

# Package ‘BBmisc’

September 29, 2022

**Title** Miscellaneous Helper Functions for B. Bischl

**Version** 1.13

**Description** Miscellaneous helper functions for and from B. Bischl and some other guys, mainly for package development.

**License** BSD\_2\_clause + file LICENSE

**URL** <https://github.com/berndbischl/BBmisc>

**BugReports** <https://github.com/berndbischl/BBmisc/issues>

**Imports** checkmate (>= 1.8.0), data.table, methods, stats, utils

**Suggests** codetools, microbenchmark, testthat

**ByteCompile** yes

**Encoding** UTF-8

**RoxygenNote** 7.2.1

**NeedsCompilation** yes

**Author** Bernd Bischl [aut, cre],  
Michel Lang [aut],  
Jakob Bossek [aut],  
Daniel Horn [aut],  
Jakob Richter [aut],  
Dirk Surmann [aut]

**Maintainer** Bernd Bischl <bernd\_bischl@gmx.net>

**Repository** CRAN

**Date/Publication** 2022-09-29 10:30:08 UTC

## R topics documented:

addClasses . . . . .	4
argsAsNamedList . . . . .	4
asMatrixCols . . . . .	5
asQuoted . . . . .	5
binPack . . . . .	6

capitalizeStrings . . . . .	7
catf . . . . .	7
cFactor . . . . .	8
checkArg . . . . .	9
checkListElementClass . . . . .	10
chunk . . . . .	11
clipString . . . . .	12
coalesce . . . . .	12
collapse . . . . .	13
collapsef . . . . .	14
computeMode . . . . .	14
convertDataFrameCols . . . . .	15
convertInteger . . . . .	16
convertIntegers . . . . .	17
convertListOfRowsToDataFrame . . . . .	17
convertMatrixType . . . . .	18
convertRowsToList . . . . .	19
convertToShortString . . . . .	20
dapply . . . . .	20
deprecated . . . . .	21
do.call2 . . . . .	22
dropNamed . . . . .	22
ensureVector . . . . .	23
explode . . . . .	24
extractSubList . . . . .	24
filterNull . . . . .	25
getAttributeNames . . . . .	26
getClass1 . . . . .	26
getFirst . . . . .	27
getMaxIndex . . . . .	27
getMaxIndexOfRows . . . . .	28
getOperatingSystem . . . . .	29
getRelativePath . . . . .	30
getUnixTime . . . . .	30
getUsedFactorLevels . . . . .	31
hasAttributes . . . . .	31
insert . . . . .	32
is.error . . . . .	32
isDirectory . . . . .	33
isEmptyDirectory . . . . .	34
isExpensiveExampleOk . . . . .	34
isFALSE . . . . .	35
isProperlyNamed . . . . .	35
isScalarNA . . . . .	36
isScalarValue . . . . .	36
isSubset . . . . .	37
isSuperset . . . . .	38
isValidName . . . . .	38

itostr . . . . .	39
lib . . . . .	39
load2 . . . . .	40
lsort . . . . .	41
makeDataFrame . . . . .	41
makeFileCache . . . . .	42
makeProgressBar . . . . .	43
makeS3Obj . . . . .	45
makeSimpleFileLogger . . . . .	45
mapValues . . . . .	46
messagef . . . . .	47
namedList . . . . .	48
names2 . . . . .	48
normalize . . . . .	49
optimizeSubInts . . . . .	50
pause . . . . .	51
printHead . . . . .	51
printStrToChar . . . . .	52
printToChar . . . . .	53
rangeVal . . . . .	53
requirePackages . . . . .	54
rowLapply . . . . .	55
save2 . . . . .	56
seq_row . . . . .	57
setAttribute . . . . .	57
setClasses . . . . .	58
setRowNames . . . . .	59
setValue . . . . .	59
sortByCol . . . . .	60
splitPath . . . . .	60
splitTime . . . . .	61
stopf . . . . .	61
strrepeat . . . . .	62
suppressAll . . . . .	63
syndiff . . . . .	63
system3 . . . . .	64
toRangeStr . . . . .	65
vlapply . . . . .	65
warningf . . . . .	66
which.first . . . . .	67
%btwn% . . . . .	67
%nin% . . . . .	68

---

addClasses	<i>A wrapper to add to the class attribute.</i>
------------	---

---

**Description**

A wrapper to add to the class attribute.

**Usage**

```
addClasses(x, classes)
```

**Arguments**

x	[any] Your object.
classes	[character] Classes to add. Will be added in front (specialization).

**Value**

Changed object x.

**Examples**

```
x = list()
print(class(x))
x = addClasses(x, c("foo1", "foo2"))
print(class(x))
```

---

argsAsNamedList	<i>Parses ... arguments to a named list.</i>
-----------------	--

---

**Description**

The deparsed name will be used for arguments with missing names. Missing names will be set to NA.

**Usage**

```
argsAsNamedList(...)
```

**Arguments**

... Arbitrary number of objects.

**Value**

list : Named list with objects.

**Examples**

```
z = 3
argsAsNamedList(x = 1, y = 2, z)
```

---

asMatrixCols	<i>Extracts a named element from a list of lists.</i>
--------------	---

---

**Description**

Extracts a named element from a list of lists.

**Usage**

```
asMatrixCols(xs, row.names, col.names)
asMatrixRows(xs, row.names, col.names)
```

**Arguments**

xs	[list] A list of vectors of the same length.
row.names	[character   integer   NULL] Row names of result. Default is to take the names of the elements of xs.
col.names	[character   integer   NULL] Column names of result. Default is to take the names of the elements of xs.

**Value**

matrix .

---

asQuoted	<i>Converts a string into a quoted expression.</i>
----------	--

---

**Description**

Works the same as if you would have entered the expression and called `quote` on it.

**Usage**

```
asQuoted(s, env = parent.frame())
```

**Arguments**

s	[character(1)] Expression as string.
env	[numeric(1)] Environment for expression. Default is parent.frame()

**Value**

Quoted expression.

**Examples**

```
asQuoted("x == 3")
```

---

binPack	<i>Simple bin packing.</i>
---------	----------------------------

---

**Description**

Maps numeric items in `x` into groups with sum less or equal than `capacity`. A very simple greedy algorithm is used, which is not really optimized for speed. This is a convenience function for smaller vectors, not a competitive solver for the real binpacking problem. If an element of `x` exceeds `capacity`, an error is thrown.

**Usage**

```
binPack(x, capacity)
```

**Arguments**

x	[numeric] Numeric vector of elements to group.
capacity	[numeric(1)] Maximum capacity of each bin, i.e., elements will be grouped so their sum does not exceed this limit.

**Value**

integer . Integer with values "1" to "n.bins" indicating bin membership.

**Examples**

```
x = 1:10
bp = binPack(x, 11)
xs = split(x, bp)
print(xs)
print(sapply(xs, sum))
```

---

capitalizeStrings      *Capitalize strings in a vector*

---

**Description**

Capitalise first word or all words of a character vector. Lower back of vector element or word, respectively.

**Usage**

```
capitalizeStrings(x, all.words = FALSE, lower.back = FALSE)
```

**Arguments**

x	[character(n)] Vector of character elements to capitalize.
all.words	[logical(1)] If TRUE all words of each vector element are capitalized. FALSE capitalizes the first word of each vector element.
lower.back	[logical(1)] TRUE lowers the back of each word or vector element (depends on all.words).

**Value**

Capitalized vector: [character(n)].

**Examples**

```
capitalizeStrings(c("the tail", "wags The dOg", "That looks fuNny!"))
capitalizeStrings(c("the tail", "wags The dOg", "That looks fuNny!")
, all.words = TRUE, lower.back = TRUE)
```

---

catf      *Wrapper for cat and sprintf.*

---

**Description**

A simple wrapper for cat(sprintf(...)).

**Usage**

```
catf(..., file = "", append = FALSE, newline = TRUE)
```

**Arguments**

...	[any] See <a href="#">sprintf</a> .
file	[character(1)] See <a href="#">cat</a> . Default is "".
append	[logical(1)] See <a href="#">cat</a> . Default is FALSE.
newline	[logical(1)] Append newline at the end? Default is TRUE.

**Value**

Nothing.

**Examples**

```
msg = "a message."  
catf("This is %s", msg)
```

---

cFactor

*Combine multiple factors and return a factor.*

---

**Description**

Note that function does not inherit from `c` to not change R semantics behind your back when this package is loaded.

**Usage**

```
cFactor(...)
```

**Arguments**

...	[factor] The factors.
-----	--------------------------

**Value**

factor .

**Examples**

```
f1 = factor(c("a", "b"))  
f2 = factor(c("b", "c"))  
print(c(f1, f2))  
print(cFactor(f1, f2))
```



---

checkArg	<i>Check for a function argument.</i>
----------	---------------------------------------

---

### Description

Throws exception if checks are not passed. Note that argument is evaluated when checked.

