Package 'statnet.common'

June 2, 2019

Version 4.3.0

Date 2019-06-02

Title Common R Scripts and Utilities Used by the Statnet Project Software

Description Non-statistical utilities used by the software developed by the Statnet Project. They may also be of use to others.

Depends R (>= 3.5)

Imports utils, methods, coda, parallel, tools

BugReports https://github.com/statnet/statnet.common/issues

License GPL-3 + file LICENSE

URL https://statnet.org

Roxygen list(markdown = TRUE)

RoxygenNote 6.1.1

Encoding UTF-8

Suggests covr

R topics documented:

`all_identical` *Test if all items in a vector or a list are identical.*

Description

Test if all items in a vector or a list are identical.

Usage

```
all_identical(x)
```

Arguments

x	a vector or a list

Value

`TRUE` if all elements of `x` are identical to each other.

See Also

`identical`

Examples

```
stopifnot(!all_identical(1:3))

stopifnot(all_identical(list("a", "a", "a")))
```

as.rle — *Coerce to rle if not already an rle object.*

Description

Coerce to rle if not already an rle object.

Usage

```
as.rle(x)

## S3 method for class 'rle'
as.rle(x)

## Default S3 method:
as.rle(x)
```

Arguments

x	the object to be coerced.

check.control.class — *Check if the class of the control list is one of those that can be used by the calling function*

Description

This function can be called to check that the control list passed is appropriate for the function to be controlled. It does so by looking up the class of the control argument (defaulting to the control variable in the calling function) and checking if it matches a list of acceptable classes.

Usage

```
check.control.class(OKnames = as.character(ult(sys.calls(), 2)[[1L]]),
  myname = as.character(ult(sys.calls(), 2)[[1L]]),
  control = get("control", pos = parent.frame()))
```

Arguments

OKnames	List of control function names which are acceptable.
myname	Name of the calling function (used in the error message).
control	The control list. Defaults to the control variable in the calling function.

Note

In earlier versions, `OKnames` and `myname` were autodetected. This capability has been deprecated and results in a warning issued once per session. They now need to be set explicitly.

See Also

set.control.class, print.control.list

`compress.data.frame` *"Compress" a data frame.*

Description

`compress.data.frame` "compresses" a data frame, returning unique rows and a tally of the number of times each row is repeated, as well as a permutation vector that can reconstruct the original data frame. `decompress.data.frame` reconstructs the original data frame.

Usage

```
compress.data.frame(x)

decompress.data.frame(x)
```

Arguments

`x`	For `compress.data.frame` a `data.frame` to be compressed. For `decompress.data.frame` a `list` as returned by `compress.data.frame`.

Value

For `compress.data.frame`, a `list` with three elements:

`rows`	Unique rows of `x`
`frequencies`	A vector of the same length as the number or rows, giving the number of times the corresponding row is repeated
`ordering`	A vector such that if `c` is the compressed data frame, `c$rows[c$ordering,,drop=FALSE]` equals the original data frame, except for row names
`rownames`	Row names of `x`

For `decompress.data.frame`, the original data frame.

See Also

`data.frame`

Examples

```
(x <- data.frame(V1=sample.int(3,30,replace=TRUE),
                 V2=sample.int(2,30,replace=TRUE),
                 V3=sample.int(4,30,replace=TRUE)))

(c <- compress.data.frame(x))

stopifnot(all(decompress.data.frame(c)==x))
```

---

compress_rows *A generic function to compress a row-weighted table*

Description

Compress a matrix or a data frame with duplicated rows, updating row weights to reflect frequencies, or reverse the process, reconstructing a matrix like the one compressed (subject to permutation of rows and weights not adding up to an integer).

Usage

```
compress_rows(x, ...)

decompress_rows(x, target.nrows = NULL, ...)
```

Arguments

x	a weighted matrix or data frame.
...	extra arguments for methods.
target.nrows	the approximate number of rows the uncompressed matrix should have; if not achievable exactly while respecting proportionality, a matrix with a slightly different number of rows will be constructed.

Value

For compress_rows A weighted matrix or data frame of the same type with duplicated rows removed and weights updated appropriately.

control.list.accessor *Named element accessor for ergm control lists*

Description

Utility method that overrides the standard '$' list accessor to disable partial matching for ergm `control.list` objects

Usage

```
## S3 method for class 'control.list'
object$name
```

Arguments

`object`	list-coearceable object with elements to be searched
`name`	literal character name of list element to search for and return

Details

Executes `getElement` instead of `$` so that element names must match exactly to be returned and partially matching names will not return the wrong object.

Value

Returns the named list element exactly matching `name`, or `NULL` if no matching elements found

Author(s)

Pavel N. Krivitsky

See Also

see `getElement`

control.remap *Overwrite control parameters of one configuration with another.*

Description

Given a `control.list`, and two prefixes, `from` and `to`, overwrite the elements starting with `to` with the corresponding elements starting with `from`.

Usage

```
control.remap(control, from, to)
```

Arguments

control	An object of class control.list.
from	Prefix of the source of control parameters.
to	Prefix of the destination of control parameters.

Value

An control.list object.

Author(s)

Pavel N. Krivitsky

See Also

print.control.list

Examples

```
(l <- set.control.class("test", list(a.x=1, a.y=2)))
control.remap(l, "a", "b")
```

---

deprecation-utilities *Utilities to help with deprecating functions.*

Description

.Deprecate_once calls .Deprecated(), passing all its arguments through, but only the first time it's called.

.Deprecate_method calls .Deprecated(), but only if a method has been called by name, i.e., *METHOD*.*CLASS*. Like .Deprecate_once it only issues a warning the first time.

Usage

```
.Deprecate_once(...)

.Deprecate_method(generic, class)
```

Arguments

...	arguments passed to .Deprecated().
generic, class	strings giving the generic function name and class name of the function to be deprecated.

