Package ‘S4Vectors’

April 15, 2017

Title S4 implementation of vectors and lists

Description The S4Vectors package defines the Vector and List virtual classes and a set of generic functions that extend the semantic of ordinary vectors and lists in R. Package developers can easily implement vector-like or list-like objects as concrete subclasses of Vector or List. In addition, a few low-level concrete subclasses of general interest (e.g. DataFrame, Rle, and Hits) are implemented in the S4Vectors package itself (many more are implemented in the IRanges package and in other Bioconductor infrastructure packages).

Version 0.12.2

Encoding UTF-8

Author H. Pagès, M. Lawrence and P. Aboyoun

Maintainer Bioconductor Package Maintainer <maintainer@bioconductor.org>

biocViews Infrastructure, DataRepresentation

Depends R (>= 3.3.0), methods, utils, stats, stats4, BiocGenerics (>= 0.15.10)

Suggests IRanges, GenomicRanges, Matrix, ShortRead, graph, data.table, RUnit

License Artistic-2.0

Collate S4-utils.R show-utils.R utils.R normarg-utils.R isSorted.R subsetting-utils.R vector-utils.R logical-utils.R int-utils.R str-utils.R eval-utils.R DataTable-class.R Annotated-class.R Vector-class.R Vector-comparison.R Vector-setops.R Vector-merge.R Hits-class.R Hits-comparison.R Hits-setops.R Rle-class.R Rle-utils.R List-class.R List-comparison.R List-utils.R SimpleList-class.R HitsList-class.R DataFrame-class.R expand-methods.R FilterRules-class.R aggregate-methods.R shiftApply-methods.R split-methods.R Pairs-class.R zzz.R

NeedsCompilation yes

R topics documented:

`aggregate-methods` *Compute summary statistics of subsets of vector-like objects*

Description

The **S4Vectors** package defines `aggregate` methods for Vector, Rle, and List objects.

Usage

```
## S4 method for signature 'Vector'
aggregate(x, by, FUN, start=NULL, end=NULL, width=NULL,
          frequency=NULL, delta=NULL, ..., simplify=TRUE)

## S4 method for signature 'Rle'
aggregate(x, by, FUN, start=NULL, end=NULL, width=NULL,
          frequency=NULL, delta=NULL, ..., simplify=TRUE)

## S4 method for signature 'List'
aggregate(x, by, FUN, start=NULL, end=NULL, width=NULL,
          frequency=NULL, delta=NULL, ..., simplify=TRUE)
```

Arguments

`x`	A Vector, Rle, or List object.
`by`	An object with `start`, `end`, and `width` methods. If `x` is a List object, the `by` parameter can be a RangesList object to aggregate within the list elements rather than across them. When `by` is a RangesList object, the output is either a SimpleAtomicList object, if possible, or a SimpleList object, if not.
`FUN`	The function, found via `match.fun`, to be applied to each subset of `x`.
`start, end, width`	The start, end, and width of the subsets. If `by` is missing, then two of the three must be supplied and have the same length.
`frequency, delta`	Optional arguments that specify the sampling frequency and increment within the subsets (in the same fashion as `window` from the **stats** package does).
`...`	Optional arguments to `FUN`.
`simplify`	A logical value specifying whether the result should be simplified to a vector or matrix if possible.

Details

Subsets of `x` can be specified either via the `by` argument or via the `start`, `end`, `width`, `frequency`, and `delta` arguments.

For example, if `start` and `end` are specified, then:

```
aggregate(x, FUN=FUN, start=start, end=end, ..., simplify=simplify)
```

is equivalent to:

```
sapply(seq_along(start),
       function(i) FUN(x[start[i]:end[i]], ...), simplify=simplify)
```

(replace `x[start[i]:end[i]]` with 2D-style subsetting `x[start[i]:end[i], ]` if `x` is a DataFrame object).

See Also

- The `aggregate` function in the **stats** package.
- Vector, Rle, List, and DataFrame objects.
- The `start`, `end`, and `width` generic functions defined in the **BiocGenerics** package.

Examples

```
x <- Rle(10:2, 1:9)
aggregate(x, x > 4, mean)
aggregate(x, FUN=mean, start=1:26, width=20)

## Note that aggregate() works on a DataFrame object the same way it
## works on an ordinary data frame:
aggregate(DataFrame(state.x77), list(Region=state.region), mean)
aggregate(weight ~ feed, data=DataFrame(chickwts), mean)
```

```
library(IRanges)
by <- IRanges(start=1:26, width=20, names=LETTERS)
aggregate(x, by, is.unsorted)
```

Annotated-class *Annotated class*

Description

The virtual class `Annotated` is used to standardize the storage of metadata with a subclass.

Details

The `Annotated` class supports the storage of global metadata in a subclass. This is done through the `metadata` slot that stores a list object.

Accessors

In the following code snippets, `x` is an `Annotated` object.

`metadata(x)`, `metadata(x) <- value`: Get or set the list holding arbitrary R objects as annotations. May be, and often is, empty.

Author(s)

P. Aboyoun

See Also

The Vector class, which extends Annotated directly.

Examples

```
showClass("Annotated")  # shows (some of) the known subclasses

## If the IRanges package was not already loaded, this will show
## more subclasses:
library(IRanges)
showClass("Annotated")
```

DataFrame-class *DataFrame objects*

Description

The DataFrame class extends the DataTable virtual class and supports the storage of any type of object (with length and [methods) as columns.

Details

On the whole, the DataFrame behaves very similarly to data.frame, in terms of construction, subsetting, splitting, combining, etc. The most notable exception is that the row names are optional. This means calling rownames(x) will return NULL if there are no row names. Of course, it could return seq_len(nrow(x)), but returning NULL informs, for example, combination functions that no row names are desired (they are often a luxury when dealing with large data).

As DataFrame derives from Vector, it is possible to set an annotation string. Also, another DataFrame can hold metadata on the columns.

For a class to be supported as a column, it must have length and [methods, where [supports subsetting only by i and respects drop=FALSE. Optionally, a method may be defined for the showAsCell generic, which should return a vector of the same length as the subset of the column passed to it. This vector is then placed into a data.frame and converted to text with format. Thus, each element of the vector should be some simple, usually character, representation of the corresponding element in the column.

