

Package ‘tidyr’

February 5, 2016

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

Version 0.4.1

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

Depends R (>= 3.1.0)

License MIT + file LICENSE

LazyData true

Imports dplyr (>= 0.4), stringi, lazyeval, magrittr, Rcpp

URL <https://github.com/hadley/tidyr>

BugReports <https://github.com/hadley/tidyr/issues>

Suggests knitr, testthat, data.table, covr, gapminder, rmarkdown

VignetteBuilder knitr

LinkingTo Rcpp

RoxygenNote 5.0.1

NeedsCompilation yes

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

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2016-02-05 21:46:58

R topics documented:

complete	2
expand	3
extract	5
extract_numeric	6
fill	6

full_seq	7
gather	8
nest	9
replace_na	10
separate	10
spread	12
unite	13
unnest	14
Index	16

complete	<i>Complete a data frame with missing combinations of data.</i>
----------	---

Description

Turns implicit missing values into explicit missing values. This is a wrapper around [expand\(\)](#), [left_join\(\)](#) and [replace_na](#) that's useful for completing missing combinations of data.

Usage

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

Arguments

data	A data frame
...	<p>Specification of columns to expand.</p> <p>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: <code>expand(df, nesting(x, y, z))</code>.</p> <p>You can combine the two forms. For example, <code>expand(df, nesting(school_id, student_id), date)</code> would produce a row for every student for each date.</p> <p>To fill in values that are missing altogether, use expressions like <code>year = 2010:2020</code> or <code>year = full_seq(year)</code>.</p>
fill	A named list that for each variable supplies a single value to use instead of NA for missing combinations.

See Also

[complete_](#) for a version that uses regular evaluation and is suitable for programming with.

Examples

```
library(dplyr)
df <- data_frame(
  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))

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

 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
...	<p>Specification of columns to expand.</p> <p>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)).</p> <p>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.</p> <p>To fill in values that are missing altogether, use expressions like year = 2010:2020 or year = full_seq(year).</p>

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.

[expand_](#) for a version that uses regular evaluation and is suitable for programming with.

Examples

```
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 <- data_frame(
  year = c(2010, 2010, 2010, 2010, 2012, 2012, 2012),
  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 <- data_frame(
  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),
  measurment_1 = runif(6),
  measurment_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
# original data
all %>% right_join(experiment)
# Or use the complete() short-hand
experiment %>% complete(nesting(name, trt), rep)
```

extract	<i>Extract one column into multiple columns.</i>
---------	--

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

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

Arguments

<code>data</code>	A data frame.
<code>col</code>	Bare column name.
<code>into</code>	Names of new variables to create as character vector.
<code>regex</code>	a regular expression used to extract the desired values.
<code>remove</code>	If TRUE, remove input column from output data frame.
<code>convert</code>	If TRUE, will run <code>type.convert</code> with <code>as.is = TRUE</code> on new columns. This is useful if the component columns are integer, numeric or logical.
<code>...</code>	Other arguments passed on to <code>regexec</code> to control how the regular expression is processed.

See Also

[extract_](#) for a version that uses regular evaluation and is suitable for programming with.

Examples

```
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:]]+")

# If no match, NA:
df %>% extract(x, c("A", "B"), "[a-d]+-[a-d]+")
```

extract_numeric	<i>Extract numeric component of variable.</i>
-----------------	---

Description

This uses a regular expression to strip all non-numeric character from a string and then coerces the result to a number. This is useful for strings that are numbers with extra formatting (e.g. \$1,200.34).

Usage

```
extract_numeric(x)
```

Arguments

x	A character vector (or a factor).
---	-----------------------------------

Examples

```
extract_numeric("$1,200.34")
extract_numeric("-2%")

# The heuristic is not perfect - it won't fail for things that
# clearly aren't numbers
extract_numeric("-2-2")
extract_numeric("12abc34")
```

fill	<i>Fill in missing values.</i>
------	--------------------------------

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

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

Arguments

data	A data frame.
...	Specification of columns to fill. Use bare variable names. Select all variables between x and z with x:z, exclude y with -y. For more options, see the select documentation.
.direction	Direction in which to fill missing values. Currently either "down" (the default) or "up".

Details

Missing values are replaced in atomic vectors; NULLs are replaced in list.

See Also

[fill_](#) for a version that uses regular evaluation and is suitable for programming with.

Examples

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

full_seq	<i>Create the full sequence of values in a vector.</i>
----------	--

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

```
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

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

gather	<i>Gather columns into key-value pairs.</i>
--------	---

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

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

Arguments

<code>data</code>	A data frame.
<code>key, value</code>	Names of key and value columns to create in output.
<code>...</code>	Specification of columns to gather. Use bare variable names. Select all variables between <code>x</code> and <code>z</code> with <code>x:z</code> , exclude <code>y</code> with <code>-y</code> . For more options, see the select documentation.
<code>na.rm</code>	If TRUE, will remove rows from output where the value column is NA.
<code>convert</code>	If TRUE will automatically run type.convert on the key column. This is useful if the column names are actually numeric, integer, or logical.
<code>factor_key</code>	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.

See Also

[gather_](#) for a version that uses regular evaluation and is suitable for programming with.