Examples

```
options(warn=1) # Print warning immediately after the call.
f <- function(){
   .Deprecate_once("new_f")
}
f() # Deprecation warning
f() # No deprecation warning


options(warn=1) # Print warning immediately after the call.
summary.packageDescription <- function(object, ...){
   .Deprecate_method("summary", "packageDescription")
   invisible(object)
}

summary(packageDescription("statnet.common")) # No warning.
summary.packageDescription(packageDescription("statnet.common")) # Warning.
summary.packageDescription(packageDescription("statnet.common")) # No warning.
```

---

despace *A one-line function to strip whitespace from its argument.*

Description

A one-line function to strip whitespace from its argument.

Usage

```
despace(s)
```

Arguments

s	a character vector.

Examples

```
stopifnot(despace("\n \t  ")=="")
```

ERRVL *Return the first argument passed (out of any number) that is not a* `try-error` *(result of* `try` *encountering an error.*

Description

This function is inspired by `NVL`, and simply returns the first argument that is not a `try-error`, raising an error if all arguments are `try-errors`.

Usage

```
ERRVL(...)
```

Arguments

`...`	Expressions to be tested; usually outputs of `try`.

Value

The first argument that is not a `try-error`. Stops with an error if all are.

Note

This function uses lazy evaluation, so, for example `ERRVL(1,stop("Error!"))` will never evaluate the `stop` call and will not produce an error, whereas `ERRVL(try(solve(0)),stop("Error!"))` would.

In addition, all expressions after the first may contain a `.`, which is substituted with the `try-error` object returned by the previous expression.

See Also

`try`, `inherits`

Examples

```
print(ERRVL(1,2,3)) # 1
print(ERRVL(try(solve(0)),2,3)) # 2
print(ERRVL(1, stop("Error!"))) # No error

## Not run:
# Error:
print(ERRVL(try(solve(0), silent=TRUE),
            stop("Error!")))

# Error with an elaborate message:
print(ERRVL(try(solve(0), silent=TRUE),
            stop("Stopped with an error: ", .)))
```

```
## End(Not run)
```

`forkTimeout` *Evaluate an* R *expression with a hard time limit by forking a process*

Description

This function uses `parallel::mcparallel()`, so the time limit is not enforced on Windows. However, unlike functions using `setTimeLimit()`, the time limit is enforced even on native code.

Usage

```
forkTimeout(expr, timeout, unsupported = c("warning", "error", "message",
  "silent"), onTimeout = NULL)
```

Arguments

`expr`	expression to be evaluated.
`timeout`	number of seconds to wait for the expression to evaluate.
`unsupported`	a character vector of length 1 specifying how to handle a platform that does not support `parallel::mcparallel()`,
	`"warning"` **or** `"message"` Issue a warning or a message, respectively, then evaluate the expression without the time limit enforced.
	`"error"` Stop with an error.
	`"silent"` Evaluate the expression without the time limit enforced, without any notice.
	Partial matching is used.
`onTimeout`	Value to be returned on time-out.

Value

Result of evaluating `expr` if completed, `onTimeout` otherwise.

Note

`onTimeout` can itself be an expression, so it is, for example, possible to stop with an error by passing `onTimeout=stop()`.

Note that this function is not completely transparent: side-effects may behave in unexpected ways. In particular, RNG state will not be updated.

Examples

```
forkTimeout({Sys.sleep(1); TRUE}, 2) # TRUE
forkTimeout({Sys.sleep(1); TRUE}, 0.5) # NULL (except on Windows)
```

`formula.utilities` *Functions for Querying, Validating and Extracting from Formulas A suite of utilities for handling model formulas of the style used in Statnet packages.*

Description

Functions for Querying, Validating and Extracting from Formulas

A suite of utilities for handling model formulas of the style used in Statnet packages.

Usage

```
append_rhs.formula(object, newterms, keep.onesided = FALSE)

append.rhs.formula(object, newterms, keep.onesided = FALSE)

filter_rhs.formula(object, f, ...)

nonsimp_update.formula(object, new, ..., from.new = FALSE)

nonsimp.update.formula(object, new, ..., from.new = FALSE)

term.list.formula(rhs, sign = +1)

list_summands.call(object)

list_rhs.formula(object)

eval_lhs.formula(object)
```

Arguments

`object`	formula object to be updated or evaluated
`newterms`	list of terms (names) to append to the formula, or a formula whose RHS terms will be used; either may have a "sign" attribute vector of the same length as the list, giving the sign of each term (`+1` or `-1`).
`keep.onesided`	if the initial formula is one-sided, keep it whether to keep it one-sided or whether to make the initial formula the new LHS
`f`	a function whose first argument is the term and whose additional arguments are forwarded from `...` that returns either `TRUE` or `FALSE`, for whether that term should be kept.
`...`	Additional arguments. Currently unused.
`new`	new formula to be used in updating
`from.new`	logical or character vector of variable names. controls how environment of formula gets updated.
`rhs, sign`	Arguments to the deprecated `term.list.formula`.

Value

append_rhs.formula each return an updated formula object

nonsimp_update.formula each return an updated formula object

list_summands.call returns a list of unevaluated calls, with an additional numerical vector attribute "sign" with of the same length, giving the corresponding term's sign as +1 or -1.

list_rhs.formula returns a list of formula terms, with an additional numerical vector attribute "sign" with of the same length, giving the corresponding term's sign as +1 or -1.

eval_lhs.formula an object of whatever type the LHS evaluates to.