Constructor

DataFrame(..., row.names = NULL, check.names = TRUE): Constructs a DataFrame in similar fashion to data.frame. Each argument in ... is coerced to a DataFrame and combined column-wise. No special effort is expended to automatically determine the row names from the arguments. The row names should be given in row.names; otherwise, there are no row names. This is by design, as row names are normally undesirable when data is large. If check.names is TRUE, the column names will be checked for syntactic validity and made unique, if necessary.

To store an object of a class that does not support coercion to DataFrame, wrap it in I(). The class must still have methods for length and [.

Accessors

In the following code snippets, x is a DataFrame.

- dim(x): Get the length two integer vector indicating in the first and second element the number of rows and columns, respectively.
- dimnames(x), dimnames(x) <- value: Get and set the two element list containing the row names (character vector of length nrow(x) or NULL) and the column names (character vector of length ncol(x)).

Coercion

as(from, "DataFrame"): By default, constructs a new DataFrame with from as its only column. If from is a matrix or data.frame, all of its columns become columns in the new DataFrame. If from is a list, each element becomes a column, recycling as necessary. Note that for the

DataFrame to behave correctly, each column object must support element-wise subsetting via the [method and return the number of elements with length. It is recommended to use the DataFrame constructor, rather than this interface.

as.list(x): Coerces x, a DataFrame, to a list.

as.data.frame(x, row.names=NULL, optional=FALSE): Coerces x, a DataFrame, to a data.frame. Each column is coerced to a data.frame and then column bound together. If row.names is NULL, they are retrieved from x, if it has any. Otherwise, they are inferred by the data.frame constructor.

NOTE: conversion of x to a data.frame is not supported if x contains any list, SimpleList, or CompressedList columns.

as(from, "data.frame"): Coerces a DataFrame to a data.frame by calling as.data.frame(from).

as.matrix(x): Coerces the DataFrame to a matrix, if possible.

Subsetting

In the following code snippets, x is a DataFrame.

x[i,j,drop]: Behaves very similarly to the [.data.frame method, except i can be a logical Rle object and subsetting by matrix indices is not supported. Indices containing NA's are also not supported.

x[i,j] <- value: Behaves very similarly to the [<-.data.frame method.

x[[i]]: Behaves very similarly to the [[.data.frame method, except arguments j and exact are not supported. Column name matching is always exact. Subsetting by matrices is not supported.

x[[i]] <- value: Behaves very similarly to the [[<-.data.frame method, except argument j is not supported.

Combining

In the following code snippets, x is a DataFrame.

rbind(...): Creates a new DataFrame by combining the rows of the DataFrame objects in .... Very similar to rbind.data.frame, except in the handling of row names. If all elements have row names, they are concatenated and made unique. Otherwise, the result does not have row names. Currently, factors are not handled well (their levels are dropped). This is not a high priority until there is an XFactor class.

cbind(...): Creates a new DataFrame by combining the columns of the DataFrame objects in .... Very similar to cbind.data.frame, except row names, if any, are dropped. Consider the DataFrame as an alternative that allows one to specify row names.

Author(s)

Michael Lawrence

See Also

- DataTable and SimpleList which DataFrame extends directly.

Examples

```
score <- c(1L, 3L, NA)
counts <- c(10L, 2L, NA)
row.names <- c("one", "two", "three")

df <- DataFrame(score) # single column
df[["score"]]
df <- DataFrame(score, row.names = row.names) #with row names
rownames(df)

df <- DataFrame(vals = score) # explicit naming
df[["vals"]]

# arrays
ary <- array(1:4, c(2,1,2))
sw <- DataFrame(I(ary))

# a data.frame
sw <- DataFrame(swiss)
as.data.frame(sw) # swiss, without row names
# now with row names
sw <- DataFrame(swiss, row.names = rownames(swiss))
as.data.frame(sw) # swiss

# subsetting

sw[] # identity subset
sw[,] # same

sw[NULL] # no columns
sw[,NULL] # no columns
sw[NULL,] # no rows

## select columns
sw[1:3]
sw[,1:3] # same as above
sw[,"Fertility"]
sw[,c(TRUE, FALSE, FALSE, FALSE, FALSE, FALSE)]

## select rows and columns
sw[4:5, 1:3]

sw[1] # one-column DataFrame
## the same
sw[, 1, drop = FALSE]
sw[, 1] # a (unnamed) vector
sw[[1]] # the same
sw[["Fertility"]]

sw[["Fert"]] # should return 'NULL'

sw[1,] # a one-row DataFrame
sw[1,, drop=TRUE] # a list

## duplicate row, unique row names are created
sw[c(1, 1:2),]
```

```
## indexing by row names
sw["Courtelary",]
subsw <- sw[1:5,1:4]
subsw["C",] # partially matches

## row and column names
cn <- paste("X", seq_len(ncol(swiss)), sep = ".")
colnames(sw) <- cn
colnames(sw)
rn <- seq(nrow(sw))
rownames(sw) <- rn
rownames(sw)

## column replacement

df[["counts"]] <- counts
df[["counts"]]
df[[3]] <- score
df[["X"]]
df[[3]] <- NULL # deletion
```

DataTable-class *DataTable objects*

Description

DataTable is an API only (i.e. virtual class with no slots) for accessing objects with a rectangular shape like DataFrame or RangedData objects. It mimics the API for standard data.frame objects.

Accessors

In the following code snippets, `x` is a `DataTable`.

`nrow(x)`, `ncol(x)`: Get the number of rows and columns, respectively.

`NROW(x)`, `NCOL(x)`: Same as `nrow(x)` and `ncol(x)`, respectively.

`dim(x)`: Length two integer vector defined as `c(nrow(x), ncol(x))`.

`rownames(x)`, `colnames(x)`: Get the names of the rows and columns, respectively.

`dimnames(x)`: Length two list of character vectors defined as `list(rownames(x), colnames(x))`.

Subsetting

In the code snippets below, `x` is a DataTable object.

`x[i, j, drop=TRUE]`: Return a new DataTable object made of the selected rows and columns. For single column selection, the `drop` argument specifies whether or not to coerce the returned sequence to a standard vector.

`head(x, n=6L)`: If `n` is non-negative, returns the first `n` rows of the DataTable object. If `n` is negative, returns all but the last `abs(n)` rows of the DataTable object.

