Package 'unpivotr'

January 2, 2019

Title Unpivot Complex and Irregular Data Layouts

Version 0.5.0

Description Tools for converting data from complex or irregular layouts to a columnar structure. For example, tables with multilevel column or row headers, or spreadsheets. Header and data cells are selected by their contents and position, as well as formatting and comments where available, and are associated with one other by their proximity in given directions. Functions for data frames and HTML tables are provided.

Depends R (>= 3.2.0)

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Imports methods, rlang, magrittr, dplyr, forcats, purrr, tidyr, pillar, tibble, cellranger, xml2

URL https://github.com/nacnudus/unpivotr

BugReports https://github.com/nacnudus/unpivotr/issues

RoxygenNote 6.1.1

Suggests knitr, rmarkdown, readr, tidyxl, readxl, stringr, htmltools, rvest, selectr, DT, testthat, covr

VignetteBuilder knitr

NeedsCompilation no

Author Duncan Garmonsway [aut, cre]

Maintainer Duncan Garmonsway <nacnudus@gmail.com>

Repository CRAN

Date/Publication 2019-01-02 10:00:03 UTC

R topics documented:

`unpivotr-package` *Un-pivot complex and irregular data layouts.*

Description

'Unpivotr' provides tools for converting data from complex or irregular layouts to a columnar structure. For example, tables with multi-level column or row headers, or spreadsheets of several tables, nested HTML tables, and data that uses several different sentinel values.

Details

The best way to learn unpivotr is the free online book Spreadsheet Munging Strategies.

Header and data cells can selected by their contents, position, data type and formatting, and can be associated with one other by their relative positions.

The input data must be a data frame with the columns 'row' and 'col' to describe the position of a 'cell' of data. For cells that are to be interpreted as data, further columns containing the 'value' of the cell are, of course, necessary for there to be any point in using this package, though they are not actually required for any of the given functions.

Data frames and HTML tables can be converted into a format meeting these requirements by using the `as_cells()` function. Excel (.xlsx) files can be imported directly into the required format with the 'tidyxl' package, available at `https://github.com/nacnudus/tidyxl`, which has the advantage that it retains cell formatting and comments.

Author(s)

Maintainer: Duncan Garmonsway <nacnudus@gmail.com>

See Also

Useful links:

- https://github.com/nacnudus/unpivotr
- Report bugs at https://github.com/nacnudus/unpivotr/issues

`as_cells` *Tokenize data frames into a tidy 'melted' structure*

Description

Data frames represent data in a tabular structure. `as_cells()` takes the row and column position of each 'cell', and returns that information in a new data frame, alongside the content and type of each cell.

This makes it easier to deal with complex or non-tabular data (e.g. pivot tables) that have been imported into R as data frames. Once they have been 'melted' by `as_cells()`, you can use functions like `behead()` and `spatter()` to reshape them into conventional, tidy, unpivoted structures.

For HTML tables, the content of each cell is returned as a standalone HTML string that can be further parsed with tools such as the rvest package. This is particularly useful when an HTML cell itself contains an HTML table, or contains both text and a URL. If the HTML itself is poorly formatted, try passing it through the htmltidy package first.

This is an S3 generic.

Usage

```
as_cells(x, row_names = FALSE, col_names = FALSE)
```

Arguments

`x`	A data.frame or an HTML document
`row_names`	Whether to treat the row names as cells, Default: FALSE
`col_names`	Whether to treat the column names as cells, Default: FALSE

Details

For certain non-rectangular data formats, it can be useful to parse the data into a melted format where each row represents a single token.

Value

A data.frame with the following columns:

- `row` and `col` (integer) giving the original position of the 'cells'
- any relevant columns for cell values in their original types: `chr`, `cplx`, `cplx`, `dbl`, `fct`, `int`, `lgl`, `list`, and `ord`

- data_type to specify for each cell which of the above columns (chr etc.) the value is in.

The columns fct and ord are, like list, list-columns (each element is independent) to avoid factor levels clashing. For HTML tables, the column html gives the HTML string of the original cell.

Row and column names, when present and required by row_names = TRUE or col_names = TRUE, are treated as though they were cells in the table, and they appear in the chr column.