Functions

- append_rhs.formula: append_rhs.formula appends a list of terms to the RHS of a formula. If the formula is one-sided, the RHS becomes the LHS, if keep.onesided==FALSE (the default).
- append.rhs.formula: append.rhs.formula has been renamed to append_rhs.formula.
- filter_rhs.formula: filter_rhs.formula filters through the terms in the RHS of a formula, returning a formula without the terms for which function f(term,...) is FALSE. Terms inside another term (e.g., parentheses or an operator other than + or -) will be unaffected.
- nonsimp_update.formula: nonsimp_update.formula is a reimplementation of update.formula that does not simplify. Note that the resulting formula's environment is set as follows. If from.new==FALSE, it is set to that of object. Otherwise, a new sub-environment of object, containing, in addition, variables in new listed in from.new (if a character vector) or all of new (if TRUE).
- nonsimp.update.formula: nonsimp.update.formula has been renamed to nonsimp_update.formula.
- term.list.formula: term.list.formula is an older version of list_rhs.formula that required the RHS call, rather than the formula itself.
- list_summands.call: list_summands.call, given an unevaluated call or expression containing the sum of one or more terms, returns a list of the terms being summed, handling + and - operators and parentheses, and keeping track of whether a term has a plus or a minus sign.
- list_rhs.formula: list_rhs.formula returns a list containing terms in a given formula, handling + and - operators and parentheses, and keeping track of whether a term has a plus or a minus sign.
- eval_lhs.formula: eval_lhs.formula extracts the LHS of a formula, evaluates it in the formula's environment, and returns the result.

Examples

```
## append_rhs.formula

(f1 <- append_rhs.formula(y~x,list(as.name("z1"),as.name("z2"))))
(f2 <- append_rhs.formula(~y,list(as.name("z"))))
(f3 <- append_rhs.formula(~y+x,structure(list(as.name("z")),sign=-1)))
(f4 <- append_rhs.formula(~y,list(as.name("z")),TRUE))
(f5 <- append_rhs.formula(y~x,~z1-z2))
```

```
## filter_rhs.formula
(f1 <- filter_rhs.formula(~a-b+c, `!=`, "a"))
(f2 <- filter_rhs.formula(~-a+b-c, `!=`, "a"))
(f3 <- filter_rhs.formula(~a-b+c, `!=`, "b"))
(f4 <- filter_rhs.formula(~-a+b-c, `!=`, "b"))
(f5 <- filter_rhs.formula(~a-b+c, `!=`, "c"))
(f6 <- filter_rhs.formula(~-a+b-c, `!=`, "c"))
(f7 <- filter_rhs.formula(~c-a+b-c(a),
                          function(x) (if(is.call(x)) x[[1]] else x)!="c"))



## eval_lhs.formula

(result <- eval_lhs.formula((2+2)~1))

stopifnot(identical(result,4))
```

`logspace.utils` *Utilities for performing calculations on logarithmic scale.*

Description

A small suite of functions to compute sums, means, and weighted means on logarithmic scale, minimizing loss of precision.

Usage

```
log_sum_exp(logx, use_ldouble = FALSE)

log_mean_exp(logx, use_ldouble = FALSE)

lweighted.mean(x, logw)

lweighted.var(x, logw)
```

Arguments

`logx`	Numeric vector of $\log(x)$, the natural logarithms of the values to be summed or averaged.
`use_ldouble`	Whether to use `long double` precision in the calculation. If `TRUE`, 's C built-in `logspace_sum()` is used. If `FALSE`, the package's own implementation based on it is used, using `double` precision, which is (on most systems) several times faster, at the cost of precision.

x	Numeric vector of x, the (raw) values to be summed or averaged. For `lweighted.mean`, `x` may also be a matrix, in which case the weighted mean will be computed for each column of `x`.
logw	Numeric vector of $\log(w)$, the natural logarithms of the weights.

Value

The functions return the equivalents of the following R expressions, but faster and with less loss of precision:

`log_sum_exp(logx)` `log(sum(exp(logx)))`

`log_mean_exp(logx)` `log(mean(exp(logx)))`

`lweighted.mean(x,logw)` `sum(x*exp(logw))/sum(exp(logw))` for `x` scalar and `colSums(x*exp(logw))/sum(exp(logw` for `x` matrix

`lweighted.var(x,logw)` `crossprod(x*exp(logw/2))/sum(exp(logw))`

Author(s)

Pavel N. Krivitsky

Examples

```
logx <- rnorm(1000)
stopifnot(all.equal(log(sum(exp(logx))), log_sum_exp(logx)))
stopifnot(all.equal(log(mean(exp(logx))), log_mean_exp(logx)))

x <- rnorm(1000)
logw <- rnorm(1000)
stopifnot(all.equal(m <- sum(x*exp(logw))/sum(exp(logw)),lweighted.mean(x, logw)))
stopifnot(all.equal(sum((x-m)^2*exp(logw))/sum(exp(logw)),
                    lweighted.var(x, logw), check.attributes=FALSE))

x <- cbind(x, rnorm(1000))
stopifnot(all.equal(m <- colSums(x*exp(logw))/sum(exp(logw)),
                    lweighted.mean(x, logw), check.attributes=FALSE))
stopifnot(all.equal(crossprod(t(t(x)-m)*exp(logw/2))/sum(exp(logw)),
                    lweighted.var(x, logw), check.attributes=FALSE))
```

`mcmc-utilities` *Utility operations for* `mcmc.list` *objects*

Description

`colMeans.mcmc.list` is a "method" for (non-generic) `colMeans` applicable to `mcmc.list` objects.

`sweep.mcmc.list` is a "method" for (non-generic) `sweep` applicable to `mcmc.list` objects.

`lapply.mcmc.list` is a "method" for (non-generic) `lapply` applicable to `mcmc.list` objects.

Usage

```
colMeans.mcmc.list(x, ...)

sweep.mcmc.list(x, STATS, FUN = "-", check.margin = TRUE, ...)

lapply.mcmc.list(X, FUN, ...)
```

Arguments

x	a mcmc.list object.
...	additional arguments to colMeans or sweep.
STATS, FUN, check.margin	See help for sweep.
X	An mcmc.list object.

Value

colMeans.mcmc returns a vector with length equal to the number of mcmc chains in x with the mean value for each chain.

sweep.mcmc.list returns an appropriately modified version of x

lapply.mcmc.list returns an mcmc.list each of whose chains had been passed through FUN.

