Package ‘tidyr’

October 28, 2018

Title  Easily Tidy Data with 'spread()' and 'gather()' Functions

Version  0.8.2

Description  An evolution of 'reshape2'. It's designed specifically for data
tidying (not general reshaping or aggregating) and works well with
'dplyr' data pipelines.

License  MIT + file LICENSE


BugReports  https://github.com/tidyverse/tidyr/issues

Encoding  UTF-8

Depends  R (>= 3.1)

Imports  dplyr (>= 0.7.0), glue, magrittr, purrr, Rcpp, rlang, stringi,
tibble, tidyselect (>= 0.2.5)

Suggests  covr, gapminder, knitr, rmarkdown, testthat

LinkingTo  Rcpp

VignetteBuilder  knitr

LazyData  true

RoxygenNote  6.1.0

NeedsCompilation  yes

Author  Hadley Wickham [aut, cre],
        Lionel Henry [aut],
        RStudio [cph]

Maintainer  Hadley Wickham <hadley@rstudio.com>

Repository  CRAN

Date/Publication  2018-10-28 17:20:03 UTC

R topics documented:

  complete ............................................................... 2
  drop_na ............................................................. 3
**Description**

Turns implicit missing values into explicit missing values. This is a wrapper around `expand()`, `dplyr::left_join()` and `replace_na()` that’s useful for completing missing combinations of data.

**Usage**

```
complete(data, ..., fill = list())
```

**Arguments**

- `data` A data frame.
- `...` Specification of columns to expand.
  
  To find all unique combinations of x, y and z, including those not found in the data, supply each variable as a separate argument. To find only the combinations that occur in the data, use `nest`: `expand(df, nesting(x, y, z))`.
  
  You can combine the two forms. For example, `expand(df, nesting(school_id, student_id), date)` would produce a row for every student for each date.
  
  For factors, the full set of levels (not just those that appear in the data) are used.
  
  For continuous variables, you may need to fill in values that don’t appear in the data: to do so use expressions like `year = 2010:2020` or `year = full_seq(year,1)`.
  
  Length-zero (empty) elements are automatically dropped.
- `fill` A named list that for each variable supplies a single value to use instead of `NA` for missing combinations.
**drop_na**

*Drop rows containing missing values*

**Details**

If you supply `fill`, these values will also replace existing explicit missing values in the data set.

**Examples**

```r
library(dplyr, warn.conflicts = FALSE)
df <- tibble(
  group = c(1:2, 1),
  item_id = c(1:2, 2),
  item_name = c("a", "b", "b"),
  value1 = 1:3,
  value2 = 4:6
)
df %>% complete(group, nesting(item_id, item_name))
```

```r
# You can also choose to fill in missing values
df %>% complete(group, nesting(item_id, item_name), fill = list(value1 = 0))
```

**Description**

Drop rows containing missing values

**Usage**

```r
drop_na(data, ...)
```

**Arguments**

- `data` A data frame.
- `...` A selection of columns. If empty, all variables are selected. You can supply bare variable names, select all variables between `x` and `z` with `x:z`, exclude `y` with `~y`. For more options, see the `dplyr::select()` documentation. See also the section on selection rules below.

**Rules for selection**

Arguments for selecting columns are passed to `tidyselect::vars_select()` and are treated specially. Unlike other verbs, selecting functions make a strict distinction between data expressions and context expressions.

- A data expression is either a bare name like `x` or an expression like `x:y` or `c(x, y)`. In a data expression, you can only refer to columns from the data frame.
- Everything else is a context expression in which you can only refer to objects that you have defined with `<-`. 
For instance, `col1:col3` is a data expression that refers to data columns, while `seq(start, end)` is a context expression that refers to objects from the contexts.

If you really need to refer to contextual objects from a data expression, you can unquote them with the tidy eval operator `!!`. This operator evaluates its argument in the context and inlines the result in the surrounding function call. For instance, `c(x, !! x)` selects the `x` column within the data frame and the column referred to by the object `x` defined in the context (which can contain either a column name as string or a column position).

### Examples

```r
library(dplyr)
df <- tibble(x = c(1, 2, NA), y = c("a", NA, "b"))
df %>% drop_na()
df %>% drop_na(x)
```

### expand

**Expand data frame to include all combinations of values**

**Description**

`expand()` is often useful in conjunction with `left_join` if you want to convert implicit missing values to explicit missing values. Or you can use it in conjunction with `anti_join()` to figure out which combinations are missing.

