Package ‘rlist’

September 3, 2021

Type Package
Title A Toolbox for Non-Tabular Data Manipulation
Version 0.4.6.2
Author Kun Ren <ken@renkun.me>
Maintainer Kun Ren <ken@renkun.me>
Description Provides a set of functions for data manipulation with list objects, including mapping, filtering, grouping, sorting, updating, searching, and other useful functions. Most functions are designed to be pipeline friendly so that data processing with lists can be chained.
Depends R (>= 2.15)
Date 2021-09-02
Suggests testthat, stringdist
Imports yaml, jsonlite, XML, data.table
License MIT + file LICENSE
URL https://renkun-ken.github.io/rlist/,
https://github.com/renkun-ken/rlist,
https://renkun-ken.github.io/rlist-tutorial/
BugReports https://github.com/renkun-ken/rlist/issues
ByteCompile TRUE
LazyData true
RoxygenNote 7.1.1
Encoding UTF-8
NeedsCompilation no
Repository CRAN
Date/Publication 2021-09-03 12:20:02 UTC
### R topics documented:

- rlist-package
- .evalwith
- args_env
- args_list
- callwith
- contains
- dots
- getnames
- is.empty
- List
- list.all
- list.any
- list.append
- list.apply
- list.cases
- list.cbind
- list.class
- list.clean
- list.common
- list.count
- list.do
- list.exclude
- list.expand
- list.extract
- list.filter
- list.find
- list.findi
- list.first
- list.flatten
- list.group
- list.insert
- list.is
- list.iter
- list.join
- list.last
- list.load
- list.map
- list.maps
- list.mapv
- list.match
- list.merge
- list.names
- list.order
- list.parse
- list.prepend
- list.rbind
Description

rlist is a set of tools for working with list objects. Its goal is to make it easier to work with lists by providing a wide range of functions that operate on non-tabular data stored in them.

The package provides a set of functions for data manipulation with list objects, including mapping, filtering, grouping, sorting, updating, searching, and other useful functions. Most functions are designed to be pipeline friendly so that data processing with lists can be chained.

rlist Tutorial (https://renkun-ken.github.io/rlist-tutorial/) is a complete guide to rlist.
**.evalwith**  
*Convert an object to evaluating environment for list elements Users should not directly use this function*

**Description**  
Convert an object to evaluating environment for list elements Users should not directly use this function

**Usage**  
```
.evalwith(x)
```

**Arguments**  
- **x**  
  the object

---

**args_env**  
*create an environment for args*

**Description**  
create an environment for args

**Usage**  
```
args_env(..., parent = parent.frame())
```

**Arguments**  
- **...**  
  objects
- **parent**  
  parent environment

---

**args_list**  
*create a list for args*

**Description**  
create a list for args

**Usage**  
```
args_list(...)```

**Arguments**  
- **...**  
  objects
**callwith**

*Evaluate a function with a modified default values*

**Description**
Evaluate a function with a modified default values

**Usage**
callwith(fun, args, dots = list(), keep.null = FALSE, envir = parent.frame())

**Arguments**
- **fun** either a function or a non-empty character string naming the function to be called
- **args** a list of values to modify the default arguments of the function
- **dots** the user-specific input (usually from ...)
- **keep.null** TRUE to keep NULL values after argument modifications
- **envir** the environment to evaluate the function call

**contains**
*Test if a vector contains certain values*

**Description**
Test if a vector contains certain values

**Usage**
contains(table, x)

**Arguments**
- **table** the values to be matched against
- **x** the values to be matched
getnames

Description
Get the names of an object

Usage
getnames(x, def = \texttt{NULL})

Arguments
\begin{itemize}
\item \texttt{x} \hspace{1cm} the object to extract names
\item \texttt{def} \hspace{1cm} the value to return if the object has \texttt{NULL} names. For vectorization purpose, set this to \texttt{character(1L)}.
\end{itemize}

Details
This function is used in vectorization when the names of an object is to be supplied. \texttt{NULL} value will break the vectorization while setting \texttt{def = character(1L)} makes the names vectorizable.
is.empty  

*Check if an object is empty (has length 0)*

---

**Description**

Check if an object is empty (has length 0)

**Usage**

```r
is.empty(x)
```

**Arguments**

- `x`  
  the object

**Details**

A NULL value, zero-length vector or list have length zero, which is called empty.

---

**List**

*Create a List environment that wraps given data and most list functions are defined for chainable operations.*

---

**Description**

Create a List environment that wraps given data and most list functions are defined for chainable operations.

**Usage**

```r
List(data = list())
```

**Arguments**

- `data`  
  A list or vector

**Details**

Most list functions are defined in List environment. In addition to these functions, `call(fun,...)` calls external function `fun` with additional parameters specifies in `...`. To extract the data from List `x`, call `x$data` or simply `x[]`. 
Examples

x <- list(p1 = list(type='A', score=list(c1=10, c2=8)),
         p2 = list(type='B', score=list(c1=9, c2=9)),
         p3 = list(type='B', score=list(c1=9, c2=7)))
m <- List(x)
m$filter(type=='B')$
   map(score$c1) []

m$group(type)$
   map(g ~ List(g)$
       map(score)$
       call(unlist)$
       call(mean) [] []

# Subsetting, extracting, and assigning

p <- List(list(a=1, b=2))
p['a']
p[['a']]
p$a <- 2
p['b'] <- NULL
p[['a']] <- 3

list.all

Examine if a condition is true for all elements of a list

Description

Examine if a condition is true for all elements of a list

Usage

list.all(.data, cond, na.rm = FALSE)

Arguments

.data A list or vector
cond A logical lambda expression
na.rm logical. If true NA values are ignored in the evaluation.

Value

TRUE if cond is evaluated to be TRUE for all elements in .data.

See Also

list.any
list.any

Examine if a condition is true for at least one list element

Description
Examine if a condition is true for at least one list element

Usage
list.any(.data, cond, na.rm = FALSE)

Arguments
.data A list or vector
cond A logical lambda expression
na.rm logical. If true NA values are ignored in the evaluation.

Value
TRUE if cond is evaluated to be TRUE for any element in .data.

See Also
list.all

Examples
x <- list(p1 = list(type='A',score=list(c1=10,c2=8)),
p2 = list(type='B',score=list(c1=9,c2=9)),
p3 = list(type='B',score=list(c1=9,c2=7)))
list.all(x, type=='B')
list.all(x, mean(unlist(score))>=6)
list.all(x, score$c2 > 8 || score$c3 > 5, na.rm = TRUE)
list.all(x, score$c2 > 8 || score$c3 > 5, na.rm = FALSE)

list.any

Examine if a condition is true for at least one list element

Description
Examine if a condition is true for at least one list element

Usage
list.any(.data, cond, na.rm = FALSE)

Arguments
.data A list or vector
cond A logical lambda expression
na.rm logical. If true NA values are ignored in the evaluation.

Value
TRUE if cond is evaluated to be TRUE for any element in .data.

See Also
list.all

Examples
x <- list(p1 = list(type='A',score=list(c1=10,c2=8)),
p2 = list(type='B',score=list(c1=9,c2=9)),
p3 = list(type='B',score=list(c1=9,c2=7)))
list.any(x, type=='B')
list.any(x, mean(unlist(score))>=6)
list.any(x, score$c2 > 8 || score$c3 > 5, na.rm = TRUE)
list.any(x, score$c2 > 8 || score$c3 > 5, na.rm = FALSE)
list.append

Append elements to a list

Description

Append elements to a list

Usage

