

# Package ‘eList’

May 8, 2026

**Title** List Comprehension and Tools

**Version** 0.2.0

**Author** Chris Mann <cmann3@unl.edu>

**Maintainer** Chris Mann <cmann3@unl.edu>

**Description** Create list comprehensions (and other types of comprehension) similar to those in 'python', 'haskell', and other languages. List comprehension in 'R' converts a regular for() loop into a vectorized lapply() function. Support for looping with multiple variables, parallelization, and across non-standard objects included. Package also contains a variety of functions to help with list comprehension.

**License** MIT + file LICENSE

**BugReports** <https://github.com/cmnn3/eList/issues>

**Encoding** UTF-8

**Suggests** knitr, rmarkdown, stats, parallel

**VignetteBuilder** knitr

**LazyData** true

**RoxygenNote** 7.1.1

**NeedsCompilation** no

**Repository** CRAN

**Date/Publication** 2021-01-22 23:00:02 UTC

## Contents

.. . . . .	2
auto_cluster . . . . .	2
comprehendSummary . . . . .	3
comprehension . . . . .	5
flatten . . . . .	7
helpers . . . . .	8
ifor . . . . .	11
iter . . . . .	12
null.omit . . . . .	13

---

.. *Create Vector*

---

### Description

The .. function allows for the quick creation of vector using either ..(...) or ..[...]. It accepts vector [comprehension](#) arguments using for .... It can also be used as a more general form of [c](#).

### Usage

```
..(..., clust = NULL, type = Vec, simplify = TRUE)
```

### Arguments

...	values to be combined within a vector. Arguments beginning with for are interpreted as comprehensions.
clust	cluster to use for <a href="#">parallel</a> computations
type	<a href="#">comprehension</a> function used when for arguments are present. Defaults to Vec.
simplify	logical; should the result be simplified to an array if possible?

### Value

vector

### Examples

```
..[for (i in 1:10) 2*(1:i)]
```

---

auto\_cluster

*Quickly Create a Cluster for Parallel Comprehension*

---

### Description

A function to quickly create a cluster for use in parallel vector comprehensions. Use [makeCluster](#) from the [parallel](#) package for greater control. It defaults to making a PSOCK cluster on Windows systems and a Fork cluster on unix-based systems. [close\\_cluster](#) is a wrapper to [stopCluster](#).

### Usage

```
auto_cluster(ncore = detectCores() - 1)
```

```
close_cluster(clust)
```

**Arguments**

`ncore`            number of cores/nodes to use. If not specified, it attempts to detect the number of cores available and uses all but 1.

`clust`            cluster to close the connection to

**Value**

an object of class `c("SOCKcluster", "cluster")`

**Functions**

- `close_cluster`: close an open connection to a cluster

**Examples**

```
## Parallel vector comprehension
cluster <- auto_cluster(2)
Num(for (i in 1:1000) exp(sqrt(i)), clust=cluster)
close_cluster(cluster)
```

**Description**

Functions that summarize the results of a Python-style comprehension. These functions extend those in [comprehension](#) by applying a post-evaluation function to the results of the loop.

**Usage**

```
All(..., clust = NULL, na.rm = FALSE)

Any(..., clust = NULL, na.rm = FALSE)

None(..., clust = NULL, na.rm = FALSE)

Sum(..., clust = NULL, na.rm = FALSE)

Prod(..., clust = NULL, na.rm = FALSE)

Min(..., clust = NULL, na.rm = FALSE)

Max(..., clust = NULL, na.rm = FALSE)

Mean(..., clust = NULL, na.rm = FALSE, trim = 0)
```

```
Stats(..., clust = NULL, na.rm = FALSE, trim = 0)
```

```
Paste(..., clust = NULL, collapse = "")
```

### Arguments

...	vectors of any type or a for loop with format: for (var in seq) <name => <if (cond)> expr. See <a href="#">comprehension</a> .
clust	cluster to use for <a href="#">parallel</a> computations
na.rm	logical; should missing values be removed? Defaults to FALSE
trim	fraction between 0 and 0.5 describing percent of observations to be trimmed from each side for the mean
collapse	character describing how the results from Paste should be collapsed. See <a href="#">paste</a> .