This function is superseded by the package **checkmate** and might get deprecated in the future. Please

### Usage

```
checkArg(  
  x,  
  cl,  
  s4 = FALSE,  
  len,  
  min.len,  
  max.len,  
  choices,  
  subset,  
  lower = NA,  
  upper = NA,  
  na.ok = TRUE,  
  formals  
)
```

### Arguments

x	[any] Argument.
cl	[character] Class that argument must “inherit” from. If multiple classes are given, x must “inherit” from at least one of these. See also argument s4.
s4	[logical(1)] If TRUE, use <code>is</code> for checking class <code>cl</code> , otherwise use <code>inherits</code> , which implies that only S3 classes are correctly checked. This is done for speed reasons as calling <code>is</code> is pretty slow. Default is FALSE.
len	[integer(1)] Length that argument must have. Not checked if not passed, which is the default.
min.len	[integer(1)] Minimal length that argument must have. Not checked if not passed, which is the default.
max.len	[integer(1)] Maximal length that argument must have. Not checked if not passed, which is the default.

choices	[any] Discrete number of choices, expressed by a vector of R objects. If passed, argument must be identical to one of these and nothing else is checked.
subset	[any] Discrete number of choices, expressed by a vector of R objects. If passed, argument must be identical to a subset of these and nothing else is checked.
lower	[numeric(1)] Lower bound for numeric vector arguments. Default is NA, which means not required.
upper	[numeric(1)] Upper bound for numeric vector arguments. Default is NA, which means not required.
na.ok	[logical(1)] Is it ok if a vector argument contains NAs? Default is TRUE.
formals	[character] If this is passed, x must be a function. It is then checked that formals are the names of the (first) formal arguments in the signature of x. Meaning checkArg(function(a, b), formals = "a") is ok. Default is missing.

**Value**

Nothing.

---

checkListElementClass *Check that a list contains only elements of a required type.*

---

**Description**

Check that argument is a list and contains only elements of a required type. Throws exception if check is not passed. Note that argument is evaluated when checked.

**Usage**

```
checkListElementClass(xs, cl)
```

**Arguments**

xs	[list] Argument.
cl	[character(1)] Class that elements must have. Checked with is.

**Value**

Nothing.

**Examples**

```
xs = as.list(1:3)
checkListElementClass(xs, "numeric")
```

---

 chunk
 

---



---

*Chunk elements of vectors into blocks of nearly equal size.*


---

**Description**

In case of shuffling and vectors that cannot be chunked evenly, it is chosen randomly which levels / chunks will receive 1 element less. If you do not shuffle, always the last chunks will receive 1 element less.

**Usage**

```
chunk(x, chunk.size, n.chunks, props, shuffle = FALSE)
```

**Arguments**

x	[ANY] Vector, list or other type supported by <a href="#">split</a> .
chunk.size	[integer(1)] Requested number of elements in each chunk. Cannot be used in combination with n.chunks or props. If x cannot be evenly chunked, some chunks will have less elements.
n.chunks	[integer(1)] Requested number of chunks. If more chunks than elements in x are requested, empty chunks are dropped. Can not be used in combination with chunks.size or props.
props	[numeric] Vector of proportions for chunk sizes. Empty chunks may occur, depending on the length of x and the given proportions. Cannot be used in combination with chunks.size or n.chunks.
shuffle	[logical(1)] Shuffle x? Default is FALSE.

**Value**

unnamed list of chunks.

**Examples**

```
xs = 1:10
chunk(xs, chunk.size = 3)
chunk(xs, n.chunks = 2)
chunk(xs, n.chunks = 2, shuffle = TRUE)
chunk(xs, props = c(7, 3))
```

---

clipString	<i>Shortens strings to a given length.</i>
------------	--

---

**Description**

Shortens strings to a given length.

**Usage**

```
clipString(x, len, tail = "...")
```

**Arguments**

x	[character] Vector of strings.
len	[integer(1)] Absolute length the string should be clipped to, including tail. Note that you cannot clip to a shorter length than tail.
tail	[character(1)] If the string has to be shortened at least 1 character, the final characters will be tail. Default is "...".

**Value**

character(1) .

**Examples**

```
print(clipString("abcdef", 10))
print(clipString("abcdef", 5))
```

---

coalesce	<i>Returns first non-missing, non-null argument.</i>
----------	--

---

**Description**

Returns first non-missing, non-null argument, otherwise NULL.

We have to perform some pretty weird [tryCatch](#) stuff internally, so you should better not pass complex function calls into the arguments that can throw exceptions, as these will be completely muffled, and return NULL in the end.

**Usage**

```
coalesce(...)
```

**Arguments**

... [any]  
Arguments.

**Value**

any .

**Examples**

```
f = function(x,y) {  
  print(coalesce(NULL, x, y))  
}  
f(y = 3)
```

---

collapse	<i>Collapse vector to string.</i>
----------	-----------------------------------

---

**Description**

A simple wrapper for `paste(x, collapse)`.

**Usage**

```
collapse(x, sep = ",")
```

**Arguments**

x [vector]  
Vector to collapse.

sep [character(1)]  
Passed to collapse in [paste](#). Default is “,”.

**Value**

character(1) .

**Examples**

```
collapse(c("foo", "bar"))  
collapse(c("foo", "bar"), sep = ";")
```

---

collapsef	<i>Collapse vector to string.</i>
-----------	-----------------------------------

---

**Description**

A simple wrapper for `collapse(sprintf, ...)`.

**Usage**

```
collapsef(..., sep = ",")
```

**Arguments**

...	[any] See <a href="#">sprintf</a> .
sep	[character(1)] See <a href="#">collapse</a> .

**Details**

Useful for vectorized call to [sprintf](#).

**Value**

character(1) .

---

computeMode	<i>Compute statistical mode of a vector (value that occurs most frequently).</i>
-------------	--

---

**Description**

Works for integer, numeric, factor and character vectors. The implementation is currently not extremely efficient.

**Usage**

```
computeMode(x, ties.method = "random", na.rm = TRUE)
```

**Arguments**

x	[vector] Factor, character, integer, numeric or logical vector.
ties.method	[character(1)] “first”, “random”, “last”: Decide which value to take in case of ties. Default is “random”.
na.rm	[logical(1)] If TRUE, missing values in the data removed. if FALSE, they are used as a separate level and this level could therefore be returned as the most frequent one. Default is TRUE.

**Value**

Modal value of length 1, data type depends on data type of x.

**Examples**

```
computeMode(c(1,2,3,3))
```

---

convertDataFrameCols *Converts columns in a data frame to characters, factors or numerics.*

---

**Description**

Converts columns in a data frame to characters, factors or numerics.

**Usage**

```
convertDataFrameCols(
  df,
  chars.as.factor = FALSE,
  factors.as.char = FALSE,
  ints.as.num = FALSE,
  logicals.as.factor = FALSE
)
```

**Arguments**

df	[data.frame] Data frame.
chars.as.factor	[logical(1)] Should characters be converted to factors? Default is FALSE.
factors.as.char	[logical(1)] Should characters be converted to factors? Default is FALSE.

ints.as.num      [logical(1)]  
                  Should integers be converted to numerics? Default is FALSE.

logicals.as.factor  
                  [logical(1)]  
                  Should logicals be converted to factors? Default is FALSE.

### Value

data.frame .

---

convertInteger	<i>Conversion for single integer.</i>
----------------	---------------------------------------

---

### Description

Convert single numeric to integer only if the numeric represents a single integer, e.g. 1 to 1L. Otherwise the argument is returned unchanged.

### Usage

```
convertInteger(x)
```

### Arguments

x                    [any]  
                  Argument.

### Value

Either a single integer if conversion was done or x unchanged.

### Examples

```
str(convertInteger(1.0))  
str(convertInteger(1.3))  
str(convertInteger(c(1.0, 2.0)))  
str(convertInteger("foo"))
```



---

convertIntegers	<i>Conversion for integer vector.</i>
-----------------	---------------------------------------

---

**Description**

Convert numeric vector to integer vector if the numeric vector fully represents an integer vector, e.g. `c(1, 5)` to `c(1L, 5L)`. Otherwise the argument is returned unchanged.

**Usage**

```
convertIntegers(x)
```

**Arguments**

x	[any] Argument.
---	--------------------

**Value**

Either an integer vector if conversion was done or x unchanged.

**Examples**

```
str(convertIntegers(1.0))
str(convertIntegers(1.3))
str(convertIntegers(c(1.0, 2.0)))
str(convertIntegers("foo"))
```

---

convertListOfRowsToDataFrame
------------------------------

---

*Convert a list of row-vector of equal structure to a data.frame.*

---

**Description**

Elements are arranged in columns according to their name in each element of rows. Variables that are not present in some row-lists, or encoded as NULL, are filled using NAs.

**Usage**

```
convertListOfRowsToDataFrame(  
  rows,  
  strings.as.factors = NULL,  
  row.names,  
  col.names  
)
```

**Arguments**

rows	[list] List of rows. Each row is a list or vector of the same structure, where all corresponding elements must have the same class. It is allowed that in some rows some elements are not present, see above.
strings.as.factors	[logical(1)] Convert character columns to factors? Default is default.stringsAsFactors() for R < "4.1.0" and FALSE otherwise.
row.names	[character   integer   NULL] Row names for result. By default the names of the list rows are taken.
col.names	[character   integer] Column names for result. By default the names of an element of rows are taken.