`tail(x, n=6L)`: If `n` is non-negative, returns the last `n` rows of the DataTable object. If `n` is negative, returns all but the first `abs(n)` rows of the DataTable object.

`subset(x, subset, select, drop=FALSE)`: Return a new DataTable object using:

subset logical expression indicating rows to keep, where missing values are taken as FALSE.

select expression indicating columns to keep.

drop passed on to `[` indexing operator.

`na.omit(object)`: Returns a subset with incomplete cases removed.

`na.exclude(object)`: Returns a subset with incomplete cases removed (but to be included with NAs in statistical results).

`is.na(x)`: Returns a logical matrix indicating which cells are missing.

`complete.cases(x)`: Returns a logical vector identifying which cases have no missing values.

Combining

In the code snippets below, `x` is a DataTable object.

`cbind(...)`: Creates a new `DataTable` by combining the columns of the `DataTable` objects in `...`.

`rbind(...)`: Creates a new `DataTable` by combining the rows of the `DataTable` objects in `...`.

`merge(x, y, ...)`: Merges two `DataTable` objects `x` and `y`, with arguments in `...` being the same as those allowed by the base `merge`. It is allowed for either `x` or `y` to be a `data.frame`.

Looping

In the code snippets below, `x` is a DataTable object.

`by(data, INDICES, FUN, ..., simplify = TRUE)`: Apply `FUN` to each group of `data`, a `DataTable`, formed by the factor (or list of factors) `INDICES`. Exactly the same contract as `as.data.frame`.

Utilities

`duplicated(x)`: Returns a logical vector indicating the rows that are identical to a previous row.

`unique(x)`: Returns a new `DataTable` after removing the duplicated rows from `x`.

`show(x)`: By default the `show` method displays 5 head and 5 tail lines. The number of lines can be altered by setting the global options `showHeadLines` and `showTailLines`. If the object length is less than the sum of the options, the full object is displayed. These options affect GRanges, GAlignments, Ranges, DataTable and XString objects.

Coercion

`as.env(x, enclos = parent.frame())`: Creates an environment from `x` with a symbol for each `colnames(x)`. The values are not actually copied into the environment. Rather, they are dynamically bound using `makeActiveBinding`. This prevents unnecessary copying of the data from the external vectors into R vectors. The values are cached, so that the data is not copied every time the symbol is accessed.

Statistical modeling with DataTable

A number of wrappers are implemented for performing statistical procedures, such as model fitting, with DataTable objects.

Tabulation:

`xtabs(formula = ~., data, subset, na.action, exclude = c(NA, NaN), drop.unused.levels = F`
Like the original `xtabs`, except `data` is a `DataTable`.

See Also

- DataFrame for an implementation that mimics `data.frame`.
- data.frame

Examples

```
showClass("DataTable")  # shows (some of) the known subclasses

library(IRanges)
df <- DataFrame(as.data.frame(UCBAdmissions))
xtabs(Freq ~ Gender + Admit, df)
```

`expand` *Unlist the list-like columns of a DataFrame object*

Description

expand transforms a DataFrame object into a new DataFrame object where the columns specified by the user are unlisted. The transformed DataFrame object has the same colnames as the original but typically more rows.

Usage

```
## S4 method for signature 'DataFrame'
expand(x, colnames, keepEmptyRows = FALSE)
```

Arguments

`x`	A DataFrame object with list-like columns or a Vector object with list-like metadata columns (i.e. with list-like columns in `mcols(x)`).
`colnames`	A `character` or `numeric` vector containing the names or indices of the list-like columns to unlist. The order in which columns are unlisted is controlled by the column order in this vector. This defaults to all of the recursive (list-like) columns in `x`.
`keepEmptyRows`	A `logical` indicating if rows containing empty list elements in the specified `colnames` should be retained or dropped. When `TRUE`, list elements are replaced with NA and all rows are kept. When `FALSE`, rows with empty list elements in the `colnames` columns are dropped.

Value

A DataFrame object that has been expanded row-wise to match the length of the unlisted columns.

See Also

- DataFrame objects.

Examples

```
library(IRanges)
aa <- CharacterList("a", paste0("d", 1:2), paste0("b", 1:3), c(), "c")
bb <- CharacterList(paste0("sna", 1:2),"foo", paste0("bar",1:3),c(),"hica")
df <- DataFrame(aa=aa, bb=bb, cc=11:15)

## Expand by all list-like columns (aa, bb), dropping rows with empty
## list elements:
expand(df)

## Expand the aa column only:
expand(df, colnames="aa", keepEmptyRows=TRUE)
expand(df, colnames="aa", keepEmptyRows=FALSE)

## Expand the aa and then the bb column:
expand(df, colnames=c("aa","bb"), keepEmptyRows=TRUE)
expand(df, colnames=c("aa","bb"), keepEmptyRows=FALSE)
```

`FilterMatrix-class` *Matrix for Filter Results*

Description

A `FilterMatrix` object is a matrix meant for storing the logical output of a set of `FilterRules`, where each rule corresponds to a column. The `FilterRules` are stored within the `FilterMatrix` object, for the sake of provenance. In general, a `FilterMatrix` behaves like an ordinary `matrix`.

Accessor methods

In the code snippets below, `x` is a `FilterMatrix` object.

`filterRules(x)`: Get the `FilterRules` corresponding to the columns of the matrix.

Constructor

`FilterMatrix(matrix, filterRules)`: Constructs a `FilterMatrix`, from a given `matrix` and `filterRules`. Not usually called by the user, see `evalSeparately`.

Utilities

`summary(object, discarded = FALSE, percent = FALSE)`: Returns a numeric vector containing the total number of records (`nrow`), the number passed by each filter, and the number of records that passed every filter. If `discarded` is `TRUE`, then the numbers are inverted (i.e., the values are subtracted from the number of rows). If `percent` is `TRUE`, then the numbers are percent of total.

Author(s)

Michael Lawrence

See Also

- `evalSeparately` is the typical way to generate this object.
- FilterRules objects.

FilterRules-class *Collection of Filter Rules*

Description

A `FilterRules` object is a collection of filter rules, which can be either `expression` or `function` objects. Rules can be disabled/enabled individually, facilitating experimenting with different combinations of filters.

