Package 'tidyr'

May 19, 2020

Title Tidy Messy Data

Version 1.1.0

Description Tools to help to create tidy data, where each column is a variable, each row is an observation, and each cell contains a single value. 'tidyr' contains tools for changing the shape (pivoting) and hierarchy (nesting and 'unnesting') of a dataset, turning deeply nested lists into rectangular data frames ('rectangling'), and extracting values out of string columns. It also includes tools for working with missing values (both implicit and explicit).

License MIT + file LICENSE

URL https://tidyr.tidyverse.org, https://github.com/tidyverse/tidyr

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

Depends R (>= 3.1)

Imports dplyr (>= 0.8.2), ellipsis (>= 0.1.0), glue, magrittr, purrr, Rcpp, rlang, stringi, tibble (>= 2.1.1), tidyselect (>= 1.1.0), utils, vctrs (>= 0.3.0), lifecycle

Suggests covr, jsonlite, knitr, repurrrsive (>= 1.0.0), rmarkdown, readr, testthat (>= 2.1.0)

LinkingTo Rcpp

VignetteBuilder knitr

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.1.0

R topics documented:

`billboard` *Song rankings for billboard top 100 in the year 2000*

Description

Song rankings for billboard top 100 in the year 2000

Usage

```
billboard
```

Format

A dataset with variables:

artist Artist name

track Song name,

date.enter Date the song entered the top 100

wk1 – wk76 Rank of the song in each week after it entered

Source

The "Whitburn" project, https://waxy.org/2008/05/the_whitburn_project/, (downloaded April 2008)

---

chop *Chop and unchop*

Description

Maturing

Chopping and unchopping preserve the width of a data frame, changing its length. `chop()` makes `df` shorter by converting rows within each group into list-columns. `unchop()` makes `df` longer by expanding list-columns so that each element of the list-column gets its own row in the output. `chop()` and `unchop()` are building blocks for more complicated functions (like `unnest()`, `unnest_longer()`, and `unnest_wider()`) and are generally more suitable for programming than interactive data analysis.

Usage

```
chop(data, cols)

unchop(data, cols, keep_empty = FALSE, ptype = NULL)
```

Arguments

`data`	A data frame.
`cols`	<`tidy-select`> Columns to chop or unchop (automatically quoted). For `unchop()`, each column should be a list-column containing generalised vectors (e.g. any mix of `NULL`s, atomic vector, S3 vectors, a lists, or data frames).
`keep_empty`	By default, you get one row of output for each element of the list your unchopping/unnesting. This means that if there's a size-0 element (like `NULL` or an empty data frame), that entire row will be dropped from the output. If you want to preserve all rows, use `keep_empty = TRUE` to replace size-0 elements with a single row of missing values.
`ptype`	Optionally, supply a data frame prototype for the output `cols`, overriding the default that will be guessed from the combination of individual values.

Details

Generally, unchopping is more useful than chopping because it simplifies a complex data structure, and `nest()`ing is usually more appropriate that `chop()`ing‘ since it better preserves the connections between observations.

`chop()` creates list-columns of class `vctrs::list_of()` to ensure consistent behaviour when the chopped data frame is emptied. For instance this helps getting back the original column types after the roundtrip chop and unchop. Because <list_of> keeps tracks of the type of its elements, `unchop()` is able to reconstitute the correct vector type even for empty list-columns.

Examples

```
# Chop ============================================================
df <- tibble(x = c(1, 1, 1, 2, 2, 3), y = 1:6, z = 6:1)
# Note that we get one row of output for each unique combination of
# non-chopped variables
df %>% chop(c(y, z))
# cf nest
df %>% nest(data = c(y, z))

# Unchop ==========================================================
df <- tibble(x = 1:4, y = list(integer(), 1L, 1:2, 1:3))
df %>% unchop(y)
df %>% unchop(y, keep_empty = TRUE)

# Incompatible types ----------------------------------------------
# If the list-col contains types that can not be natively
df <- tibble(x = 1:2, y = list("1", 1:3))
try(df %>% unchop(y))

# Unchopping data frames ------------------------------------------------------
# Unchopping a list-col of data frames must generate a df-col because
# unchop leaves the column names unchanged
df <- tibble(x = 1:3, y = list(NULL, tibble(x = 1), tibble(y = 1:2)))
df %>% unchop(y)
df %>% unchop(y, keep_empty = TRUE)
```

---

complete *Complete a data frame with missing combinations of data*

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. Columns can be atomic vectors or lists. • To find all unique combinations of x, y and z, including those not present in the data, supply each variable as a separate argument: expand(df,x,y,z). • To find only the combinations that occur in the data, use nesting: expand(df,nesting(x,y,z)) • You can combine the two forms. For example, expand(df,nesting(school_id,student_id), would produce a row for each present school-student combination for all possible dates. When used with factors, expand() uses the full set of levels, not just those that appear in the data. If you want to use only the values seen in the data, use forcats::fct_drop(). When used with continuous variables, you may need to fill in values that do not appear in the data: to do so use expressions like year = 2010:2020 or year = \link{full_seq}(year,1).
fill	A named list that for each variable supplies a single value to use instead of NA for missing combinations.