**Value**

data.frame .

**Examples**

```
convertListOfRowsToDataFrame(list(list(x = 1, y = "a"), list(x = 2, y = "b")))
```

---

convertMatrixType      *Converts storage type of a matrix.*

---

**Description**

Works by setting [mode](#).

**Usage**

```
convertMatrixType(x, type)
```

**Arguments**

x	[matrix] . Matrix to convert.
type	[character(1)] New storage type.

**Value**

matrix .

**Note**

as.mytype drops dimension when used on a matrix.

---

convertRowsToList      *Convert rows (columns) of data.frame or matrix to lists.*

---

### Description

For each row, one list/vector is constructed, each entry of the row becomes a list/vector element.

### Usage

```
convertRowsToList(  
  x,  
  name.list = TRUE,  
  name.vector = FALSE,  
  factors.as.char = TRUE,  
  as.vector = TRUE  
)
```

```
convertColsToList(  
  x,  
  name.list = FALSE,  
  name.vector = FALSE,  
  factors.as.char = TRUE,  
  as.vector = TRUE  
)
```

### Arguments

x	[matrix   data.frame] Object to convert.
name.list	[logical(1)] Name resulting list with names of rows (cols) of x? Default is FALSE.
name.vector	[logical(1)] Name vector elements in resulting list with names of cols (rows) of x? Default is FALSE.
factors.as.char	[logical(1)] If x is a data.frame, convert factor columns to string elements in the resulting lists? Default is TRUE.
as.vector	[logical(1)] If x is a matrix, store rows as vectors in the resulting list - or otherwise as lists? Default is TRUE.

### Value

list of lists or vectors .

---

`convertToShortString` *Converts any R object to a descriptive string so it can be used in messages.*

---

### Description

Atomics: If of length 0 or 1, they are basically printed as they are. Numerics are formatted with `num.format`. If of length greater than 1, they are collapsed with “,” and clipped. so they do not become excessively long. Expressions will be converted to plain text.

All others: Currently, only their class is simply printed like “<data.frame>”.

Lists: The mechanism above is applied (non-recursively) to their elements. The result looks like this: “a=1, <unnamed>=2, b=<data.frame>, c=<list>”.

### Usage

```
convertToShortString(x, num.format = "%.4g", clip.len = 15L)
```

### Arguments

<code>x</code>	[any] The object.
<code>num.format</code>	[character(1)] Used to format numerical scalars via <code>sprintf</code> . Default is “%.4g”.
<code>clip.len</code>	[integer(1)] Used clip atomic vectors via <code>clipString</code> . Default is 15.

### Value

character(1) .

### Examples

```
convertToShortString(list(a = 1, b = NULL, "foo", c = 1:10))
```

---

`dapply` *Call lapply on an object and return a data.frame.*

---

### Description

Applies a function `fun` on each element of input `x` and combines the results as `data.frame` columns. The results will get replicated to have equal length if necessary and possible.

### Usage

```
dapply(x, fun, ..., col.names)
```

**Arguments**

x	[data.frame] Data frame.
fun	[function] The function to apply.
...	[any] Further arguments passed down to fun.
col.names	[character(1)] Column names for result. Default are the names of x.

**Value**

data.frame .

---

deprecated	<i>Deprecated function. Do not use!</i>
------------	---

---

**Description**

Deprecated function. Do not use!

**Usage**

```
convertDfCols(
  df,
  chars.as.factor = FALSE,
  factors.as.char = FALSE,
  ints.as.num = FALSE,
  logicals.as.factor = FALSE
)

listToShortString(x, num.format = "%.4g", clip.len = 15L)
```

**Arguments**

df	No text
chars.as.factor	
factors.as.char	No text
ints.as.num	No text
logicals.as.factor	No text
x	No text
num.format	No text
clip.len	No text

---

do.call2	<i>Execute a function call similar to do.call.</i>
----------	--

---

### Description

This function is supposed to be a replacement for `do.call` in situations where you need to pass big R objects. Unlike `do.call`, this function allows to pass objects via `...` to avoid a copy.

### Usage

```
do.call2(fun, ..., .args = list())
```

### Arguments

<code>fun</code>	[character(1)] Name of the function to call.
<code>...</code>	[any] Arguments to fun. Best practice is to specify them in a key = value syntax.
<code>.args</code>	[list] Arguments to fun as a (named) list. Will be passed after arguments in <code>...</code> . Default is <code>list()</code> .

### Value

Return value of fun.

### Examples

```
## Not run:
library(microbenchmark)
x = 1:1e7
microbenchmark(do.call(head, list(x, n = 1)), do.call2("head", x, n = 1))

## End(Not run)
```

---

dropNamed	<i>Drop named elements of an object.</i>
-----------	--

---

### Description

Drop named elements of an object.

### Usage

```
dropNamed(x, drop = character(0L))
```

**Arguments**

x	[any] Object to drop named elements from. For a matrix or a data frames this function drops named columns via the second argument of the binary index operator [,]. Otherwise, the unary index operator [] is used for dropping.
drop	[character] Names of elements to drop.

**Value**

Subset of object of same type as x. The object is not simplified, i.e, no dimensions are dropped as [, , drop = FALSE] is used.

---

ensureVector	<i>Blow up single scalars / objects to vectors / list by replication.</i>
--------------	---

---

**Description**

Useful for standard argument conversion where a user can input a single element, but this has to be replicated now n times for a resulting vector or list.

**Usage**

```
ensureVector(x, n = 1L, cl = NULL, names = NULL, ensure.list = FALSE)
```

**Arguments**

x	[any] Input element.
n	[integer(1)] Desired length. Default is 1 (the most common case).
cl	[character*] Only do the operation if x inherits from this one of these classes, otherwise simply let x pass. Default is NULL which means to always do the operation.
names	[character*] Names for result. Default is NULL, which means no names.
ensure.list	[logical(1)] Should x be wrapped in a list in any case? Default is FALSE, i.e., if x is a scalar value, a vector is returned.

**Value**

Ether a vector or list of length n with replicated x or x unchanged..

---

explode	<i>Split up a string into substrings.</i>
---------	---

---

**Description**

Split up a string into substrings according to a seperator.

**Usage**

```
explode(x, sep = " ")
```

**Arguments**

x	[character] Source string.
sep	[character] Seperator whcih is used to split x into substrings. Default is " ".

**Value**

vector Vector of substrings.

**Examples**

```
explode("foo bar")  
explode("comma,seperated,values", sep = ",")
```

---

extractSubList	<i>Extracts a named element from a list of lists.</i>
----------------	---

---

**Description**

Extracts a named element from a list of lists.

**Usage**

```
extractSubList(xs, element, element.value, simplify = TRUE, use.names = TRUE)
```



**Arguments**

xs	[list] A list of named lists.
element	[character] Name of element(s) to extract from the list elements of xs. What happens is this: x\$el1\$el2.....
element.value	[any] If given, <code>vapply</code> is used and this argument is passed to FUN.VALUE. Note that even for repeated indexing (if <code>length(element) &gt; 1</code> ) you only pass one value here which refers to the data type of the final result.
simplify	[logical(1)   character(1)] If FALSE <code>lapply</code> is used, otherwise <code>sapply</code> . If “cols”, we expect the elements to be vectors of the same length and they are arranged as the columns of the resulting matrix. If “rows”, likewise, but rows of the resulting matrix. Default is TRUE.
use.names	[logical(1)] If TRUE and xs is named, the result is named as xs, otherwise the result is unnamed. Default is TRUE.

**Value**

list | simplified vector | matrix . See above.

**Examples**

```
xs = list(list(a = 1, b = 2), list(a = 5, b = 7))
extractSubList(xs, "a")
extractSubList(xs, "a", simplify = FALSE)
```

---

filterNull *Filter a list for NULL values*

---

**Description**

Filter a list for NULL values

**Usage**

```
filterNull(li)
```

**Arguments**

li	[list] List.
----	-----------------

**Value**

list .

---

getAttributeNames	<i>Helper function for determining the vector of attribute names of a given object.</i>
-------------------	---

---

**Description**

Helper function for determining the vector of attribute names of a given object.

**Usage**

```
getAttributeNames(obj)
```

**Arguments**

obj	[any] Source object.
-----	-------------------------

**Value**

character Vector of attribute names for the source object.

---

getClass1	<i>Wrapper for class(x)[1].</i>
-----------	---------------------------------

---

**Description**

Wrapper for class(x)[1].

**Usage**

```
getClass1(x)
```

**Arguments**

x	[any] Input object.
---	------------------------

**Value**

character(1) .

**Note**

getClass is a function in methods. Do not confuse.

---

getFirst	<i>Get the first/last element of a list/vector.</i>
----------	---

---

**Description**

Get the first/last element of a list/vector.

**Usage**

```
getFirst(x)
```

```
getLast(x)
```

**Arguments**

x	[list   vector] The list or vector.
---	--

**Value**

Selected element. The element name is dropped.

---

getMaxIndex	<i>Return index of maximal/minimal/best element in numerical vector.</i>
-------------	--

---

**Description**

If x is empty or only contains NAs which are to be removed, -1 is returned.