Details

It is common to split a dataset into subsets during data analysis. When data is large, however, representing subsets (e.g. by logical vectors) and storing them as copies might become too costly in terms of space. The `FilterRules` class represents subsets as lightweight `expression` and/or `function` objects. Subsets can then be calculated when needed (on the fly). This avoids copying and storing a large number of subsets. Although it might take longer to frequently recalculate a subset, it often is a relatively fast operation and the space savings tend to be more than worth it when data is large.

Rules may be either expressions or functions. Evaluating an expression or invoking a function should result in a logical vector. Expressions are often more convenient, but functions (i.e. closures) are generally safer and more powerful, because the user can specify the enclosing environment. If a rule is an expression, it is evaluated inside the `envir` argument to the `eval` method (see below). If a function, it is invoked with `envir` as its only argument. See examples.

Accessor methods

In the code snippets below, `x` is a `FilterRules` object.

`active(x)`: Get the logical vector of length `length(x)`, where `TRUE` for an element indicates that the corresponding rule in `x` is active (and inactive otherwise). Note that `names(active(x))` is equal to `names(x)`.

`active(x) <- value`: Replace the active state of the filter rules. If `value` is a logical vector, it should be of length `length(x)` and indicate which rules are active. Otherwise, it can be either numeric or character vector, in which case it sets the indicated rules (after dropping NA's) to active and all others to inactive. See examples.

Constructor

`FilterRules(exprs = list(), ..., active = TRUE)`: Constructs a `FilterRules` with the rules given in the list `exprs` or in `...`. The initial active state of the rules is given by `active`, which is recycled as necessary. Elements in `exprs` may be either character (parsed into an expression), a language object (coerced to an expression), an expression, or a function that takes at least one argument. **IMPORTANTLY**, all arguments in `...` are `quote()`'d and then coerced to an expression. So, for example, character data is only parsed if it is a literal. The names of the filters are taken from the names of `exprs` and `...`, if given. Otherwise, the character vectors take themselves as their name and the others are deparsed (before any coercion). Thus, it is recommended to always specify meaningful names. In any case, the names are made valid and unique.

Subsetting and Replacement

In the code snippets below, `x` is a `FilterRules` object.

`x[i]`: Subsets the filter rules using the same interface as for `Vector`.

`x[[i]]`: Extracts an expression or function via the same interface as for `List`.

`x[[i]] <- value`: The same interface as for `List`. The default active state for new rules is `TRUE`.

Combining

In the code snippets below, `x` is a `FilterRules` object.

`append(x, values, after = length(x))`: Appends the `values` `FilterRules` instance onto `x` at the index given by `after`.

`c(x, ..., recursive = FALSE)`: Concatenates the `FilterRule` instances in `...` onto the end of `x`.

Evaluating

`eval(expr, envir = parent.frame(), enclos = if (is.list(envir) || is.pairlist(envir)) parent.frame() else baseenv())`: Evaluates a `FilterRules` instance (passed as the `expr` argument). Expression rules are evaluated in `envir`, while function rules are invoked with `envir` as their only argument. The evaluation of a rule should yield a logical vector. The results from the rule evaluations are combined via the AND operation (i.e. `&`) so that a single logical vector is returned from `eval`.

`evalSeparately(expr, envir = parent.frame(), enclos = if (is.list(envir) || is.pairlist(en baseenv())`: Evaluates separately each rule in a `FilterRules` instance (passed as the `expr` argument). Expression rules are evaluated in `envir`, while function rules are invoked with `envir` as their only argument. The evaluation of a rule should yield a logical vector. The results from the rule evaluations are combined into a logical matrix, with a column for each rule. This is essentially the parallel evaluator, while `eval` is the serial evaluator.

`subsetByFilter(x, filter)`: Evaluates `filter` on `x` and uses the result to subset `x`. The result contains only the elements in `x` for which `filter` evaluates to `TRUE`.

`summary(object, subject)`: Returns an integer vector with the number of elements in `subject` that pass each rule in `object`, along with a count of the elements that pass all filters.

Filter Closures

When a closure (function) is included as a filter in a `FilterRules` object, it is converted to a `FilterClosure`, which is currently nothing more than a marker class that extends `function`. When a `FilterClosure` filter is extracted, there are some accessors and utilities for manipulating it:

`params`: Gets a named list of the objects that are present in the enclosing environment (without inheritance). This assumes that a filter is constructed via a constructor function, and the objects in the frame of the constructor (typically, the formal arguments) are the parameters of the filter.

Author(s)

Michael Lawrence

See Also

FilterMatrix objects for storing the logical output of a set of FilterRules objects.

Examples

```
## constructing a FilterRules instance

## an empty set of filters
filters <- FilterRules()

## as a simple character vector
filts <- c("peaks", "promoters")
filters <- FilterRules(filts)
active(filters) # all TRUE

## with functions and expressions
filts <- list(peaks = expression(peaks), promoters = expression(promoters),
              find_eboxes = function(rd) rep(FALSE, nrow(rd)))
filters <- FilterRules(filts, active = FALSE)
active(filters) # all FALSE

## direct, quoted args (character literal parsed)
filters <- FilterRules(under_peaks = peaks, in_promoters = "promoters")
filts <- list(under_peaks = expression(peaks),
              in_promoters = expression(promoters))

## specify both exprs and additional args
filters <- FilterRules(filts, diffexp = de)

filts <- c("promoters", "peaks", "introns")
filters <- FilterRules(filts)

## evaluation
df <- DataFrame(peaks = c(TRUE, TRUE, FALSE, FALSE),
                promoters = c(TRUE, FALSE, FALSE, TRUE),
                introns = c(TRUE, FALSE, FALSE, FALSE))
eval(filters, df)
fm <- evalSeparately(filters, df)
identical(filterRules(fm), filters)
summary(fm)
summary(fm, percent = TRUE)
fm <- evalSeparately(filters, df, serial = TRUE)

## set the active state directly

active(filters) <- FALSE # all FALSE
active(filters) <- TRUE # all TRUE
active(filters) <- c(FALSE, FALSE, TRUE)
active(filters)["promoters"] <- TRUE # use a filter name

## toggle the active state by name or index

active(filters) <- c(NA, 2) # NA's are dropped
active(filters) <- c("peaks", NA)
```

`Hits-class` *Hits objects*

Description

The Hits class is a container for representing a set of hits between a set of *left nodes* and a set of *right nodes*. Note that only the hits are stored in the object. No information about the left or right nodes is stored, except their number.