See Also

colMeans, mcmc.list

sweep

lapply

Examples

```
data(line, package="coda")
summary(line) # coda
colMeans.mcmc.list(line) # "Method"

data(line, package="coda")
colMeans.mcmc.list(line)-1:3
colMeans.mcmc.list(sweep.mcmc.list(line, 1:3))

data(line, package="coda")
colMeans.mcmc.list(line)[c(2,3,1)]
colMeans.mcmc.list(lapply.mcmc.list(line, `[`,,c(2,3,1)))
```

message_print *print objects to the message output.*

Description

A thin wrapper around print that captures its output and prints it as a message, usually to STDERR.

Usage

```
message_print(..., messageArgs = NULL)
```

Arguments

...	arguments to print.
messageArgs	a list of arguments to be passed directly to message.

Examples

```
cat(1:5)

print(1:5)
message_print(1:5) # Looks the same (though may be in a different color on some frontends).

suppressMessages(print(1:5)) # Still prints
suppressMessages(message_print(1:5)) # Silenced
```

NVL *Convenience functions for handling NULL objects.*

Description

Convenience functions for handling NULL objects.

Usage

```
NVL(...)

NVL2(test, notnull, null = NULL)

NVL3(test, notnull, null = NULL)

EVL(...)

EVL2(test, notnull, null = NULL)

EVL3(test, notnull, null = NULL)
```

```
NVL(x) <- value

EVL(x) <- value
```

Arguments

`..., test`	expressions to be tested.
`notnull`	expression to be returned if `test` is not `NULL`.
`null`	expression to be returned if `test` is `NULL`.
`x`	an object to be overwritten if `NULL`.
`value`	new value for `x`.

Functions

- `NVL`: Inspired by SQL function `NVL`, returns the first argument that is not `NULL`, or `NULL` if all arguments are `NULL`.
- `NVL2`: Inspired by Oracle SQL function `NVL2`, returns the second argument if the first argument is not `NULL` and the third argument if the first argument is `NULL`. The third argument defaults to `NULL`, so `NVL2(a,b)` can serve as shorthand for `(if(!is.null(a)) b)`.
- `NVL3`: Inspired by Oracle SQL `NVL2` function and `magittr` `%>%` operator, behaves as `NVL2` but `.s` in the second argument are substituted with the first argument.
- `EVL`: As `NVL`, but for any objects of length 0 (*E*mpty) rather than just `NULL`. Note that if no non-zero-length arguments are given, `NULL` is returned.
- `EVL2`: As `NVL2`, but for any objects of length 0 (*E*mpty) rather than just `NULL`.
- `EVL3`: As `NVL3`, but for any objects of length 0 (*E*mpty) rather than just `NULL`.
- `NVL<-`: Assigning to `NVL` overwrites its first argument if that argument is `NULL`. Note that it will *always* return the right-hand-side of the assignment (`value`), regardless of what `x` is.
- `EVL<-`: As assignment to `NVL`, but for any objects of length 0 (*E*mpty) rather than just `NULL`.

Note

Whenever possible, these functions use lazy evaluation, so, for example `NVL(1,stop("Error!"))` will never evaluate the `stop` call and will not produce an error, whereas `NVL(NULL,stop("Error!"))` would.

See Also

`NULL`, `is.null`, `if`

Examples

```
a <- NULL

a # NULL
NVL(a,0) # 0
```

```
b <- 1

b # 1
NVL(b,0) # 1

# Here, object x does not exist, but since b is not NULL, x is
# never evaluated, so the statement finishes.
NVL(b,x) # 1

# Also,
NVL(NULL,1,0) # 1
NVL(NULL,0,1) # 0
NVL(NULL,NULL,0) # 0
NVL(NULL,NULL,NULL) # NULL

NVL2(a, "not null!", "null!") # "null!"
NVL2(b, "not null!", "null!") # "not null!"

NVL3(a, "not null!", "null!") # "null!"
NVL3(b, .+1, "null!") # 2

NVL(NULL*2, 1) # numeric(0) is not NULL
EVL(NULL*2, 1) # 1


NVL(a) <- 2
a # 2
NVL(b) <- 2
b # still 1
```

`once` *Evaluate a function once for a given input.*

Description

This is a `purrr`-style adverb that checks if a given function has already been called with a given configuration of arguments and skips it if it has.

Usage

```
once(f, expire_after = Inf, max_entries = Inf)
```

Arguments

`f`	A function to modify.
`expire_after`	The number of seconds since it was added to the database before a particular configuration is "forgotten". This can be used to periodically remind the user without overwhelming them.

`max_entries` The number of distinct configurations to remember. If not `Inf`, *earliest-inserted* configurations will be removed from the database when capacity is exceeded. (This exact behavior may change in the future.)

Details

Each modified function instance returned by `once()` maintains a database of previous argument configurations. They are not in any way compressed, so this database may grow over time. Thus, this wrapper should be used with caution if arguments are large objects. This may be replaced with hashing in the future. In the meantime, you may want to set the `max_entries` argument to be safe.

Different instances of a modified function do not share databases, even if the function is the same. This means that if you, say, modify a function within another function, the modified function will call once per call to the outer function. Modified functions defined at package level count as the same "instance", however. See example.

Note

Because the function needs to test whether a particular configuration of arguments have already been used, do not rely on lazy evaluation behaviour.

Examples

```
msg <- once(message)
msg("abc") # Prints.
msg("abc") # Silent.

msg <- once(message) # Starts over.
msg("abc") # Prints.

f <- function(){
  innermsg  <- once(message)
  innermsg("efg") # Prints once per call to f().
  innermsg("efg") # Silent.
  msg("abcd") # Prints only the first time f() is called.
  msg("abcd") # Silent.
}
f() # Prints "efg" and "abcd".
f() # Prints only "efg".

msg3 <- once(message, max_entries=3)
msg3("a") # 1 remembered.
msg3("a") # Silent.
msg3("b") # 2 remembered.
msg3("a") # Silent.
msg3("c") # 3 remembered.
msg3("a") # Silent.
msg3("d") # "a" forgotten.
msg3("a") # Printed.

msg2s <- once(message, expire_after=2)
msg2s("abc") # Prints.
```

```
msg2s("abc") # Silent.
Sys.sleep(1)
msg2s("abc") # Silent after 1 sec.
Sys.sleep(1.1)
msg2s("abc") # Prints after 2.1 sec.
```

`opttest` *Optionally test code depending on environment variable.*

Description

A convenience wrapper to run code based on whether an environment variable is defined.