**Usage**

```
getMaxIndex(x, weights = NULL, ties.method = "random", na.rm = FALSE)
```

```
getMinIndex(x, weights = NULL, ties.method = "random", na.rm = FALSE)
```

```
getBestIndex(x, weights = NULL, minimize = TRUE, ...)
```

**Arguments**

x	[numeric] Input vector.
weights	[numeric] Weights (same length as x). If these are specified, the index is selected from x * w. Default is NULL which means no weights.

ties.method	[character(1)] How should ties be handled? Possible are: “random”, “first”, “last”. Default is “random”.
na.rm	[logical(1)] If FALSE, NA is returned if an NA is encountered in x. If TRUE, NAs are disregarded. Default is FALSE
minimize	[logical(1)] Minimal element is considered best? Default is TRUE.
...	[any] Further arguments passed down to the delegate.

**Value**

integer(1) .

**Note**

Function `getBestIndex` is a simple wrapper for `getMinIndex` or `getMaxIndex` respectively depending on the argument `minimize`.

---

<code>getMaxIndexOfRows</code>	<i>Find row- or columnwise the index of the maximal / minimal element in a matrix.</i>
--------------------------------	--

---

**Description**

`getMaxIndexOfRows` returns the index of the maximal element of each row. `getMinIndexOfRows` returns the index of the minimal element of each row. `getMaxIndexOfCols` returns the index of the maximal element of each col. `getMinIndexOfCols` returns the index of the minimal element of each col. If a corresponding vector (row or col) is empty, possibly after NA removal, -1 is returned as index.

**Usage**

```
getMaxIndexOfRows(x, weights = NULL, ties.method = "random", na.rm = FALSE)
```

```
getMinIndexOfRows(x, weights = NULL, ties.method = "random", na.rm = FALSE)
```

```
getMaxIndexOfCols(x, weights = NULL, ties.method = "random", na.rm = FALSE)
```

```
getMinIndexOfCols(x, weights = NULL, ties.method = "random", na.rm = FALSE)
```

**Arguments**

x	[matrix(n,m)] Numerical input matrix.
weights	[numeric] Weights (same length as number of rows/cols). If these are specified, the index is selected from the weighted elements (see <a href="#">getMaxIndex</a> ). Default is NULL which means no weights.
ties.method	[character(1)] How should ties be handled? Possible are: “random”, “first”, “last”. Default is “random”.
na.rm	[logical(1)] If FALSE, NA is returned if an NA is encountered in x. If TRUE, NAs are disregarded. Default is FALSE

**Value**

integer(n) .

**Examples**

```
x = matrix(runif(5 * 3), ncol = 3)
print(x)
print(getMaxIndexofRows(x))
print(getMinIndexofRows(x))
```

---

getOperatingSystem      *Functions to determine the operating system.*

---

**Description**

- `getOperatingSystem` Simple wrapper for `.Platform$OS`. type, returns `character(1)`.
- `isUnixPredicate` for OS string, returns `logical(1)`. Currently this would include Unix, Linux and Mac flavours.
- `isLinuxPredicate` for `sysname` string, returns `logical(1)`.
- `isDarwinPredicate` for `sysname` string, returns `logical(1)`.
- `isWindowsPredicate` for OS string, returns `logical(1)`.

**Usage**

```
getOperatingSystem()

isWindows()

isUnix()
```

isLinux()

isDarwin()

### Value

See above.

---

getRelativePath	<i>Construct a path relative to another</i>
-----------------	---

---

### Description

Constructs a relative path from path from to path to. If this is not possible (i.e. different drive letters on windows systems), NA is returned.

### Usage

```
getRelativePath(to, from = getwd(), ignore.case = isWindows())
```

### Arguments

to	[character(1)] Where the relative path should point to.
from	[character(1)] From which part to start. Default is <a href="#">getwd</a> .
ignore.case	[logical(1)] Should path comparisons be made case insensitive? Default is TRUE on Windows systems and FALSE on other systems.

### Value

character(1) : A relative path.

---

getUnixTime	<i>Current time in seconds.</i>
-------------	---------------------------------

---

### Description

Simple wrapper for `as.integer(Sys.time())`.

### Usage

```
getUnixTime()
```

### Value

integer(1) .

---

getUsedFactorLevels     *Determines used factor levels.*

---

**Description**

Determines the factor levels of a factor type vector that are actually occurring in it.

**Usage**

```
getUsedFactorLevels(x)
```

**Arguments**

x	[factor] The factor.
---	-------------------------

**Value**

character

---

hasAttributes     *Check if given object has certain attributes.*

---

**Description**

Check if given object has certain attributes.

**Usage**

```
hasAttributes(obj, attribute.names)
```

**Arguments**

obj	[mixed] Arbitrary R object.
attribute.names	[character] Vector of strings, i.e., attribute names.

**Value**

logical(1) TRUE if object x contains all attributes from attributeNames and FALSE otherwise.

---

insert	<i>Insert elements from one list/vector into another list/vector.</i>
--------	---

---

**Description**

Inserts elements from xs2 into xs1 by name, overwriting elements of equal names.

**Usage**

```
insert(xs1, xs2, elements)
```

**Arguments**

xs1	[list] First list/vector.
xs2	[list] Second vector/list. Must be fully and uniquely named.
elements	[character] Elements from xs2 to insert into xs1. Default is all.

**Value**

x1 with replaced elements from x2.

**Examples**

```
xs1 = list(a = 1, b = 2)
xs2 = list(b = 1, c = 4)
insert(xs1, xs2)
insert(xs1, xs2, elements = "c")
```

---

is.error	<i>Is return value of try an exception?</i>
----------	---

---

**Description**

Checks if an object is of class “try-error” or “error”.

**Usage**

```
is.error(x)
```

**Arguments**

x	[any] Any object, usually the return value of <a href="#">try</a> , <a href="#">tryCatch</a> , or a function which may return a <a href="#">simpleError</a> .
---	--



**Value**

logical(1) .

**Examples**

```
x = try(stop("foo"))
print(is.error(x))
x = 1
print(is.error(x))
```

---

isDirectory	<i>Is one / are several files a directory?</i>
-------------	--

---

**Description**

If a file does not exist, FALSE is returned.

**Usage**

```
isDirectory(...)
```

**Arguments**

```
... [character(1)]
File names, all strings.
```

**Value**

logical .

**Examples**

```
print(isDirectory(tempdir()))
print(isDirectory(tempfile()))
```

---

isEmptyDirectory      *Is one / are several directories empty?*

---

**Description**

If file does not exist or is not a directory, FALSE is returned.

**Usage**

```
isEmptyDirectory(...)
```

**Arguments**

```
...                    [character(1)]  
                      Directory names, all strings.
```

**Value**

logical .

**Examples**

```
print(isEmptyDirectory(tempdir()))  
print(isEmptyDirectory(tempfile()))
```

---

isExpensiveExampleOk      *Conditional checking for expensive examples.*

---

**Description**

Queries environment variable “R\_EXPENSIVE\_EXAMPLE\_OK”. Returns TRUE iff set exactly to “TRUE”. This allows conditional checking of expensive examples in packages via R CMD CHECK, so they are not run on CRAN, but at least on your local computer. A better option than “dont\_run” in many cases, where such examples are not checked at all.

**Usage**

```
isExpensiveExampleOk()
```

**Value**

logical(1) .

**Examples**

```
# extremely costly random number generation, that we dont want checked on CRAN
if (isExpensiveExampleOk()) {
  runif(1)
}
```

---

isFALSE	<i>A wrapper for identical(x, FALSE).</i>
---------	---

---

**Description**

A wrapper for identical(x, FALSE).

**Usage**

```
isFALSE(x)
```

**Arguments**

x	[any] Your object.
---	-----------------------

**Value**

logical(1) .

**Examples**

```
isFALSE(0)
isFALSE(FALSE)
```

---

isProperlyNamed	<i>Are all elements of a list / vector uniquely named?</i>
-----------------	--

---

**Description**

NA or "" are not allowed as names.

**Usage**

```
isProperlyNamed(x)
```

**Arguments**

x	[vector] The vector or list.
---	---------------------------------

**Value**

logical(1) .

**Examples**

```
isProperlyNamed(list(1))
isProperlyNamed(list(a = 1))
isProperlyNamed(list(a = 1, 2))
```

---

isScalarNA	<i>Checks whether an object is a scalar NA value.</i>
------------	---

---

**Description**

Checks whether object is from (NA, NA\_integer\_, NA\_real\_, NA\_character\_, NA\_complex\_).

**Usage**

```
isScalarNA(x)
```

**Arguments**

x	[any] Object to check.
---	---------------------------

**Value**

logical(1) .

---

isScalarValue	<i>Is given argument an atomic vector or factor of length 1?</i>
---------------	--

---

**Description**

More specific functions for scalars of a given type exist, too.