Examples

```
x <- data.frame(a = c(10, 20),
                b = c("foo", "bar"),
                stringsAsFactors = FALSE)
x
as_cells(x)
as_cells(x, row_names = TRUE)
as_cells(x, col_names = TRUE)

# 'list' columns are undisturbed
y <- data.frame(a = c("a", "b"), stringsAsFactors = FALSE)
y$b <- list(1:2, 3:4)
y
as_cells(y)

# Factors are preserved by being wrapped in lists so that their levels don't
# conflict.  Blanks are NULLs.
z <- data.frame(x = factor(c("a", "b")),
                y = factor(c("c", "d"), ordered = TRUE))
as_cells(z)
as_cells(z)$fct
as_cells(z)$ord

# HTML tables can be extracted from the output of xml2::read_html().  These
# are returned as a list of tables, similar to rvest::html_table().  The
# value of each cell is its standalone HTML string, which can contain
# anything -- even another table.

colspan <- system.file("extdata", "colspan.html", package = "unpivotr")
rowspan <- system.file("extdata", "rowspan.html", package = "unpivotr")
nested <- system.file("extdata", "nested.html", package = "unpivotr")

## Not run:
browseURL(colspan)
browseURL(rowspan)
browseURL(nestedspan)

## End(Not run)

as_cells(xml2::read_html(colspan))
as_cells(xml2::read_html(rowspan))
as_cells(xml2::read_html(nested))
```

behead *Strip a level of headers from a pivot table*

Description

`behead()` takes one level of headers from a pivot table and makes it part of the data. Think of it like `tidyr::gather()`, except that it works when there is more than one row of headers (or more than one column of row-headers), and it only works on tables that have first come through `as_cells()` or `tidyxl::xlsx_cells()`.

Usage

```
behead(cells, direction, name, values = NULL, types = data_type,
  formatters = list(), drop_na = TRUE)

behead_if(cells, ..., direction, name, values = NULL,
  types = data_type, formatters = list(), drop_na = TRUE)
```

Arguments

`cells`	Data frame. The cells of a pivot table, usually the output of `as_cells()` or `tidyxl::xlsx_cells()`, or of a subsequent operation on those outputs.
`direction`	The direction between a data cell and its header, one of `"N"`, `"E"`, `"S"`, `"W"`, `"NNW"`, `"NNE"`, `"ENE"`, `"ESE"`, `"SSE"`, `"SSW"`. `"WSW"` and `"WNW"`. See 'details'. `"ABOVE"`, `"BELOW"`, `"LEFT"` and `"RIGHT"` aren't available because they require certain ambiguities that are better handled by using `enhead()` directly rather than via `behead()`.
`name`	A name to give the new column that will be created, e.g. `"location"` if the headers are locations. Quoted (`"location"`, not `location`) because it doesn't refer to an actual object.
`values`	Optional. The column of `cells` to use as the values of each header. Given as a bare variable name. If omitted (the default), the `types` argument will be used instead.
`types`	The name of the column that names the data type of each cell. Usually called `data_types` (the default), this is a character column that names the other columns in `cells` that contain the values of each cell. E.g. a cell with a character value will have `"character"` in this column. Unquoted(`data_types`, not `"data_types"`) because it refers to an actual object.
`formatters`	A named list of functions for formatting each data type in a set of headers of mixed data types, e.g. when some headers are dates and others are characters. These can be given as `character = toupper` or `character = ~ toupper(.x)`, similar to purrr::map.
`drop_na`	logical Whether to filter out headers that have `NA` in the `value` column. Default: `TRUE`. This can happen with the output of `tidyxl::xlsx_cells()`, when an empty cell exists because it has formatting applied to it, but should be ignored.

... Passed to dplyr::filter. logical predicates defined in terms of the variables in .data. Multiple conditions are combined with &. Only rows where the condition evaluates to TRUE are kept.

The arguments in ... are automatically quoted and evaluated in the context of the data frame. They support unquoting and splicing. See the dplyr vignette("programming") for an introduction to these concepts.

Value

A data frame

Examples

```
# A simple table with a row of headers
(x <- data.frame(a = 1:2, b = 3:4))

# Make a tidy representation of each cell
(cells <- as_cells(x, col_names = TRUE))

# Strip the cells in row 1 (the original headers) and use them as data
behead(cells, "N", foo)

# More complex example: pivot table with several layers of headers
(x <- purpose$`NNW WNW`)

# Make a tidy representation
cells <- as_cells(x)
head(cells)
tail(cells)

# Strip the headers and make them into data
tidy <-
  cells %>%
  behead("NNW", Sex) %>%
  behead("N", `Sense of purpose`) %>%
  behead("WNW", `Highest qualification`) %>%
  behead("W", `Age group (Life-stages)`) %>%
  dplyr::mutate(count = as.integer(chr)) %>%
  dplyr::select(-row, -col, -data_type, -chr)
head(tidy)

# Check against the provided 'tidy' version of the data.
dplyr::anti_join(tidy, purpose$Tidy)

# The provided 'tidy' data is missing a row for Male 15-24-year-olds with a
# postgraduate qualification and a sense of purpose between 0 and 6.  That
# seems to have been an oversight by Statistics New Zealand.

cells <- tibble::tribble(
       ~X1, ~adult, ~juvenile,
    "LION",    855,       677,
    "male",    496,       322,
```

```
  "female",    359,       355,
   "TIGER",    690,       324,
    "male",    381,       222,
  "female",    309,       102
  )
cells <- as_cells(cells, col_names = TRUE)

cells %>%
  behead_if(chr == toupper(chr), direction = "WNW", name = "species") %>%
  behead("W", "sex") %>%
  behead("N", "age") %>%
  dplyr::select(species, sex, age, population = dbl)
```