**Usage**

```
expand(data, ...)
```

```
crossing(...) 
```

```
nesting(...) 
```

**Arguments**

- `data` A data frame.
- `...` Specification of columns to expand.

To find all unique combinations of `x`, `y` and `z`, including those not found in the data, supply each variable as a separate argument. To find only the combinations that occur in the data, use `nest`: `expand(df, nesting(x, y, z))`.

You can combine the two forms. For example, `expand(df, nesting(school_id, student_id), date)` would produce a row for every student for each date.

For factors, the full set of levels (not just those that appear in the data) are used. For continuous variables, you may need to fill in values that don’t appear in the data: to do so use expressions like `year = 2010:2020` or `year = full_seq(year, 1)`.

Length-zero (empty) elements are automatically dropped.
Details

crossing() is similar to `expand.grid()`, this never converts strings to factors, returns a tbl_df without additional attributes, and first factors vary slowest. `nesting()` is the complement to `crossing()`: it only keeps combinations of all variables that appear in the data.

See Also

`complete()` for a common application of expand: completing a data frame with missing combinations.

Examples

```r
library(dplyr)
# All possible combinations of vs & cyl, even those that aren't present in the data
expand(mtcars, vs, cyl)

# Only combinations of vs and cyl that appear in the data
expand(mtcars, nesting(vs, cyl))

# Implicit missings -----------------------------------------------
df <- tibble(
  qtr = c(1, 2, 3, 4, 1, 2, 3),
  return = rnorm(7)
)
df %>% expand(year, qtr)
df %>% expand(year = 2010:2012, qtr)
df %>% expand(year = full_seq(year, 1), qtr)
df %>% complete(year = full_seq(year, 1), qtr)

# Nesting -----------------------------------------------------------
# Each person was given one of two treatments, repeated three times
# But some of the replications haven't happened yet, so we have incomplete data:
experiment <- tibble(
  name = rep(c("Alex", "Robert", "Sam"), c(3, 2, 1)),
  trt = rep(c("a", "b", "a"), c(3, 2, 1)),
  rep = c(1, 2, 3, 1, 2, 1),
  measurement_1 = runif(6),
  measurement_2 = runif(6)
)

# We can figure out the complete set of data with expand()
# Each person only gets one treatment, so we nest name and trt together:
all <- experiment %>% expand(nesting(name, trt), rep)
all

# We can use anti_join to figure out which observations are missing
all %>% anti_join(experiment)

# And use right_join to add in the appropriate missing values to the
```
extract

Extract one column into multiple columns.

Description

Given a regular expression with capturing groups, `extract()` turns each group into a new column. If the groups don’t match, or the input is NA, the output will be NA.

Usage

```r
extract(data, col, into, regex = "\([[:alnum:]]+\)\)(\([[:alnum:]]+\)\)(\([[:alnum:]]+\)\)(\([[:alnum:]]+\)\), remove = TRUE,
convert = FALSE, ...)
```

Arguments

- `data` A data frame.
- `col` Column name or position. This is passed to `tidyselect::vars_pull()`. This argument is passed by expression and supports quasiquotation (you can unquote column names or column positions).
- `into` Names of new variables to create as character vector.
- `regex` a regular expression used to extract the desired values. The should be one group (defined by \( \)) for each element of into.
- `remove` If TRUE, remove input column from output data frame.
- `convert` If TRUE, will run `type.convert()` with as.is = TRUE on new columns. This is useful if the component columns are integer, numeric or logical.
- `...` Other arguments passed on to `regexec()` to control how the regular expression is processed.

Examples

```r
library(dplyr)
df <- data.frame(x = c(NA, "a-b", "a-d", "b-c", "d-e"))
df %>% extract(x, "A")
df %>% extract(x, c("A", "B"), "([[:alnum:]]+)-(\([[:alnum:]]+\)\)(\([[:alnum:]]+\)\)(\([[:alnum:]]+\)\))")
# If no match, NA:
df %>% extract(x, c("A", "B"), "([A-d]+)-(\[a-d]+)\)"
```
**fill**  
*Fill in missing values.*

**Description**

Fills missing values in using the previous entry. This is useful in the common output format where values are not repeated, they're recorded each time they change.