**Usage**

```
isScalarValue(x, na.ok = TRUE, null.ok = FALSE, type = "atomic")
isScalarLogical(x, na.ok = TRUE, null.ok = FALSE)
isScalarNumeric(x, na.ok = TRUE, null.ok = FALSE)
isScalarInteger(x, na.ok = TRUE, null.ok = FALSE)
```

```
isScalarComplex(x, na.ok = TRUE, null.ok = FALSE)
```

```
isScalarCharacter(x, na.ok = TRUE, null.ok = FALSE)
```

```
isScalarFactor(x, na.ok = TRUE, null.ok = FALSE)
```

### Arguments

x	[any] Argument.
na.ok	[logical(1)] Is NA considered a scalar? Default is TRUE.
null.ok	[logical(1)] Is NULL considered a scalar? Default is FALSE.
type	[character(1)] Allows to restrict to specific type, e.g., “numeric”? But instead of this argument you might want to consider using isScalar<Type>. Default is “atomic”, so no special restriction.

### Value

logical(1) .

---

isSubset	<i>Check subset relation on two vectors.</i>
----------	--

---

### Description

Check subset relation on two vectors.

### Usage

```
isSubset(x, y, strict = FALSE)
```

### Arguments

x	[vector] Source vector.
y	[vector] Vector of the same mode as x.
strict	[logical(1)] Checks for strict/proper subset relation.

### Value

logical(1) TRUE if each element of x is also contained in y, i. e., if x is a subset of y and FALSE otherwise.

---

isSuperset	<i>Check superset relation on two vectors.</i>
------------	--

---

**Description**

Check superset relation on two vectors.

**Usage**

```
isSuperset(x, y, strict = FALSE)
```

**Arguments**

x	[vector] Source vector.
y	[vector] Vector of the same mode as x.
strict	[logical(1)] Checks for strict/proper superset relation.

**Value**

logical(1) TRUE if each element of y is also contained in x, i. e., if y is a subset of x and FALSE otherwise.

---

isValidName	<i>Can some strings be used for column or list element names without problems?</i>
-------------	--

---

**Description**

Can some strings be used for column or list element names without problems?

**Usage**

```
isValidName(x, unique = TRUE)
```

**Arguments**

x	[character] Character vector to check.
unique	[logical(1)] Should the names be unique? Default is TRUE.

**Value**

logical . One Boolean entry for each string in x. If the entries are not unique and unique is enabled, the first duplicate will be FALSE.

---

itostr                      *Convert Integers to Strings*

---

**Description**

This is the counterpart of `strtoi`. For a base greater than '10', letters 'a' to 'z' are used to represent '10' to '35'.

**Usage**

```
itostr(x, base = 10L)
```

**Arguments**

x	[integer] Vector of integers to convert.
base	[integer(1)] Base for conversion. Values between 2 and 36 (inclusive) are allowed.

**Value**

character(length(x)).

**Examples**

```
# binary representation of the first 10 natural numbers
itostr(1:10, 2)

# base36 encoding of a large number
itostr(1e7, 36)
```

---

lib                      *A wrapper for library.*

---

**Description**

Tries to load packages. If the packages are not found, they will be installed from the default repository. This function is intended for use in interactive sessions and should not be used by other packages.

**Usage**

```
lib(...)
```

**Arguments**

```
...          [any]
             Package names.
```

**Value**

logical : Named logical vector determining the success of package load.

**Examples**

```
## Not run:
lib("BBmisc", "MASS", "rpart")

## End(Not run)
```

---

load2	<i>Load RData file and return objects in it.</i>
-------	--

---

**Description**

Load RData file and return objects in it.

**Usage**

```
load2(file, parts, simplify = TRUE, envir, impute)
```

**Arguments**

file	[character(1)] File to load.
parts	[character] Elements in file to load. Default is all.
simplify	[logical(1)] If TRUE, a list is only returned if parts and the file contain both more than 1 element, otherwise the element is directly returned. Default is TRUE.
envir	[environment(1)] Assign objects to this environment. Default is not to assign.
impute	[ANY] If file does not exists, return impute instead. Default is missing which will result in an exception if file is not found.



**Value**

Either a single object or a list.

**Examples**

```
fn = tempfile()
save2(file = fn, a = 1, b = 2, c = 3)
load2(fn, parts = "a")
load2(fn, parts = c("a", "c"))
```

---

lsort	<i>A wrapper for <a href="#">sort</a> to sort using the “C” collating rules.</i>
-------	--

---

**Description**

A wrapper for [sort](#) to sort using the “C” collating rules.

**Usage**

```
lsort(...)
```

**Arguments**

... Options passed to [sort](#).

**Value**

See [sort](#).

---

makeDataFrame	<i>Initialize data.frame in a convenient way.</i>
---------------	---

---

**Description**

Initialize data.frame in a convenient way.

**Usage**

```
makeDataFrame(
  nrow,
  ncol,
  col.types,
  init,
  row.names = NULL,
  col.names = sprintf("%i", seq_len(ncol))
)
```

**Arguments**

nrow	[integer(1)] Number of rows.
ncol	[integer(1)] Number of columns.
col.types	[character(ncol)   character(1)] Data types of columns. If you only pass one type, it will be replicated. Supported are all atomic modes also supported by <a href="#">vector</a> , i.e. all common data frame types except factors.
init	[any] Scalar object to initialize all elements of the data.frame. You do not need to specify col.types if you pass this.
row.names	[character   integer   NULL] Row names. Default is NULL.
col.names	[character   integer] Column names. Default is “V1”, “V2”, and so on.

**Examples**

```
print(makeDataFrame(3, 2, init = 7))
print(makeDataFrame(3, 2, "logical"))
print(makeDataFrame(3, 2, c("logical", "numeric")))
```

---

makeFileCache	<i>A caching wrapper around load2.</i>
---------------	--

---

**Description**

This closure returns a wrapper around [load2](#) which per default caches loaded objects and returns the cached version in subsequent calls.

**Usage**

```
makeFileCache(use.cache = TRUE)
```

**Arguments**

use.cache	[logical(1)] Enable the cache? Default is TRUE.
-----------	--

**Value**

function() with argument slot (name of the slot to cache the object in, default is “default”). All other arguments are passed down to [load2](#).

---

makeProgressBar	<i>Create a progress bar with estimated time.</i>
-----------------	---

---

## Description

Create a progress bar function that displays the estimated time till completion and optional messages. Call the returned functions `set` or `inc` during a loop to change the display. Note that you are not allowed to decrease the value of the bar. If you call these function without setting any of the arguments the bar is simply redrawn with the current value. For errorhandling use `error` and have a look at the example below.

You can globally change the behavior of all bars by setting the option `options(BBmisc.ProgressBar.style)` either to `"text"` (the default) or `"off"`, which display no bars at all.

You can globally change the width of all bars by setting the option `options(BBmisc.ProgressBar.width)`. By default this is `getOption("width")`.

You can globally set the stream where the output of the bar is directed by setting the option `options(BBmisc.ProgressBar.stream)` either to `"stderr"` (the default) or `"stdout"`. Note that using the latter will result in the bar being shown in reports generated by Sweave or knitr, what you probably do not want.

## Usage

```
makeProgressBar(
  min = 0,
  max = 100,
  label = "",
  char = "+",
  style = getOption("BBmisc.ProgressBar.style", "text"),
  width = getOption("BBmisc.ProgressBar.width", getOption("width")),
  stream = getOption("BBmisc.ProgressBar.stream", "stderr")
)
```

## Arguments

min	[numeric(1)] Minimum value, default is 0.
max	[numeric(1)] Maximum value, default is 100.
label	[character(1)] Label shown in front of the progress bar. Note that if you later set <code>msg</code> in the progress bar function, the message will be left-padded to the length of this label, therefore it should be at least as long as the longest message you want to display. Default is <code>""</code> .
char	[character(1)] A single character used to display progress in the bar. Default is <code>'+'</code> .

style	[character(1)] Style of the progress bar. Default is set via options (see details).
width	[integer(1)] Width of the progress bar. Default is set via options (see details).
stream	[character(1)] Stream to use. Default is set via options (see details).

## Value

`ProgressBar` . A list with following functions:

set [function(value, msg = label)]	Set the bar to a value and possibly display a message instead of the label.
inc [function(value, msg = label)]	Increase the bar and possibly display a message instead of the label.
kill [function(clear = FALSE)]	Kill the bar so it cannot be used anymore. Cursor is moved to new line. You can also erase its display.
error [function(e)]	Useful in tryCatch to properly display error messages below the bar. See the example.

## Examples

```

bar = makeProgressBar(max = 5, label = "test-bar")
for (i in 0:5) {
  bar$set(i)
  Sys.sleep(0.2)
}
bar = makeProgressBar(max = 5, label = "test-bar")
for (i in 1:5) {
  bar$inc(1)
  Sys.sleep(0.2)
}
# display errors properly (in next line)
## Not run:
f = function(i) if (i>2) stop("foo")
bar = makeProgressBar(max = 5, label = "test-bar")
for (i in 1:5) {
  tryCatch ({
    f(i)
    bar$set(i)
  }, error = bar$error)
}

## End(Not run)

```

---

makeS3Obj                      *Simple constructor for S3 objects based on lists.*

---

**Description**

Simple wrapper for `as.list` and `setClasses`.

