Package: listr (via r-universe)

August 23, 2024

Type Package

Title Tools for Lists

Version 0.1.0

Description Tools for common operations on lists. Provided are short-cuts to operations like selecting and merging data stored in lists. The functions in this package are designed to be used with pipes.

License EUPL

Encoding UTF-8

RoxygenNote 7.2.1

Suggests knitr, rmarkdown, spelling, testthat (>= 3.0.0), tibble

Config/testthat/edition 3

Language en-US

Imports tidyselect, rlang

VignetteBuilder knitr

NeedsCompilation no

Author Christian Hohenfeld [aut, cre, cph]

Maintainer Christian Hohenfeld <r@hohenfeld.is>

Repository CRAN

Date/Publication 2022-10-06 13:40:01 UTC

Contents

`list_bind` *Bind list elements together.*

Description

Bind list elements together.

Usage

```
list_bind(in_list, ..., what = "rows", name = NULL)
```

Arguments

`in_list`	The list to work on.
`...`	A selection of elements to bind together.
`what`	Either 'rows' or 'cols'.
`name`	Optional name for the resulting element.

Details

The element to bind together must be compatible in the dimension you want to bind them together, if not there will either be an error or an unexpected result.

The 'what' parameter specifies whether to call 'rbind' or 'cbind' on the selected elements.

Using 'name' you can optionally specify a new name for the result. It will be in the position of the first selected element, while the other selected elements will be removed from the input list.

Value

A list with the selected elements bound together as specified.

Examples

```
dfl <- list(data.frame(idx = 1:20, y = rnorm(20)),
            data.frame(idx = 21:40, y = rnorm(20)),
            data.frame(idx = 41:60, y = rnorm(20)))
list_bind(dfl, 1, 2, 3)
```

list_bind_all *Bind all elements together and extract them.*

Description

Bind all elements together and extract them.

Usage

```
list_bind_all(in_list, what = "rows")
```

Arguments

in_list	The list to work on.
what	Either 'rows' or 'cols'

Details

This a convenient wrapper around 'list_bind' which selects everything in the list and extracts the result.

Value

All elements in the list bound together.

Examples

```
dfl <- list(data.frame(idx = 1:20, y = rnorm(20)),
            data.frame(idx = 21:40, y = rnorm(20)),
            data.frame(idx = 41:60, y = rnorm(20)))
list_bind_all(dfl)
```

list_extract *Extract an element from a list using tidy selection.*

Description

This is pretty much an equivalent of calling '[[' on a list, but allows for cleaner use inside pipes.

Usage

```
list_extract(in_list, ...)
```

Arguments

in_list	The list to extract an element from.
...	A selection of what to extract. Must be a single element.

Value

The selected list element.

Examples

```
my_list <- list(rnorm(20), data.frame(x = 1:10, y = rnorm(10)), letters[1:5])
list_extract(my_list, 3)
```

`list_filter` *Filter a list.*

Description

Filter a list.

Usage

```
list_filter(in_list, filter_fun)
```

Arguments

`in_list`	The list to filter.
`filter_fun`	The function to use for filtering.

Details

'filter_fun' must evaluate to TRUE or FALSE for filtering, where elements returning TRUE are kept.

Value

A list, 'in_list' but filtered according to 'filter_fun.'

Examples

```
my_list <- list(aa = 1:3, bb =  1:4, cc = 2:5)
list_filter(my_list, function(x) min(x) == 1)
list_filter(my_list, function(x) max(x) > 3)
```

list_flatten *Flatten nested lists.*

Description

'list_flatten()' works recursively through an input list and puts all elements of nested list to the top level. If there are no nested lists then the input is returned unchanged.

Usage

```
list_flatten(in_list, max_depth = -1)
```

Arguments

in_list	The list to flatten
max_depth	Maximum depth to recurse into.

Details

Using 'max_depth' you can control whether to flatten all nested lists. Negative values will cause all nested lists to be flattened, positive depths will limit the depth of the recursion.

Value

A list without nested lists.

Examples

```
my_list <- list(a = list(1, 2, 3), b = list(4, 5, 6))
list_flatten(my_list)
```

list_insert *Insert an element into a list.*

Description

Insert an element into a list.

Usage

```
list_insert(in_list, item, index, name = NULL, pad = FALSE)

list_append(in_list, item, name = NULL)

list_prepend(in_list, item, name = NULL)
```

Arguments

in_list	The list to work on.
item	The item to add to the list.
index	The index to insert at.
name	Optional name for the new item.
pad	Add 'NULL' elements for too large indices?

Details

The 'index' behaves in the way that everything including the specified index will be moved one position forward. Thus, if you insert at index 2, the old item at index 2 will be moved to index 3. If 'index' is larger than the length of 'in_list' the default behaviour is to just add the new item to the end of the list, however if you specify 'pad = TRUE' then as many 'NULL' elements as needed are added to the list to insert 'item' at the specified location.

The functions 'list_append' and 'list_prepend' exist as a simple short-cut for appending and prepending to a list.

Value

A list, the same as 'in_list' but with 'item' added at 'index'.

Examples

```
my_list <- list(foo1 = 1:10, foo2 = LETTERS[1:10])
list_insert(my_list, rnorm(3), 2, name = "bar")
```

---

list_is_same_class *Check whether all elements of a list have the same class.*

Description

This is a convenience function to check whether all elements of a list have the same class. It will only return TRUE if all elements in a list are of the exact same class. This means that if a list has two vectors TRUE will only be returned if they have the same mode or in case list has elements of compatible classes like data.frame and tbl.df the result will be false.

For the latter case there is 'list_is_compatible_class' that checks whether elements of vectors of classes overlap. Note that this does not necessarily mean that elements can be safely combined, this depends on the respective implementations.