Details

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

Examples

```
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))

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

---

construction *Completed construction in the US in 2018*

Description

Completed construction in the US in 2018

Usage

```
construction
```

Format

A dataset with variables:

Year,Month Record date

1 unit**,** 2 to 4 units**,** 5 units or mote Number of completed units of each size

Northeast,Midwest,South,West Number of completed units in each region

Source

Completions of "New Residential Construction" found in Table 5 at https://www.census.gov/construction/nrc/xls/newresconst.xls (downloaded March 2019)

---

drop_na *Drop rows containing missing values*

Description

Drop rows containing missing values

Usage

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

Arguments

data	A data frame.
...	<tidy-select> Columns to inspect for missing values.

Examples

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

vars <- "y"
df %>% drop_na(x, any_of(vars))
```

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

Description

`expand()` generates all combination of variables found in a dataset. It is paired with `nesting()` and `crossing()` helpers. `crossing()` is a wrapper around `expand_grid()` that de-duplicates and sorts its inputs; `nesting()` is a helper that only finds combinations already present in the data.

`expand()` is often useful in conjunction with joins:

- use it with `right_join()` to convert implicit missing values to explicit missing values (e.g., fill in gaps in your data frame).
- use it with `anti_join()` to figure out which combinations are missing (e.g., identify gaps in your data frame).

Usage

```
expand(data, ..., .name_repair = "check_unique")

crossing(..., .name_repair = "check_unique")

nesting(..., .name_repair = "check_unique")
```

Arguments

`data`	A data frame.
`...`	Specification of columns to expand. Columns can be atomic vectors or lists. • To find all unique combinations of `x`, `y` and `z`, including those not present in the data, supply each variable as a separate argument: `expand(df,x,y,z)`. • To find only the combinations that occur in the data, use `nesting`: `expand(df,nesting(x,y,z))` • You can combine the two forms. For example, `expand(df,nesting(school_id,student_id),` would produce a row for each present school-student combination for all possible dates. When used with factors, `expand()` uses the full set of levels, not just those that appear in the data. If you want to use only the values seen in the data, use `forcats::fct_drop()`. When used with continuous variables, you may need to fill in values that do not appear in the data: to do so use expressions like `year = 2010:2020` or year = \link{full_seq}(year,1).
`.name_repair`	Treatment of problematic column names: • `"minimal"`: No name repair or checks, beyond basic existence, • `"unique"`: Make sure names are unique and not empty, • `"check_unique"`: (default value), no name repair, but check they are `unique`, • `"universal"`: Make the names `unique` and syntactic • a function: apply custom name repair (e.g., `.name_repair = make.names` for names in the style of base R). • A purrr-style anonymous function, see `rlang::as_function()` This argument is passed on as `repair` to `vctrs::vec_as_names()`. See there for more details on these terms and the strategies used to enforce them.

See Also

complete() to expand list objects. expand_grid() to input vectors rather than a data frame.

Examples

```
fruits <- tibble(
  type   = c("apple", "orange", "apple", "orange", "orange", "orange"),
  year   = c(2010, 2010, 2012, 2010, 2010, 2012),
  size  =  factor(
    c("XS", "S",  "M", "S", "S", "M"),
    levels = c("XS", "S", "M", "L")
  ),
  weights = rnorm(6, as.numeric(size) + 2)
)

# All possible combinations --------------------------------------
# Note that all defined, but not necessarily present, levels of the
# factor variable `size` are retained.
fruits %>% expand(type)
fruits %>% expand(type, size)
fruits %>% expand(type, size, year)

# Only combinations that already appear in the data ---------------
fruits %>% expand(nesting(type))
fruits %>% expand(nesting(type, size))
fruits %>% expand(nesting(type, size, year))

# Other uses -----------------------------------------------------------
# Use with `full_seq()` to fill in values of continuous variables
fruits %>% expand(type, size, full_seq(year, 1))
fruits %>% expand(type, size, 2010:2012)

# Use `anti_join()` to determine which observations are missing
all <- fruits %>% expand(type, size, year)
all
all %>% dplyr::anti_join(fruits)

# Use with `right_join()` to fill in missing rows
fruits %>% dplyr::right_join(all)
```

---

expand_grid *Create a tibble from all combinations of inputs*

Description

Create a tibble from all combinations of inputs

Usage

```
expand_grid(..., .name_repair = "check_unique")
```

Arguments

`...`	Name-value pairs. The name will become the column name in the output.
`.name_repair`	Treatment of problematic column names: • `"minimal"`: No name repair or checks, beyond basic existence, • `"unique"`: Make sure names are unique and not empty, • `"check_unique"`: (default value), no name repair, but check they are `unique`, • `"universal"`: Make the names `unique` and syntactic • a function: apply custom name repair (e.g., `.name_repair = make.names` for names in the style of base R). • A purrr-style anonymous function, see `rlang::as_function()` This argument is passed on as `repair` to `vctrs::vec_as_names()`. See there for more details on these terms and the strategies used to enforce them.

Value

A tibble with one column for each input in .... The output will have one row for each combination of the inputs, i.e. the size be equal to the product of the sizes of the inputs. This implies that if any input has length 0, the output will have zero rows.

Compared to expand.grid

- Varies the first element fastest.
- Never converts strings to factors.
- Does not add any additional attributes.
- Returns a tibble, not a data frame.
- Can expand any generalised vector, including data frames.

Examples

```
expand_grid(x = 1:3, y = 1:2)
expand_grid(l1 = letters, l2 = LETTERS)

# Can also expand data frames
expand_grid(df = data.frame(x = 1:2, y = c(2, 1)), z = 1:3)
# And matrices
expand_grid(x1 = matrix(1:4, nrow = 2), x2 = matrix(5:8, nrow = 2))
```

---

extract — *Extract a character column into multiple columns using regular expression groups*

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

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.
regex	a regular expression used to extract the desired values. There 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. NB: this will cause string "NA"s to be converted to NAs.
...	Additional arguments passed on to methods.