### Value

Single numeric or character value, or a list of results for Stats

### Functions

- All: Are all results TRUE?
- Any: Are any results TRUE?
- None: Are all results FALSE?
- Sum: Calculate the [sum](#) of results
- Prod: Calculate the [prod](#) of results
- Min: Find the minimum in the result
- Max: Find the maximum in the result
- Mean: Calculate the arithmetic mean of the result
- Stats: Find the 7 number summary (5 number + mean & sd) of the result
- Paste: Collapse the result into a single character

### Examples

```
## Calculate the sum of all even numbers to 100
Sum(for (i in seq(2, 100, 2)) i)

## Find the mean
Mean(for (i in 1:10) log(i))

## Combine character values
greet <- c("Hello", "World", "Nice", "To", "Meet", "You")
val <- Paste(for (i,j in enum(greet)) paste0(i, ":", j), collapse="\n")
cat(val)
```

## Description

Functions that provide Python-style list (and related) comprehension. Comprehensions convert `for` loops into `lapply` functions before evaluation. Support for multiple variables, name assignment, nested loops, custom iterators, if-else statements, and variety of return types included.

## Usage

```
Comp(map = lapply, fun = NULL)
```

```
List(loop, clust = NULL, fun = NULL)
```

```
Env(loop, clust = NULL)
```

```
Vec(loop, clust = NULL, drop.names = FALSE)
```

```
Num(loop, clust = NULL, drop.names = FALSE)
```

```
Chr(loop, clust = NULL, drop.names = FALSE)
```

```
Logical(loop, clust = NULL, drop.names = FALSE)
```

```
Mat(loop, clust = NULL, by.col = TRUE)
```

```
DF(loop, clust = NULL)
```

## Arguments

<code>map</code>	function, such as <code>lapply</code> , that is used for the comprehension
<code>fun</code>	function to be called on result after comprehension
<code>loop</code>	a for loop with format: <code>for (var in seq) &lt;name =&gt; &lt;if (cond)&gt; expr</code> . See "details" below.
<code>clust</code>	cluster to use for <a href="#">parallel</a> computations
<code>drop.names</code>	logical; should names be dropped after conversion? Defaults to <code>FALSE</code> .
<code>by.col</code>	should comprehension on matrix group by columns or rows? Defaults to <code>TRUE</code> .

## Details

The comprehension functions parse an R loop expression into `lapply` functions to allow for more readable code and easy creation and conversion of vectors. The general syntax for a loop expression is as follows:

```
for (var in seq) <name=> <if (cond)> expr
```

where `<...>` denotes optional statements. The `seq` can be any R object: a list, matrix, data.frame, environment, function, etc. The function `iter` is called on the `seq`. So the behavior can be easily described for custom classes or objects. See [helpers](#) for functions like `zip` that can be used with `seq`.

Multiple variables can be used in `var` by separating the names with a period `"."`. For example, `i.j` is equivalent looping with variables `i` and `j`. The downside is that periods cannot be used in the `var` name. When multiple variables are used, the object received from the sequence at each iteration is split and its elements assigned in order to each of the variables. If the `var` is `i.j` and the object received in the iteration was `c(2, 4, 6)`, then `i=2`, `j=4`, and `6` would not be assigned. Since variables are split on periods, `i.j` could be used to assign the first and third elements, or `.i.j` the second and third. Any number of variables can be used. Note that the entire object is returned if there are no periods in the name, so use `i.` if only the first object is needed.

To provide names within a loop, preface the expression with the desired name for that particular object followed by `=`. `name` can be any expression, just make sure to surround any `if` chain for the name with parentheses, or the R parser will not detect that the assignment operator is associated with the `expr`. Behind the scenes, the expression on the left-hand side of `"="` is wrapped in an `sapply` function and the results are assigned to the `names` of the right-hand side result.

The `if` statement can contain any number of `if-else` statements and can be nested. Similarly, for statements can be nested any number of times and converted to `lapply` as long as the expression is a self-contained for loop.

Though comprehensions are functions, both `List(for ...)` and `List[for ...]` syntax are supported. See [..](#) for a convenience wrapper around `Vec`.