Usage

```
opttest(expr, testname = NULL, testvar = "ENABLE_statnet_TESTS",
  yesvals = c("y", "yes", "t", "true", "1"), lowercase = TRUE)
```

Arguments

`expr`	An expression to be evaluated only if `testvar` is set to a non-empty value.
`testname`	Optional name of the test. If given, and the test is skipped, will print a message to that end, including the name of the test, and instructions on how to enable it.
`testvar`	Environment variable name. If set to one of the `yesvals`, `expr` is run. Otherwise, an optional message is printed.
`yesvals`	A character vector of strings considered affirmative values for `testvar`.
`lowercase`	Whether to convert the value of `testvar` to lower case before comparing it to `yesvals`.

`order` *Implement the* `sort` *and* `order` *methods for* `data.frame` *and* `matrix`*, sorting it in lexicographic order.*

Description

These function return a data frame sorted in lexcographic order or a permutation that will rearrange it into lexicographic order: first by the first column, ties broken by the second, remaining ties by the third, etc..

Usage

```
order(..., na.last = TRUE, decreasing = FALSE)

## Default S3 method:
order(..., na.last = TRUE, decreasing = FALSE)

## S3 method for class 'data.frame'
order(..., na.last = TRUE, decreasing = FALSE)

## S3 method for class 'matrix'
order(..., na.last = TRUE, decreasing = FALSE)

## S3 method for class 'data.frame'
sort(x, decreasing = FALSE, ...)
```

Arguments

`...`	Ignored for `sort`. For `order`, first argument is the data frame to be ordered. (This is needed for compatibility with `order`.)
`na.last`	See `order` documentation.
`decreasing`	Whether to sort in decreasing order.
`x`	A `data.frame` to sort.

Value

For `sort`, a data frame, sorted lexicographically. For `order`, a permutation `I` (of a vector `1:nrow(x)`) such that `x[I,,drop=FALSE]` equals `x` ordered lexicographically.

See Also

`data.frame`, `sort`, `order`, `matrix`

Examples

```
data(iris)

head(iris)

head(order(iris))

head(sort(iris))

stopifnot(identical(sort(iris),iris[order(iris),]))
```

paste.and — *Concatenates the elements of a vector (optionaly enclosing them in quotation marks or parentheses) adding appropriate punctuation and conjunctions.*

Description

A vector x becomes "x[1]", "x[1] and x[2]", or "x[1], x[2], and x[3]", depending on the langth of x.

Usage

```
paste.and(x, oq = "", cq = "", con = "and")
```

Arguments

x	A vector.
oq	Opening quotation symbol. (Defaults to none.)
cq	Closing quotation symbol. (Defaults to none.)
con	Conjunction to be used if length(x)>1. (Defaults to "and".)

Value

A string with the output.

See Also

paste, cat

Examples

```
print(paste.and(c()))

print(paste.and(1))

print(paste.and(1:2))

print(paste.and(1:3))

print(paste.and(1:4,con='or'))
```

persistEval *Evaluate an expression, restarting on error*

Description

A pair of functions paralleling `eval()` and `evalq()` that make multiple attempts at evaluating an expression, retrying on error up to a specified number of attempts, and optionally evaluating another expression before restarting.

Usage

```
persistEval(expr, retries = NVL(getOption("eval.retries"), 5),
  beforeRetry, envir = parent.frame(), enclos = if (is.list(envir) ||
  is.pairlist(envir)) parent.frame() else baseenv(), verbose = FALSE)

persistEvalQ(expr, retries = NVL(getOption("eval.retries"), 5),
  beforeRetry, envir = parent.frame(), enclos = if (is.list(envir) ||
  is.pairlist(envir)) parent.frame() else baseenv(), verbose = FALSE)
```

Arguments

`expr`	an expression to be retried; note the difference between `eval()` and `evalq()`.
`retries`	number of retries to make; defaults to `"eval.retries"` option, or 5.
`beforeRetry`	if given, an expression that will be evaluated before each retry if the initial attempt fails; it is evaluated in the same environment and with the same quoting semantics as `expr`, but its errors are not handled.
`envir, enclos`	see `eval()`.
`verbose`	Whether to output retries.

Value

Results of evaluating `expr`, including side-effects such as variable assignments, if successful in `retries` retries.

Note

If `expr` returns a `"try-error"` object (returned by `try()`), it will be treated as an error. This behavior may change in the future.

Examples

```
x <- 0
persistEvalQ({if((x<-x+1)<3) stop("x < 3") else x},
             beforeRetry = {cat("Will try incrementing...\n")})

x <- 0
e <- quote(if((x<-x+1)<3) stop("x < 3") else x)
```

```
persistEval(e,
            beforeRetry = quote(cat("Will try incrementing...\n")))
```

`print.control.list` *Pretty print the control list*

Description

This function prints the control list, including what it can control and the elements.

Usage

```
## S3 method for class 'control.list'
print(x, ...)
```

Arguments

`x`	A list generated by a `control.*` function.
`...`	Unused at this time.

See Also

`check.control.class`, `set.control.class`

`rle.utils` *RLE utilities*

Description

Simple utilities for operations on RLE-encoded vectors.

