with `.method="month"`, the time data should include day as well. |

`.skip` A numeric vector containing the values of year, month, or day-of-week (where Monday = 1, Sunday = 7, no matter what value `.start` takes) you'd like to skip over (for `.method="year","month","week","day"`, respectively). For example, with `.method="month"` and `.skip=12`, an observation in January would be determined to come one period after November. Commonly this might be `.skip=c(6,7)` with `.method="day"` to skip weekends so that Monday immediately follows Friday. If `.breaks` is also specified, select the values of `.breaks` you would like to skip, but do be aware that combining `.skip` and `.breaks` can be tricky.

`.breaks` A numeric vector containing the starting breakpoints of year or month you'd like to clump together (for `.method="year','month"`, respectively). Commonly, this might be `.breaks=c(1,4,7,10)` with `.method="month"` to go by quarter-year. The first element of `.breaks` should usually be 1.

`.turnover` A numeric vector the same length as the number of variables included indicating the maximum value that the corresponding variable in the list of variables takes, where NA indicates no maximum value, for use with `.method="turnover"` and required for that method. For example, if the variable list is `year,month` then you might have `.turnover=c(NA,12)`. Or if the variable list is `days-since-jan1-1970,hour,minute,s` you might have `.turnover=c(NA,23,59,59)`. Defaults to the maximum observed value of each variable if not specified, and NA for the first variable. Note that in almost all cases, the first element of `.turnover` should be `NA`, and all others should be non-NA.

`.turnover_start`
A numeric vector the same length as the number of variables included indicating the minimum value that the corresponding variable in the list of variables takes, where NA indicates no minimum value, for use with `method="turnover"`. For example, if the variable list is `year,month` then you might have `.turnover=c(NA,1)`. Or if the variable list is `days-since-jan1-1970,hour,minute,second` you might have `.turnover=c(NA,0,0,0)`. By default this is a vector of 1s the same length as the number of variables, except for the first element, which is NA. Note that in almost all cases, the first element of `.turnover_start` should be `NA`, and all others should be non-NA.

## Details

The `pmdplyr` library accepts only two kinds of time variables:

1. Ordinal time variables: Variables of any ordered type (`numeric`, `Date`, `character`) where the size of the gap between one value and the next does not matter. So if someone has two observations - one in period 3 and one in period 1, the period immediately before 3 is period 1, and two periods before 3 is missing. Set `.d=0` in your data to use this.

2. Cardinal time variables: Numeric variables with a fixed gap between one observation and the next, where the size of that gap is given by `.d`. So if `.d=1` and someone has two observations - one in period 3 and one in period 1, the period immediately before 3 is missing, and two periods before 3 is period 1.

If you would like to have a cardinal time variable but your data is not currently in that format, `time_variable()` will help you create a new variable that works with a setting of `.d=1`, the default.

If you have a date variable that is not in Date format (perhaps it's a string) and would like to use one of the Date-reliant methods below, I recommend converting it to Date using the convenient ymd(), mdy(), etc. functions from the lubridate package. If you only have partial date information (i.e. only year and month) and so converting to a Date doesn't work, see the .datepos option below.

Methods available include:

- .method="present" will assume that, even if each individual may have some missing periods, each period is present in your data *somewhere*, and so simply numbers, in order, all the time periods observed in the data.
- .method="year" can be used with a single Date/POSIX/etc.-type variable (anything that allows lubridate::date()) and will extract the year from it. Or, use it with a character or numeric variable and indicate with .datepos the character/digit positions that hold the year in YY or YYYY format. If combined with .breaks or .skip, will instead set the earliest year in the data to 1 rather than returning the actual year.
- .method="month" can be used with a single Date/POSIX/etc.-type variable (anything that allows lubridate::date()). It will give the earliest-observed month in the data set a value of 1, and will increment from there. Or, use it with a character or numeric variable and indicate with .datepos the character/digit positions that hold the year and month in YYMM or YYYYMM format (note that if your variable is in MMYYYY format, for example, you can just give a .datepos argument like c(3:6,1:2)). Months turn over on the .start day of the month, which is by default 1.
- .method="week" can be used with a single Date/POSIX/etc.-type variable (anything that allows lubridate::date()). It will give the earliest-observed week in the data set a value of 1, and will increment from there. Weeks turn over on the .start day, which is by default 1 (Monday). Note that this method always starts weeks on the same day of the week, which is different from standard lubridate procedure of counting sets of 7 days starting from January 1.
- .method="day" can be used with a single Date/POSIX/etc.-type variable (anything that allows lubridate::date()). It will give the earliest-observed day in the data set a value of 1, and increment from there. Or, use it with a character or numeric variable and indicate with .datepos the character/digit positions that hold the year and month in YYMMDD or YYYYMMDD format. To skip certain days of the week, such as weekends, use the .skip option.
- .method="turnover" can be used when you have more than one variable in variable and they are all numeric nonnegative integers. Set the .turnover option to indicate the highest value each variable takes before it starts over, and set .turnover_start to indicate what value it takes when it starts over. Cannot be combined with .skip or .breaks. Doesn't work with any variable for which the turnover values change, i.e. it doesn't play well with days-in-month - if you'd like to do something like year-month-day-hour, I recommend running .method="day" once with just the year-month-day variable, and then taking the result and combining *that* with hour in .method="turnover".