**Usage**

```r
fill(data, ..., .direction = c("down", "up"))
```

**Arguments**

- `data`: A data frame.
- `...`: A selection of columns. If empty, all variables are selected. You can supply bare variable names, select all variables between x and z with `x:z`, exclude y with `-y`. For more options, see the `dplyr::select()` documentation. See also the section on selection rules below.
- `.direction`: Direction in which to fill missing values. Currently either "down" (the default) or "up".

**Details**

Missing values are replaced in atomic vectors; `NULL`s are replaced in list.

**Examples**

```r
df <- data.frame(Month = 1:12, Year = c(2000, rep(NA, 11)))
df %>% fill(Year)
```

---

**full_seq**  
*Create the full sequence of values in a vector.*

**Description**

This is useful if you want to fill in missing values that should have been observed but weren't. For example, `full_seq(c(1, 2, 4, 6), 1)` will return 1:6.

**Usage**

```r
full_seq(x, period, tol = 1e-06)
```
Arguments

- **x**: A numeric vector.
- **period**: Gap between each observation. The existing data will be checked to ensure that it is actually of this periodicity.
- **tol**: Numerical tolerance for checking periodicity.

Examples

```r
full_seq(c(1L, 2, 4, 5, 10), 1)
```

---

### Description

Gather takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed. You use `gather()` when you notice that you have columns that are not variables.

### Usage

```r
gather(data, key = "key", value = "value", ..., na.rm = FALSE, convert = FALSE, factor_key = FALSE)
```

### Arguments

- **data**: A data frame.
- **key, value**: Names of new key and value columns, as strings or symbols.
  
  This argument is passed by expression and supports quasiquotation (you can unquote strings and symbols). The name is captured from the expression with `rlang::ensym()` (note that this kind of interface where symbols do not represent actual objects is now discouraged in the tidyverse; we support it here for backward compatibility).

- **...**: A selection of columns. If empty, all variables are selected. You can supply bare variable names, select all variables between x and z with `x:z`, exclude y with `~y`. For more options, see the `dplyr::select()` documentation. See also the section on selection rules below.

- **na.rm**: If `TRUE`, will remove rows from output where the value column is `NA`.

- **convert**: If `TRUE` will automatically run `type.convert()` on the key column. This is useful if the column types are actually numeric, integer, or logical.

- **factor_key**: If `FALSE`, the default, the key values will be stored as a character vector. If `TRUE`, will be stored as a factor, which preserves the original ordering of the columns.
**Rules for selection**

Arguments for selecting columns are passed to `tidyselect::vars_select()` and are treated specially. Unlike other verbs, selecting functions make a strict distinction between data expressions and context expressions.

- A data expression is either a bare name like `x` or an expression like `x:y` or `c(x, y)`. In a data expression, you can only refer to columns from the data frame.
- Everything else is a context expression in which you can only refer to objects that you have defined with `<-`.

For instance, `col1:col3` is a data expression that refers to data columns, while `seq(start, end)` is a context expression that refers to objects from the contexts.

If you really need to refer to contextual objects from a data expression, you can unquote them with the tidy eval operator `AA`. This operator evaluates its argument in the context and inlines the result in the surrounding function call. For instance, `c(x, AA x)` selects the `x` column within the data frame and the column referred to by the object `x` defined in the context (which can contain either a column name as string or a column position).

**Examples**

```r
library(dplyr)
# From http://stackoverflow.com/questions/118060
stocks <- tibble(
  time = as.Date('2009-01-01') + 0:9,
  X = rnorm(10, 0, 1),
  Y = rnorm(10, 0, 2),
  Z = rnorm(10, 0, 4)
)

gather(stocks, stock, price, -time)
stocks %>% gather(stock, price, -time)

# get first observation for each Species in iris data -- base R
mini_iris <- iris[c(1, 51, 101), ]
# gather Sepal.Length, Sepal.Width, Petal.Length, Petal.Width
gather(mini_iris, key = flower_att, value = measurement, 
  Sepal.Length, Sepal.Width, Petal.Length, Petal.Width)
# same result but less verbose
gather(mini_iris, key = flower_att, value = measurement, -Species)

# repeat iris example using dplyr and the pipe operator
library(dplyr)
mini_iris <-
  iris %>%
    group_by(Species) %>%
    slice(1)
mini_iris %>% gather(key = flower_att, value = measurement, -Species)
```
nest

Nest repeated values in a list-variable.