```r
list.append(.data, ...)  
```

Arguments

- `.data` A list or vector
- `...` A vector or list to append after `x`

See Also

- `list.prepend`, `list.insert`

Examples

```r
## Not run:
x <- list(a=1,b=2,c=3)
list.append(x,d=4,e=5)
list.append(x,d=4,f=c(2,3))
## End(Not run)
```

list.apply

Apply a function to each list element (lapply)

Description

Apply a function to each list element (lapply)

Usage

```r
list.apply(.data, .fun, ...)  
```

Arguments

- `.data` A list or vector
- `.fun` function
- `...` Additional parameters passed to `FUN.`
list.cases

Get all unique cases of a list field by expression

Description
Get all unique cases of a list field by expression

Usage
list.cases(.data, expr, simplify = TRUE, sorted = TRUE)

Arguments
.data A list or vector
expr A lambda expression. The function will returns all cases of the elements if expr is missing.
simplify logical. Should atomic vectors be simplified by unlist?
sorted logical. Should the cases be sorted in ascending order?

Examples
x <- list(p1 = list(type='A',score=list(c1=10,c2=8)),
p2 = list(type='B',score=list(c1=9,c2=9)),
p3 = list(type='B',score=list(c1=9,c2=7)))
list.cases(x,type)
list.cases(x,mean(unlist(score)))

foo <- list(x = LETTERS[1:3], y = LETTERS[3:5])
list.cases(foo)

list.cbind

Bind all list elements by column

Description
The function binds all list elements by column. Each element of the list is expected to be an atomic vector, data.frame, or data.table of the same length. If list elements are also lists, the binding will flatten the lists and may produce undesired results.

Usage
list.cbind(.data)

Arguments
.data list
See Also

list.cbind, list.stack

Examples

```r
x <- list(data.frame(i=1:5,x=rnorm(5)),
        data.frame(y=rnorm(5),z=rnorm(5)))
list.cbind(x)
```

---

**list.class**

*Classify list elements into unique but non-exclusive cases*

Description

In non-tabular data, a certain field may take multiple values in a collection non-exclusively. To classify these elements into different cases, this function detects all possible cases and for each case all elements are examined whether to belong to that case.

Usage

```r
list.class(.data, ..., sorted = TRUE)
```

Arguments

- `.data` A list or vector
- `...` keys
- `sorted` TRUE to sort the group keys. Ignored when the key has multiple entries.

Value

A list of possible cases each of which contains elements belonging to the case non-exclusively.

Examples

```r
x <- list(
  p1=list(name='Ken', age=24,
           interest=c('reading', 'music', 'movies'),
           lang=list(r=2, csharp=4, python=3)),
  p2=list(name='James', age=25,
           interest=c('sports', 'music'),
           lang=list(r=3, java=2, cpp=5)),
  p3=list(name='Penny', age=24,
           interest=c('movies', 'reading'),
           lang=list(r=1, cpp=4, python=2)))
list.class(x, interest)
list.class(x, names(lang))
```
list.clean

Clean a list by a function

Description

This function removes all elements evaluated to be TRUE by an indicator function. The removal can be recursive so that the resulted list surely does not include such elements in any level.

Usage

list.clean(.data, fun = is.null, recursive = FALSE)

Arguments

.data A list or vector to operate over.

.fun A character or a function that returns TRUE or FALSE to indicate if an element of .data should be removed.

recursive logical. Should the list be cleaned recursively? Set to FALSE by default.

Details

Raw data is usually not completely ready for analysis, and needs to be cleaned up to certain standards. For example, some data operations require that the input does not include NULL values in any level, therefore fun = "is.null" and recursive = TRUE can be useful to clean out all NULL values in a list at any level.

Sometimes, not only NULL values are undesired, empty vectors or lists are also unwanted. In this case, fun = function(x) length(x) == 0L can be useful to remove all empty elements of zero length. This works because length(NULL) == 0L, length(list()) == 0L and length(numeric()) == 0L are all TRUE.

Examples

x <- list(a=NULL, b=list(x=NULL, y=character()), d=1, e=2)
list.clean(x)
list.clean(x, recursive = TRUE)
list.clean(x, function(x) length(x) == 0L, TRUE)
**list.common**  
*Get all common cases by expression for a list*

**Description**
Get all common cases by expression for a list

**Usage**
```
list.common(.data, expr)
```

**Arguments**

- **.data**: list
- **expr**: An anonymous (or "lambda") expression to determine common cases. If one is not specified, list.common simply returns all identical sub-elements within lists.

**Examples**
```
x <- list(c('a','b','c'),c('a','b'),c('b','c'))
list.common(x, .)
x <- list(p1 = list(type='A',score=list(c1=10,c2=8)),
          p2 = list(type='B',score=list(c1=9,c2=9)),
          p3 = list(type='B',score=list(c1=9,c2=7)))
list.common(x,type)
list.common(x, names(score))

foo <- list(x = LETTERS[1:3], y = LETTERS[3:5])
list.common(foo)
```

**list.count**  
*Count the number of elements that satisfy given condition*

**Description**
Count the number of elements that satisfy given condition

**Usage**
```
list.count(.data, cond)
```

**Arguments**

- **.data**: A list or vector
- **cond**: A logical lambda expression for each element of .data to evaluate. If cond is missing then the total number of elements in .data will be returned.
Value

An integer that indicates the number of elements with which `cond` is evaluated to be TRUE.

Examples