See Also

separate() to split up by a separator.

Examples

```
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]+)")
```

---

fill *Fill in missing values with previous or next value*

Description

Fills missing values in selected columns using the next or previous entry. This is useful in the common output format where values are not repeated, and are only recorded when they change.

Usage

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

Arguments

data	A data frame.
...	<tidy-select> Columns to fill.
.direction	Direction in which to fill missing values. Currently either "down" (the default), "up", "downup" (i.e. first down and then up) or "updown" (first up and then down).

Details

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

Examples

```
# Value (year) is recorded only when it changes
sales <- tibble::tribble(
  ~quarter, ~year, ~sales,
  "Q1",    2000,    66013,
  "Q2",      NA,    69182,
  "Q3",      NA,    53175,
  "Q4",      NA,    21001,
  "Q1",    2001,    46036,
  "Q2",      NA,    58842,
  "Q3",      NA,    44568,
  "Q4",      NA,    50197,
  "Q1",    2002,    39113,
  "Q2",      NA,    41668,
  "Q3",      NA,    30144,
  "Q4",      NA,    52897,
  "Q1",    2004,    32129,
  "Q2",      NA,    67686,
  "Q3",      NA,    31768,
  "Q4",      NA,    49094
)

# `fill()` defaults to replacing missing data from top to bottom
sales %>% fill(year)

# Value (pet_type) is missing above
tidy_pets <- tibble::tribble(
  ~rank, ~pet_type, ~breed,
  1L,        NA,    "Boston Terrier",
  2L,        NA,    "Retrievers (Labrador)",
  3L,        NA,    "Retrievers (Golden)",
  4L,        NA,    "French Bulldogs",
  5L,        NA,    "Bulldogs",
  6L,     "Dog",    "Beagles",
  1L,        NA,    "Persian",
  2L,        NA,    "Maine Coon",
  3L,        NA,    "Ragdoll",
  4L,        NA,    "Exotic",
  5L,        NA,    "Siamese",
```

```
  6L,     "Cat",    "American Short"
)

# For values that are missing above you can use `.direction = "up"`
tidy_pets %>%
  fill(pet_type, .direction = "up")

# Value (n_squirrels) is missing above and below within a group
squirrels <- tibble::tribble(
  ~group,    ~name,     ~role,     ~n_squirrels,
  1,      "Sam",    "Observer",   NA,
  1,     "Mara", "Scorekeeper",    8,
  1,    "Jesse",    "Observer",   NA,
  1,      "Tom",    "Observer",   NA,
  2,     "Mike",    "Observer",   NA,
  2,  "Rachael",    "Observer",   NA,
  2,  "Sydekea", "Scorekeeper",   14,
  2, "Gabriela",    "Observer",   NA,
  3,  "Derrick",    "Observer",   NA,
  3,     "Kara", "Scorekeeper",    9,
  3,    "Emily",    "Observer",   NA,
  3, "Danielle",    "Observer",   NA
)

# The values are inconsistently missing by position within the group
# Use .direction = "downup" to fill missing values in both directions
squirrels %>%
  dplyr::group_by(group) %>%
  fill(n_squirrels, .direction = "downup") %>%
  dplyr::ungroup()

# Using `.direction = "updown"` accomplishes the same goal in this example
```

`fish_encounters` *Fish encounters*

Description

Information about fish swimming down a river: each station represents an autonomous monitor that records if a tagged fish was seen at that location. Fish travel in one direction (migrating downstream). Information about misses is just as important as hits, but is not directly recorded in this form of the data.

Usage

```
fish_encounters
```

Format

A dataset with variables:

fish Fish identifier

station Measurement station

seen Was the fish seen? (1 if yes, and true for all rows)

Source

Dataset provided by Myfanwy Johnston; more details at https://fishsciences.github.io/post/visualizing-fish-encounter-histories/

`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

```
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` *Gather columns into key-value pairs*

Description

Retired

Development on `gather()` is complete, and for new code we recommend switching to `pivot_longer()`, which is easier to use, more featureful, and still under active development. `df %>% gather("key","value",x,y,z)` is equivalent to `df %>% pivot_longer(c(x,y,z),names_to = "key",values_to = "value")`

See more details in `vignette("pivot")`.

Usage

```
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 need to refer to contextual objects from a data expression, you can use all_of() or any_of(). These functions are used to select data-variables whose names are stored in a env-variable. For instance, all_of(a) selects the variables listed in the character vector a. For more details, see the tidyselect::select_helpers() documentation.

Examples

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

`hoist` *Rectangle a nested list into a tidy tibble*

Description

Maturing

`hoist()`, `unnest_longer()`, and `unnest_wider()` provide tools for rectangling, collapsing deeply nested lists into regular columns. `hoist()` allows you to selectively pull components of a list-column out in to their own top-level columns, using the same syntax as `purrr::pluck()`. `unnest_wider()` turns each element of a list-column into a column, and `unnest_longer()` turns each element of a list-column into a row. `unnest_auto()` picks between `unnest_wider()` or `unnest_longer()` based heuristics described below.

Learn more in `vignette("rectangle")`.