The different comprehensions primarily describe the return value, with `List` return a "list" and `Num` returning a numeric vector. If the object cannot be converted, then an error will be produced. For `Env`, the objects must be named. This means that either the name must be assigned within the loop or the loop is performed across a named object and the name is preserved. Another difference is that is some comprehensions - though related to atomic vectors - convert `for` to `sapply` while others convert to `lapply`.

The `Comp` function is used to create custom comprehensions. It should be supplied with a `map` function such as `lapply` that accepts arguments: `X` for the argument over which the comprehension iterates, `FUN` a function applied to each element, and `...` for additional arguments passed to the `FUN`. `Comp` also accepts a post-evaluation function, `fun`, that is applied to the result. This could be used to ensure that the result complies to some class or other restriction.

Users can also specify a cluster to use. If specified, then a parallel version of `lapply` or `sapply` is used based on `parLapply` and `parSapply` from the [parallel](#) package. This can greatly reduce the calculation time for different operations, but has additional overhead that makes the cost greater than the benefit for relatively small vectors. See [auto\\_cluster](#) for auto-creation.

## Value

Determined by the function. `List` returns an object of class 'list', `Num` returns a numeric vector, etc. See the descriptions of each function for their return type.

## Functions

- `Comp`: Create generalized comprehension function

- List: Generate a 'list' from a for loop
- Env: Generate an 'environment' from a for loop
- Vec: Generate a flat, atomic 'vector' from a for loop
- Num: Generate a 'numeric' vector from a for loop
- Chr: Generate a 'character' vector from a for loop
- Logical: Generate a 'logical' vector from a for loop
- Mat: Generate a 'matrix' from a for loop
- DF: Generate a 'data.frame' from a for loop

## Examples

```
people <- list(
  John = list(age = 30, weight = 180, mood = "happy", gender = "male"),
  April = list(age = 26, weight = 110, mood = "sad", gender = "female"),
  Jill = list(age = 42, weight = 125, mood = "ok", gender = "female")
)

weight_kg <- Num(for (i in people) i$weight/2.2)
gender <- Chr(for (i in people) i$gender)
gender_tab <- List(for (i in c("male", "female")) i = length(which(gender == i)))

Chr(for (..i.j in people) paste0(i, " & ", j))

Chr(for (i.j in items(people)) paste0(i, " is ", j$age, " years old."))

e <- Env(for (i.j in items(people)) i = j$age)
e$John

Num(for (i in 1:10) for (j in 2:6) if (i == j) i^2)
```

---

 flatten

*Flatten a List or Other Object*


---

## Description

Reduces the depth of a list or other object. Most non-atomic objects (matrix, data.frame, environments, etc.) are converted to a "list" in the first level flattening. Atomic vectors, functions, and other special objects return themselves.

## Usage

```
flatten(x, level = -1, ...)
```

**Arguments**

<code>x</code>	object of any class, but primarily designed for lists and other "deep" objects
<code>level</code>	numeric integer describing the depth at which to flatten the object. If <code>level &lt; 0</code> , the object will become as flat as possible.
<code>...</code>	objects passed to methods

**Details**

`flatten` maps itself to each object with the aggregate `x`, combining the results. Each time it is mapped, the level is reduced by 1. When `level == 0`, or an atomic vector or other special object is reached, `flatten` returns the object without mapping itself.

**Value**

flatter object

**Examples**

```
x <- list(a = 1, b = 2:5, c = list(list(1,2,3), 4, 5), 6)
flatten(x)
## returns: [1 2 3 4 5 1 2 3 4 5 6]

flatten(x, level=1)
## returns: [1 2 3 4 5 [1 2 3] 4 5 6]
```

---

helpers

*Helpers for Vector Comprehension*

---

**Description**

These functions help to create sequences for use in vector [comprehension](#).