For example, the `findOverlaps` function, defined and documented in the **IRanges** package, returns the hits between the `query` and `subject` arguments in a `Hits` object.

Usage

```
## Constructor functions

Hits(from=integer(0), to=integer(0), nLnode=0L, nRnode=0L, ...,
     sort.by.query=FALSE)

SelfHits(from=integer(0), to=integer(0), nnode=0L, ...,
     sort.by.query=FALSE)
```

Arguments

`from, to`	2 integer vectors of the same length. The values in `from` must be >= 1 and <= `nLnode`. The values in `to` must be >= 1 and <= `nRnode`.
`nLnode, nRnode`	Number of left and right nodes.
`...`	Metadata columns to set on the Hits object. All the metadata columns must be vector-like objects of the same length as `from` and `to`.
`sort.by.query`	Should the hits in the returned object be sorted by query? If yes, then a SortedByQueryHits object is returned (SortedByQueryHits is a subclass of Hits).
`nnode`	Number of nodes.

Accessors

In the code snippets below, `x` is a `Hits` object.

`length(x)`: get the number of hits

`from(x)`: Equivalent to `as.data.frame(x)[[1]]`.

`to(x)`: Equivalent to `as.data.frame(x)[[2]]`.

`nLnode(x)`, `nrow(x)`: get the number of left nodes

`nRnode(x)`, `ncol(x)`: get the number of right nodes

`countLnodeHits(x)`: Counts the number of hits for each left node, returning an integer vector.

`countRnodeHits(x)`: Counts the number of hits for each right node, returning an integer vector.

The following accessors are just aliases for the above accessors:

`queryHits(x)`: alias for `from(x)`.

`subjectHits(x)`: alias for `to(x)`.

`queryLength(x)`: alias for `nLnode(x)`.

`subjectLength(x)`: alias for `nRnode(x)`.

`countQueryHits(x)`: alias for `countLnodeHits(x)`.

`countSubjectHits(x)`: alias for `countRnodeHits(x)`.

Coercion

In the code snippets below, `x` is a `Hits` object.

`as.matrix(x)`: Coerces `x` to a two column integer matrix, with each row representing a hit between a left node (first column) and a right node (second column).

`as.table(x)`: Counts the number of hits for each left node in `x` and outputs the counts as a `table`.

Subsetting

In the code snippets below, `x` is a `Hits` object.

`x[i]`: Subset the Hits object.

Other transformations

In the code snippets below, `x` is a `Hits` object.

`t(x)`: Transpose `x` by interchanging the left and right nodes. This allows, for example, counting the number of hits for each right node using `as.table`.

`remapHits(x, Lnodes.remapping=NULL, new.nLnode=NA, Rnodes.remapping=NULL`
Only supports SortedByQueryHits objects at the moment.

Remaps the left and/or right nodes in `x`. The left nodes are remapped thru the map specified via the `Lnodes.remapping` and `new.nLnode` arguments. The right nodes are remapped thru the map specified via the `Rnodes.remapping` and `new.nRnode` arguments.

`Lnodes.remapping` must represent a function defined on the 1..M interval that takes values in the 1..N interval, where N is `nLnode(x)` and M is the value specified by the user via the `new.nLnode` argument. Note that this mapping function doesn't need to be injective or surjective. Also it is not represented by an R function but by an integer vector of length M with no NAs. More precisely `Lnodes.remapping` can be NULL (identity map), or a vector of `nLnode(x)` non-NA integers that are >= 1 and <= `new.nLnode`, or a factor of length `nLnode(x)` with no NAs (a factor is treated as an integer vector, and, if missing, `new.nLnode` is taken to be its number of levels). Note that a factor will typically be used to represent a mapping function that is not injective.

The same applies to the `Rnodes.remapping`.

`remapHits` returns a Hits object where `from(x)` and `to(x)` have been remapped thru the 2 specified maps. This remapping is actually only the 1st step of the transformation, and is followed by 2 additional steps: (2) the removal of duplicated hits, and (3) the reordering of the hits (first by query hits, then by subject hits). Note that if the 2 maps are injective then the remapping won't introduce duplicated hits, so, in that case, step (2) is a no-op (but is still performed). Also if the "query map" is strictly ascending and the "subject map" ascending then the remapping will preserve the order of the hits, so, in that case, step (3) is also a no-op (but is still performed).

`breakTies(x, method=c("first", "last"))`: Restrict the hits so that every left node maps to at most one right node. If `method` is "first", for each left node, select the edge with the first (lowest rank) right node, if any. If `method` is "last", select the edge with the last (highest rank) right node.

SelfHits

A SelfHits object is a Hits object where the left and right nodes are identical. For a SelfHits object `x`, `nLnode(x)` is equal to `nRnode(x)`. The object can be seen as an oriented graph where `nLnode` is the nb of nodes and the hits are the (oriented) edges. SelfHits objects support the same set of accessors as Hits objects plus the `nnode()` accessor that is equivalent to `nLnode()` and `nRnode()`.

We also provide two little utilities to operate on a SelfHits object `x`:

`isSelfHit(x)`: A *self hit* is an edge from a node to itself. `isSelfHit(x)` returns a logical vector *parallel* to `x` indicating which elements in `x` are self hits.

`isRedundantHit(x)`: When there is more than 1 edge between 2 given nodes (regardless of orientation), the extra edges are considered to be *redundant hits*. `isRedundantHit(x)` returns a logical vector *parallel* to `x` indicating which elements in `x` are redundant hits.

Author(s)

Michael Lawrence and Hervé Pagès

See Also

- Hits-comparison for comparing and ordering hits.
- The `findOverlaps` function in the **IRanges** package which returns SortedByQueryHits object.
- Hits-examples in the **IRanges** package, for some examples of Hits object basic manipulation.
- setops-methods in the **IRanges** package, for set operations on Hits objects.