```r
x <- list(p1 = list(type='A', score=list(c1=10, c2=8)),
          p2 = list(type='B', score=list(c1=9, c2=9)),
          p3 = list(type='B', score=list(c1=9, c2=7)))
list.count(x, type=='B')
list.count(x, min(unlist(score)) >= 9)
```

---

**list.do**

Call a function with a list of arguments

Description

Call a function with a list of arguments

Usage

```r
list.do(.data, fun, ...)
```

Arguments

- `.data` list. vector will be coerced to list before being passed to `fun`.
- `fun` The function to call
- `...` The additional parameters passed to `do.call`

Examples

```r
x <- lapply(1:3, function(i) { c(a=i, b=i^2)})
df <- lapply(1:3, function(i) { data.frame(a=i, b=i^2, c=letters[i])})
list.do(x, rbind)
```

---

**list.exclude**

Exclude members of a list that meet given condition.

Description

Exclude members of a list that meet given condition.

Usage

```r
list.exclude(.data, cond)
```
list.expand

Create a list from all combinations of factors

Description

Create a list from all combinations of the supplied vectors or lists, extending the functionality of `expand.grid` from data frame to list.

Usage

list.expand(...)

Arguments

... vectors or lists

Value

A list of all combinations of the supplied vectors or lists.

Examples

```r
list.expand(x=1:10, y=c("a","b","c"))
list.expand(x=list(c(1,2), c(2,3)), y = c("a","b","c"))
list.expand(
    a=list(list(x=1,y="a"), list(x=2, y="b")),
    b=list(c("x","y"), c("y","z","w")))
```
list.extract  

Extract an element from a list or vector

Description
Extract an element from a list or vector

Usage
list.extract()

Examples
x <- list(a=1, b=2, c=3)
list.extract(x, 1)
list.extract(x, 'a')

list.filter  

Filter a list or vector by a series of conditions

Description
The function recursively filters the data by a given series of conditions. The filter can be a single condition or multiple conditions. .data will be filtered by the first condition; then the results will be filtered by the second condition, if any; then the results will be filtered by the third, if any, etc. The results only contain elements satisfying all conditions specified in . . .

Usage
list.filter(.data, ...)

Arguments
.data
A list or vector

...  
logical conditions

Value
elements in .data satisfying all conditions

Examples
x <- list(p1 = list(type='A',score=list(c1=10,c2=8)),
p2 = list(type='B',score=list(c1=9,c2=9)),
p3 = list(type='B',score=list(c1=9,c2=7)))
list.filter(x, type=='B')
list.filter(x, min(score$c1, score$c2) >= 8)
list.filter(x, type=='B', score$c2 >= 8)
list.find

Find a specific number of elements in a list or vector satisfying a given condition

Description
Find a specific number of elements in a list or vector satisfying a given condition

Usage
list.find(.data, cond, n = 1L)

Arguments
- .data: A list or vector
- cond: A logical lambda expression
- n: The number of items to find. (n = 1L by default)

Value
A list or vector of at most n elements in .data found to satisfy cond.

Examples
x <- list(p1 = list(type='A', score=list(c1=10, c2=8)),
          p2 = list(type='B', score=list(c1=9, c2=9)),
          p3 = list(type='B', score=list(c1=9, c2=7)))
list.find(x, type=='B', 1)
list.find(x, min(score$c1, score$c2) >= 9)

list.findi

Find the indices of a number of elements in a list or vector satisfying a given condition

Description
Find the indices of a number of elements in a list or vector satisfying a given condition

Usage
list.findi(.data, cond, n = 1L)

Arguments
- .data: A list or vector
- cond: A logical lambda expression
- n: The number of items to find. (n = 1L by default)
Value

an integer vector consisting of the elements indices

Examples

```r
x <- list(p1 = list(type='Var A', score=list(c1=10,c2=8)),
          p2 = list(type='Var B', score=list(c1=9,c2=9)),
          p3 = list(type='Var B', score=list(c1=9,c2=7)))
list.first(x, type=='B')
list.first(x, min(score$c1, score$c2) >= 8)
list.first(x, min(score$c1, score$c2) <= 8, n = 2)
```

---

**list.first**

*Find the first element that meets a condition*

Description

Find the first element that meets a condition

Usage

```r
list.first(.data, cond)
```

Arguments

- `.data` A list or vector
- `cond` a logical lambda expression

See Also

- `list.last`

Examples

```r
x <- list(p1 = list(type='A', score=list(c1=10,c2=8)),
          p2 = list(type='B', score=list(c1=9,c2=9)),
          p3 = list(type='B', score=list(c1=9,c2=7)))
list.first(x, score$c1 < 10)
list.first(x, score$c1 < 9 || score$c3 >= 5) # NULL for all results are NA or FALSE
```
list.flatten  Flatten a nested list to a one-level list

Description
Flatten a nested list to a one-level list

Usage
list.flatten(x, use.names = TRUE, classes = "ANY")

Arguments
x list
use.names logical. Should the names of x be kept?
classes A character vector of class names, or "ANY" to match any class.

Details
The function is essentially a slightly modified version of flatten2 provided by Tommy at stackoverflow.com who has full credit of the implementation of this function.

Author(s)
Tommy

Examples
p <- list(a=1,b=list(b1=2,b2=3),c=list(c1=list(c11='Var',c12='x'),c2=3))
list.flatten(p)

p <- list(a=1,b=list(x="a",y="b",z=10))
list.flatten(p, classes = "numeric")
list.flatten(p, classes = "character")

list.group  Divide list/vector elements into exclusive groups

Description
Divide list/vector elements into exclusive groups

Usage
list.group(.data, ..., sorted = TRUE)
list.insert

Insert a series of lists at the given index

Description
Insert a series of lists at the given index

Usage
list.insert(.data, index, ...)

Arguments
.data A list or vector
index The index at which the lists are inserted
...

See Also
list.append, listprepend

Arguments
.data A list or vector
... One or more expressions in the scope of each element to evaluate as keys
sorted TRUE to sort the group keys. Ignored when the key has multiple entries.

Value
A list of group elements each contain all the elements in .data belonging to the group

See Also
list.ungroup

Examples
x <- list(p1 = list(type='A', score=list(c1=10, c2=8)),
p2 = list(type='B', score=list(c1=9, c2=9)),
p3 = list(type='B', score=list(c1=9, c2=7)))
list.group(x, type)
list.group(x, mean(unlist(score)))
### list.is

Return a logical vector that indicates if each member of a list satisfies a given condition.

#### Description

Return a logical vector that indicates if each member of a list satisfies a given condition.

#### Usage