**Usage**

```
items(x)

vals(x)

enum(x)

rows(x, ...)

cols(x, ...)

zip(..., fill = NA, longest = TRUE)
```

```
lrep(x, n = 2, axis = 0)
transpose(x, fill = NA, longest = TRUE)
slice(x, start, end, by = 1L)
roll(x, n = 2, fill = NULL, head = TRUE, ...)
unroll(x)
lagg(x, k = 1, fill = NA, axis = 0)
groups(x, g)
chars(x)
chain(x)
separate(x, n = 2, fill = NA)
first(x)
rest(x)
splitn(x, n = 1)
```

### Arguments

x	list, environment, or other vector
...	vectors to combine
fill	object with which to fill the vector when operating on elements with varying lengths or shifts.
longest	logical; should the longest item be used to determine the new length or shortest? Defaults to TRUE.
n	size of window for roll and separate, or position of item in which to split each element in splitn
axis	which axis to perform different operations? axis=0, the default, performs operations on each element in the list (columns), while axis=1 performs operations on each object within the elements of a list (rows).
start, end, by	integers of length 1 describing the sequence for slicing the vector. If missing, they will default to the start or end of the vector.
head	logical; should fill be at the head of the vector or the tail?
k	number of elements to shift right. Negative values of k shift to the left
g	vector of objects used to define groups

## Details

These functions transform vectors or other objects into lists, by adding elements, grouping objects, extracting certain elements, and so forth. These can be used in conjunction with vector [comprehension](#) to develop quick and readable code.

An example of how each of these can be used is seen here. Let `x` and `y` be given as follows.

```
x = list(a = 2, b = 4, c = 8) y = list(1:2, 2:3, 3:4)
```

Then the various helper functions will have the following effect.

- `chain(y)` => [1, 2, 2, 3, 3, 4]
- `chars("hello")` => ['h', 'e', 'l', 'l', 'o']
- `enum(x)` => [[1, 2], [2, 4], [3, 8]]
- `first(y)` => [1, 2, 3]
- `groups(x, c("z", "w", "z"))` => [{"z", [2, 8]}, {"w", [4]}]
- `items(x)` => [{"a", 2}, {"b", 4}, {"c", 8}]
- `lagg(x, 2)` => [[2, 4, 8], [NA, 2, 4], [NA, NA, 2]]
- `lrep(x, 3)` => [[2, 4, 8], [2, 4, 8], [2, 4, 8]]
- `rest(y)` => [[2], [3], [4]]
- `roll(x, 2)` => [[2, 4] [4, 8]]
- `separate(x, 2)` => [[2, 4], [8, NA]]
- `slice(x, 1, 2)` => [2, 8]
- `splitn(y)` => [[1], [2]], [[2], [3]], [[3], [4]]]
- `transpose(y)` => [[1, 2, 3], [2, 3, 4]]
- `unroll(y)` => [1, 2, 3, 4]
- `vals(x)` => [2, 4, 8]
- `zip(x, 1:3)` => [[2, 1], [4, 2], [8, 3]]

## Value

list or other vector

## Functions

- `items`: Create a list containing the name of each element of `x` and its value.
- `vals`: Extract the values of `x` without their names.
- `enum`: Create a list containing the index of each element of `x` and its value.
- `rows`: Create a list containing the rows of a `data.frame` or matrix
- `cols`: Create a list containing the columns of a `data.frame` or matrix
- `zip`: Merge two or more vectors into a list with each index containing values from each vector at that index.
- `lrep`: Repeat `x`, `n` times, with each repetition being an item in a list.

- `transpose`: Transpose a list or other object into a list. Opposite of `zip`.
- `slice`: Subset an object by a sequence: `start`, `end`, `by`. If `start` is missing, it is assumed to be 1. If `end` is missing, it is assumed to be the length of the object.
- `roll`: Create a list of objects containing `n` items from `x`, with `n-1` elements overlapping in a chain. Opposite of `unroll`.
- `unroll`: Flatten a list by combining the unique elements between each group of two elements. Opposite of `roll`.
- `lagg`: Create a list containing an object and each the first `k` lags of an object.
- `groups`: Create a list where each element is a list with the first element equal to a unique value in `g` and the other element is a list containing all values of `x` at the same indices as the value of `g`.
- `chars`: Convert a character string into a vector of single character values.
- `chain`: Combine each object in a list. Opposite of `separate`.
- `separate`: Separate vector into a list of objects with length `n`. Opposite of `chain`.
- `first`: Take the first element of each item in a list.
- `rest`: Remove the first element of each item in a list.
- `splitn`: Split each element in a list into two parts: one with the first `n` elements and the second with the rest.