Usage

```
## S3 method for class 'rle'
c(...)

## S3 method for class 'rle'
!x

binop.rle(e1, e2, FUN)

## S3 method for class 'rle'
e1 | e2

## S3 method for class 'rle'
e1 & e2
```

```
compact.rle(x)

## S3 method for class 'rle'
any(..., na.rm = FALSE)

## S3 method for class 'rle'
all(..., na.rm = FALSE)

## S3 method for class 'rle'
e1 * e2

## S3 method for class 'rle'
e1 / e2

## S3 method for class 'rle'
e1 - e2

## S3 method for class 'rle'
e1 + e2

## S3 method for class 'rle'
e1 ^ e2

## S3 method for class 'rle'
e1 %% e2

## S3 method for class 'rle'
e1 %/% e2

## S3 method for class 'rle'
e1 == e2

## S3 method for class 'rle'
e1 > e2

## S3 method for class 'rle'
e1 < e2

## S3 method for class 'rle'
e1 != e2

## S3 method for class 'rle'
e1 <= e2

## S3 method for class 'rle'
e1 >= e2
```

```
## S3 method for class 'rle'
sum(..., na.rm = FALSE)

## S3 method for class 'rle'
mean(x, na.rm = FALSE, ...)

## S3 method for class 'rle'
length(x)

## S3 method for class 'rle'
is.na(x)

## S3 method for class 'rle'
rep(x, ..., scale = c("element", "run"),
  doNotCompact = FALSE)
```

Arguments

`...`	For `c`, objects to be concatenated. The first object must be of class `rle`. For `rep`, see documentation for `rep`. For `sum`, objects to be summed.
`x, e1, e2`	Arguments to unary (`x`) and binary (`e1` and `e2`) operators.
`FUN`	A binary function or operator or a name of one. It is assumed to be vectorized: it expects two vectors of equal lengths and outputs a vector of the same length.
`na.rm`	see documentation for `any`, `all`, and `sum`.
`scale`	whether to replicate the elements of the RLE-compressed vector or the runs.
`doNotCompact`	whether the method should call `compact.rle` the results before returning. Methods liable to produce very long output vectors, like `rep`, have this set `FALSE` by default.

Value

Unless otherwise stated, all functions return an `rle` object. By default, the functions and the operators do not merge adjacent runs with the same value. This must be done explicitly with `compact.rle`.

`any`, `all`, `sum`, and `length` return logical, logical, numeric, and numeric vectors, respectively.

Functions

- `binop.rle`: Perform an arbitrary binary operation on the pair of vectors represented by the `rle` objects.
- `compact.rle`: Compact the `rle` object by merging adjacent runs.

Note

Since `rle` stores run lengths as integers, `compact.rle` will not merge runs that add up to lengths greater than what can be represented by a 32-bit signed integer (2147483647).

The length method returns the length of the vector represented by the object, obtained by summing the lengths of individual runs.

The rep method for rle objects is very limited at this time. Even though the default setting is to replicate elements of the vector, only the run-replicating functionality is implemented at this time except for the simplest case (scalar times argument).

Examples

```
x <- rle(as.logical(rbinom(10,1,.7)))
y <- rle(as.logical(rbinom(10,1,.3)))

stopifnot(isTRUE(all.equal(c(inverse.rle(x),inverse.rle(y)),inverse.rle(c(x,y)))))

stopifnot(isTRUE(all.equal((!inverse.rle(x)),inverse.rle(!x))))
stopifnot(isTRUE(all.equal((inverse.rle(x)|inverse.rle(y)),inverse.rle(x|y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)&inverse.rle(y)),inverse.rle(x&y))))
stopifnot(identical(rle(inverse.rle(x)&inverse.rle(y)),compact.rle(x&y)))

big <- structure(list(lengths=as.integer(rep(.Machine$integer.max/4,6)),
                      values=rep(TRUE,6)), class="rle")

stopifnot(all(aggregate(as.numeric(lengths)~values,
                        data=as.data.frame(unclass(big)),FUN=sum)
              ==
              aggregate(as.numeric(lengths)~values,
                        data=as.data.frame(unclass(compact.rle(big))),
                        FUN=sum)))

x <- rle(as.logical(rbinom(10,1,.9)))
y <- rle(as.logical(rbinom(10,1,.1)))

stopifnot(isTRUE(all.equal(any(x),any(inverse.rle(x)))))
stopifnot(isTRUE(all.equal(any(y),any(inverse.rle(y)))))


stopifnot(isTRUE(all.equal(all(x),all(inverse.rle(x)))))
stopifnot(isTRUE(all.equal(all(y),all(inverse.rle(y)))))


x <- rle(sample(c(-1,+1), 10, c(.7,.3), replace=TRUE))
y <- rle(sample(c(-1,+1), 10, c(.3,.7), replace=TRUE))

stopifnot(isTRUE(all.equal((inverse.rle(x)*inverse.rle(y)),inverse.rle(x*y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)/inverse.rle(y)),inverse.rle(x/y))))
stopifnot(isTRUE(all.equal((-inverse.rle(y)),inverse.rle(-y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)-inverse.rle(y)),inverse.rle(x-y))))
stopifnot(isTRUE(all.equal((+inverse.rle(y)),inverse.rle(+y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)+inverse.rle(y)),inverse.rle(x+y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)^inverse.rle(y)),inverse.rle(x^y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)%%inverse.rle(y)),inverse.rle(x%%y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)%/%inverse.rle(y)),inverse.rle(x%/%y))))
```

```
stopifnot(isTRUE(all.equal(inverse.rle(x)==inverse.rle(y),inverse.rle(x==y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)>inverse.rle(y)),inverse.rle(x>y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)<inverse.rle(y)),inverse.rle(x<y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)!=inverse.rle(y)),inverse.rle(x!=y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)<=inverse.rle(y)),inverse.rle(x<=y))))
stopifnot(isTRUE(all.equal((inverse.rle(x)>=inverse.rle(y)),inverse.rle(x>=y))))

stopifnot(isTRUE(all.equal(sum(inverse.rle(x)),sum(x))))
stopifnot(isTRUE(all.equal(sum(inverse.rle(y)),sum(y))))


stopifnot(isTRUE(all.equal(mean(inverse.rle(x)),mean(x))))
stopifnot(isTRUE(all.equal(mean(inverse.rle(y)),mean(y))))


stopifnot(isTRUE(all.equal(length(inverse.rle(x)),length(x))))
stopifnot(isTRUE(all.equal(length(inverse.rle(y)),length(y))))

x$values[1] <- NA
y$values[1] <- NA
stopifnot(isTRUE(all.equal(is.na(inverse.rle(x)),inverse.rle(is.na(x)))))
stopifnot(isTRUE(all.equal(is.na(inverse.rle(y)),inverse.rle(is.na(y)))))


x <- rle(sample(c(-1,+1), 10, c(.7,.3), replace=TRUE))
y <- rpois(length(x$lengths), 2)

stopifnot(isTRUE(all.equal(rep(inverse.rle(x), rep(y, x$lengths)),
                           inverse.rle(rep(x, y, scale="run")))))

stopifnot(isTRUE(all.equal(rep(inverse.rle(x), max(y)),
                           inverse.rle(rep(x, max(y), scale="element")))))
```

`set.control.class` *Set the class of the control list*

Description

This function sets the class of the control list, with the default being the name of the calling function.