```r
list.is(.data, cond, use.names = TRUE)
list.if(.data, cond, use.names = TRUE)
```

#### Arguments

- `.data` list
- `cond` A logical lambda expression
- `use.names` logical Should the names of `.data` be kept?

#### Examples

```r
x <- list(p1 = list(type='A', score=list(c1=10,c2=8)),
          p2 = list(type='B', score=list(c1=9,c2=9)),
          p3 = list(type='B', score=list(c1=9,c2=7)))
list.insert(x, 2, p2.1 = list(type='B', score=list(c1=8,c2=9)))

## Not run:
x <- list(p1 = list(type='A', score=list(c1=10,c2=8)),
p2 = list(type='B', score=list(c1=9,c2=9)),
p3 = list(type='B', score=list(c1=9,c2=7)))
list.insert(x, 2, p2.1 = list(type='B', score=list(c1=8,c2=9)))

## End(Not run)
```
**list.iter**

*Iterate a list by evaluating an expression on each list element*

**Description**

Iterate a list by evaluating an expression on each list element

**Usage**

`list.iter(.data, expr)`

**Arguments**

- `.data` list
- `expr` A lambda expression

**Value**

`invisible(.data)`

**Examples**

```r
x <- list(p1 = list(type='A', score=list(c1=10, c2=8)),
          p2 = list(type='B', score=list(c1=9, c2=9)),
          p3 = list(type='B', score=list(c1=9, c2=7)))
list.iter(x, cat(paste(type, '\n')))
list.iter(x, cat(str(.)))
```

---

**list.join**

*Join two lists by single or multiple keys*

**Description**

Join two lists by single or multiple keys

**Usage**

`list.join(x, y, xkey, ykey, ..., keep.order = TRUE)`

**Arguments**

- `x` The first list
- `y` The second list
- `xkey` A lambda expression that determines the key for list x
- `ykey` A lambda expression that determines the key for list y, same to xkey if missing
- `...` The additional parameters passed to `merge.data.frame`
- `keep.order` Should the order of x be kept?
Examples

```r
l1 <- list(p1=list(name='Ken', age=20),
           p2=list(name='James', age=21),
           p3=list(name='Jenny', age=20))
l2 <- list(p1=list(name='Jenny', age=20, type='A'),
           p2=list(name='Ken', age=20, type='B'),
           p3=list(name='James', age=22, type='A'))
list.join(l1, l2, name)
list.join(l1, l2, .[c('name', 'age')])
```

---

### list.last

**Find the last element that meets a condition**

#### Description

Find the last element that meets a condition

#### Usage

```r
list.last(.data, cond)
```

#### Arguments

- `.data` A list or vector
- `cond` a logical lambda expression

#### See Also

- `list.first`

#### Examples

```r
x <- list(p1 = list(type='A', score=list(c1=10, c2=8)),
          p2 = list(type='B', score=list(c1=9, c2=9)),
          p3 = list(type='B', score=list(c1=9, c2=7)))
list.last(x, score$c1 < 10)
list.last(x, score$c1 < 9 || score$c3 >= 5) # NULL for all results are NA or FALSE
```
list.load  

*Load a list from file*

**Description**

Load a list from file

**Usage**

```r
list.load(
  file,
  type = tools::file_ext(file),
  ..., 
  guess = c("json", "yaml", "rds", "rdata", "xml"),
  action = c("none", "merge", "ungroup"),
  progress = length(file) >= 5L
)
```

**Arguments**

- `file`  
  a character vector. The file as input.
- `type`  
  The type of input which, by default, is determined by file extension. Currently supports RData, RDS, JSON, YAML.
- `...`  
  Additional parameters passed to the loader function
- `guess`  
  a character vector to guess iteratively if type of file is unrecognized, NA or empty string.
- `action`  
  The post-processing action if multiple files are supplied. This parameter will be ignored if only a single file is supplied.  
  'none' (default) to leave the resulted list as a list of elements corresponding to elements in `file` vector.  
  'merge' to merge the list elements iteratively, the later lists always modify the former ones through `modifyList`.  
  'ungroup' to ungroup the list elements, especially when each file is a page of elements with identical structure.
- `progress`  
  TRUE to show a text progress bar in console while loading files. By default, if file contains 5 elements, then the progress bar will automatically be triggered to indicate loading progress.

**Examples**

```r
## Not run:
list.load('list.rds')
list.load('list.rdata')
list.load('list.yaml')
list.load('list.json')

## End(Not run)
```
**list.map**  
*Map each element in a list or vector by an expression.*

**Description**
Map each element in a list or vector by an expression.

**Usage**

```r
list.map(.data, expr)
```

**Arguments**

- `.data`  
a list or vector  
- `expr`  
A lambda expression

**Value**

A list in which each element is mapped by `expr` in `.data`

**See Also**

`list.mapv`

**Examples**

```r
x <- list(p1 = list(type="A", score=list(c1=10,c2=8)),  
p2 = list(type="B", score=list(c1=9,c2=9)),  
p3 = list(type="B", score=list(c1=9,c2=7)))
list.map(x, type)  
list.map(x, min(score$c1, score$c2))
```

---

**list.maps**  
*Map multiple lists with an expression*

**Description**
Map multiple lists with an expression

**Usage**

```r
list.maps(expr, ...)
```
list.mapv

Arguments

expr          An implicit lambda expression where only .i and .name are defined.
...           Named arguments of lists with equal length. The names of the lists are available as symbols that represent the element for each list.

Examples

## Not run:
l1 <- list(p1=list(x=1,y=2), p2=list(x=3,y=4), p3=list(x=1,y=3))
l2 <- list(2,3,5)
list.maps(a$x*b+a$y,a=l1,b=l2)
list.maps(.1$x+.2+.1$y,l1,l2)
## End(Not run)

list.mapv            Map each member of a list by an expression to a vector.

Description

Map each member of a list by an expression to a vector.

Usage

list.mapv(.data, expr, as, use.names = TRUE)

Arguments

.data           a list or vector
expr            a lambda expression
as              the mode to corece. Missing to unlist the mapped results.
use.names       Should the names of the results be preserved?

Value

A vector in which each element is mapped by expr in .data

See Also

list.map

Examples

x <- list(p1 = list(type='A',score=list(c1=10,c2=8)),
          p2 = list(type='B',score=list(c1=9,c2=9)),
          p3 = list(type='B',score=list(c1=9,c2=7)))
list.mapv(x, type)
list.mapv(x, min(score$c1, score$c2))
**list.match**

Select members of a list that match given regex pattern

**Description**

Select members of a list that match given regex pattern

**Usage**