## Examples

```
x <- 1:10
y <- 32:35

n <- Num(for (i,j in zip(x,y)) i+j)
# Note that the result is different from x+y since the shortest does not repeat
mean(n[1:4])

e <- new.env()
e$a <- 1:5
e$b <- 6:10

e2 <- Env(for (key.val in items(e)) key = sqrt(val))
e2$a

# row product
mat <- matrix(1:9, nrow=3)
Num(for (i in rows(mat)) prod(i))
```

**Description**

ifor evaluates an expression within a for loop, after applying `iter` to the sequence. ifor also allows multiple indexes by separating each variable name with a ".", such that `ifor(x, i.j, ...)` is similar to `for (i,j in x) ...` if for loops accepted multiple index values. See [comprehension](#) for more details. Assignment to a variable outside of the function can be accomplished through `assign` or `<<-`.

**Usage**

```
ifor(ind, x, expr)
```

**Arguments**

<code>ind</code>	variable name whose values are updated each round in the loop. Separate names with "." to allow for multiple variables
<code>x</code>	sequence over which to loop
<code>expr</code>	expression that is evaluated each round within the loop

**Value**

NULL invisibly

**Examples**

```
ifor(i.j, zip(1:4, 0:3),{
  print(i+j)
})
```

---

iter

*Create an Iterable Object*

---

**Description**

Vector [comprehension](#) iterates over an object, but the default behavior may not be desirable for custom classes. `iter` allows the user to specify how the object behaves within a comprehension, or other loop in the `eList` package. Unless a method is specified for an object, `iter` will attempt to convert it to a list except for atomic vectors.

**Usage**

```
iter(x)
```

**Arguments**

<code>x</code>	object to be looped across
----------------	----------------------------

**Value**

a vector

**Examples**

```
e <- new.env()
e$x <- 10
e$y <- letters[1:10]
iter(e)
```

---

null.omit

*Remove 'NULL' Entries from List*

---

**Description**

Function removes all items that are NULL or empty from a list or other object.

**Usage**

```
null.omit(x)
```

**Arguments**

x                    object to be checked

**Value**

x without NULL entries

**Examples**

```
l <- list(a=2, b=NULL, c = 3)
length(l) == 3

k <- null.omit(l)
length(k) == 2
```

# Index

..., 2, 6

All (comprehendSummary), 3  
Any (comprehendSummary), 3  
auto\_cluster, 2, 6

c, 2  
chain (helpers), 8  
chars (helpers), 8  
Chr (comprehension), 5  
close\_cluster (auto\_cluster), 2  
cols (helpers), 8  
Comp (comprehension), 5  
comprehendSummary, 3  
comprehension, 2–4, 5, 8, 10, 12

DF (comprehension), 5

enum (helpers), 8  
Env (comprehension), 5

first (helpers), 8  
flatten, 7  
for, 5

groups (helpers), 8

helpers, 6, 8

ifor, 11  
items (helpers), 8  
iter, 6, 12, 12

lagg (helpers), 8  
lapply, 5, 6  
List (comprehension), 5  
Logical (comprehension), 5  
lrep (helpers), 8

makeCluster, 2  
Mat (comprehension), 5

Max (comprehendSummary), 3  
Mean (comprehendSummary), 3  
Min (comprehendSummary), 3

names, 6  
None (comprehendSummary), 3  
null.omit, 13  
Num (comprehension), 5

parallel, 2, 4–6  
Paste (comprehendSummary), 3  
paste, 4  
Prod (comprehendSummary), 3  
prod, 4

rest (helpers), 8  
roll (helpers), 8  
rows (helpers), 8

separate (helpers), 8  
slice (helpers), 8  
splitn (helpers), 8  
Stats (comprehendSummary), 3  
stopCluster, 2  
Sum (comprehendSummary), 3  
sum, 4

transpose (helpers), 8

unroll (helpers), 8

vals (helpers), 8  
Vec (comprehension), 5

zip, 6  
zip (helpers), 8