## Examples

```
data(SPrail)
```

```
# Since we have a date variable, we can easily create integers that increment for each
# year, or for each month, etc.
# Likely we'd only really need one of these four, depending on our purposes
SPrail <- SPrail %>%
  dplyr::mutate(
    year_time_id = time_variable(insert_date, .method = "year"),
    month_time_id = time_variable(insert_date, .method = "month"),
    week_time_id = time_variable(insert_date, .method = "week"),
    day_time_id = time_variable(insert_date, .method = "day")
  )

# Perhaps I'd like quarterly data
# (although in this case there are only two months, not much variation there)
SPrail <- SPrail %>%
  dplyr::mutate(quarter_time_id = time_variable(insert_date,
    .method = "month",
    .breaks = c(1, 4, 7, 10)
  ))
table(SPrail$month_time_id, SPrail$quarter_time_id)

# Maybe I'd like Monday to come immediately after Friday!
SPrail <- SPrail %>%
  dplyr::mutate(weekday_id = time_variable(insert_date,
    .method = "day",
    .skip = c(6, 7)
  ))

# Perhaps I'm interested in ANY time period in the data and just want to enumerate them in order
SPrail <- SPrail %>%
  dplyr::mutate(any_present_time_id = time_variable(insert_date,
    .method = "present"
  ))


# Maybe instead of being given a nice time variable, I was given it in string form
SPrail <- SPrail %>% dplyr::mutate(time_string = as.character(insert_date))
# As long as the character positions are consistent we can still use it
SPrail <- SPrail %>%
  dplyr::mutate(day_from_string_id = time_variable(time_string,
    .method = "day",
    .datepos = c(3, 4, 6, 7, 9, 10)
  ))
# Results are identical
cor(SPrail$day_time_id, SPrail$day_from_string_id)


# Or, maybe instead of being given a nice time variable, we have separate year and month variables
SPrail <- SPrail %>%
  dplyr::mutate(
    year = lubridate::year(insert_date),
    month = lubridate::month(insert_date)
  )
# We can use the turnover method to tell it that there are 12 months in a year,
```

```
# and get an integer year-month variable
SPrail <- SPrail %>%
  dplyr::mutate(month_from_two_vars_id = time_variable(year, month,
    .method = "turnover",
    .turnover = c(NA, 12)
  ))
# Results are identical
cor(SPrail$month_time_id, SPrail$month_from_two_vars_id)

# I could also use turnover to make the data hourly.
# Note that I'm using the day variable from earlier to avoid having
# to specify when day turns over (since that could be 28, 30, or 31)
SPrail <- SPrail %>%
  dplyr::mutate(hour_id = time_variable(day_time_id, lubridate::hour(insert_date),
    .method = "turnover",
    .turnover = c(NA, 23),
    .turnover_start = c(NA, 0)
  ))
# This could be easily extended to make the data by-minute, by-second, etc.
```

---

`tlag` *Time-lag a variable*

---

## Description

This function retrieves the time-lagged values of a variable, using the time variable defined in `.t` in the function or by `as_pibble()`. `tlag()` is highly unusual among time-lag functions in that it is usable even if observations are not uniquely identified by `.t` (and `.i`, if defined).

## Usage

```
tlag(
  .var,
  .df = get(".", envir = parent.frame()),
  .n = 1,
  .default = NA,
  .quick = FALSE,
  .resolve = "error",
  .group_i = TRUE,
  .i = NULL,
  .t = NULL,
  .d = NA,
  .uniqcheck = FALSE
)
```