**Usage**

```
makeS3Obj(classes, ...)
```

**Arguments**

classes	[character] Class(es) for constructed object.
...	[any] Key-value pairs for class members.

**Value**

Object.

**Examples**

```
makeS3Obj("car", speed = 100, color = "red")
```

---

makeSimpleFileLogger    *Simple logger which outputs to a file.*

---

**Description**

Creates a simple file logger closure to log to a file, including time stamps. An optional buffer holds the last few log messages.

**Usage**

```
makeSimpleFileLogger(logfile, touch = FALSE, keep = 10L)
```

**Arguments**

logfile	[character(1)] File to log to.
touch	[logical(1)] Should the file be created before the first log message? Default is FALSE.
keep	[integer(1)] Number of log messages to keep in memory for quick access. Default is 10.

**Value**

`SimpleFileLogger` . A list with following functions:

```
log [function(msg)]
    Send log message.
getMessages [function(n)]
    Get last n log messages.
clear [function()]
    Resets logger and deletes log file.
getSize [function()]
    Returns the number of logs written.
getLogfile [function()]
    Returns the full file name logs are written to.
```

---

mapValues	<i>Replace values in atomic vectors</i>
-----------	---

---

**Description**

Replace values in atomic vectors

**Usage**

```
mapValues(
  x,
  from,
  to,
  regex = FALSE,
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE
)
```

**Arguments**

x	[atomic] Atomic vector. If x is a factor, all replacements work on the levels.
from	[atomic] Atomic vector with values to replace, same length as to.
to	[atomic] Atomic vector with replacements, same length as from.
regex	[logical] Use regular expression matching? Default is FALSE.
ignore.case	[logical] Argument passed to <code>gsub</code> .

perl	[logical] Argument passed to <a href="#">gsub</a> .
fixed	[logical] Argument passed to <a href="#">gsub</a> .

**Details**

Replaces values specified in `from` with values in `to`. Regular expression matching can be enabled which calls [gsub](#) iteratively on `x` to replace all patterns in `from` with replacements in `to`.

**Value**

atomic .

**Examples**

```
# replace integers
x = 1:5
mapValues(x, c(2, 3), c(99, 100))

# replace factor levels using regex matching
x = factor(c("aab", "aba", "baa"))
mapValues(x, "a.a", "zzz", regex = TRUE)
```

---

messagef	<i>Wrapper for message and sprintf.</i>
----------	---

---

**Description**

A simple wrapper for `message(sprintf(...))`.

**Usage**

```
messagef(..., .newline = TRUE)
```

**Arguments**

...	[any] See <a href="#">sprintf</a> .
.newline	[logical(1)] Add a newline to the message. Default is TRUE.

**Value**

Nothing.

**Examples**

```
msg = "a message"
warningf("this is %s", msg)
```

---

namedList	<i>Create named list, possibly initialized with a certain element.</i>
-----------	--

---

**Description**

Even an empty list will always be named.

**Usage**

```
namedList(names, init)
```

**Arguments**

names	[character] Names of elements.
init	[valid R expression] If given all list elements are initialized to this, otherwise NULL is used.

**Value**

list .

**Examples**

```
namedList(c("a", "b"))
namedList(c("a", "b"), init = 1)
```

---

names2	<i>Replacement for names which always returns a vector.</i>
--------	---

---

**Description**

A simple wrapper for [names](#). Returns a vector even if no names attribute is set. Values NA and "" are treated as missing and replaced with the value provided in `missing.val`.

**Usage**

```
names2(x, missing.val = NA_character_)
```

**Arguments**

x	[ANY] Object, probably named.
missing.val	[ANY] Value to set for missing names. Default is NA_character_.



**Value**

character : vector of the same length as x.

**Examples**

```
x = 1:3
names(x)
names2(x)
names(x[1:2]) = letters[1:2]
names(x)
names2(x)
```

---

normalize

*Normalizes numeric data to a given scale.*

---

**Description**

Currently implemented for numeric vectors, numeric matrices and data.frame. For matrixes one can operate on rows or columns For data.frames, only the numeric columns are touched, all others are left unchanged. For constant vectors / rows / columns most methods fail, special behaviour for this case is implemented.

The method also handles NAs in in x and leaves them untouched.

**Usage**

```
normalize(
  x,
  method = "standardize",
  range = c(0, 1),
  margin = 1L,
  on.constant = "quiet"
)
```

**Arguments**

x	[numeric   matrix   data.frame] Input vector.
method	[character(1)] Normalizing method. Available are: “center”: Subtract mean. “scale”: Divide by standard deviation. “standardize”: Center and scale. “range”: Scale to a given range.

range	[numeric(2)] Range for method “range”. The first value represents the replacement for the min value, the second is the substitute for the max value. So it is possible to reverse the order by giving range = c(1, 0). Default is c(0, 1).
margin	[integer(1)] 1 = rows, 2 = cols. Same is in <a href="#">apply</a> Default is 1.
on.constant	[character(1)] How should constant vectors be treated? Only used, of “method != center”, since this methods does not fail for constant vectors. Possible actions are: “quiet”: Depending on the method, treat them quietly: “scale”: No division by standard deviation is done, input values. will be returned untouched. “standardize”: Only the mean is subtracted, no division is done. “range”: All values are mapped to the mean of the given range. “warn”: Same behaviour as “quiet”, but print a warning message. “stop”: Stop with an error.

**Value**

numeric | matrix | data.frame .

**See Also**

[scale](#)

---

optimizeSubInts

*Naive multi-start version of [optimize](#) for global optimization.*

---

**Description**

The univariate [optimize](#) can stop at arbitrarily bad points when  $f$  is not unimodal. This functions mitigates this effect in a very naive way: interval is subdivided into `nsub` equally sized subintervals, [optimize](#) is run on all of them (and on the original big interval) and the best obtained point is returned.

**Usage**

```
optimizeSubInts(
  f,
  interval,
  ...,
  lower = min(interval),
  upper = max(interval),
  maximum = FALSE,
  tol = .Machine$double.eps^0.25,
  nsub = 50L
)
```

**Arguments**

f	See <a href="#">optimize</a> .
interval	See <a href="#">optimize</a> .
...	See <a href="#">optimize</a> .
lower	See <a href="#">optimize</a> .
upper	See <a href="#">optimize</a> .
maximum	See <a href="#">optimize</a> .
tol	See <a href="#">optimize</a> .
nsub	[integer(1)] Number of subintervals. A value of 1 implies normal <a href="#">optimize</a> behavior. Default is 50L.

**Value**

See [optimize](#).

---

pause	<i>Pause in interactive mode and continue on &lt;Enter&gt;.</i>
-------	---

---

**Description**

Pause in interactive mode and continue on <Enter>.

**Usage**

```
pause()
```

---

printhead	<i>More meaningful head(df) output.</i>
-----------	---

---

**Description**

The behaviour is similar to `print(head(x, n))`. The difference is, that if the number of rows in a data.frame/matrix or the number of elements in a list or vector is larger than n, additional information is printed about the total number of rows or elements respectively.

**Usage**

```
printhead(x, n = 6L)
```

**Arguments**

x	[data.frame   matrix   list   vector] Object.
n	[integer(1)] Single positive integer: number of rows for a matrix/data.frame or number of elements for vectors/lists respectively.

**Value**

Nothing.

---

printStrToChar      *Print str(x) of an object to a string / character vector.*

---

**Description**

Print str(x) of an object to a string / character vector.

**Usage**

```
printStrToChar(x, collapse = "\n")
```

**Arguments**

x	[any] Object to print
collapse	[character(1)] Used to collapse multiple lines. NULL means no collapsing, vector is returned. Default is “\n”.

**Value**

character .

**Examples**

```
printStrToChar(iris)
```

---

printToChar	<i>Prints object to a string / character vector.</i>
-------------	--

---

**Description**

Prints object to a string / character vector.

**Usage**

```
printToChar(x, collapse = "\n")
```

**Arguments**

x	[any] Object to print
collapse	[character(1)] Used to collapse multiple lines. NULL means no collapsing, vector is returned. Default is “\n”.

**Value**

character .

**Examples**

```
x = data.frame(a = 1:2, b = 3:4)
str(printToChar(x))
```

---

rangeVal	<i>Calculate range statistic.</i>
----------	-----------------------------------

---

**Description**

A simple wrapper for `diff(range(x))`, so `max(x) - min(x)`.

**Usage**

```
rangeVal(x, na.rm = FALSE)
```

**Arguments**

x	[numeric] The vector.
na.rm	[logical(1)] If FALSE, NA is returned if an NA is encountered in x. If TRUE, NAs are disregarded. Default is FALSE

**Value**

numeric(1) .

---

requirePackages	<i>Require some packages.</i>
-----------------	-------------------------------

---

**Description**

Packages are loaded either via [requireNamespace](#) or [require](#).

If some packages could not be loaded and stop is TRUE the following exception is thrown: “For <why> please install the following packages: <missing packages>”. If why is NULL the message is: “Please install the following packages: <missing packages>”.