```
list.match(.data, pattern, ...)
```

**Arguments**

- `.data` A list or vector
- `pattern` character. The regex pattern to match the name of the members
- `...` Additional parameters to pass to `grep`

**Examples**

```r
x <- list(p1 = list(type='A', score=list(c1=10,c2=8)),
         p2 = list(type='B', score=list(c1=9,c2=9)),
         p3 = list(type='B', score=list(c1=9,c2=7)))
list.match(x,'p[12]')
list.match(x,'3')
```

**list.merge**

Merge a number of named lists in sequential order

**Description**

The function merges a number of lists in sequential order by `modifyList`, that is, the later list always modifies the former list and form a merged list, and the resulted list is again being merged with the next list. The process is repeated until all lists in `...` or `list` are exhausted.

**Usage**

```
list.merge(...)
```

**Arguments**

- `...` named lists
List merging is usually useful in the merging of program settings or configuration with multiple versions across time, or multiple administrative levels. For example, a program settings may have an initial version in which most keys are defined and specified. In later versions, partial modifications are recorded. In this case, list merging can be useful to merge all versions of settings in release order of these versions. The result is an fully updated settings with all later modifications applied.

Examples

```r
l1 <- list(a=1,b=list(x=1,y=1))
l2 <- list(a=2,b=list(z=2))
l3 <- list(a=2,b=list(x=3))
list.merge(l1,l2,l3)
```

---

**list.names**

*Get or set the names of a list by expression*

**Description**

Get or set the names of a list by expression

**Usage**

`list.names(.data, expr)`

**Arguments**

- `data`: A list or vector
- `expr`: the expression whose value will be set as the name for each list element. If missing then the names of the list will be returned. If `NULL` then the names of the list will be removed.

**Examples**

```r
list.names(c(1,2,3))
list.names(c(a=1,b=2,c=3))
list.names(c(1,2,3),letters[.])
list.names(list(list(name='A',value=10),list(name='B',value=20)), name)
```
list.order

Give the order of each list element by expression

Description

Give the order of each list element by expression

Usage

list.order(.data, ..., keep.names = FALSE, na.last = TRUE)

Arguments

.data A list or vector
... A group of lambda expressions
keep.names Whether to keep the names of x in the result
na.last The way to deal with NAs.

Value

an integer vector.

See Also

list.sort

Examples

x <- list(p1 = list(type='A',score=list(c1=10,c2=8)),
p2 = list(type='B',score=list(c1=9,c2=9)),
p3 = list(type='B',score=list(c1=9,c2=7)))
list.order(x, type, (score$c2)) # order by type (ascending) and score$c2 (descending)
list.order(x, min(score$c1,score$c2))
list.order(x, min(score$c1,score$c2), keep.names=TRUE)

list.parse

Convert an object to list with identical structure

Description

This function converts an object representing data to list that represents the same data. For example, a data.frame stored tabular data column-wisely, that is, each column represents a vector of a certain type. list.parse converts a data.frame to a list which represents the data row-wisely so that it can be more convinient to perform other non-tabular data manipulation methods.
Usage

list.parse(x, ...)

## Default S3 method:
list.parse(x, ...)

## S3 method for class 'matrix'
list.parse(x, ...)

## S3 method for class 'data.frame'
list.parse(x, ...)

## S3 method for class 'character'
list.parse(x, type, ...)

Arguments

x  An object

... Additional parameters passed to converter function

type  The type of data to parse. Currently json and yaml are supported.

Value

list object representing the data in x

Examples

x <- data.frame(a=1:3,type=c('A','C','B'))
list.parse(x)

x <- matrix(rnorm(1000),ncol=5)
rownames(x) <- paste0('item',1:nrow(x))
colnames(x) <- c('a','b','c','d','e')
list.parse(x)

z <- 
   a:
    type: x
class: A
    registered: yes

list.parse(z, type='yaml')
list.prepend

Prepend elements to a list

Description

Prepend elements to a list

Usage

list.prepend(.data, ...)

Arguments

.data A list or vector
... The vector or list to prepend before x

See Also

list.append, list.insert

Examples

x <- list(a=1,b=2,c=3)
list.prepend(x, d=4, e=5)
list.prepend(x, d=4, f=c(2,3))

list.rbind

Bind all list elements by row

Description

The function binds all list elements by row. Each element of the list is expected to be an atomic vector, data.frame, or data.table. If list elements are also lists, the result can be a list-valued matrix. In this case, list.stack may produce a better result.

Usage

list.rbind(.data)

Arguments

.data list

See Also

list.cbind, list.stack
Examples

\[
x \leftarrow lapply(1:3, function(i) \{ \text{c}(a=i, b=i^2) \})
\]
\[
df \leftarrow lapply(1:3, function(i) \{ \text{data.frame}(a=i, b=i^2, c=letters[i]) \})
\]
list.rbind(x)
list.rbind(df)

list.remove
Remove members from a list by index or name

Description
Remove members from a list by index or name

Usage

list.remove(.data, range = integer())

Arguments

.data A list or vector
range A numeric vector of indices or a character vector of names to remove from .data

Examples

\[
x \leftarrow \text{list}(p1 = \text{list}(\text{type}='A', \text{score}=\text{list}(c1=10, c2=8)),
\]
\[
p2 = \text{list}(\text{type}='B', \text{score}=\text{list}(c1=9, c2=9)),
\]
\[
p3 = \text{list}(\text{type}='B', \text{score}=\text{list}(c1=9, c2=7))
\]
list.remove(x, 'p1')
list.remove(x, c(1,2))

list.reverse
Reverse a list

Description
Reverse a list

Usage

list.reverse(.data)

Arguments

.data A list or vector
Example

```r
x <- list(a=1, b=2, c=3)
list.reverse(x)
```

---

**list.sample**  
Sample a list or vector

**Description**

Sample a list or vector

**Usage**

```r
list.sample(.data, size, replace = FALSE, weight = 1, prob = NULL)
```

**Arguments**

- `.data` A list or vector
- `size` integer. The size of the sample
- `replace` logical. Should sampling be with replacement?
- `weight` A lambda expression to determine the weight of each list member, which only takes effect if `prob` is `NULL`.
- `prob` A vector of probability weights for obtaining the elements of the list being sampled.

**Examples**

```r
x <- list(a = 1, b = c(1,2,3), c = c(2,3,4))
list.sample(x, 2, weight = sum(.))
```

---

**list.save**  
Save a list to a file

**Description**

Save a list to a file

**Usage**