Description

There are many possible ways one could choose to nest columns inside a data frame. `nest()` creates a list of data frames containing all the nested variables: this seems to be the most useful form in practice.

Usage

```
nest(data, ..., .key = "data")
```

Arguments

- `data` A data frame.
- `...` A selection of columns. If empty, all variables are selected. You can supply bare variable names, select all variables between `x` and `z` with `x\:z`, exclude `y` with `\!\!y`. For more options, see the `dplyr::select()` documentation. See also the section on selection rules below.
- `key` The name of the new column, as a string or symbol.

This argument is passed by expression and supports quasiquotation (you can unquote strings and symbols). The name is captured from the expression with `rlang::ensym()` (note that this kind of interface where symbols do not represent actual objects is now discouraged in the tidyverse; we support it here for backward compatibility).

Rules for selection

Arguments for selecting columns are passed to `tidyselect::vars_select()` and are treated specially. Unlike other verbs, selecting functions make a strict distinction between data expressions and context expressions.

- A data expression is either a bare name like `x` or an expression like `x\:y` or `c(x, y)`. In a data expression, you can only refer to columns from the data frame.
- Everything else is a context expression in which you can only refer to objects that you have defined with `<-`.

For instance, `col1:col3` is a data expression that refers to data columns, while `seq(start, end)` is a context expression that refers to objects from the contexts.

If you really need to refer to contextual objects from a data expression, you can unquote them with the tidy eval operator `!!`. This operator evaluates its argument in the context and inlines the result in the surrounding function call. For instance, `c(x, !! x)` selects the `x` column within the data frame and the column referred to by the object `x` defined in the context (which can contain either a column name as string or a column position).
replace_na

See Also

unnest() for the inverse operation.

Examples

library(dplyr)
as_tibble(iris) %>% nest(-Species)as_tibble(chickwts) %>% nest(weight)

if (require("gapminder")) {
gapminder %>%
  group_by(country, continent) %>%
  nest()

gapminder %>%
  nest(-country, -continent)
}

---

replace_na Replace missing values

Description

Replace missing values

Usage

replace_na(data, replace, ...)

Arguments

data A data frame or vector.
replace If data is a data frame, a named list giving the value to replace NA with for each column. If data is a vector, a single value used for replacement.

See Also

na_if to replace specified values with a NA, coalesce to replace missing values with a specified value. recode to more generally replace values.
Examples

```r
library(dplyr)

df <- tibble(x = c(1L, 2L, NA), y = c("a", NA, "b"), z = list(1:5, NULL, 10:20))
df %>% replace_na(list(x = 0, y = "unknown"))
df %>% mutate(x = replace_na(x, 0))

# NULL are the list-col equivalent of NAs
df %>% replace_na(list(z = list(5)))

df$x %>% replace_na(0)
df$y %>% replace_na("unknown")
```

separate

Separate one column into multiple columns.

Description

Given either regular expression or a vector of character positions, separate() turns a single character column into multiple columns.

Usage

```r
separate(data, col, into, sep = "^[[:alnum:]]+", remove = TRUE,
          convert = FALSE, extra = "warn", fill = "warn", ...)
```

Arguments

- **data**: A data frame.
- **col**: Column name or position. This is passed to `tidyselect::vars_pull()`. This argument is passed by expression and supports quasiquotation (you can unquote column names or column positions).
- **into**: Names of new variables to create as character vector. Use `NA` to omit the variable in the output.
- **sep**: Separator between columns.
  - If character, is interpreted as a regular expression. The default value is a regular expression that matches any sequence of non-alphanumeric values.
  - If numeric, interpreted as positions to split at. Positive values start at 1 at the far-left of the string; negative values start at -1 at the far-right of the string. The length of `sep` should be one less than `into`.
- **remove**: If `TRUE`, remove input column from output data frame.
- **convert**: If `TRUE`, will run `type.convert()` with `as.is = TRUE` on new columns. This is useful if the component columns are integer, numeric or logical.
- **extra**: If `sep` is a character vector, this controls what happens when there are too many pieces. There are three valid options:
  - "warn" (the default): emit a warning and drop extra values.
separate_rows

Separate a collapsed column into multiple rows.