Usage

```
hoist(
  .data,
  .col,
  ...,
  .remove = TRUE,
  .simplify = TRUE,
  .ptype = list(),
  .transform = list()
)

unnest_longer(
  data,
  col,
  values_to = NULL,
  indices_to = NULL,
  indices_include = NULL,
  names_repair = "check_unique",
  simplify = TRUE,
  ptype = list(),
  transform = list()
)
```

```
unnest_wider(
  data,
  col,
  names_sep = NULL,
  simplify = TRUE,
  names_repair = "check_unique",
  ptype = list(),
  transform = list()
)

unnest_auto(data, col)
```

Arguments

`.data, data`	A data frame.
`.col, col`	List-column to extract components from.
`...`	Components of `.col` to turn into columns in the form `col_name = "pluck_specification"`. You can pluck by name with a character vector, by position with an integer vector, or with a combination of the two with a list. See `purrr::pluck()` for details. The column names must be unique in a call to `hoist()`, although existing columns with the same name will be overwritten. When plucking with a single string you can choose to omit the name, i.e. `hoist(df,col,"x")` is short-hand for `hoist(df,col,x = "x")`.
`.remove`	If `TRUE`, the default, will remove extracted components from `.col`. This ensures that each value lives only in one place.
`.simplify, simplify`	If `TRUE`, will attempt to simplify lists of length-1 vectors to an atomic vector
`.ptype, ptype`	Optionally, a named list of prototypes declaring the desired output type of each component. Use this argument if you want to check each element has the types you expect when simplifying.
`.transform, transform`	Optionally, a named list of transformation functions applied to each component. Use this function if you want transform or parse individual elements as they are hoisted.
`values_to`	Name of column to store vector values. Defaults to `col`.
`indices_to`	A string giving the name of column which will contain the inner names or position (if not named) of the values. Defaults to `col` with _id suffix
`indices_include`	Add an index column? Defaults to `TRUE` when `col` has inner names.
`names_repair`	Used to check that output data frame has valid names. Must be one of the following options: • "minimal": no name repair or checks, beyond basic existence, • "unique": make sure names are unique and not empty, • "check_unique": (the default), no name repair, but check they are unique, • "universal": make the names unique and syntactic • a function: apply custom name repair. • tidyr_legacy: use the name repair from tidyr 0.8.

- a formula: a purrr-style anonymous function (see `rlang::as_function()`)

See `vctrs::vec_as_names()` for more details on these terms and the strategies used to enforce them.

`names_sep` If `NULL`, the default, the names will be left as is. If a string, the inner and outer names will be paste together using `names_sep` as a separator.

Unnest variants

The three `unnest()` functions differ in how they change the shape of the output data frame:

- `unnest_wider()` preserves the rows, but changes the columns.
- `unnest_longer()` preserves the columns, but changes the rows
- `unnest()` can change both rows and columns.

These principles guide their behaviour when they are called with a non-primary data type. For example, if you `unnest_wider()` a list of data frames, the number of rows must be preserved, so each column is turned into a list column of length one. Or if you `unnest_longer()` a list of data frame, the number of columns must be preserved so it creates a packed column. I'm not sure how if these behaviours are useful in practice, but they are theoretically pleasing.

`unnest_auto()` heuristics

`unnest_auto()` inspects the inner names of the list-col:

- If all elements are unnamed, it uses `unnest_longer()`
- If all elements are named, and there's at least one name in common acros all components, it uses `unnest_wider()`
- Otherwise, it falls back to `unnest_longer(indices_include = TRUE)`.

Examples

```
df <- tibble(
  character = c("Toothless", "Dory"),
  metadata = list(
    list(
      species = "dragon",
      color = "black",
      films = c(
        "How to Train Your Dragon",
        "How to Train Your Dragon 2",
        "How to Train Your Dragon: The Hidden World"
       )
    ),
    list(
      species = "blue tang",
      color = "blue",
      films = c("Finding Nemo", "Finding Dory")
    )
  )
)
df

# Turn all components of metadata into columns
df %>% unnest_wider(metadata)
```

```
# Extract only specified components
df %>% hoist(metadata,
  "species",
  first_film = list("films", 1L),
  third_film = list("films", 3L)
)

df %>%
  unnest_wider(metadata) %>%
  unnest_longer(films)

# unnest_longer() is useful when each component of the list should
# form a row
df <- tibble(
  x = 1:3,
  y = list(NULL, 1:3, 4:5)
)
df %>% unnest_longer(y)
# Automatically creates names if widening
df %>% unnest_wider(y)
# But you'll usually want to provide names_sep:
df %>% unnest_wider(y, names_sep = "_")

# And similarly if the vectors are named
df <- tibble(
  x = 1:2,
  y = list(c(a = 1, b = 2), c(a = 10, b = 11, c = 12))
)
df %>% unnest_wider(y)
df %>% unnest_longer(y)
```

`nest` *Nest and unnest*

Description

Nesting creates a list-column of data frames; unnesting flattens it back out into regular columns. Nesting is implicitly a summarising operation: you get one row for each group defined by the non-nested columns. This is useful in conjunction with other summaries that work with whole datasets, most notably models.

Learn more in `vignette("nest")`.