Examples

```
from <- c(5, 2, 3, 3, 3, 2)
to <- c(11, 15, 5, 4, 5, 11)
id <- letters[1:6]

Hits(from, to, 7, 15, id)
Hits(from, to, 7, 15, id, sort.by.query=TRUE)

## ---------------------------------------------------------------------
## selectHits()
## ---------------------------------------------------------------------

x <- c("a", "b", "a", "c", "d")
table <- c("a", "e", "d", "a", "a", "d")
hits <- findMatches(x, table)  # sorts the hits by query
hits

selectHits(hits, select="all")  # no-op
selectHits(hits, select="first")
selectHits(hits, select="last")
selectHits(hits, select="arbitrary")
selectHits(hits, select="count")

## ---------------------------------------------------------------------
## remapHits()
## ---------------------------------------------------------------------

Lnodes.remapping <- factor(c(a="A", b="B", c="C", d="D")[x],
                           levels=LETTERS[1:4])
remapHits(hits, Lnodes.remapping=Lnodes.remapping)

## See ?`Hits-examples` in the IRanges package for more examples of basic
```

```
## manipulation of Hits objects.

## ---------------------------------------------------------------------
## SelfHits objects
## ---------------------------------------------------------------------

hits2 <- SelfHits(c(2, 3, 3, 3, 3, 3, 4, 4, 4), c(4, 3, 2:4, 2, 2:3, 2), 4)
## Hits 2 and 4 are self hits (from 3rd node to itself):
which(isSelfHit(hits2))
## Hits 4, 6, 7, 8, and 9, are redundant hits:
which(isRedundantHit(hits2))

hits3 <- findMatches(x)
hits3[!isSelfHit(hits3)]
hits3[!(isSelfHit(hits3) | isRedundantHit(hits3))]
```

`Hits-comparison` *Comparing and ordering hits*

Description

==, !=, <=, >=, <, >, `match()`, `%in%`, `order()`, `sort()`, and `rank()` can be used on Hits objects to compare and order hits.

Note that only the `"pcompare"`, `"match"`, and `"order"` methods are actually defined for Hits objects. This is all what is needed to make all the other comparing and ordering operations (i.e. ==, !=, <=, >=, <, >, `%in%`, `sort()`, and `rank()`) work on these objects (see ?`` `Vector-comparison` `` for more information about this).

Usage

```
## S4 method for signature 'Hits,Hits'
pcompare(x, y)

## S4 method for signature 'Hits,Hits'
match(x, table, nomatch=NA_integer_, incomparables=NULL,
      method=c("auto", "quick", "hash"))

## S4 method for signature 'Hits'
order(..., na.last=TRUE, decreasing=FALSE, method=c("shell", "radix"))
```

Arguments

`x, y, table` *Compatible* Hits objects, that is, Hits objects with the same subject and query lengths.

`nomatch` The value to be returned in the case when no match is found. It is coerced to an `integer`.

`incomparables` Not supported.

`method` For `match`: Use a Quicksort-based (`method="quick"`) or a hash-based (`method="hash"`) algorithm. The latter tends to give better performance, except maybe for some pathological input that we've not encountered so far. When `method="auto"` is specified, the most efficient algorithm will be used, that is, the hash-based algorithm if `length(x) <= 2^29`, otherwise the Quicksort-based algorithm.

For `order`: The `method` argument is ignored.

`...` One or more Hits objects. The additional Hits objects are used to break ties.

`na.last` Ignored.

`decreasing` `TRUE` or `FALSE`.

Details

Only hits that belong to Hits objects with same subject and query lengths can be compared.

Hits are ordered by query hit first, and then by subject hit. On a Hits object, `order`, `sort`, and `rank` are consistent with this order.

`pcompare(x, y)`: Performs element-wise (aka "parallel") comparison of 2 Hits objects `x` and `y`, that is, returns an integer vector where the i-th element is less than, equal to, or greater than zero if `x[i]` is considered to be respectively less than, equal to, or greater than `y[i]`. See ?`` `Vector-comparison` `` for how `x` or `y` is recycled when the 2 objects don't have the same length.

`match(x, table, nomatch=NA_integer_, method=c("auto", "quick", "hash"))`: Returns an integer vector of the length of `x`, containing the index of the first matching hit in `table` (or `nomatch` if there is no matching hit) for each hit in `x`.

`order(...)`: Returns a permutation which rearranges its first argument (a Hits object) into ascending order, breaking ties by further arguments (also Hits objects).

Author(s)

Hervé Pagès

See Also

- Hits objects.
- Vector-comparison for general information about comparing, ordering, and tabulating vector-like objects.

Examples

```
## ---------------------------------------------------------------------
## A. ELEMENT-WISE (AKA "PARALLEL") COMPARISON OF 2 Hits OBJECTS
## ---------------------------------------------------------------------
hits <- Hits(c(2, 4, 4, 4, 5, 5), c(3, 1, 3, 2, 3, 2), 6, 3)
hits

pcompare(hits, hits[3])
pcompare(hits[3], hits)

hits == hits[3]
hits != hits[3]
hits >= hits[3]
hits < hits[3]

## ---------------------------------------------------------------------
## B. match(), %in%
## ---------------------------------------------------------------------
```

```
table <- hits[-c(1, 3)]
match(hits, table)

hits %in% table

## ---------------------------------------------------------------------
## C. order(), sort(), rank()
## ---------------------------------------------------------------------
order(hits)
sort(hits)
rank(hits)
```

Hits-setops *Set operations on Hits objects*

Description

Perform set operations on Hits objects.

Details

`union(x, y)`, `intersect(x, y)`, `setdiff(x, y)`, and `setequal(x, y)` work on Hits objects `x` and `y` only if the objects are *compatible Hits objects*, that is, if they have the same subject and query lengths. These operations return respectively the union, intersection, (asymmetric!) difference, and equality of the *sets* of hits in `x` and `y`.

Value

`union` returns a Hits object obtained by appending to `x` the hits in `y` that are not already in `x`.

`intersect` returns a Hits object obtained by keeping only the hits in `x` that are also in `y`.

`setdiff` returns a Hits object obtained by dropping from `x` the hits that are in `y`.

`setequal` returns `TRUE` if `x` and `y` contain the same *sets* of hits and `FALSE` otherwise.

`union`, `intersect`, and `setdiff` propagate the names and metadata columns of their first argument (`x`).