**Usage**

```
requirePackages(
  packs,
  min.versions = NULL,
  why = "",
  stop = TRUE,
  suppress.warnings = FALSE,
  default.method = "attach"
)
```

**Arguments**

packs	[character] Names of packages. If a package name is prefixed with “!”, it will be attached using <a href="#">require</a> . If a package name is prefixed with “_”, its namespace will be loaded using <a href="#">requireNamespace</a> . If there is no prefix, argument <code>default.method</code> determines how to deal with package loading.
min.versions	[character] A char vector specifying required minimal version numbers for a subset of packages in packs. Must be named and all names must be in packs. The only exception is when packs is only a single string, then you are allowed to pass an unnamed version string here. Default is NULL, meaning no special version requirements
why	[character(1)] Short string explaining why packages are required. Default is an empty string.
stop	[logical(1)] Should an exception be thrown for missing packages? Default is TRUE.
suppress.warnings	[logical(1)] Should warnings be suppressed while requiring? Default is FALSE.

default.method [character(1)]

If the packages are not explicitly prefixed with “!” or “\_”, this arguments determines the default. Possible values are “attach” and “load”. Note that the default is “attach”, but this might/will change in a future version, so please make sure to always explicitly set this.

### Value

logical . Named logical vector describing which packages could be loaded (with required version). Same length as packs.

### Examples

```
requirePackages(c("BBmisc", "base"), why = "BBmisc example")
```

---

rowLapply	<i>Apply function to rows of a data frame.</i>
-----------	--

---

### Description

Just like an [lapply](#) on data frames, but on the rows.

### Usage

```
rowLapply(df, fun, ..., unlist = FALSE)
```

```
rowSapply(df, fun, ..., unlist = FALSE, simplify = TRUE, use.names = TRUE)
```

### Arguments

df	[data.frame] Data frame.
fun	[function] Function to apply. Rows are passed as list or vector, depending on argument unlist, as first argument.
...	[ANY] Additional arguments for fun.
unlist	[logical(1)] Unlist the row? Note that automatic conversion may be triggered for lists of mixed data types Default is FALSE.
simplify	[logical(1)   character(1)] Should the result be simplified? See <a href="#">sapply</a> . If “cols”, we expect the call results to be vectors of the same length and they are arranged as the columns of the resulting matrix. If “rows”, likewise, but rows of the resulting matrix. Default is TRUE.
use.names	[logical(1)] Should result be named by the row names of df? Default is TRUE.

**Value**

list or simplified object . Length is nrow(df).

**Examples**

```
rowLapply(iris, function(x) x$Sepal.Length + x$Sepal.Width)
```

---

save2	<i>Save multiple objects to a file.</i>
-------	---

---

**Description**

A simple wrapper for [save](#). Understands key = value syntax to save objects using arbitrary variable names. All options of [save](#), except list and envir, are available and passed to [save](#).

**Usage**

```
save2(
  file,
  ...,
  ascii = FALSE,
  version = NULL,
  compress = !ascii,
  compression_level,
  eval.promises = TRUE,
  precheck = TRUE
)
```

**Arguments**

file	File to save.
...	[any] Will be converted to an environment and then passed to <a href="#">save</a> .
ascii	See help of <a href="#">save</a> .
version	See help of <a href="#">save</a> .
compress	See help of <a href="#">save</a> .
compression_level	See help of <a href="#">save</a> .
eval.promises	See help of <a href="#">save</a> .
precheck	See help of <a href="#">save</a> .

**Value**

See help of [save](#).



**Examples**

```
x = 1
save2(y = x, file = tempfile())
```

---

seq_row	<i>Generate sequences along rows or cols.</i>
---------	---

---

**Description**

A simple convenience wrapper around [seq\\_len](#).

**Usage**

```
seq_row(x)
seq_col(x)
```

**Arguments**

x	[data.frame   matrix] Data frame, matrix or any object which supports <a href="#">nrow</a> or <a href="#">ncol</a> , respectively.
---	---

**Value**

Vector of type [integer].

**Examples**

```
data(iris)
seq_row(iris)
seq_col(iris)
```

---

setAttribute	<i>A wrapper for attr(x, which) = y.</i>
--------------	--

---

**Description**

A wrapper for `attr(x, which) = y`.

**Usage**

```
setAttribute(x, which, value)
```

**Arguments**

x	[any] Your object.
which	[character(1)] Name of the attribute to set
value	[ANY] Value for the attribute.

**Value**

Changed object x.

**Examples**

```
setAttribute(list(), "foo", 1)
```

---

setClasses	<i>A wrapper for class(x) = classes.</i>
------------	--

---

**Description**

A wrapper for class(x) = classes.

**Usage**

```
setClasses(x, classes)
```

**Arguments**

x	[any] Your object.
classes	[character] New classes.

**Value**

Changed object x.

**Examples**

```
setClasses(list(), c("foo1", "foo2"))
```

---

setRowNames	<i>Wrapper for rownames(x) = y, colnames(x) = y.</i>
-------------	--

---

**Description**

Wrapper for rownames(x) = y, colnames(x) = y.

**Usage**

```
setRowNames(x, names)
```

```
setColNames(x, names)
```

**Arguments**

x	[matrix   data.frame] Matrix or data.frame.
names	[character] New names for rows / columns.

**Value**

Changed object x.

**Examples**

```
setColNames(matrix(1:4, 2, 2), c("a", "b"))
```

---

setValue	<i>Set a list element to a new value.</i>
----------	---

---

**Description**

This wrapper supports setting elements to NULL.

**Usage**

```
setValue(obj, index, newval)
```

**Arguments**

obj	[list]
index	[character   integer] Index or indices where to insert the new values.
newval	[any] Inserted element(s). Has to be a list if index is a vector.

**Value**

list

---

sortByCol	<i>Sort the rows of a data.frame according to one or more columns.</i>
-----------	--

---

**Description**

Sort the rows of a data.frame according to one or more columns.

**Usage**

```
sortByCol(x, col, asc = TRUE)
```

**Arguments**

x	[data.frame] Data.frame to sort.
col	[character] One or more column names to sort x by. In order of preference.
asc	[logical] Sort ascending (or descending)? One value per entry of col. If a scalar logical is passed, it is replicated. Default is TRUE.

**Value**

data.frame .

---

splitPath	<i>Split a path into components</i>
-----------	-------------------------------------

---

**Description**

The first normalized path is split on forward and backward slashes and its components returned as character vector. The drive or network home are extracted separately on windows systems and empty on all other systems.

**Usage**

```
splitPath(path)
```

**Arguments**

path	[character(1)] Path to split as string
------	---

**Value**

named list: List with components “drive” (character(1)) and “path” (character(n)).

---

splitTime	<i>Split seconds into handy chunks of time.</i>
-----------	---

---

**Description**

Note that a year is simply defined as exactly 365 days.

**Usage**

```
splitTime(seconds, unit = "years")
```

**Arguments**

seconds	[numeric(1)] Number of seconds. If not an integer, it is rounded down.
unit	[character(1)] Largest unit to split seconds into. Must be one of: c("years", "days", "hours", "minutes", "seconds"). Default is “years”.

**Value**

numeric(5) . A named vector containing the “years”, “days”, “hours”, “minutes” and “seconds”. Units larger than the given unit are NA.

**Examples**

```
splitTime(1000)
```

---

stopf	<i>Wrapper for stop and sprintf.</i>
-------	--------------------------------------

---

**Description**

A wrapper for `stop` with `sprintf` applied to the arguments. Notable difference is that error messages are not truncated to 1000 characters by default.

**Usage**

```
stopf(..., warning.length = 8170L)
```

**Arguments**

... [any]  
See [sprintf](#).

warning.length [integer(1)]  
Number of chars after which the error message gets truncated, see ?options.  
Default is 8170.

**Value**

Nothing.

**Examples**

```
err = "an error."
try(stopf("This is %s", err))
```

---

strrepeat	<i>Repeat and join a string</i>
-----------	---------------------------------

---

**Description**

Repeat and join a string

**Usage**

```
strrepeat(x, n, sep = "")
```

**Arguments**

x [character]  
Vector of characters.

n [integer(1)]  
Times the vector x is repeated.

sep [character(1)]  
Separator to use to collapse the vector of characters.

**Value**

character(1).

**Examples**

```
strrepeat("x", 3)
```

---

suppressAll	<i>Suppresses all output except for errors.</i>
-------------	---

---

**Description**

Evaluates an expression and suppresses all output except for errors, meaning: prints, messages, warnings and package startup messages.

**Usage**

```
suppressAll(expr)
```

**Arguments**

expr	[valid R expression] Expression.
------	-------------------------------------

**Value**

Return value of expression invisibly.

**Examples**

```
suppressAll({  
  print("foo")  
  message("foo")  
  warning("foo")  
})
```

---

symdiff	<i>Calculates symmetric set difference between two sets.</i>
---------	--

---

**Description**

Calculates symmetric set difference between two sets.

**Usage**

```
symdiff(x, y)
```

**Arguments**

x	[vector] Set 1.
y	[vector] Set 2.

**Value**

vector .

system3

*Wrapper for system2 with better return type and errorhandling.***Description**

Wrapper for [system2](#) with better return type and errorhandling.