Usage

```
nest(.data, ..., .names_sep = NULL, .key = deprecated())

unnest(
  data,
  cols,
  ...,
  keep_empty = FALSE,
```

```
  ptype = NULL,
  names_sep = NULL,
  names_repair = "check_unique",
  .drop = deprecated(),
  .id = deprecated(),
  .sep = deprecated(),
  .preserve = deprecated()
)
```

Arguments

.data	A data frame.
...	<tidy-select> Columns to nest, specified using name-variable pairs of the form new_col = c(col1,col2,col3). The right hand side can be any valid tidy select expression. **Deprecated**: previously you could write df %>% nest(x,y,z) and df %>% unnest(x,y,z). Convert to df %>% nest(data = c(x,y,z)). and df %>% unnest(c(x,y,z)). If you previously created new variable in unnest() you'll now need to do it explicitly with mutate(). Convert df %>% unnest(y = fun(x,y,z)) to df %>% mutate(y = fun(x,y,z)) %>% unnest(y).
.key	**Deprecated**: No longer needed because of the new new_col = c(col1,col2,col3) syntax.
data	A data frame.
cols	<tidy-select> Columns to unnest. If you unnest() multiple columns, parallel entries must be of compatible sizes, i.e. they're either equal or length 1 (following the standard tidyverse recycling rules).
keep_empty	By default, you get one row of output for each element of the list your unchopping/unnesting. This means that if there's a size-0 element (like NULL or an empty data frame), that entire row will be dropped from the output. If you want to preserve all rows, use keep_empty = TRUE to replace size-0 elements with a single row of missing values.
ptype	Optionally, supply a data frame prototype for the output cols, overriding the default that will be guessed from the combination of individual values.
names_sep, .names_sep	If NULL, the default, the names will be left as is. In nest(), inner names will come from the former outer names; in unnest(), the new outer names will come from the inner names. If a string, the inner and outer names will be used together. In nest(), the names of the new outer columns will be formed by pasting together the outer and the inner column names, separated by names_sep. In unnest(), the new inner names will have the outer names (+ names_sep) automatically stripped. This makes names_sep roughly symmetric between nesting and unnesting.
names_repair	Used to check that output data frame has valid names. Must be one of the following options: • "minimal": no name repair or checks, beyond basic existence, • "unique": make sure names are unique and not empty, • "check_unique": (the default), no name repair, but check they are unique, • "universal": make the names unique and syntactic

- a function: apply custom name repair.
- tidyr_legacy: use the name repair from tidyr 0.8.
- a formula: a purrr-style anonymous function (see `rlang::as_function()`)

See `vctrs::vec_as_names()` for more details on these terms and the strategies used to enforce them.

`.drop, .preserve`
: **Deprecated**: all list-columns are now preserved; If there are any that you don't want in the output use `select()` to remove them prior to unnesting.

`.id`
: **Deprecated**: convert `df %>% unnest(x, .id = "id")` to df %>% mutate(id = names(x)) %>% unnest(

`.sep`
: **Deprecated**: use `names_sep` instead.

New syntax

tidyr 1.0.0 introduced a new syntax for `nest()` and `unnest()` that's designed to be more similar to other functions. Converting to the new syntax should be straightforward (guided by the message you'll recieve) but if you just need to run an old analysis, you can easily revert to the previous behaviour using `nest_legacy()` and `unnest_legacy()` as follows:

```
library(tidyr)
nest <- nest_legacy
unnest <- unnest_legacy
```

Grouped data frames

`df %>% nest(data = c(x,y))` specifies the columns to be nested; i.e. the columns that will appear in the inner data frame. Alternatively, you can `nest()` a grouped data frame created by `dplyr::group_by()`. The grouping variables remain in the outer data frame and the others are nested. The result preserves the grouping of the input.

Variables supplied to `nest()` will override grouping variables so that `df %>% group_by(x,y) %>% nest(data = -z)` will be equivalent to `df %>% nest(data = -z)`.

Examples

```
df <- tibble(x = c(1, 1, 1, 2, 2, 3), y = 1:6, z = 6:1)
# Note that we get one row of output for each unique combination of
# non-nested variables
df %>% nest(data = c(y, z))
# chop does something similar, but retains individual columns
df %>% chop(c(y, z))

# use tidyselect syntax and helpers, just like in dplyr::select()
df %>% nest(data = one_of("y", "z"))

iris %>% nest(data = -Species)
nest_vars <- names(iris)[1:4]
iris %>% nest(data = one_of(nest_vars))
iris %>%
  nest(petal = starts_with("Petal"), sepal = starts_with("Sepal"))
iris %>%
  nest(width = contains("Width"), length = contains("Length"))

# Nesting a grouped data frame nests all variables apart from the group vars
library(dplyr)
```

```
fish_encounters %>%
  group_by(fish) %>%
  nest()

# Nesting is often useful for creating per group models
mtcars %>%
  group_by(cyl) %>%
  nest() %>%
  mutate(models = lapply(data, function(df) lm(mpg ~ wt, data = df)))

# unnest() is primarily designed to work with lists of data frames
df <- tibble(
  x = 1:3,
  y = list(
    NULL,
    tibble(a = 1, b = 2),
    tibble(a = 1:3, b = 3:1)
  )
)
df %>% unnest(y)
df %>% unnest(y, keep_empty = TRUE)

# If you have lists of lists, or lists of atomic vectors, instead
# see hoist(), unnest_wider(), and unnest_longer()

#' # You can unnest multiple columns simultaneously
df <- tibble(
 a = list(c("a", "b"), "c"),
 b = list(1:2, 3),
 c = c(11, 22)
)
df %>% unnest(c(a, b))

# Compare with unnesting one column at a time, which generates
# the Cartesian product
df %>% unnest(a) %>% unnest(b)
```

`nest_legacy` *Legacy versions of* `nest()` *and* `unnest()`

Description

Retired

tidyr 1.0.0 introduced a new syntax for `nest()` and `unnest()`. The majority of existing usage should be automatically translated to the new syntax with a warning. However, if you need to quickly roll back to the previous behaviour, these functions provide the previous interface. To make old code work as is, add the following code to the top of your script:

```
library(tidyr)
nest <- nest_legacy
unnest <- unnest_legacy
```

Usage

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

unnest_legacy(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.
.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).
.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 the 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")`.