---

enhead *Join data cells to headers*

Description

Data cells in a table are associated with header cells by proximity. `enhead()` joins a data frame of data cells to a data frame of header cells, choosing the nearest header cells in the given direction.

Usage

```
enhead(data_cells, header_cells, direction, drop = TRUE)
```

Arguments

`data_cells`	Data frame of data cells with at least the columns 'row' and 'column', which are `numeric` or `integer`.
`header_cells`	Data frame of header cells with at least the columns 'row' and 'column', which are numeric/integer vectors.
`direction`	The direction between a data cell and its header, one of `"N"`, `"E"`, `"S"`, `"W"`, `"NNW"`, `"NNE"`, `"ENE"`, `"ESE"`, `"SSE"`, `"SSW"`. `"WSW"`, `"WNW"`, `"ABOVE"`, `"BELOW"`, `"LEFT"` and `"RIGHT"`. See 'details'.
`drop`	Logical vector length 1. Whether data cells that can't be associated with a header should be dropped. Default: `TRUE`.

Details

Headers are associated with data by proximity in a given direction. The directions are mapped to the points of the compass, where 'N' is north (up), 'E' is east (right), and so on. `enhead()` finds the nearest header to a given data cell in a given direction, and joins it to the data cell.

The most common directions to search are `"NNW"` (for left-aligned headers at the top of the table) and `"WNW"` for top-aligned headers at the side of the table.

The full list of available directions is `"N"`, `"E"`, `"S"`, `"W"`, `"NNW"`, `"NNE"`, `"ENE"`, `"ESE"`, `"SSE"`, `"SSW"`, `"WSW"`, `"WNW"`, `"ABOVE"`, `"BELOW"`, `"LEFT"`, `"RIGHT"`. For convenience, these directions are provided as their own functions, wrapping the concept of `enhead()`.

The difference between "N" and "ABOVE" (and similar pairs of directions) is that "N" finds headers directly above the data cell, whereas "ABOVE" matches the nearest header, whether above-left, above-right or directly above the data cell. This is useful for matching headers that are not aligned to the edge of the data cells that they refer to. There can be a tie in the directions "ABOVE", "BELOW", "LEFT" and "RIGHT" , causing NAs to be returned in the place of header values. Avoid ties by using justify() first to align header cells to the corner of the data cells they describe.

Examples

```
library(dplyr)
# Load some pivoted data
(x <- purpose$`NNW WNW`)
# Make a tidy representation
cells <- as_cells(x)
cells <- cells[!is.na(cells$chr), ]
head(cells)
# Select the cells containing the values
data_cells <-
  filter(cells, row >= 3, col >= 3) %>%
  transmute(row, col, count = as.integer(chr))
head(data_cells)
# Select the headers
qualification <-
  filter(cells, col == 1) %>%
  select(row, col, qualification = chr)
age <-
  filter(cells, col == 2) %>%
  select(row, col, age = chr)
gender <-
  filter(cells, row == 1) %>%
  select(row, col, gender = chr)
satisfaction <-
  filter(cells, row == 2) %>%
  select(row, col, satisfaction = chr)
# From each data cell, search for the nearest one of each of the headers
data_cells %>%
  enhead(gender, "NNW") %>%
  enhead(satisfaction, "N") %>%
  enhead(qualification, "WNW") %>%
  enhead(age, "W") %>%
  select(-row, -col)

# The `drop` argument controls what happens when for some cells there is no
# header in the given direction. When `drop = TRUE` (the default), cells that
# can't be joined to a header are dropped.  Otherwise they are kept.
enhead(data_cells, gender, "N")
enhead(data_cells, gender, "N", drop = FALSE)
```

---

isolate_sentinels *Move sentinel values into a separate column leaving NA behind*

Description

A sentinel value, takes the place of a value that isn't available for some reason. `isolate_sentinels()` removes these values from a column of data into a separate column, and optionally converts the data left behind into an appropriate data type.