Description
If a variable contains observations with multiple delimited values, this separates the values and places each one in its own row.

Usage
separate_rows(data, ..., sep = "[^[:alnum:]\.]\+", convert = FALSE)

Examples

library(dplyr)
df <- data.frame(x = c(NA, "a.b", "a.d", "b.c"))
df %>% separate(x, c("A", "B"))

# If you just want the second variable:
df %>% separate(x, c(NA, "B"))

# If every row doesn't split into the same number of pieces, use
# the extra and fill arguments to control what happens
df <- data.frame(x = c("a", "a b", "a b c", NA))
df %>% separate(x, c("a", "b"))

# The same behaviour but no warnings
df %>% separate(x, c("a", "b"), extra = "drop", fill = "right")
# Another option:
df %>% separate(x, c("a", "b"), extra = "merge", fill = "left")

# If only want to split specified number of times use extra = "merge"
df <- data.frame(x = c("x: 123", "y: error: 7"))
df %>% separate(x, c("key", "value"), ": ", extra = "merge")
Arguments

data A data frame.

... A selection of columns. If empty, all variables are selected. You can supply bare variable names, select all variables between x and z with x:z, exclude y with ~y. For more options, see the dplyr::select() documentation. See also the section on selection rules below.

sep Separator delimiting collapsed values.

convert If TRUE will automatically run type.convert() on the key column. This is useful if the column types are actually numeric, integer, or logical.

Rules for selection

Arguments for selecting columns are passed to tidyselect::vars_select() and are treated specially. Unlike other verbs, selecting functions make a strict distinction between data expressions and context expressions.

- A data expression is either a bare name like x or an expression like x:y or c(x, y). In a data expression, you can only refer to columns from the data frame.
- Everything else is a context expression in which you can only refer to objects that you have defined with <M.

For instance, col1:col3 is a data expression that refers to data columns, while seq(start, end) is a context expression that refers to objects from the contexts.

If you really need to refer to contextual objects from a data expression, you can unquote them with the tidy eval operator !!. This operator evaluates its argument in the context and inlines the result in the surrounding function call. For instance, c(x, !! x) selects the x column within the data frame and the column referred to by the object x defined in the context (which can contain either a column name as string or a column position).

Examples

def <- data.frame(
  x = 1:3,
  y = c("a", "d,e,f", "g,h"),
  z = c("1", "2,3,4", "5,6"),
  stringsAsFactors = FALSE
)
separate_rows(df, y, z, convert = TRUE)

---

smiths Some data about the Smith family.

---

Description

A small demo dataset describing John and Mary Smith.
**spread**

**Usage**

```
smaths
```

**Format**

A data frame with 2 rows and 5 columns.

---

**spread**  
*Spread a key-value pair across multiple columns.*

**Description**

Spread a key-value pair across multiple columns.

**Usage**

```
spread(data, key, value, fill = NA, convert = FALSE, drop = TRUE, sep = NULL)
```

**Arguments**

- **data**  
  A data frame.

- **key, value**  
  Column names or positions. This is passed to `tidyselect::vars_pull()`. These arguments are passed by expression and support quasiquotation (you can unquote column names or column positions).

- **fill**  
  If set, missing values will be replaced with this value. Note that there are two types of missingness in the input: explicit missing values (i.e. `NA`), and implicit missings, rows that simply aren’t present. Both types of missing value will be replaced by `fill`.

- **convert**  
  If `TRUE`, `type_convert()` with `asis = TRUE` will be run on each of the new columns. This is useful if the value column was a mix of variables that was coerced to a string. If the class of the value column was factor or date, note that will not be true of the new columns that are produced, which are coerced to character before type conversion.

- **drop**  
  If `FALSE`, will keep factor levels that don’t appear in the data, filling in missing combinations with `fill`.