Examples

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

# nesting ------------------------------------------------------------------
as_tibble(iris) %>% nest_legacy(-Species)
as_tibble(chickwts) %>% nest_legacy(weight)

# unnesting ----------------------------------------------------------------
df <- tibble(
  x = 1:2,
  y = list(
    tibble(z = 1),
    tibble(z = 3:4)
  )
)
df %>% unnest_legacy(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_legacy(a, b)
# If you omit the column names, it'll unnest all list-cols
df %>% unnest_legacy()
```

---

pack *Pack and unpack*

Description

Maturing

Packing and unpacking preserve the length of a data frame, changing its width. `pack()` makes `df` narrow by collapsing a set of columns into a single df-column. `unpack()` makes `data` wider by expanding df-columns back out into individual columns.

Usage

```
pack(.data, ..., .names_sep = NULL)

unpack(data, cols, names_sep = NULL, names_repair = "check_unique")
```

Arguments

`...`	<tidy-select> Columns to pack, specified using name-variable pairs of the form `new_col = c(col1,col2,col3)`. The right hand side can be any valid tidy select expression.
`data, .data`	A data frame.
`cols`	<tidy-select> Column to unpack.
`names_sep, .names_sep`	If `NULL`, the default, the names will be left as is. In `pack()`, inner names will come from the former outer names; in `unpack()`, the new outer names will come from the inner names. If a string, the inner and outer names will be used together. In `pack()`, the names of the new outer columns will be formed by pasting together the outer and the inner column names, separated by `names_sep`. In `unpack()`, the new inner names will have the outer names (+ `names_sep`) automatically stripped. This makes `names_sep` roughly symmetric between packing and unpacking.
`names_repair`	Used to check that output data frame has valid names. Must be one of the following options: • "minimal": no name repair or checks, beyond basic existence, • "unique": make sure names are unique and not empty, • "check_unique": (the default), no name repair, but check they are unique, • "universal": make the names unique and syntactic • a function: apply custom name repair. • tidyr_legacy: use the name repair from tidyr 0.8. • a formula: a purrr-style anonymous function (see `rlang::as_function()`) See `vctrs::vec_as_names()` for more details on these terms and the strategies used to enforce them.

Details

Generally, unpacking is more useful than packing because it simplifies a complex data structure. Currently, few functions work with df-cols, and they are mostly a curiosity, but seem worth exploring further because they mimic the nested column headers that are so popular in Excel.

Examples

```
# Packing ========================================================
# It's not currently clear why you would ever want to pack columns
# since few functions work with this sort of data.
df <- tibble(x1 = 1:3, x2 = 4:6, x3 = 7:9, y = 1:3)
df
df %>% pack(x = starts_with("x"))
df %>% pack(x = c(x1, x2, x3), y = y)

# .names_sep allows you to strip off common prefixes; this
# acts as a natural inverse to name_sep in unpack()
iris %>%
  as_tibble() %>%
  pack(
    Sepal = starts_with("Sepal"),
    Petal = starts_with("Petal"),
    .names_sep = "."
  )

# Unpacking ======================================================
df <- tibble(
  x = 1:3,
  y = tibble(a = 1:3, b = 3:1),
  z = tibble(X = c("a", "b", "c"), Y = runif(3), Z = c(TRUE, FALSE, NA))
)
df
df %>% unpack(y)
df %>% unpack(c(y, z))
df %>% unpack(c(y, z), names_sep = "_")
```

---

pivot_longer *Pivot data from wide to long*

Description

Maturing

`pivot_longer()` "lengthens" data, increasing the number of rows and decreasing the number of columns. The inverse transformation is `pivot_wider()`

Learn more in `vignette("pivot")`.

Usage

```
pivot_longer(
  data,
  cols,
  names_to = "name",
```

```
  names_prefix = NULL,
  names_sep = NULL,
  names_pattern = NULL,
  names_ptypes = list(),
  names_transform = list(),
  names_repair = "check_unique",
  values_to = "value",
  values_drop_na = FALSE,
  values_ptypes = list(),
  values_transform = list(),
  ...
)
```

Arguments

`data`	A data frame to pivot.
`cols`	<tidy-select> Columns to pivot into longer format.
`names_to`	A string specifying the name of the column to create from the data stored in the column names of `data`. Can be a character vector, creating multiple columns, if `names_sep` or `names_pattern` is provided. In this case, there are two special values you can take advantage of: • `NA` will discard that component of the name. • `.value` indicates that component of the name defines the name of the column containing the cell values, overriding `values_to`.
`names_prefix`	A regular expression used to remove matching text from the start of each variable name.
`names_sep, names_pattern`	If `names_to` contains multiple values, these arguments control how the column name is broken up. `names_sep` takes the same specification as `separate()`, and can either be a numeric vector (specifying positions to break on), or a single string (specifying a regular expression to split on). `names_pattern` takes the same specification as `extract()`, a regular expression containing matching groups (()). If these arguments do not give you enough control, use `pivot_longer_spec()` to create a spec object and process manually as needed.
`names_ptypes, values_ptypes`	A list of column name-prototype pairs. A prototype (or ptype for short) is a zero-length vector (like `integer()` or `numeric()`) that defines the type, class, and attributes of a vector. Use these arguments to confirm that the created columns are the types that you expect. If not specified, the type of the columns generated from `names_to` will be character, and the type of the variables generated from `values_to` will be the common type of the input columns used to generate them.
`names_transform, values_transform`	A list of column name-function pairs. Use these arguments if you need to change the type of specific columns. For example, `names_transform = list(week = as.integer)` would convert a character week variable to an integer.
`names_repair`	What happens if the output has invalid column names? The default, `"check_unique"` is to error if the columns are duplicated. Use `"minimal"` to allow duplicates

	in the output, or "unique" to de-duplicated by adding numeric suffixes. See vctrs::vec_as_names() for more options.
values_to	A string specifying the name of the column to create from the data stored in cell values. If names_to is a character containing the special .value sentinel, this value will be ignored, and the name of the value column will be derived from part of the existing column names.
values_drop_na	If TRUE, will drop rows that contain only NAs in the value_to column. This effectively converts explicit missing values to implicit missing values, and should generally be used only when missing values in data were created by its structure.
...	Additional arguments passed on to methods.

Details

pivot_longer() is an updated approach to gather(), designed to be both simpler to use and to handle more use cases. We recommend you use pivot_longer() for new code; gather() isn't going away but is no longer under active development.

Examples