Usage

```
set.control.class(myname = as.character(ult(sys.calls(), 2)[[1L]]),
  control = get("control", pos = parent.frame()))
```

Arguments

`myname`	Name of the class to set.
`control`	Control list. Defaults to the `control` variable in the calling function.

Value

The control list with class set.

Note

In earlier versions, `OKnames` and `myname` were autodetected. This capability has been deprecated and results in a warning issued once per session. They now need to be set explicitly.

See Also

check.control.class, print.control.list

`statnet.cite` `CITATION` *file utilities for Statnet packages (DEPRECATED)*

Description

These functions automate citation generation for Statnet Project packages. They no longer appear to work with CRAN and are thus deprecated.

Usage

```
statnet.cite.head(pkg)

statnet.cite.foot(pkg)

statnet.cite.pkg(pkg)
```

Arguments

`pkg`	Name of the package whose citation is being generated.

Value

For `statnet.cite.head` and `statnet.cite.foot`, an object of type `citationHeader` and `citationFooter`, respectively, understood by the `citation` function, with package name substituted into the template.

For `statnet.cite.pkg`, an object of class `bibentry` containing a 'software manual' citation for the package constructed from the current version and author information in the `DESCRIPTION` and a template.

See Also

citation, citHeader, citFooter, bibentry

Examples

```
## Not run:
statnet.cite.head("statnet.common")

statnet.cite.pkg("statnet.common")

statnet.cite.foot("statnet.common")

## End(Not run)
```

`statnetStartupMessage` *Construct a "standard" startup message to be printed when the package is loaded.*

Description

This function uses information returned by `packageDescription` to construct a standard package startup message according to the policy of the Statnet Project. To determine institutional affiliation, it uses a lookup table that maps domain names to institutions. (E.g., *.uw.edu or *.washington.edu maps to University of Washington.)

Usage

```
statnetStartupMessage(pkgname, friends, nofriends)
```

Arguments

`pkgname`	Name of the package whose information is used.
`friends`	This argument is required, but will only be interpreted if the Statnet Project policy makes use of "friendly" package information. A character vector of names of packages whose attribution information incorporates the attribution information of this package, or `TRUE`. (This may, in the future, lead the package to suppress its own startup message when loaded by a "friendly" package.) If `TRUE`, the package considers all other packages "friendly". (This may, in the future, lead the package to suppress its own startup message when loaded by another package, but print it when loaded directly by the user.)
`nofriends`	This argument controls the startup message if the Statnet Project policy does not make use of "friendly" package information but does make use of whether or not the package is being loaded directly or as a dependency. If `TRUE`, the package is willing to suppress its startup message if loaded as a dependency. If `FALSE`, it is not.

Value

A string containing the startup message, to be passed to the packageStartupMessage call or NULL, if policy prescribes printing 's default startup message. (Thus, if statnetStartupMessage returns NULL, the calling package should not call packageStartupMessage at all.)

Note that arguments to friends and nofriends are merely requests, to be interpreted (or ignored) by the statnetStartupMessage according to the Statnet Project policy.

See Also

packageDescription

Examples

```
## Not run:
.onAttach <- function(lib, pkg){
 sm <- statnetStartupMessage("ergm", friends=c("statnet","ergm.count","tergm"), nofriends=FALSE)
  if(!is.null(sm)) packageStartupMessage(sm)
}

## End(Not run)
```

---

sweep_cols.matrix *Suptract a elements of a vector from respective columns of a matrix*

Description

An optimized function equivalent to sweep(x,2,STATS) for a matrix x.

Usage

```
sweep_cols.matrix(x, STATS, disable_checks = FALSE)
```

Arguments

x	a numeric matrix;
STATS	a numeric vector whose length equals to the number of columns of x.
disable_checks	if TRUE, do not check that x is a numeric matrix and its number of columns matches the length of STATS; set in production code for a significant speed-up.

Value

A matrix of the same attributes as x.

Examples

```
x <- matrix(runif(1000), ncol=4)
s <- 1:4

stopifnot(all.equal(sweep_cols.matrix(x, s), sweep(x, 2, s)))
```

`ult` *Extract or replace the* ult*imate (last) element of a vector or a list, or an element counting from the end.*

Description

Extract or replace the *ult*imate (last) element of a vector or a list, or an element counting from the end.

Usage

```
ult(x, i = 1L)

ult(x, i = 1L) <- value
```

Arguments

`x`	a vector or a list.
`i`	index from the end of the list to extract or replace (where 1 is the last element, 2 is the penultimate element, etc.).
`value`	Replacement value for the `i`th element from the end.

Value

An element of `x`.

Note

Due to the way in which assigning to a function is implemented in R, `ult(x) <-e` may be less efficient than `x[[length(x)]] <-e`.

Examples

```
x <- 1:5
(last <- ult(x))
(penultimate <- ult(x, 2)) # 2nd last.



(ult(x) <- 6)
```

```
(ult(x, 2) <- 7) # 2nd last.
x
```

---

unwhich *Construct a logical vector with* TRUE *in specified positions.*

Description

This function is basically an inverse of which.

Usage

```
unwhich(which, n)
```

Arguments

which	a numeric vector of indices to set to TRUE.
n	total length of the output vector.

Value

A logical vector of length n whose elements listed in which are set to TRUE, and whose other elements are set to FALSE.