Examples

```
library(dplyr)
# From http://stackoverflow.com/questions/1181060
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)
)

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

<code>data</code>	A data frame.
<code>...</code>	Specification of columns to nest. Use bare variable names. Select all variables between <code>x</code> and <code>z</code> with <code>x:z</code> , exclude <code>y</code> with <code>-y</code> . For more options, see the select documentation.
<code>.key</code>	The name of the new column.

See Also

[unnest](#) for the inverse operation.

[nest_](#) for a version that uses regular evaluation and is suitable for programming with.

Examples

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

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

```

    nest()

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

```

replace_na	<i>Replace missing values</i>
------------	-------------------------------

Description

Replace missing values

Usage

```
replace_na(data, replace = list(), ...)
```

Arguments

data	A data frame.
replace	A named list given the value to replace NA with for each column.
...	Additional arguments for methods. Currently unused.

Examples

```

library(dplyr)
df <- data_frame(x = c(1, 2, NA), y = c("a", NA, "b"))
df %>% replace_na(list(x = 0, y = "unknown"))

```

separate	<i>Separate one column into multiple columns.</i>
----------	---

Description

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

Usage

```

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

```

Arguments

<code>data</code>	A data frame.
<code>col</code>	Bare column name.
<code>into</code>	Names of new variables to create as character vector.
<code>sep</code>	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 value start at -1 at the far-right of the string. The length of <code>sep</code> should be one less than <code>into</code> .
<code>remove</code>	If TRUE, remove input column from output data frame.
<code>convert</code>	If TRUE, will run <code>type.convert</code> with <code>as.is = TRUE</code> on new columns. This is useful if the component columns are integer, numeric or logical.
<code>extra</code>	If <code>sep</code> is a character vector, this controls what happens when there are too many pieces. There are three valid options: <ul style="list-style-type: none"> • "warn" (the default): emit a warning and drop extra values. • "drop": drop any extra values without a warning. • "merge": only splits at most <code>length(into)</code> times
<code>fill</code>	If <code>sep</code> is a character vector, this controls what happens when there are not enough pieces. There are three valid options: <ul style="list-style-type: none"> • "warn" (the default): emit a warning and fill from the right • "right": fill with missing values on the right • "left": fill with missing values on the left
<code>...</code>	Defunct, will be removed in the next version of the package.

See Also

[unite\(\)](#), the complement.

[separate_](#) for a version that uses regular evaluation and is suitable for programming with.

Examples

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

# If every row doesn't split into the same number of pieces, use
# the extra and file 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")
```

spread	<i>Spread a key-value pair across multiple columns.</i>
--------	---

Description

Spread a key-value pair across multiple columns.

Usage

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

Arguments

data	A data frame.
key	The bare (unquoted) name of the column whose values will be used as column headings.
value	The bare (unquoted) name of the column whose values will populate the cells.
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, <code>type.convert</code> with <code>asis = TRUE</code> 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.

See Also

[spread_](#) for a version that uses regular evaluation and is suitable for programming with.

Examples

```
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)
stocksm %>% spread(time, 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

```

unite	<i>Unite multiple columns into one.</i>
-------	---

Description

Convenience function to paste together multiple columns into one.

Usage

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

Arguments

data	A data frame.
col	(Bare) name of column to add
...	Specification of columns to unite. Use bare variable names. Select all variables between x and z with x:z, exclude y with -y. For more options, see the select documentation.
sep	Separator to use between values.
remove	If TRUE, remove input columns from output data frame.

See Also

[separate\(\)](#), the complement.

[unite_](#) for a version that uses regular evaluation and is suitable for programming with.

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. List-columns can either be atomic vectors or data frames. Each row must have the same number of entries.

Usage

```
unnest(data, ..., .drop = NA)
```

Arguments

<code>data</code>	A data frame.
<code>...</code>	Specification of columns to nest. Use bare variable names or functions of variables. If omitted, defaults to all list-cols.
<code>.drop</code>	Should additional list columns be dropped? By default, <code>unnest</code> will drop them if unnesting the specified columns requires the rows to be duplicated.

See Also

[nest](#) for the inverse operation.

[unnest_](#) for a version that uses regular evaluation and is suitable for programming with.

Examples

```
library(dplyr)
df <- data_frame(
  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 <- data_frame(
  x = 1:2,
  y = list(
    data_frame(z = 1),
    data_frame(z = 3:4)
  )
)
```

```
df %>% unnest(y)

# You can also unnest multiple columns simultaneously
df <- data_frame(
  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()

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

Index

complete, [2](#), [4](#)
complete_, [2](#)
crossing (expand), [3](#)

expand, [2](#), [3](#)
expand.grid, [3](#)
expand_, [4](#)
extract, [5](#)
extract_, [5](#)
extract_numeric, [6](#)

fill, [6](#)
fill_, [7](#)
full_seq, [2](#), [3](#), [7](#)

gather, [8](#)
gather_, [8](#)

left_join, [2](#)

nest, [9](#), [14](#)
nest_, [9](#)
nesting (expand), [3](#)

regexec, [5](#)
replace_na, [2](#), [10](#)

select, [6](#), [8](#), [9](#), [13](#)
separate, [10](#), [13](#)
separate_, [11](#)
spread, [12](#)
spread_, [12](#)

type.convert, [5](#), [8](#), [11](#), [12](#)

unite, [11](#), [13](#)
unite_, [13](#)
unnest, [9](#), [14](#)
unnest_, [14](#)