Usage

```
isolate_sentinels(.data, col, sentinels, into = "sentinel")
```

Arguments

`.data`	A data frame.
`col`	The name of the column of data containing sentinel values.
`sentinels`	A vector of sentinel values to be removed.
`into`	A name to give the new column of sentinel values.

Examples

```
x <- data.frame(name = c("Matilda", "Nicholas", "Olivia", "Paul"),
                score = c(10, "confidential", "N/A", 12),
                stringsAsFactors = FALSE)
x
isolate_sentinels(x, score, c("confidential", "N/A"))
isolate_sentinels(x, score, c("confidential", "N/A"), "flag")
```

`justify` *Align one set of cells with another set*

Description

If the header cells of a table aren't aligned to the left, right, top or bottom of the data cells that they describe, then use `justify()` to re-align them, using a second set of cells as a guide.

Usage

```
justify(header_cells, corner_cells)
```

Arguments

`header_cells`	Data frame of data cells with at least the columns 'row' and 'column', which are `numeric` or `integer`.
`corner_cells`	Data frame of header cells with at least the columns 'row' and 'column', which are numeric/integer vectors. The same length as `header_cells`.

Examples

```
header_cells <- tibble::tibble(row = c(1L, 1L, 1L, 1L),
                               col = c(3L, 5L, 8L, 10L),
                               value = LETTERS[1:4])
corner_cells <- tibble::tibble(row = c(2L, 2L, 2L, 2L),
                               col = c(1L, 4L, 6L, 9L))
justify(header_cells, corner_cells)
```

`merge_cells` *Merge cell values into a single cell by rows or columns*

Description

When a single column header is split across cells, merge the cells with `merge_rows()` or `merge_cols()`. E.g. if a column header "Mean GDP" is split over two cells, where the top cell has the value "Mean" and the bottom cell has the value "GDP", then `merge_rows()` will combine them into a single cell with the value "Mean GDP".

`merge_rows()` keeps the top cell, and `merge_cols()` keeps the left-most cell. When there are several columns of headers, `merge_rows()` aligns the output cells so that they are all in the same row, and similarly `merge_cols()` aligns to the same column.

These functions apply only to cells with character values because it doesn't make sense to concatenate non-character values. Convert cell values to characters first if you need to merge non-character cells.

Usage

```
merge_rows(cells, rows, values, collapse = " ")

merge_cols(cells, cols, values, collapse = " ")
```

Arguments

`cells`	Data frame. The cells of a pivot table, usually the output of `as_cells()` or `tidyxl::xlsx_cells()`, or of a subsequent operation on those outputs.
`rows`	The numbers of the rows to be merged.
`values`	The column of `cells` to use as the values of each cell to be merged. Given as a bare variable name.
`collapse`	A character string to separate the values of each cell.
`cols`	The numbers of the columns to be merged.

Value

A data frame

Examples

```
 x <- tibble::tribble(
~row, ~col, ~data_type,     ~chr,
   1,    1,      "chr",   "Katy",
   2,    1,      "chr",  "Perry",
   3,    1,      "chr",      "a",
   4,    1,      "chr",      "b",
   5,    1,      "chr",      "c",
   2,    2,      "chr",  "Adele",
   3,    2,      "chr",      "d",
   4,    2,      "chr",      "e",
   5,    2,      "chr",      "f",
   1,    3,      "chr", "Ariana",
   2,    3,      "chr", "Grande",
   3,    3,      "chr",      "g",
   4,    3,      "chr",      "h",
   5,    3,      "chr",      "i"
)
rectify(x)
y <- merge_rows(x, 1:2, chr)
rectify(y)
z <- merge_cols(x, 1:2, chr)
rectify(z)
```

---

pack *Pack cell values from separate columns per data type into one list-column*

Description

Pack cell values from separate columns per data type into one list-column

Usage

```
pack(cells, types = data_type, name = "value", drop_types = TRUE,
  drop_type_cols = TRUE)

unpack(cells, values = value, name = "data_type", drop_packed = TRUE)
```

Arguments

cells	A data frame of cells, one row per cell. For pack() it must have a column that names, for each cell/row, which of the other columns the value is in. For unpack() it must have a list-column of cell values, where each element is named according to the data type of the value.
types	For pack(), the name of the column that that names, for each cell/row, which of the other columns the value is in.

name	A string. For pack(), the name to give the new list-column of values. For unpack(), the name to give the new column that will name, for each cell, which of the other columns the value is in.
drop_types	For pack(), whether to drop the column named by types.
drop_type_cols	For pack(), whether to drop the original columns of cell values.
values	For unpack(), the name of the list-column of cell values.
drop_packed	For unpack(), whether to drop the column named by values.