- **sep**  
  If `NULL`, the column names will be taken from the values of `key` variable. If non-`NULL`, the column names will be given by "<key_name><sep><key_value>".
Examples

```r
library(dplyr)

stocks <- data.frame(
  time = as.Date('2009-01-01') + 0:9,
  X = rnorm(10, 0, 1),
  Y = rnorm(10, 0, 2),
  Z = rnorm(10, 0, 4)
)

stocksm <- stocks %>% gather(stock, price, -time)
stocksm %>% spread(stock, price)

# spread and gather are complements
df <- data.frame(x = c("a", "b"), y = c(3, 4), z = c(5, 6))

df %>% spread(x, y) %>% gather(x, y, a:b, na.rm = TRUE)

# Use 'convert = TRUE' to produce variables of mixed type
df <- data.frame(row = rep(c(1, 51), each = 3),
  var = c("Sepal.Length", "Species", "Species_num"),
  value = c(5.1, "setosa", 1, 7.0, "versicolor", 2))

df %>% spread(var, value) %>% str

df %>% spread(var, value, convert = TRUE) %>% str
```

table 1  

<table>
<thead>
<tr>
<th>Description</th>
<th>Example tabular representations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data sets that demonstrate multiple ways to layout the same tabular data.</td>
<td></td>
</tr>
</tbody>
</table>

Usage

```
table1
table2
table3
table4a
table4b
table5
```

Format

An object of class tbl_df (inherits from tbl.data.frame) with 6 rows and 4 columns.
table1, table2, table3, table4a, table4b, and table5 all display the number of TB cases documented by the World Health Organization in Afghanistan, Brazil, and China between 1999 and 2000. The data contains values associated with four variables (country, year, cases, and population), but each table organizes the values in a different layout.

The data is a subset of the data contained in the World Health Organization Global Tuberculosis Report.

Source

http://www.who.int/tb/country/data/download/en/

uncount

"Uncount" a data frame

Description

Performs the opposite operation to dplyr::count(), duplicating rows according to a weighting variable (or expression).

Usage

uncount(data, weights, .remove = TRUE, .id = NULL)

Arguments

data A data frame, tibble, or grouped tibble.
weights A vector of weights. Evaluated in the context of data; supports quasiquotation.
.remove If TRUE, and weights is a single
.id Supply a string to create a new variable which gives a unique identifier for each created row.

Examples

df <- tibble::tibble(x = c("a", "b"), n = c(1, 2))
uncount(df, n)
uncount(df, n, .id = "id")

# You can also use constants
uncount(df, 2)

# Or expressions
uncount(df, 2 / n)
Unite multiple columns into one.

Description

Convenience function to paste together multiple columns into one.

Usage

unite(data, col, ..., sep = "_", remove = TRUE)

Arguments

data

A data frame.

col

The name of the new column, as a string or symbol.

This argument is passed by expression and supports quasiquotation (you can unquote strings and symbols). The name is captured from the expression with `rlang::ensym()` (note that this kind of interface where symbols do not represent actual objects is now discouraged in the tidyverse; we support it here for backward compatibility).

... A selection of columns. If empty, all variables are selected. You can supply bare variable names, select all variables between x and z with `x:z`, exclude y with `~y`. For more options, see the `dplyr::select()` documentation. See also the section on selection rules below.

sep

Separator to use between values.

remove

If TRUE, remove input columns from output data frame.

Rules for selection

Arguments for selecting columns are passed to `tidyselect::vars_select()` and are treated specially. Unlike other verbs, selecting functions make a strict distinction between data expressions and context expressions.

- A data expression is either a bare name like x or an expression like `x:y` or `c(x, y)`. In a data expression, you can only refer to columns from the data frame.
- Everything else is a context expression in which you can only refer to objects that you have defined with `<-`.

For instance, `col1:col3` is a data expression that refers to data columns, while `seq(start, end)` is a context expression that refers to objects from the contexts.

If you really need to refer to contextual objects from a data expression, you can unquote them with the tidy eval operator `!!`. This operator evaluates its argument in the context and inlines the result in the surrounding function call. For instance, `c(x, !! x)` selects the x column within the data frame and the column referred to by the object x defined in the context (which can contain either a column name as string or a column position).
unnest

See Also

separate(), the complement.

Examples

library(dplyr)
unite(mtcars, "vs_am", c("vs","am"))

# Separate is the complement of unite
mtcars %>%
  unite(vs_am, vs, am) %>%
  separate(vs_am, c("vs", "am"))

unnest

Unnest a list column.

Description

If you have a list-column, this makes each element of the list its own row. unnest() can handle list-columns that contain atomic vectors, lists, or data frames (but not a mixture of the different types).

Usage

unnest(data, ..., .drop = NA, .id = NULL, .sep = NULL, 
  .preserve = NULL)

Arguments

data A data frame.