```r
list.save(x, file, type = tools::file_ext(file), ...)
```
Arguments

- **x**  
  The list to save
- **file**  
  The file for output
- **type**  
  The type of output which, by default, is determined by file extension. Currently supports RData, RDS, JSON, YAML.
- **...**  
  Additional parameters passed to the output function

Value

x will be returned.

Examples

```r
## Not run:
x <- lapply(1:5,function(i) data.frame(a=i,b=i^2))
list.save(x,quotesingle.Var 'list.rds'quotesingle.Var)
list.save(x,quotesingle.Var 'list.rds'quotesingle.Var)
list.save(x,quotesingle.Var 'list.yaml'quotesingle.Var)
list.save(x,quotesingle.Var 'list.json'quotesingle.Var)
## End(Not run)
```

list.search

Search a list recursively by an expression

Description

Search a list recursively by an expression

Usage

```
list.search(.data, expr, classes = "ANY", n, unlist = FALSE)
```

Arguments

- **.data**  
  A list or vector
- **expr**  
  a lambda expression
- **classes**  
  a character vector of class names that restrict the search. By default, the range is unrestricted (ANY).
- **n**  
  the maximal number of vectors to return
- **unlist**  
  logical Should the result be unlisted?
Details

`list.search` evaluates an expression (expr) recursively along a list (.data).

If the expression results in a single-valued logical vector and its value is `TRUE`, the whole vector will be collected. If it results in multi-valued or non-logical vector, the non-NA values resulted from the expression will be collected.

To search whole vectors that meet certain condition, specify the expression that returns a single logical value.

To search the specific values within the vectors, use subsetting in the expression, that is, `. [cond]` or lambda expression like `x -> x[cond]` where cond is a logical vector used to select the elements in the vector.

Examples

# Exact search

```r
x <- list(p1 = list(type='A', score=c(c1=9)),
          p2 = list(type=c('A', 'B'), score=c(c1=8, c2=9)),
          p3 = list(type=c('B', 'C'), score=c(c1=9, c2=7)),
          p4 = list(type=c('B', 'C'), score=c(c1=8, c2=NA)))
```

## Search exact values

```r
list.search(x, identical(., 'A'))
list.search(x, identical(., c('A', 'B')))  
list.search(x, identical(., c(9,7)))
list.search(x, identical(., c(c1=9, c2=7)))
```

## Search all equal values

```r
list.search(x, all( . == 9))
list.search(x, all( . == c(8,9)))
list.search(x, all( . == c(8,9), na.rm = TRUE))
```

## Search any equal values

```r
list.search(x, any( . == 9))
list.search(x, any( . == c(8,9)))
```

# Fuzzy search

```r
data <- list(
          p1 = list(name='Ken', age=24),
          p2 = list(name='Kent', age=26),
          p3 = list(name='Sam', age=24),
          p4 = list(name='Keynes', age=30),
          p5 = list(name='Kwen', age=31)
)
```

```r
list.search(data, grepl('^K\w*n$', .), 'character')
```

## Not run:

```r
library(stringdist)
list.search(data, stringdist(., 'Ken') <= 1, 'character')
```
list.select

Select by name or expression for each member of a list

Description

Select by name or expression for each member of a list

Usage

list.select(.data, ...)

Arguments

.data A list or vector

... A group of implicit labmda expressions

Examples

x <- list(p1 = list(type='A', score=list(c1=10,c2=8)),
          p2 = list(type='B', score=list(c1=9,c2=9)),
          p3 = list(type='B', score=list(c1=9,c2=7)))
list.select(x, type)
list.select(x, tp = type)
list.select(x, type, score)
list.select(x, type, score.range = range(unlist(score)))
list.serialize  Serialize a list

Description
Serialize a list

Usage
list.serialize(x, file, type = tools::file_ext(file), ...)

Arguments
- x  list
- file  The file for output
- type  The type of serialization, including native serializer and json serializer, which is by default determined by file extension
- ...  Additional parameters passed to the serializer function

See Also
list.unserialize

Examples
## Not run:
x <- list(a=1,b=2,c=3)
list.serialize(x, 'test.dat')
list.serialize(x, 'test.json')

## End(Not run)

list.skip  Skip a number of elements

Description
Skip the first n elements of a list or vector and return the remaining elements if any.

Usage
list.skip(.data, n)
**list.skipWhile**

**Arguments**

- `.data` A list or vector
- `n` integer. The number of elements to skip

**See Also**

`list.skipWhile, list.take, list.takeWhile`

**Examples**

```r
x <- list(a=1, b=2, c=3)
list.skip(x, 1)
list.skip(x, 2)
```

---

**list.skipWhile**

Keep skipping elements while a condition holds

**Description**

Keep skipping elements in a list or vector while a condition holds for the element. As long as the condition is violated, the element will be kept and all remaining elements are returned.

**Usage**

`list.skipWhile(.data, cond)`

**Arguments**

- `.data` A list or vector
- `cond` A logical lambda expression

**See Also**

`list.skip, list.take, list.takeWhile`

**Examples**

```r
x <- list(p1 = list(type='A', score=list(c1=10, c2=8)),
          p2 = list(type='B', score=list(c1=9, c2=9)),
          p3 = list(type='B', score=list(c1=9, c2=7)))
list.skipWhile(x, type=='A')
list.skipWhile(x, min(score$c1, score$c2) >= 8)
```
list.sort

Sort a list by given expressions

Description

Sort a list by given expressions

Usage

list.sort(.data, ..., na.last = NA)

Arguments

.data a list or vector

... A group of lambda expressions. For each expression, the data is sorted ascending by default unless the expression is enclosed by ()

na.last The way to deal with NAs.

See Also

list.order

Examples

x <- list(p1 = list(type='A', score=list(c1=10, c2=8)),
    p2 = list(type='B', score=list(c1=9, c2=9)),
    p3 = list(type='B', score=list(c1=9, c2=7)))
list.sort(x, type, (score$c2))  # sort by score$c2 in descending order
list.sort(x, min(score$c1, score$c2))

list.stack

Stack all list elements to tabular data

Description

Stack all list elements to tabular data

Usage

list.stack(.data, ..., data.table = FALSE)

Arguments

.data list of vectors, lists, data.frames or data.tables.

... additional parameters passed to data.table::rbindlist.

data.table TRUE to keep the result as data.table
Examples