Examples

```
x <- as.logical(rbinom(10,1,0.5))
stopifnot(all(x == unwhich(which(x), 10)))
```

---

vector.namesmatch *reorder vector v into order determined by matching the names of its elements to a vector of names*

Description

A helper function to reorder vector v (if named) into order specified by matching its names to the argument names

Usage

```
vector.namesmatch(v, names, errname = NULL)
```

Arguments

v	a vector (or list) with named elements, to be reorderd
names	a character vector of element names, corresponding to names of v, specificying desired orering of v
errname	optional, name to be reported in any error messages. default to deparse(substitute(v))

Details

does some checking of appropriateness of arguments, and reorders v by matching its names to character vector names

Value

returns v, with elements reordered

Note

earlier versions of this function did not order as advertiased

Examples

```
test<-list(c=1,b=2,a=3)
vector.namesmatch(test,names=c('a','c','b'))
```

---

wmatrix *A data matrix with row weights*

Description

A representation of a numeric matrix with row weights, represented on either linear (linwmatrix) or logarithmic (logwmatrix) scale.

Usage

```
logwmatrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
  dimnames = NULL, w = NULL)

linwmatrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
  dimnames = NULL, w = NULL)

is.wmatrix(x)

is.logwmatrix(x)

is.linwmatrix(x)
```

```
as.linwmatrix(x, ...)

as.logwmatrix(x, ...)

## S3 method for class 'linwmatrix'
as.linwmatrix(x, ...)

## S3 method for class 'logwmatrix'
as.linwmatrix(x, ...)

## S3 method for class 'logwmatrix'
as.logwmatrix(x, ...)

## S3 method for class 'linwmatrix'
as.logwmatrix(x, ...)

## S3 method for class 'matrix'
as.linwmatrix(x, w = NULL, ...)

## S3 method for class 'matrix'
as.logwmatrix(x, w = NULL, ...)

## S3 method for class 'wmatrix'
print(x, ...)

## S3 method for class 'logwmatrix'
print(x, ...)

## S3 method for class 'linwmatrix'
print(x, ...)

## S3 method for class 'logwmatrix'
compress_rows(x, ...)

## S3 method for class 'linwmatrix'
compress_rows(x, ...)

## S3 method for class 'wmatrix'
decompress_rows(x, target.nrows = NULL, ...)

## S3 method for class 'wmatrix'
x[i, j, ..., drop = FALSE]

## S3 replacement method for class 'wmatrix'
x[i, j, ...] <- value
```

Arguments

data, nrow, ncol, byrow, dimnames	passed to matrix.
w	row weights on the appropriate scale.
x	an object to be coerced or tested.
...	extra arguments, currently unused.
target.nrows	see decompress_rows.
i, j, value	rows and columns and values for extraction or replacement; as matrix.
drop	Used for consistency with the generic. Ignored, and always treated as FALSE.

Value

An object of class linwmatrix/logwmatrix and wmatrix, which is a matrix but also has an attribute w containing row weights on the linear or the natural-log-transformed scale.

Note

Note that wmatrix itself is an "abstract" class: you cannot instantiate it.

Note that at this time, wmatrix is designed as, first and foremost, as class for storing compressed data matrices, so most methods that operate on matrices may not handle the weights correctly and may even cause them to be lost.

See Also

rowweights, lrowweights, compress_rows

Examples

```
(m <- matrix(1:3, 2, 3, byrow=TRUE))
(m <- rbind(m, 3*m, 2*m, m))
(mlog <- as.logwmatrix(m))
(mlin <- as.linwmatrix(m))
(cmlog <- compress_rows(mlog))
(cmlin <- compress_rows(mlin))

stopifnot(all.equal(as.linwmatrix(cmlog),cmlin))

cmlog[2,] <- 1:3
(cmlog <- compress_rows(cmlog))
stopifnot(sum(rowweights(cmlog))==nrow(m))

(m3 <- matrix(c(1:3,(1:3)*2,(1:3)*3), 3, 3, byrow=TRUE))
(rowweights(m3) <- c(4, 2, 2))

stopifnot(all.equal(compress_rows(as.logwmatrix(m)), as.logwmatrix(m3),check.attributes=FALSE))
stopifnot(all.equal(rowweights(compress_rows(as.logwmatrix(m))),
                    rowweights(as.logwmatrix(m3)),check.attributes=FALSE))
```

wmatrix_weights *Set or extract weighted matrix row weights*

Description

Set or extract weighted matrix row weights

Usage

```
rowweights(x, ...)

## S3 method for class 'linwmatrix'
rowweights(x, ...)

## S3 method for class 'logwmatrix'
rowweights(x, ...)

lrowweights(x, ...)

## S3 method for class 'logwmatrix'
lrowweights(x, ...)

## S3 method for class 'linwmatrix'
lrowweights(x, ...)

rowweights(x, ...) <- value

## S3 replacement method for class 'linwmatrix'
rowweights(x, update = TRUE, ...) <- value

## S3 replacement method for class 'logwmatrix'
rowweights(x, update = TRUE, ...) <- value

lrowweights(x, ...) <- value

## S3 replacement method for class 'linwmatrix'
lrowweights(x, update = TRUE, ...) <- value

## S3 replacement method for class 'logwmatrix'
lrowweights(x, update = TRUE, ...) <- value

## S3 replacement method for class 'matrix'
rowweights(x, ...) <- value

## S3 replacement method for class 'matrix'
lrowweights(x, ...) <- value
```

Arguments

x	a linwmatrix, a logwmatrix, or a matrix; a matrix is coerced to a weighted matrix of an appropriate type.
...	extra arguments for methods.
value	weights to set, on the appropriate scale.
update	if TRUE (the default), the old weights are updated with the new weights (i.e., corresponding weights are multiplied on linear scale or added on on log scale); otherwise, they are overwritten.

Value

For the accessor functions, the row weights or the row log-weights; otherwise, a weighted matrix with modified weights. The type of weight (linear or logarithmic) is converted to the required type and the type of weighting of the matrix is preserved.

Index