... Specification of columns to unnest. Use bare variable names or functions of variables. If omitted, defaults to all list-cols.

.drop Should additional list columns be dropped? By default, unnest will drop them if unnesting the specified columns requires the rows to be duplicated.

.id Data frame identifier - if supplied, will create a new column with name .id, giving a unique identifier. This is most useful if the list column is named.

.sep If non-NULL, the names of unnested data frame columns will combine the name of the original list-col with the names from nested data frame, separated by .sep.

.preserve Optionally, list-columns to preserve in the output. These will be duplicated in the same way as atomic vectors. This has dplyr::select semantics so you can preserve multiple variables with .preserve = c(x, y) or .preserve = starts_with("list").

Details

If you unnest multiple columns, parallel entries must have the same length or number of rows (if a data frame).
See Also

nest() for the inverse operation.

Examples

```r
library(dplyr)
df <- tibble(
  x = 1:3,
  y = c("a", "d,e,f", "g,h")
)
df %>%
  transform(y = strsplit(y, ",") %>% unnest(y))

# Or just
df %>%
  unnest(y = strsplit(y, ","))

# It also works if you have a column that contains other data frames!
df <- tibble(
  x = 1:2,
  y = list(
    tibble(z = 1),
    tibble(z = 3:4)
  )
)
df %>% unnest(y)

# You can also unnest multiple columns simultaneously
df <- tibble(
  a = list(c("a", "b"), "c"),
  b = list(1:2, 3),
  c = c(11, 22)
)
df %>% unnest(a, b)
# If you omit the column names, it'll unnest all list-cols
df %>% unnest()

# You can also choose to preserve one or more list-cols
df %>% unnest(a, .preserve = b)

# Nest and unnest are inverses
df <- data.frame(x = c(1, 1, 2), y = 3:1)
df %>% nest(y)
df %>% nest(y) %>% unnest()

# If you have a named list-column, you may want to supply .id
df <- tibble(
  x = 1:2,
  y = list(a = 1, b = 3:4)
)
unnest(df, .id = "name")
```
World Health Organization TB data

Description
A subset of data from the World Health Organization Global Tuberculosis Report, and accompanying global populations.

Usage
who
population

Format
A dataset with the variables

country  Country name
iso2, iso3  2 & 3 letter ISO country codes
year  Year
new_sp_m014 - new_rel_f65  Counts of new TB cases recorded by group. Column names encode three variables that describe the group (see details).

Details
The data uses the original codes given by the World Health Organization. The column names for columns five through 60 are made by combining new_ to a code for method of diagnosis (rel = relapse, sn = negative pulmonary smear, sp = positive pulmonary smear, ep = extrapulmonary) to a code for gender (f = female, m = male) to a code for age group (01T = 0-14 yrs of age, 152T = 15-24 years of age, 253T = 25 to 34 years of age, 354T = 35 to 44 years of age, 455T = 45 to 54 years of age, 556T = 55 to 64 years of age, 65r = 65 years of age or older).

Source
http://www.who.int/tb/country/data/download/en/
Index

*Topic datasets
    smiths, 14
    table1, 16
    who, 21

coalesce, 11
complete, 2
complete(), 5
crossing (expand), 4
dplyr::count(), 17
dplyr::left_join(), 2
dplyr::select, 19
dplyr::select(), 3, 7, 8, 10, 14, 18
drop_na, 3

expand, 4
expand(), 2
expand.grid(), 5
extract, 6
extract(), 13

fill, 7
full_seq, 2, 4, 7

gather, 8

na_if, 11
nest, 10
nest(), 20
nesting (expand), 4

population (who), 21

quasiquotation, 6, 8, 10, 12, 15, 18

recode, 11
regexec(), 6
replace_na, 11
replace_na(), 2
rlang::ensym(), 8, 10, 18

separate, 12
separate(), 19
separate_rows, 13
smiths, 14
spread, 15

table1, 16
table2 (table1), 16
table3 (table1), 16
table4a (table1), 16
table4b (table1), 16
table5 (table1), 16
tidyselect::vars_pull(), 6, 12, 15
tidyselect::vars_select(), 3, 9, 10, 14, 18
type.convert(), 6, 8, 12, 14, 15

uncount, 17
unite, 18
unite(), 13
unnest, 19
unnest(), 11

who, 21