## Arguments

| | |
|---|---|
| `.var` | Unquoted variable from `.df` to be lagged. |

| | |
|---|---|
| .df | Data frame, pibble, or tibble (usually the object that contains .var) that contains the panel structure variables either listed in .i and .t, or earlier declared with as_pibble(). If tlag is called inside of a dplyr verb, this can be omitted and the data will be picked up automatically. |
| .n | Number of periods to lag by. 1 by default. Note that this is automatically scaled by .d. If .d = 2 and .n = 1, then the lag of .t = 3 will be .t = 1. Allows negative values, equivalent to tlead() with the same value but positive. Note that .n is ignored if .d = 0. |
| .default | Fill-in value used when lagged observation is not present. Defaults to NA. |
| .quick | If .i and .t uniquely identify observations in your data, **and** there either .d = 0 or there are no time gaps for any individuals (perhaps use panel_fill() first), set .quick = TRUE to improve speed. tlag() will not check if either of these things are true (except unique identification, which will be checked if .uniqcheck = 1 or if .i or .t are specified in-function), so make sure they are or you will get strange results. |
| .resolve | If there is more than one observation per individal/period, and the value of .var is identical for all of them, that's no problem. But what should tlag() do if they're not identical? Set .resolve = 'error' (or, really, any string) to throw an error in this circumstance. Or, set .resolve to a function (ideally, a vectorized one) that can be used within dplyr::summarize() to select a single value per individual/period. For example, .resolve = mean to get the mean value of all observations present for that individual/period. |
| .group_i | By default, if .i is specified or found in the data, tlag() will group the data by .i, ignoring any grouping already implemented. Set .group_i = FALSE to avoid this. |
| .i | Quoted or unquotes variable(s) that identify the individual cases. Note that setting any one of .i, .t, or .d will override all three already applied to the data, and will return data that is as_pibble()d with all three, unless .setpanel=FALSE. |
| .t | Quoted or unquoted variable indicating the time. pmdplyr accepts two kinds of time variables: numeric variables where a fixed distance .d will take you from one observation to the next, or, if .d=0, any standard variable type with an order. Consider using the time_variable() function to create the necessary variable if your data uses a Date variable for time. |
| .d | Number indicating the gap in .t between one period and the next. For example, if .t indicates a single day but data is collected once a week, you might set .d=7. To ignore gap length and assume that "one period ago" is always the most recent prior observation in the data, set .d = 0. The default .d = NA here will become .d = 1 if either .i or .t are declared. |
| .uniqcheck | Logical parameter. Set to TRUE to always check whether .i and .t uniquely identify observations in the data. By default this is set to FALSE and the check is only performed once per session, and only if at least one of .i, .t, or .d is set. |

## Examples

```
data(Scorecard)

# The Scorecard data is uniquely identified by unitid and year.
# However, there are sometimes gaps between years.
# In cases like this, using dplyr::lag() will still use the row before,
# whereas tlag() will respect the gap and give a NA, much like plm::lag()
# (although tlag is slower than either, sorry)
Scorecard <- Scorecard %>%
  dplyr::mutate(pmdplyr_tlag = tlag(earnings_med,
    .i = unitid,
    .t = year
  ))
Scorecard <- Scorecard %>%
  dplyr::arrange(year) %>%
  dplyr::group_by(unitid) %>%
  dplyr::mutate(dplyr_lag = dplyr::lag(earnings_med)) %>%
  dplyr::ungroup()

# more NAs in the pmdplyr version - observations with a gap and thus no real lag present in data
sum(is.na(Scorecard$pmdplyr_tlag))
sum(is.na(Scorecard$dplyr_lag))

# If we want to ignore gaps, or have .d = 0, and .i and .t uniquely identify observations,
# we can use the .quick option to match dplyr::lag()
Scorecard <- Scorecard %>%
  dplyr::mutate(pmdplyr_quick_tlag = tlag(earnings_med,
    .i = unitid,
    .t = year,
    .d = 0,
    .quick = TRUE
  ))
sum(Scorecard$dplyr_lag != Scorecard$pmdplyr_quick_tlag, na.rm = TRUE)

# Where tlag shines is when you have multiple observations per .i/.t
# If the value of .var is constant within .i/.t, it will work just as you expect.
# If it's not, it will throw an error, or you can set
# .resolve to tell tlag how to select a single value from the many
# Maybe we want to get the lagged average earnings within degree award type
Scorecard <- Scorecard %>%
  dplyr::mutate(
    last_year_earnings_by_category =
      tlag(earnings_med,
        .i = pred_degree_awarded_ipeds, .t = year,
        .resolve = function(x) mean(x, na.rm = TRUE)
      )
  )
# Or maybe I want the lagged earnings across all types - .i isn't necessary!
Scorecard <- Scorecard %>%
  dplyr::mutate(last_year_earnings_all = tlag(earnings_med,
    .t = "year",
    .resolve = function(x) mean(x, na.rm = TRUE)
  ))
# Curious why the first nonmissing obs show up in 2012?
```

```
# It's because there's no 2008 or 2010 in the data, so when 2009 or 2011 look back
# a year, they find nothing!
# We could get around this by setting .d = 0 to ignore gap length
# Note this can be a little slow.
Scorecard <- Scorecard %>%
  dplyr::mutate(last_year_earnings_all = tlag(earnings_med,
    .t = year, .d = 0,
    .resolve = function(x) mean(x, na.rm = TRUE)
  ))
```

# Index