Details

When cells are represented by rows of a data frame, the values of the cells will be in different columns according to their data type. For example, the value of a cell containing text will be in a column called chr (or character if it came via tidyxl). A column called data_type names, for each cell, which column its value is in.

pack() rearranges the cell values in a different way, so that they are all in one column, by

1. taking each cell value, from whichever column.
2. making it an element of a list.
3. naming each element according to the column it came from.
4. making the list into a new list-column of the original data frame.

By default, the original columns are dropped, and so is the data_type column.

unpack() is the complement.

This can be useful for dropping all columns of cells except the ones that contain data. For example, tidyxl::xlsx_cells() returns a very wide data frame, and to make it narrow you might do:

```
select(cells, row, col, character, numeric, date)
```

But what if you don't know in advance that the data types you need are character, numeric and date? You might also need logical and error.

Instead, pack() all the data types into a single column, select it, and then unpack.

```
pack(cells) %>%
  select(row, col, value) %>%
  unpack()
```

Functions

- unpack: Unpack cell values from one list-column into separate columns per data type

Examples

```
# A normal data frame
w <- data.frame(foo = 1:2,
                bar = c("a", "b"),
                stringsAsFactors = FALSE)
w
```

```
# The same data, represented by one row per cell, with integer values in the
# `int` column and character values in the `chr` column.
x <- as_cells(w)
x

# pack() and unpack() are complements
pack(x)
unpack(pack(x))

# Drop non-data columns from a wide data frame of cells from tidyxl
if (require(tidyxl)) {
  cells <- tidyxl::xlsx_cells(system.file("extdata", "purpose.xlsx", package = "unpivotr"))
  cells

  pack(cells) %>%
    dplyr::select(row, col, value) %>%
    unpack()
}
```

`partition` *Divide a grid of cells into partitions containing individual tables*

Description

Given the positions of corner cells that mark individual tables in a single spreadsheet, `partion()` works out which table cells belong to which corner cells. The individual tables can then be worked on independently.

`partition()` partitions along both dimensions (rows and columns) at once. `partition_dim()` partitions along one dimension at a time.

Usage

```
partition(cells, corners, align = "top_left", nest = TRUE,
  strict = TRUE)

partition_dim(positions, cutpoints, bound = "upper")
```

Arguments

`cells`	Data frame or tbl, the cells to be partitioned, from `as_cells()` or `tidyxl::xlsx_cells()`.
`corners`	usually a subset of `cells`, being the corners of individual tables. Can also be cells that aren't among `cells`, in which case see the `strict` argument.
`align`	Character, the position of the corner cells relative to their tables, one of `"top-left"` (default), `"top-right"`, `"bottom-left"`, `"bottom-right"`.
`nest`	Logical, whether to nest the partitions in a list-column of data frames.
`strict`	Logical, whether to omit partitions that don't contain a corner cell.

positions — Integer vector, the positions of cells (either the row position or the column position), which are to be grouped between cutpoints.

cutpoints — Integer vector. The positions will be separated into groups either side of each cutpoint.

bound — One of "upper" or "lower", controls whether cells that lie on a cutpoint are should be grouped with cells below or above the cutpoint. For example, if column 5 is a cutpoint, and a cell is in column 5, "lower" would group it with cells in columns 1 to 4, whereas "upper" would group it with cells in columns 6 to 10. This is so that you can use cells at the bottom or the right-hand side of a table as the cutpoints (either of which would be 'upper' bounds because row and column numbers count from 1 in the top-left row and column). When "upper", any cell_positions above the first cutpoint will be in group 0; when "lower", any cell_positions below the final cutpoint will be 0.