Usage

```
list_is_same_class(list)

list_is_compatible_class(list)
```

Arguments

list	The list to check.

Value

Boolean value.

Examples

```
test_list_false <- list(c(1, 2), c(3, 4), c("abc", "def"))
list_is_same_class(test_list_false)

test_list_true <- list(c(1, 2), c(3, 4))
list_is_same_class(test_list_true)
```

---

list_join_df *Join a list of data frames on a common index.*

Description

Join a list of data frames on a common index.

Usage

```
list_join_df(in_list, join_type = "inner", by = NULL, skip_non_df = FALSE)
```

Arguments

in_list	A list of data frames.
join_type	A string specifying the join strategy to use. Must be one of "inner", "left", "right" or "full".
by	Optional vector of strings specifying columns to merge by.
skip_non_df	Should elements that are not data.frames be skipped?

Details

Using ‘join_type‘ you can specify how to join the data. The default ‘inner‘ will keep only observations present in all data frames. ‘left‘ will keep all observations from the first data frame in the list and merge the matching items from the rest, while ‘right‘ will keep all observations from the last data frame in the list. Using ‘full‘ will keep all observations.

If ‘by‘ is not supplied, then data frames will be merged on columns with names they all have. Otherwise merging is done on the specified columns.

Using ‘skip_non_df‘ you can specify to omit elements from the input list that are not data frames. If FALSE (the default) an error will be thrown if elements are present that are not data frames.

Value

A data frame.

Examples

```
dfl <- list(data.frame(idx = sample(100, 30), x = rnorm(30)),
            data.frame(idx = sample(100, 30), y = rnorm(30)),
            data.frame(idx = sample(100, 30), z = rnorm(30)))
list_join_df(dfl)
```

---

list_name_to_df *Add the names of list items to data frames.*

Description

Add the names of list items to data frames.

Usage

```
list_name_to_df(in_list, column_name = ".group", skip_non_df = TRUE)
```

Arguments

in_list	The list to work on. Must have names.
column_name	The name of the column to add to the data frames.
skip_non_df	Whether to skip items that are not data frames.

Details

With ‘column_name‘ you can specify the name the new columns in the data.frames should have. The default is ‘.group‘.

Using ‘skip_non_df‘ you can specify to omit elements from the input list that are not data frames. If FALSE an error will be thrown if elements are present that are not data frames. If TRUE (the default) then items that are not data frames will be ignored and remain unchanged.

Value

The input list with the name of the list item added in a new column for all data frames.

Examples

```
my_list <- list(group1 = data.frame(x = 1:10, y = rnorm(10)),
                group2 = data.frame(x = 1:10, y = rnorm(10)))
list_name_to_df(my_list)
```

list_remove *Remove elements from a list.*

Description

Remove elements from a list.

Usage

```
list_remove(in_list, ...)
```

Arguments

in_list	The list to remove elements from.
...	Names or numeric positions of elements to remove.

Value

The list with the specified elements removed.

Examples

```
my_list <- list(a = rnorm(10), b = rnorm(10), c = rnorm(10))
list_remove(my_list, b)
```

list_rename *Rename elements of a named list.*

Description

Rename elements of a named list.

Usage

```
list_rename(in_list, ...)
```

Arguments

in_list	The list to rename elements in.
...	The renaming definitions.

Details

'list_rename()' changes the name of elements in a named list. The definitions for renaming are given in '...' in the style 'new_name = old_name'. You can specify as many renaming definitions as you like as long as there are not more definitions than elements in the list.

If no renaming definition is given the input list is returned. If you try to rename elements not present in the list nothing happens; unless you provide more renaming definitions than elements in the list, in that case an error is raised.

Value

The list provided in 'in_list' with elements renamed according to the definition in '...'.

Examples

```
my_list <- list(a = 1, b = 2, c = 3)
list_rename(my_list, AAA = "a", CCC = "c")
```

---

list_select *Select parts of a list.*

Description

Select parts of a list.

Usage

```
list_select(in_list, ...)
```

Arguments

in_list	The list to select from.
...	The selection, can both be names and numeric positions.

Details

'list_select()' lets you select parts of a list either by position or by name. Names can be supplied as bare variable names and do not need to be supplied as strings or otherwise be quoted.

Elements are returned in the order they are given, this is useful if you want to reorder elements in a list. You can also rename while selecting, writing your selection like 'new_name = old_name'.

Value

A list of the selected elements.

Examples

```
my_list <- list(a = c(1, 2), b = c(3, 4), c(5, 6))
list_select(my_list, a, 3)
```

`list_sort` *Sort a list.*

Description

Sort a list.

Usage

```
list_sort(in_list, sort_fun, descending = FALSE, use_na = "drop")
```

Arguments

`in_list`	The list to work on.
`sort_fun`	The function to sort the list by.
`descending`	Boolean, if TRUE items will be sorted in descending order.
`use_na`	String, one of 'drop', 'first' or 'last'.

Details

The list will be sorted according to the return value of 'sort_fun'. The function must return a numeric value that will then be used to order the list items.

If the function returns 'NA' the sorting will depend on the string specified to 'use_missing'. The default is to drop these values, but they can optionally be put first or last in the sorted list.

If the sorting function returns 'NA' for some items and 'use_na' is 'first' or 'last' then the order of the 'NA' items will most likely be the same as in 'in_list' (but this cannot be guaranteed). The same is true for ties.

Value

A list, 'in_list' sorted according to 'sort_fun'.

Examples

```
my_list <- list(data.frame(x = 1:10),
                data.frame(x = 1:5, y = rnorm(5)),
                data.frame(x = 1:20, y = rnorm(20), z = LETTERS[1:20]))
list_sort(my_list, nrow)
list_sort(my_list, function(x) sum(x$x), descending = TRUE)
```

Index