Author(s)

Hervé Pagès and Michael Lawrence

See Also

- Hits objects.
- Hits-comparison for comparing and ordering hits.
- `BiocGenerics::union`, `BiocGenerics::intersect`, and `BiocGenerics::setdiff` in the **BiocGenerics** package for general information about these generic functions.

Examples

```
x <- Hits(c(2, 4, 4, 4, 5, 5), c(3, 1, 3, 2, 3, 2), 6, 3,
          score=11:16)
x

y <- Hits(c(1, 3, 4, 4, 5, 5, 5), c(3, 3, 2, 1, 2, 1, 3), 6, 3,
          score=21:27)
y

union(x, y)
union(y, x)  # same hits as in union(x, y), but in different order

intersect(x, y)
intersect(y, x)  # same hits as in intersect(x, y), but in
                 # different order

setdiff(x, y)
setdiff(y, x)

setequal(x, y)
```

`HitsList-class` *List of Hits objects*

Description

The HitsList class stores a set of Hits objects. It's typically used to represent the result of `findOverlaps` on two RangesList objects.

Details

Roughly the same set of utilities are provided for HitsList as for Hits:

The `as.matrix` method coerces a HitsList object in a similar way to Hits, except a column is prepended that indicates which space (or element in the query RangesList) to which the row corresponds.

The `as.table` method flattens or unlists the list, counts the number of hits for each query range and outputs the counts as a `table`, which has the same shape as from a single Hits object.

To transpose a HitsList object `x`, so that the subject and query in each space are interchanged, call `t(x)`. This allows, for example, counting the number of hits for each subject element using `as.table`.

Accessors

`queryHits(x)`: Equivalent to `unname(as.matrix(x)[,1])`.

`subjectHits(x)`: Equivalent to `unname(as.matrix(x)[,2])`.

`space(x)`: gets the character vector naming the space in the query RangesList for each hit, or `NULL` if the query did not have any names.

Coercion

In the code snippets below, x is a HitsList object.

- `as.matrix(x)`: calls `as.matrix` on each Hits, combines them row-wise and offsets the indices so that they are aligned with the result of calling `unlist` on the query and subject.
- `as.table(x)`: counts the number of hits for each query element in x and outputs the counts as a `table`, which is aligned with the result of calling `unlist` on the query.
- `t(x)`: Interchange the query and subject in each space of x, returns a transposed HitsList object.

Note

This class is highly experimental. It has not been well tested and may disappear at any time.

Author(s)

Michael Lawrence

See Also

- `findOverlaps` in the **IRanges** package, which returns a HitsList object when the query and subject are RangesList objects.

`isSorted` *Test if a vector-like object is sorted*

Description

`isSorted` and `isStrictlySorted` test if a vector-like object is sorted or strictly sorted, respectively.

`isConstant` tests if a vector-like or array-like object is constant. Currently only `isConstant` methods for vectors or arrays of type integer or double are implemented.

Usage

```
isSorted(x)
isStrictlySorted(x)
isConstant(x)
```

Arguments

x	A vector-like object. Can also be an array-like object for `isConstant`.

Details

Vector-like objects of length 0 or 1 are always considered to be sorted, strictly sorted, and constant.

Strictly sorted and constant objects are particular cases of sorted objects.

`isStrictlySorted(x)` is equivalent to `isSorted(x) && !anyDuplicated(x)`

Value

A single logical i.e. `TRUE`, `FALSE` or `NA`.

Author(s)

Hervé Pagès

See Also

- is.unsorted.
- duplicated and unique.
- all.equal.
- NA and is.finite.

Examples

```
## ---------------------------------------------------------------------
## A. isSorted() and isStrictlySorted()
## ---------------------------------------------------------------------

x <- 1:10

isSorted(x)           # TRUE
isSorted(-x)          # FALSE
isSorted(rev(x))      # FALSE
isSorted(-rev(x))     # TRUE

isStrictlySorted(x)   # TRUE

x2 <- rep(x, each=2)
isSorted(x2)          # TRUE
isStrictlySorted(x2)  # FALSE

## ---------------------------------------------------------------------
## B. "isConstant" METHOD FOR integer VECTORS
## ---------------------------------------------------------------------

## On a vector with no NAs:
stopifnot(isConstant(rep(-29L, 10000)))

## On a vector with NAs:
stopifnot(!isConstant(c(0L, NA, -29L)))
stopifnot(is.na(isConstant(c(-29L, -29L, NA))))

## On a vector of length <= 1:
stopifnot(isConstant(NA_integer_))


## ---------------------------------------------------------------------
## C. "isConstant" METHOD FOR numeric VECTORS
## ---------------------------------------------------------------------
## This method does its best to handle rounding errors and special
## values NA, NaN, Inf and -Inf in a way that "makes sense".
## Below we only illustrate handling of rounding errors.

## Here values in 'x' are "conceptually" the same:
x <- c(11/3,
       2/3 + 4/3 + 5/3,
       50 + 11/3 - 50,
```

```
      7.00001 - 1000003/300000)
## However, due to machine rounding errors, they are not *strictly*
## equal:
duplicated(x)
unique(x)
## only *nearly* equal:
all.equal(x, rep(11/3, 4))  # TRUE

## 'isConstant(x)' uses 'all.equal()' internally to decide whether
## the values in 'x' are all the same or not:
stopifnot(isConstant(x))

## This is not perfect though:
isConstant((x - 11/3) * 1e8)  # FALSE on Intel Pentium paltforms
                              # (but this is highly machine dependent!)
```

List-class *List objects*

Description

List objects are Vector objects with a "[[", elementType and elementNROWS method. The List class serves a similar role as list in base R.

It adds one slot, the elementType slot, to the two slots shared by all Vector objects.

The elementType slot is the preferred location for List subclasses to store the type of data represented in the sequence. It is designed to take a character of length 1 representing the class of the sequence elements. While the List class performs no validity checking based on elementType, if a subclass expects elements to be of a given type, that subclass is expected to perform the necessary validity checking. For example, the subclass IntegerList (defined in the **IRanges** package) has elementType = "integer" and its validity method checks if this condition is TRUE.

To be functional, a class that inherits from List must define at least a "[[" method (in addition to the minimum set of Vector methods).