Value

partition_dim() returns an integer vector, numbering the groups of cells. Group 0 represents the cells above the first cutpoint (when bound = "upper"), or below the first cutpoint (when bound = "lower"). The other groups are numbered from 1, where group 1 is adjacent to group 0.

partition_dim() returns an integer vector, numbering the groups of cells. Group 0 represents the cells above the first cutpoint (when bound = "upper"), or below the first cutpoint (when bound = "lower"). The other groups are numbered from 1, where group 1 is adjacent to group 0. Divide a grid of cells into chunks along both dimensions

Functions

- partition_dim: Divide a grid of cells into chunks along one dimension

Examples

```
# The `purpose` dataset, represented in four summary tables
multiples <- purpose$small_multiples
rectify(multiples, character, numeric)

# The same thing in its raw 'melted' form that can be filtered
multiples

# First, find the cells that mark a corner of each table
corners <-
  dplyr::filter(multiples,
                !is.na(character),
                !(character %in% c("Sex", "Value", "Female", "Male")))

# Then find out which cells fall into which partition
partition(multiples, corners)

# You can also use bottom-left corners (or top-right or bottom-right)
bl_corners <- dplyr::filter(multiples, character == "Male")
partition(multiples, bl_corners, align = "bottom_left")
```

```
# To complete the grid even when not all corners are supplied, use `strict`
bl_corners <- bl_corners[-1, ]
partition(multiples, bl_corners, align = "bottom_left")
partition(multiples, bl_corners, align = "bottom_left", strict = FALSE)
# Given a set of cells in rows 1 to 10, partition them at the 3rd, 5th and 7th
# rows.
partition_dim(1:10, c(3, 5, 7))

# Given a set of cells in columns 1 to 10, partition them at the 3rd, 5th and
# 7th column.  This example is exactly the same as the previous one, to show
# that the function works the same way on columns as rows.
partition_dim(1:10, c(3, 5, 7))

# Given a set of cells in rows 1 to 10, partition them at the 3rd, 5th and
# 7th rows, aligned to the bottom of the group.
partition_dim(1:10, c(3, 5, 7), bound = "lower")

# Non-integer row/column numbers and cutpoints can be used, even though they
# make no sense in the context of partioning grids of cells.  They are
# rounded towards zero first.
partition_dim(1:10 - .5, c(3, 5, 7))
partition_dim(1:10, c(3, 5, 7) + 1.5)
```

`purpose` *Sense-of-purpose in the 2014 New Zealand General Social Survey*

Description

A dataset containing the self-rated sense-of-purpose of respondents to the 2014 New Zealand General Social Survey.

Usage

```
purpose
```

Format

A list of eight data frames. The first data frame, `Tidy`, contains the raw data in a standard tabular format:

- `Sex` Character, two levels
- `Age group (Life-stages)` Character, age-range in years, four levels
- `Highest qualification` Character, five levels
- `Sense of purpose` Character, score-range, two levels and NA
- `Value` Numeric, number of respondents (weighted? rounded?), has NAs
- `Flags` Character, metadata flags, two levels and NA

The next six data frames are pivot tables of the first data frame. The data frames are named by the compass directions that are suggested for unpivoting them.

The final data frame is a 'tidy' representation of small-multiple pivot tables.

Details

The description provided by Statistics New Zealand is below.

"The 2014 New Zealand General Social Survey (NZGSS) is the fourth of the survey series. We run the NZGSS every two years and interview around 8,500 people about a range of social and economic outcomes.

It provides new and redeveloped data about different aspects of people's lives and their well-being. In particular, the survey provides a view of how well-being outcomes are distributed across different groups within the New Zealand population.

Symbols used in this table:

- S Data has been suppressed.
- * Relative sampling error of 50 percent or more. Numbers may not add to the total because 'Don't know' and 'Refused' have been excluded.

For more tables using the NZGSS 2014 first release see http://archive.stats.govt.nz/browse_for_stats/people_and_communities/Well-being/nzgss-info-releases.aspx.

Data quality: These statistics have been produced in accordance with the Official Statistics System principles and protocols for quality. They conform to the Statistics NZ Methodological Standard for Reporting of Data Quality."

Source

The data is 'Sense of purpose by highest qualification, age group, and sex, 2014' from the Statistics New Zealand portal NZ.Stat http://nzdotstat.stats.govt.nz/wbos/Index.aspx#, retrieved on 2016-08-19. It can be found in the section 'People and communities' > 'Self-rated well-being (NZGSS)'. The data was exported in the Excel (.xlsx) file format and is available at 'extdata/purpose.xlsx' in the package directory.

---

rectify *Display cells as though in a spreadsheet*

Description

Takes the 'melted' output of `as_cells()` or `tidyxl::xlsx_cells()` (each row represents one cell) and projects the cells into their original positions. By default this prints to the terminal/console, but with `display = "browser"` or `display = "rstudio"` it will be displayed in the browser or the RStudio viewer pane.

This is for viewing only; the output is not designed to be used in other functions.

Example: The following cells

```
row col value
  1   1   "a"
  1   2   "b"
  2   1   "c"
  2   2   "d"
```

Would be presented as

```
row/col 1(A) 2(B)
      1 "a"  "b"
      2 "c"  "d"
```

The letters in the column names are for comparing this view with a spreadsheet application.