```r
# Not run:
x <- lapply(1:3, function(i) { list(a=i, b=i^2) })
list.stack(x)

x <- lapply(1:3, function(i) { list(a=i, b=i^2, c=letters[i]) })
list.stack(x)

x <- lapply(1:3, function(i) { data.frame(a=i, b=i^2, c=letters[i]) })
list.stack(x)

x <- lapply(1:3, function(i) { data.frame(a=c(i,i+1), b=c(i^2,i^2+1)) })
list.stack(x)

# End(Not run)
```

---

**list.subset**

*Subset a list*

**Description**

Subset a list

**Usage**

`list.subset()`

**Examples**

```r
x <- list(p1 = list(type='A', score=list(c1=10, c2=8)),
          p2 = list(type='B', score=list(c1=9, c2=9)),
          p3 = list(type='B', score=list(c1=9, c2=7))

list.subset(x, c('p1', 'p2'))
list.subset(x, grepl('p', names(x)))

# Not run:
list.subset(x, stringdist::stringdist(names(x), 'x1') <= 1)

# End(Not run)
```

---

**list.table**

*Generate a table for a list by expression*

**Description**

Generate a table for a list by expression
list.take

Take a number of elements

Usage

list.take(.data, n, force = FALSE)

Arguments

.data list or vector

n integer. The number of elements to take

force TRUE to disable the length check

See Also

list.takeWhile, list.skip, list.skipWhile

Examples

x <- list(a=1,b=2,c=3)
list.take(x,1)
list.take(x,10)
**list.takeWhile**

Keep taking elements while a condition holds

**Description**

Keep taking elements out from a list or vector while a condition holds for the element. If the condition is violated for an element, the element will not be taken and all taken elements will be returned.

**Usage**

```
list.takeWhile(.data, cond)
```

**Arguments**

- `.data` list or vector
- `cond` A logical lambda expression

**See Also**

`list.take, list.skip, list.skipWhile`

**Examples**

```
x <- list(p1 = list(type='A', score=list(c1=10, c2=8)),
         p2 = list(type='B', score=list(c1=9, c2=9)),
         p3 = list(type='B', score=list(c1=9, c2=7)))
list.takeWhile(x, type=='B')
list.takeWhile(x, min(score$c1, score$c2) >= 8)
```

---

**list.ungroup**

Ungroup a list by taking out second-level elements

**Description**

This function reverses the grouping operation by taking out second-level elements of a nested list and removing the labels of the first-level elements. For example, a list may be created from paged data, that is, its first-level elements only indicate the page container. To unpage the list, the first-level elements must be removed and their inner elements should be taken out to to the first level.

**Usage**

```
list.ungroup(.data, level = 1L, ..., group.names = FALSE, sort.names = FALSE)
```
Arguments

.data list
level integer to indicate to which level of list elements should be ungrouped to the first level.
...
Preserved use of parameter passing
group.names logical. Should the group names be preserved?
sort.names logical. Should the members be sorted after ungrouping?

See Also
list.group

Examples

x <- list(p1 = list(type='Var A', score=list(c1=10,c2=8)),
        p2 = list(type='Var B', score=list(c1=9,c2=9)),
        p3 = list(type='Var B', score=list(c1=9,c2=7)))
xg <- list.group(x, type)
list.ungroup(xg)

x <- list(a = list(a1 = list(x=list(x1=2,x2=3), y=list(y1=1,y2=3))),
        b = list(b1 = list(x=list(x1=2,x2=6), y=list(y1=3,y2=2))))
list.ungroup(x, level = 1)
list.ungroup(x, level = 2)
list.ungroup(x, level = 2, group.names = TRUE)

list.unserialize Unserialize a file

Description
Unserialize a file

Usage
list.unserialize(file, type = tolower(tools::file_ext(file)), ...)

Arguments
file The file as input
type The type of serialization, including native unserializer and json unserializer, which is by default determined by file extension
...
Additional parameters passed to the unserializer function

See Also
list.serialize
Examples

```r
## Not run:
list.unserialize('test.dat')
list.unserialize('test.json')

## End(Not run)
```

### list.unzip

Transform a list of elements with similar structure into a list of decoupled fields

#### Description

Transform a list of elements with similar structure into a list of decoupled fields

#### Usage

```r
list.unzip(
  .data,
  .fields = c("intersect", "union"),
  ..., 
  .aggregate = "simplify2array",
  .missing = NA
)
```

#### Arguments

- `.data` A list of elements containing common fields
- `.fields` 'intersect' to select only common fields for all `.data`'s elements. 'union' to select any field that is defined in any elements in `.data`
- `...` The custom aggregate functions. Can be a named list of functions or character vectors. If a function is specified as a list of functions, then the functions will be evaluated recursively on the result of the field. Use `identity` to avoid aggregating results. Use `NULL` to remove certain field.
- `.aggregate` The default aggregate function, by default, `simplify2array`. Can be a function, character vector or a list of functions. Use `identity` to avoid aggregating results.
- `.missing` When `.fields` is 'union' and some elements do not contain certain fields, then `NULL` will be replaced by the value of `.missing`, by default, `NA`. This often makes the result more friendly.

#### See Also

- `list.zip`
Examples

```r
list.unzip(list(p1 = list(a = 1, b = 2), p2 = list(a = 2, b = 3)))
list.unzip(list(p1 = list(a = 1, b = 2), p2 = list(a = 2, b = 3, c = 4)))
list.unzip(list(p1 = list(a = 1, b = 2), p2 = list(a = 2, b = 3, c = 4)), 'union')
list.unzip(list(p1 = list(a = 1, b = 2), p2 = list(a = 2, b = 3, c = 4)), 'union', a = 'identity')
list.unzip(list(p1 = list(a = 1, b = 2), p2 = list(a = 2, b = 3, c = 4)), 'intersect', a = NULL)
```

```r
x <- list(april = list(n_days = 30,
                      holidays = list(list(2015-04-01, 'april fools'),
                                      list(2015-04-05, 'easter')),
                      month_info = c(number = 4, season = 'spring')),
                   july = list(n_days = 31,
                               holidays = list(list(2014-07-04, 'july 4th'),
                                               list(2014-07-04, 'july 4th')),
                               month_info = c(number = 7, season = 'summer')))
list.unzip(x, holidays = c('list.ungroup', 'unname', 'list.stack',
                            function(df) setNames(df, c("date", "name"))))
```

**Description**

The function updates each element of a list by evaluating a group of expressions in the scope of the element. If the name of an expression already exists in an list element, then the field with the name will be updated. Otherwise, the value with the name will be appended to the list element. The functionality is essentially done by `modifyList`.

**Usage**