```
# See vignette("pivot") for examples and explanation

# Simplest case where column names are character data
relig_income
relig_income %>%
  pivot_longer(-religion, names_to = "income", values_to = "count")

# Slightly more complex case where columns have common prefix,
# and missing missings are structural so should be dropped.
billboard
billboard %>%
 pivot_longer(
   cols = starts_with("wk"),
   names_to = "week",
   names_prefix = "wk",
   values_to = "rank",
   values_drop_na = TRUE
 )

# Multiple variables stored in column names
who %>% pivot_longer(
  cols = new_sp_m014:newrel_f65,
  names_to = c("diagnosis", "gender", "age"),
  names_pattern = "new_?(.*)_(.)(.*)",
  values_to = "count"
)

# Multiple observations per row
anscombe
anscombe %>%
 pivot_longer(everything(),
   names_to = c(".value", "set"),
   names_pattern = "(.)(.)"
 )
```

pivot_wider *Pivot data from long to wide*

Description

Maturing

pivot_wider() "widens" data, increasing the number of columns and decreasing the number of rows. The inverse transformation is pivot_longer().

Learn more in vignette("pivot").

Usage

```
pivot_wider(
  data,
  id_cols = NULL,
  names_from = name,
  names_prefix = "",
  names_sep = "_",
  names_glue = NULL,
  names_sort = FALSE,
  names_repair = "check_unique",
  values_from = value,
  values_fill = NULL,
  values_fn = NULL,
  ...
)
```

Arguments

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

names_sort	Should the column names be sorted? If FALSE, the default, column names are ordered by first appearance.
names_repair	What happens if the output has invalid column names? The default, "check_unique" is to error if the columns are duplicated. Use "minimal" to allow duplicates in the output, or "unique" to de-duplicated by adding numeric suffixes. See vctrs::vec_as_names() for more options.
values_fill	Optionally, a (scalar) value that specifies what each value should be filled in with when missing. This can be a named list if you want to apply different aggregations to different value columns.
values_fn	Optionally, a function applied to the value in each cell in the output. You will typically use this when the combination of id_cols and value column does not uniquely identify an observation. This can be a named list if you want to apply different aggregations to different value columns.
...	Additional arguments passed on to methods.

Details

pivot_wider() is an updated approach to spread(), designed to be both simpler to use and to handle more use cases. We recommend you use pivot_wider() for new code; spread() isn't going away but is no longer under active development.

See Also

pivot_wider_spec() to pivot "by hand" with a data frame that defines a pivotting specification.

Examples

```
# See vignette("pivot") for examples and explanation

fish_encounters
fish_encounters %>%
  pivot_wider(names_from = station, values_from = seen)
# Fill in missing values
fish_encounters %>%
  pivot_wider(names_from = station, values_from = seen, values_fill = 0)

# Generate column names from multiple variables
us_rent_income
us_rent_income %>%
  pivot_wider(names_from = variable, values_from = c(estimate, moe))

# When there are multiple `names_from` or `values_from`, you can use
# use `names_sep` or `names_glue` to control the output variable names
us_rent_income %>%
  pivot_wider(
    names_from = variable,
    names_sep = ".",
    values_from = c(estimate, moe)
  )
us_rent_income %>%
  pivot_wider(
    names_from = variable,
```

```
    names_glue = "{variable}_{.value}",
    values_from = c(estimate, moe)
  )

# Can perform aggregation with values_fn
warpbreaks <- as_tibble(warpbreaks[c("wool", "tension", "breaks")])
warpbreaks
warpbreaks %>%
  pivot_wider(
    names_from = wool,
    values_from = breaks,
    values_fn = mean
  )
```

`relig_income` *Pew religion and income survey*

Description

Pew religion and income survey

Usage

```
relig_income
```

Format

A dataset with variables:

religion Name of religion

<$10k-Don\'t know/refused Number of respondees with income range in column name

Source

Downloaded from https://www.pewforum.org/religious-landscape-study/ (downloaded November 2009)

`replace_na` *Replace NAs with specified values*

Description

Replace NAs with specified values

Usage

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

Arguments

data	A data frame or vector.
replace	If data is a data frame, replace takes a list of values, with one value for each column that has NA values to be replaced. If data is a vector, replace takes a single value. This single value replaces all of the NA values in the vector.
...	Additional arguments for methods. Currently unused.

Value

- If data is a data frame, replace_na() returns a data frame.
- If data is a vector, replace_na() returns a vector, with class given by the union of data and replace.

See Also

dplyr::na_if() to replace specified values with NAs; dplyr::coalesce() to replaces NAs with values from other vectors.

Examples