Usage

```
rectify(cells, values = NULL, types = data_type, formatters = list())

## S3 method for class 'cell_grid'
print(x, display = "terminal", ...)
```

Arguments

cells	Data frame or tbl, the cells to be displayed.
values	Optional. The column of cells to use as the values of each cell. Given as a bare variable name. If omitted (the default), the types argument will be used instead.
types	The column of cells that names, for each cell, which column to use for the value of the cell. E.g. a cell with a character value will have "character" in this column.
formatters	A named list of functions to format cell values for display, named according to the column that the cell value is in.
x	The output of rectify()
display	One of "terminal" (default), "browser", "rstudio". To display in the browser you must have the DT package installed.
...	Arguments passed on to print()

Methods (by class)

- cell_grid: S3 method for class cell_grid

Examples

```
x <- data.frame(name = c("Matilda", "Nicholas"),
                score = c(14L, 10L),
                stringsAsFactors = FALSE)

# This is the original form of the table, which is easy to read.
x

# This is the 'tidy' arrangement that is difficult for humans to read (but
# easy for computers)
y <- as_cells(x, col_names = TRUE)
y
```

```
# rectify() projects the cells as a spreadsheet again, for humans to read.
rectify(y)

# You can choose to use a particular column of the data
rectify(y, values = chr)
rectify(y, values = int)

# You can also show which row or which column each cell came from, which
# helps with understanding what this function does.
rectify(y, values = row)
rectify(y, values = col)

# Empty rows and columns up to the first occupied cell are dropped, but the
# row and column names reflect the original row and column numbers.
y$row <- y$row + 5
y$col <- y$col + 5
rectify(y)

# Supply named functions to format cell values for display.
rectify(y, formatters = list(chr = toupper, int = ~ . * 10))
#
# Print in the browser or in the RStudio viewer pane
## Not run:
  z <- rectify(y)
  print(z, "browser")
  print(z, "rstudio")

## End(Not run)
```

`spatter` *Spread key-value pairs of mixed types across multiple columns*

Description

`spatter()` is like `tidyr::spread()` but for when different columns have different data types. It works on data that has come via `as_cells()` or `tidyxl::xlsx_cells()`, where each row represents one cell of a table, and the value of the cell is represented in a different column, depending on the data type.

Usage

```
spatter(cells, key, values = NULL, types = data_type,
  formatters = list())
```

Arguments

`cells`	A data frame where each row represents a cell, with columns `row` and `col`, usually a column `data_type`, and additional columns of cell values.
`key`	The name of the column whose values will become column names

values	Optional. The column of cells to use as the value of each cell. Given as a bare variable name. If omitted (the default), the type argument will be used instead.
types	Optional. The column that names, for each row of cells, which column contains the cell value. Defaults to data_type.
formatters	A named list of functions for formatting particular data types, named by the data type (the name of the column of cells that contains the cell value.

Examples

```
# A tidy representation of cells of mixed data types
x <- data.frame(stringsAsFactors = FALSE,
        row = c(1L, 1L, 2L, 2L, 3L, 3L, 4L, 4L),
        col = c(1L, 2L, 1L, 2L, 1L, 2L, 1L, 2L),
  data_type = c("character", "character", "character", "numeric", "character",
                "numeric", "character", "numeric"),
  character = c("Name", "Age", "Matilda", NA, "Nicholas", NA, "Olivia", NA),
    numeric = c(NA, NA, NA, 1, NA, 3, NA, 5))
x

# How it would look in a spreadsheet
rectify(x)

# How it looks after treating the cells in row 1 as headers
y <- behead(x, "N", header)
y$col <- NULL # Drop the 'col' column
y

# At this point you might want to do tidyr::spread(), but it won't work because
# you want to use both the `character` and `numeric` columns as the values.
tidyr::spread(y, header, numeric)
tidyr::spread(y, header, character)
spatter(y, header)

# The difference between spatter() and tidyr::spread() is that spatter()
# needs to know which data-type to use for each cell beneath the headers.  By
# default, it looks at the `data_type` column to decide, but you can change
# that with the `types` argument.
y %>%
  dplyr::select(-data_type, -numeric) %>%
  dplyr::mutate(data_type_2 = "character") %>%
  spatter(header, types = data_type_2)

# Alternatively you can name one specific column to use for the cell values.
y %>%
  dplyr::mutate(foo = letters[1:6]) %>%
  dplyr::select(header, row, foo) %>%
  spatter(header, values = foo)

# The column used for the values is consumed before the spread occurs. If
# it's necessary for demarking the rows, then make a copy of it first,
# otherwise you'll get an error like "Duplicate identifiers for rows ..."
y %>%
```

```
  dplyr::mutate(row2 = row) %>%
  dplyr::select(row, header, row2) %>%
  spatter(header, values = row2)

# Like tidyr::spread(), you need to discard extraneous columns beforehand.
# Otherwise you can get more rows out than you want.
y$extra <- 11:16
spatter(y, header)

# pack() is an easy way to keep just the columns you need, without knowing
# in advance which data-type columns you need.  This examples adds a new
# column, which is then removed by the pack-unpack sequence without having to
# mention it by name.
x$extra <- 11:18
x %>%
  pack() %>%
  dplyr::select(row, col, value) %>%
  unpack()

# spatter() automatically converts data types so that they can coexist in the
# same column.  Ordered factors in particular will always be coerced to
# unordered factors.

# You can control data type conversion by supplying custom functions, named
# by the data type of the cells they are to convert (look at the `data_type`
# column).  If your custom functions aren't sufficient to avoid the need for
# coercion, then they will be overridden.
spatter(y, header,
        formatters = list(character = ~ toupper(.), numeric = as.complex))
```