```
list.update(.data, ..., keep.null = FALSE)
```

**Arguments**

- `.data` list
- `...` A group of lambda expressions
- `keep.null` Should NULL values be preserved for `modifyList`

**Examples**

```r
x <- list(p1 = list(type='A', score=list(c1=10,c2=8)),
          p2 = list(type='B', score=list(c1=9,c2=9)),
          p3 = list(type='B', score=list(c1=9,c2=7)))
list.update(x, high=max(score$c1, score$c2), low=min(score$c1, score$c2))
list.update(x, exams=length(score))
list.update(x, grade=ifelse(type=='A', score$c1, score$c2))
list.update(x, score=list(min=0, max=10))
```
**list.which**

*Give the indices of list elements satisfying a given condition*

**Description**

Give the indices of list elements satisfying a given condition

**Usage**

```r
list.which(.data, cond)
```

**Arguments**

- `.data` A list or vector
- `cond` A logical lambda expression

**Value**

an integer vector

**Examples**

```r
x <- list(p1 = list(type='A', score=list(c1=10,c2=8)),
          p2 = list(type='B', score=list(c1=9,c2=9)),
          p3 = list(type='B', score=list(c1=9,c2=7)))
list.which(x, type == 'B')
list.which(x, min(score$c1, score$c2) >= 8)
```

---

**list.zip**

*Combine multiple lists element-wisely.*

**Description**

Combine multiple lists element-wisely.

**Usage**

```r
list.zip(..., use.argnames = TRUE, use.names = TRUE)
```

**Arguments**

- `...` lists
- `use.argnames` logical. Should the names of the arguments be used as the names of list items?
- `use.names` logical. Should the names of the first argument be used as the zipped list?
See Also

list.unzip

Examples

```r
x <- list(1, 2, 3)
y <- list('x', 'y', 'z')
list.zip(num = x, sym = y)
```

nyweather

*New York hourly weather data*

**Description**

A non-tabular data of the hourly weather conditions of the New York City from 2013-01-01 to 2013-03-01.

**Usage**

```r
nyweather
```

**Format**

See https://openweathermap.org/weather-data

**Details**

Fetch date: 2014-11-23.
Processed by rlist.
To retrieve the data, please visit https://openweathermap.org/api for API usage.

---

**set_argnames**

*Make names for unnamed symbol arguments*

**Description**

Make names for unnamed symbol arguments

**Usage**

```r
set_argnames(args, data = args)
```

**Arguments**

- `args` - the unevaluated argument list
- `data` - the list to be named (args by default)
Details

The elements of an unevaluated list of arguments may or may not have names as given by user. For example, `list.select` requires user to specify the fields to select. These fields are unevaluated arguments, some of which are symbols and others are calls. For the symbols, it is natural to make the resulted lists to have the same name for the particular arguments.

subset.list

Subset a list by a logical condition

Description

Subset a list by a logical condition

Usage

```r
## S3 method for class 'list'
subset(x, subset, select, ...)
```

Arguments

- `x` The list to subset
- `subset` A logical lambda expression of subsetting condition
- `select` A lambda expression to evaluate for each selected item
- `...` Additional parameters

Examples

```r
x <- list(p1 = list(type='A', score=list(c1=10, c2=8)),
          p2 = list(type='B', score=list(c1=9, c2=9)),
          p3 = list(type='B', score=list(c1=9, c2=7))
subset(x, type == 'B')
subset(x, select = score)
subset(x, min(score$c1, score$c2) >= 8, data.frame(score))
subset(x, type == 'B', score$c1)
do.call(rbind,
        subset(x, min(score$c1, score$c2) >= 8, data.frame(score)))
```
tryEval

Try to evaluate an expression and return a default value if an error occurs or otherwise return its value.

Description
Try to evaluate an expression and return a default value if an error occurs or otherwise return its value.

Usage
tryEval(expr, def = NULL)

Arguments
expr the expression to evaluate
def the default value if an error occurs in the evaluation of expr

Examples
x <- list(a=c(x=1, y=2), b=c(x=2, p=3))
list.map(x, tryEval(x+y, NA))

tryGet
Try to get the value of a symbol if exists or return a default value

Description
Try to get the value of a symbol if exists or return a default value

Usage
tryGet(symbol, def = NULL, ..., envir = parent.frame())

Arguments
symbol the symbol to examine
def the default value if the symbol does not exist
... additional parameters passed to exists and get
envir the environment to examine whether the symbol exists and get the symbol

Details
By default, the symbol is examined in envir without inheritance, that is, if the symbol does not exist in envir the default value def will be returned.
Examples

```r
x <- list(a=c(x=1,y=2), b=c(x=2,p=3))
list.map(x, tryGet(y, 0))
```
Index

* datasets
  nyweather, 48
  .evalwith, 4
  args_env, 4
  args_list, 4
  callwith, 5
  contains, 5
  dots, 6
  expand.grid, 16
  getnames, 6
  is.empty, 7

List, 7
list.all, 8, 9
list.any, 8, 9
list.append, 10, 21, 32
list.apply, 10
list.cases, 11
list.cbind, 11, 12, 32
list.class, 12
list.clean, 13
list.common, 14
list.count, 14
list.do, 15
list.exclude, 15
list.expand, 16
list.extract, 17
list.filter, 17
list.find, 18
list.findi, 18
list.first, 19, 24
list.flatten, 20
list.group, 20, 44
list.if(list.is), 22
list.insert, 10, 21, 32

list.is, 22
list.iter, 23
list.join, 23
list.last, 19, 24
list.load, 25
list.map, 26, 27
list.maps, 26
list.mapv, 26, 27
list.match, 28
list.merge, 28
list.names, 29
list.order, 30, 40
list.parse, 30
list.prepend, 10, 21, 32
list.rbinding, 32
list.remove, 33
list.reverse, 33
list.sample, 34
list.save, 34
list.search, 35
list.select, 37
list.serialize, 38, 44
list.skip, 38, 39, 42, 43
list.skipWhile, 39, 39, 42, 43
list.sort, 30, 40
list.stack, 12, 32, 40
list.subset, 41
list.table, 41
list.take, 39, 42, 43
list.takeWhile, 39, 42, 43
list.ungroup, 21, 43
list.unserialize, 38, 44
list.unzip, 45, 48
list.update, 46
list.which, 47
list.zip, 45, 47

nyweather, 48
rlist-package, 3
INDEX

set_argnames, 48
subset.list, 49

tryEval, 50
tryGet, 50