```
# Replace NAs in a data frame
df <- tibble(x = c(1, 2, NA), y = c("a", NA, "b"))
df %>% replace_na(list(x = 0, y = "unknown"))

# Replace NAs in a vector
df %>% dplyr::mutate(x = replace_na(x, 0))
# OR
df$x %>% replace_na(0)
df$y %>% replace_na("unknown")

# Replace NULLs in a list: NULLs are the list-col equivalent of NAs
df_list <- tibble(z = list(1:5, NULL, 10:20))
df_list %>% replace_na(list(z = list(5)))
```

---

separate *Separate a character column into multiple columns with a regular expression or numeric locations*

Description

Given either a 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

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, sep is interpreted as a regular expression. The default value is a regular expression that matches any sequence of non-alphanumeric values. If numeric, sep is interpreted as character 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 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. NB: this will cause string "NA"s to be converted to NAs.
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. • "drop": drop any extra values without a warning. • "merge": only splits at most `length(into)` times
fill	If sep is a character vector, this controls what happens when there are not enough pieces. There are three valid options: • "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
...	Additional arguments passed on to methods.

See Also

`unite()`, the complement, `extract()` which uses regular expression capturing groups.

Examples

```
library(dplyr)
# If you want to split by any non-alphanumeric value (the default):
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 as previous, but drops the c without warnings:
df %>% separate(x, c("a", "b"), extra = "drop", fill = "right")
# Opposite of previous, keeping the c and filling left:
df %>% separate(x, c("a", "b"), extra = "merge", fill = "left")
# Or you can keep all three:
df %>% separate(x, c("a", "b", "c"))

# To only split a 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")

# Use regular expressions to separate on multiple characters:
df <- data.frame(x = c(NA, "a?b", "a.d", "b:c"))
df %>% separate(x, c("A","B"), sep = "([.?:])")

# convert = TRUE detects column classes:
df <- data.frame(x = c("a:1", "a:2", "c:4", "d", NA))
df %>% separate(x, c("key","value"), ":") %>% str
df %>% separate(x, c("key","value"), ":", convert = TRUE) %>% str
```

`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)
```

Arguments

`data`	A data frame.
`...`	<tidy-select> Columns to separate across multiple rows
`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.

Examples

```
df <- tibble(
  x = 1:3,
  y = c("a", "d,e,f", "g,h"),
  z = c("1", "2,3,4", "5,6")
)
separate_rows(df, y, z, convert = TRUE)
```

smiths *Some data about the Smith family*

Description

A small demo dataset describing John and Mary Smith.

Usage

```
smiths
```

Format

A data frame with 2 rows and 5 columns.

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

Description

Retired

Development on `spread()` is complete, and for new code we recommend switching to `pivot_wider()`, which is easier to use, more featureful, and still under active development. `df %>% spread(key, value)` is equivalent to `df %>% pivot_wider(names_from = key, values_from = value)`

See more details in `vignette("pivot")`.

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

```
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
```

---

table1 *Example tabular representations*

Description

Data sets that demonstrate multiple ways to layout the same tabular data.

Usage

```
table1

table2

table3

table4a

table4b

table5
```

Details

`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

https://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(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 *Unite multiple columns into one by pasting strings together*

Description

Convenience function to paste together multiple columns into one.

Usage

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

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).
...	<tidy-select> Columns to unite
sep	Separator to use between values.
remove	If TRUE, remove input columns from output data frame.
na.rm	If TRUE, missing values will be remove prior to uniting each value.

See Also

separate(), the complement.

Examples

```
df <- expand_grid(x = c("a", NA), y = c("b", NA))
df

df %>% unite("z", x:y, remove = FALSE)
# To remove missing values:
df %>% unite("z", x:y, na.rm = TRUE, remove = FALSE)

# Separate is almost the complement of unite
df %>%
  unite("xy", x:y) %>%
  separate(xy, c("x", "y"))
# (but note `x` and `y` contain now "NA" not NA)
```

---

us_rent_income *US rent and income data*

Description

Captured from the 2017 American Community Survey using the tidycensus package.

Usage

```
us_rent_income
```

Format

A dataset with variables:

GEOID FIP state identifier

NAME Name of state

variable Variable name: income = median yearly income, rent = median monthly rent

estimate Estimated value

moe 90% margin of error

who *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

`who`: a data frame with 7,240 rows and the columns:

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).

`population`: a data frame with 4,060 rows and three columns:

country Country name

year Year

population Population

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 (`014` = 0-14 yrs of age, `1524` = 15-24 years of age, `2534` = 25 to 34 years of age, `3544` = 35 to 44 years of age, `4554` = 45 to 54 years of age, `5564` = 55 to 64 years of age, `65` = 65 years of age or older).

Source

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

`world_bank_pop` *Population data from the world bank*

Description

Data about population from the World Bank.

Usage

```
world_bank_pop
```

Format

A dataset with variables:

country Three letter country code

indicator Indicator name: `SP.POP.GROW` = population growth, `SP.POP.TOTL` = total population, `SP.URB.GROW` = urban population growth, `SP.URB.TOTL` = total urban population

2000-2018 Value for each year

Source

Dataset from the World Bank data bank: https://data.worldbank.org

Index