**Usage**

```
system3(
  command,
  args = character(0L),
  stdout = "",
  stderr = "",
  wait = TRUE,
  ...,
  stop.on.exit.code = wait
)
```

**Arguments**

command	See <a href="#">system2</a> .
args	See <a href="#">system2</a> .
stdout	See <a href="#">system2</a> .
stderr	See <a href="#">system2</a> .
wait	See <a href="#">system2</a> .
...	Further arguments passed to <a href="#">system2</a> .
stop.on.exit.code	[logical(1)] Should an exception be thrown if an exit code greater 0 is generated? Can only be used if wait is TRUE. Default is wait.

**Value**

list .

exit.code [integer(1)]	Exit code of command. Given if wait is TRUE, otherwise NA. 0L means success. 127L means command was not found
output [character]	Output of command on streams. Only given is stdout or stderr was set to TRUE, otherwise NA.



---

toRangeStr	<i>Convert a numerical vector into a range string.</i>
------------	--

---

**Description**

Convert a numerical vector into a range string.

**Usage**

```
toRangeStr(x, range.sep = " - ", block.sep = ", ")
```

**Arguments**

x	[integer] Vector to convert into a range string.
range.sep	[character(1)] Separator between the first and last element of a range of consecutive elements in x. Default is "-".
block.sep	[character(1)] Separator between non consecutive elements of x or ranges. Default is ",".

**Value**

character(1)

**Examples**

```
x = sample(1:10, 7)
toRangeStr(x)
```

---

vapply	<i>Apply a function with a predefined return value</i>
--------	--

---

**Description**

These are just wrappers around [vapply](#) with argument FUN.VALUE set. The function is expected to return a single logical, integer, numeric or character value, depending on the second letter of the function name.

**Usage**

```
vapply(x, fun, ..., use.names = TRUE)
viapply(x, fun, ..., use.names = TRUE)
vnapply(x, fun, ..., use.names = TRUE)
vcapply(x, fun, ..., use.names = TRUE)
```

**Arguments**

x	[vector or list] Object to apply function on.
fun	[function] Function to apply on each element of x.
...	[ANY] Additional arguments for fun.
use.names	[logical(1)] Should result be named? Default is TRUE.

---

warningf	<i>Wrapper for warning and sprintf.</i>
----------	---

---

**Description**

A wrapper for [warning](#) with [sprintf](#) applied to the arguments.

**Usage**

```
warningf(..., immediate = TRUE, warning.length = 8170L)
```

**Arguments**

...	[any] See <a href="#">sprintf</a> .
immediate	[logical(1)] See <a href="#">warning</a> . Default is TRUE.
warning.length	[integer(1)] Number of chars after which the warning message gets truncated, see <a href="#">?options</a> . Default is 8170.

**Value**

Nothing.

**Examples**

```
msg = "a warning"  
warningf("this is %s", msg)
```

---

which.first	<i>Find the index of first/last TRUE value in a logical vector.</i>
-------------	---

---

**Description**

Find the index of first/last TRUE value in a logical vector.

**Usage**

```
which.first(x, use.names = TRUE)
```

```
which.last(x, use.names = TRUE)
```

**Arguments**

x	[logical] Logical vector.
use.names	[logical(1)] If TRUE and x is named, the result is also named.

**Value**

integer(1) | integer(0) . Returns the index of the first/last TRUE value in x or an empty integer vector if none is found.

**Examples**

```
which.first(c(FALSE, TRUE))
which.last(c(FALSE, FALSE))
```

---

%btwn%	<i>Check if some values are covered by the range of the values in a second vector.</i>
--------	--

---

**Description**

Check if some values are covered by the range of the values in a second vector.

**Usage**

```
x %btwn% y
```

**Arguments**

x	[numeric(n)] Value(s) that should be within the range of y.
y	[numeric] Numeric vector which defines the range.

**Value**

logical(n) . For each value in x: Is it in the range of y?

**Examples**

```
x = 3
y = c(-1,2,5)
x %btwn% y
```

---

%nin%

*Simply a negated in operator.*

---

**Description**

Simply a negated in operator.

**Usage**

```
x %nin% y
```

**Arguments**

x	[vector] Values that should not be in y.
y	[vector] Values to match against.

# Index

`%btwn%`, 67  
`%nin%`, 68

`addClasses`, 4  
`apply`, 50  
`argsAsNamedList`, 4  
`asMatrixCols`, 5  
`asMatrixRows` (`asMatrixCols`), 5  
`asQuoted`, 5

`binPack`, 6

`c`, 8  
`capitalizeStrings`, 7  
`cat`, 8  
`catf`, 7  
`cFactor`, 8  
`checkArg`, 9  
`checkListElementClass`, 10  
`chunk`, 11  
`clipString`, 12, 20  
`coalesce`, 12  
`collapse`, 13, 14  
`collapsef`, 14  
`computeMode`, 14  
`convertColsToList` (`convertRowsToList`), 19  
`convertDataFrameCols`, 15  
`convertDfCols` (deprecated), 21  
`convertInteger`, 16  
`convertIntegers`, 17  
`convertListOfRowsToDataFrame`, 17  
`convertMatrixType`, 18  
`convertRowsToList`, 19  
`convertToShortString`, 20

`dapply`, 20  
deprecated, 21  
`do.call`, 22  
`do.call2`, 22

`dropNamed`, 22

`ensureVector`, 23  
`explode`, 24  
`extractSubList`, 24

`filterNull`, 25

`getAttributeNames`, 26  
`getBestIndex` (`getMaxIndex`), 27  
`getClass1`, 26  
`getFirst`, 27  
`getLast` (`getFirst`), 27  
`getMaxIndex`, 27, 29  
`getMaxIndexOfCols` (`getMaxIndexOfRows`), 28  
`getMaxIndexOfRows`, 28  
`getMinIndex` (`getMaxIndex`), 27  
`getMinIndexOfCols` (`getMaxIndexOfRows`), 28  
`getMinIndexOfRows` (`getMaxIndexOfRows`), 28  
`getOperatingSystem`, 29  
`getRelativePath`, 30  
`getUnixTime`, 30  
`getUsedFactorLevels`, 31  
`getwd`, 30  
`gsub`, 46, 47

`hasAttributes`, 31

`inherits`, 9  
`insert`, 32  
`is`, 9  
`is.error`, 32  
`isDarwin` (`getOperatingSystem`), 29  
`isDirectory`, 33  
`isEmptyDirectory`, 34  
`isExpensiveExampleOk`, 34  
`isFALSE`, 35  
`isLinux` (`getOperatingSystem`), 29

- isProperlyNamed, 35
- isScalarCharacter (isScalarValue), 36
- isScalarComplex (isScalarValue), 36
- isScalarFactor (isScalarValue), 36
- isScalarInteger (isScalarValue), 36
- isScalarLogical (isScalarValue), 36
- isScalarNA, 36
- isScalarNumeric (isScalarValue), 36
- isScalarValue, 36
- isSubset, 37
- isSuperset, 38
- isUnix (getOperatingSystem), 29
- isValidName, 38
- isWindows (getOperatingSystem), 29
- itostr, 39
  
- lapply, 25, 55
- lib, 39
- listToShortString (deprecated), 21
- load2, 40, 42
- lsort, 41
  
- makeDataFrame, 41
- makeFileCache, 42
- makeProgressBar, 43
- makeS3Obj, 45
- makeSimpleFileLogger, 45
- mapValues, 46
- messagef, 47
- mode, 18
  
- namedList, 48
- names, 48
- names2, 48
- ncol, 57
- normalize, 49
- nrow, 57
  
- optimize, 50, 51
- optimizeSubInts, 50
  
- paste, 13
- pause, 51
- printHead, 51
- printStrToChar, 52
- printToChar, 53
- ProgressBar, 44
- ProgressBar (makeProgressBar), 43
  
- quote, 5
  
- rangeVal, 53
- require, 54
- requireNamespace, 54
- requirePackages, 54
- rowLapply, 55
- rowSapply (rowLapply), 55
  
- sapply, 25, 55
- save, 56
- save2, 56
- scale, 50
- seq\_col (seq\_row), 57
- seq\_len, 57
- seq\_row, 57
- setAttribute, 57
- setClasses, 45, 58
- setColNames (setRowNames), 59
- setRowNames, 59
- setValue, 59
- simpleError, 32
- SimpleFileLogger, 46
- SimpleFileLogger
  - (makeSimpleFileLogger), 45
- sort, 41
- sortByCol, 60
- split, 11
- splitPath, 60
- splitTime, 61
- sprintf, 8, 14, 20, 47, 61, 62, 66
- stop, 61
- stopf, 61
- strepeat, 62
- strtoi, 39
- suppressAll, 63
- symdiff, 63
- system2, 64
- system3, 64
  
- toRangeStr, 65
- try, 32
- tryCatch, 12, 32
  
- vapply, 25, 65
- vcapply (vlapply), 65
- vector, 42
- viapply (vlapply), 65
- vlapply, 65
- vnapply (vlapply), 65
  
- warning, 66

warningf, [66](#)

which.first, [67](#)

which.last (which.first), [67](#)