`tidy_table` *Tokenize data frames into a tidy 'melted' structure*

Description

`tidy_table()` will be deprecated. Use `as_cells()` instead.

For certain non-rectangular data formats, it can be useful to parse the data into a melted format where each row represents a single token.

Data frames represent data in a tabular structure. `tidy_table` takes the row and column position of each 'cell', and returns that information in a new data frame, alongside the content and type of each cell.

This makes it easier to deal with complex or non-tabular data (e.g. pivot tables) that have been imported into R as data frames. Once they have been 'melted' by `tidy_table()`, you can use functions like `behead()` and `spatter()` to reshape them into conventional, tidy, unpivoted structures.

For HTML tables, the content of each cell is returned as a standalone HTML string that can be further parsed with tools such as the rvest package. This is particularly useful when an HTML cell itself contains an HTML table, or contains both text and a URL. If the HTML itself is poorly formatted, try passing it through the htmltidy package first.

This is an S3 generic.

Usage

```
tidy_table(x, row_names = FALSE, col_names = FALSE)
```

Arguments

x	A data.frame or an HTML document
row_names	Whether to treat the row names as cells, Default: FALSE
col_names	Whether to treat the column names as cells, Default: FALSE

Value

A data.frame with the following columns:

- row and col (integer) giving the original position of the 'cells'
- any relevant columns for cell values in their original types: chr, cplx, cplx, dbl, fct, int, lgl, list, and ord
- data_type to specify for each cell which of the above columns (chr etc.) the value is in.

The columns fct and ord are, like list, list-columns (each element is independent) to avoid factor levels clashing. For HTML tables, the column html gives the HTML string of the original cell.

Row and column names, when present and required by row_names = TRUE or col_names = TRUE, are treated as though they were cells in the table, and they appear in the chr column.

Examples

```
x <- data.frame(a = c(10, 20),
                b = c("foo", "bar"),
                stringsAsFactors = FALSE)
x
tidy_table(x)
tidy_table(x, row_names = TRUE)
tidy_table(x, col_names = TRUE)

# 'list' columns are undisturbed
y <- data.frame(a = c("a", "b"), stringsAsFactors = FALSE)
y$b <- list(1:2, 3:4)
y
tidy_table(y)

# Factors are preserved by being wrapped in lists so that their levels don't
# conflict.  Blanks are NULLs.
z <- data.frame(x = factor(c("a", "b")),
                y = factor(c("c", "d"), ordered = TRUE))
tidy_table(z)
tidy_table(z)$fct
tidy_table(z)$ord

# HTML tables can be extracted from the output of xml2::read_html().  These
# are returned as a list of tables, similar to rvest::html_table().  The
# value of each cell is its standalone HTML string, which can contain
```

```
# anything -- even another table.

colspan <- system.file("extdata", "colspan.html", package = "unpivotr")
rowspan <- system.file("extdata", "rowspan.html", package = "unpivotr")
nested <- system.file("extdata", "nested.html", package = "unpivotr")

## Not run:
browseURL(colspan)
browseURL(rowspan)
browseURL(nestedspan)

## End(Not run)

tidy_table(xml2::read_html(colspan))
tidy_table(xml2::read_html(rowspan))
tidy_table(xml2::read_html(nested))
```

Index