Construction

List objects are typically constructed using one of the 3 following methods:

- Use of a constructor function. Many constructor functions are provided for List objects e.g. List, IntegerList, RleList, RangesList, GRangesList, etc... Which one to use depends on the particular type of List object to construct. The name of a constructor function is always the name of a valid class. If it's the name of a *concrete* class (e.g. the GRangesList constructor defined in the **GenomicRanges** package), then the constructor function returns an instance of that class. If it's the name of a *virtual* class (e.g. the List constructor defined in this package, or the IntegerList or RleList or RangesList constructors defined in the **IRanges** package), then the returned object belongs to a concrete subclass of that virtual class. Which subclass exactly depends on each constructor function (see man page of a particular constructor function for the details).
- Coercion to List or to a List subclass. Many coercion methods are provided to turn any object into a List object. One general and convenient way to convert any vector-like object into a List is to call as(x, "List"). This will typically yield an object from a subclass of CompressedList.

- Use of `extractList`. This function, defined in the **IRanges** package, extracts user-specified groups of elements from a vector-like object and returns them in a List (or sometimes list) object.

Accessors

In the following code snippets, `x` is a List object.

`length(x)`: Get the number of list elements in `x`.

`names(x)`, `names(x) <- value`: Get or set the names of the elements in the List.

`mcols(x, use.names=FALSE)`, `mcols(x) <- value`: Get or set the metadata columns. See Vector man page for more information.

`elementType(x)`: Get the scalar string naming the class from which all elements must derive.

`elementNROWS(x)`: Get the length (or nb of row for a matrix-like object) of each of the elements. Equivalent to `sapply(x, NROW)`.

`isEmpty(x)`: Returns a logical indicating either if the sequence has no elements or if all its elements are empty.

Coercion

To List.

`as(x, "List")`: Converts a vector-like object into a List, usually a CompressedList derivative. One notable exception is when `x` is an ordinary list, in which case `as(x, "List")` returns a SimpleList derivative.

To explicitly request a SimpleList derivative, call `as(x, "SimpleList")`.

See ?`CompressedList` (you might need to load the **IRanges** package first) and ?`SimpleList` for more information about the CompressedList and SimpleList representations.

From List. In the code snippets below, `x` is a List object.

`as.list(x, ...)`, `as(from, "list")`: Turns `x` into an ordinary list.

`unlist(x, recursive=TRUE, use.names=TRUE)`: Concatenates the elements of `x` into a single vector-like object (of class `elementType(x)`).

`as.data.frame(x, row.names=NULL, optional=FALSE ,                    value.name="value", use.ou`
Coerces a `List` to a `data.frame`. The result has the same length as unlisted `x` with two additional columns, `group` and `group_name`. `group` is an `integer` that indicates which list element the record came from. `group_name` holds the list name associated with each record; value is `character` by default and `factor` when `group_name.as.factor` is TRUE.

When `use.outer.mcols` is TRUE the metadata columns on the outer list elements of `x` are replicated out and included in the `data.frame`. List objects that unlist to a single vector (column) are given the column name ‘value‘ by default. A custom name can be provided in `value.name`.

Splitting values in the resulting `data.frame` by the original groups in `x` should be done using the `group` column as the `f` argument to `splitAsList`. To relist data, use `x` as the `skeleton` argument to `relist`.

Subsetting

In the code snippets below, `x` is a List object.

`x[i]`: Return a new List object made of the list elements selected by subscript `i`. Subscript `i` can be of any type supported by subsetting of a Vector object (see Vector man page for the details), plus the following types: IntegerList, LogicalList, CharacterList, integer-RleList, logical-RleList, character-RleList, and RangesList. Those additional types perform subsetting within the list elements rather than across them.

`x[i] <- value`: Replacement version of `x[i]`.

`x[[i]]`: Return the selected list element `i`, where `i` is an numeric or character vector of length 1.

`x[[i]] <- value`: Replacement version of `x[[i]]`.

`x$name`, `x$name <- value`: Similar to `x[[name]]` and `x[[name]] <- value`, but `name` is taken literally as an element name.

relistToClass

`relistToClass(x)` is the opposite of `elementType(y)` in the sense that the former returns the class of the result of relisting (or splitting) `x` while the latter returns the class of the result of unlisting (or unsplitting) `y`. More formally, if `x` is an object that is relistable and `y` a list-like object:

```
relistToClass(x) is class(relist(x, some_skeleton))
elementType(y) is class(unlist(y))
```

As a consequence, for any object `x` for which `relistToClass(x)` is defined and returns a valid class, `elementType(new(relistToClass(x)))` should return `class(x)`.

Author(s)

P. Aboyoun and H. Pagès

See Also

- List-utils for common operations on List objects.
- Vector objects for the parent class.
- The SimpleList class for a direct extension of the List class.
- The CompressedList class defined in the **IRanges** package for another direct extension of the List class.
- The IntegerList, RleList, and IRanges classes and constructors defined in the **IRanges** package for more examples of concrete List subclasses.
- The extractList function defined in the **IRanges** package for grouping elements of a vector-like object into a list-like object.

Examples

```
showClass("List")  # shows (some of) the known subclasses
```

List-utils *Common operations on List objects*

Description

Various functions and methods for looping on List objects, functional programming on List objects, and evaluation of an expression in a List object.

Usage

```
## Looping on List objects:
## ------------------------

## S4 method for signature 'List'
lapply(X, FUN, ...)

## S4 method for signature 'List'
sapply(X, FUN, ..., simplify=TRUE, USE.NAMES=TRUE)

endoapply(X, FUN, ...)

revElements(x, i)

mendoapply(FUN, ..., MoreArgs=NULL)

pc(...)

## Functional programming methods for List objects:
## ------------------------------------------------

## S4 method for signature 'List'
Reduce(f, x, init, right=FALSE, accumulate=FALSE)
## S4 method for signature 'List'
Filter(f, x)
## S4 method for signature 'List'
Find(f, x, right=FALSE, nomatch=NULL)
## S4 method for signature 'List'
Map(f, ...)
## S4 method for signature 'List'
Position(f, x, right=FALSE, nomatch=NA_integer_)

## Evaluation of an expression in a List object:
## ---------------------------------------------

## S4 method for signature 'List'
within(data, expr, ...)

## Constructing list matrices:
## ---------------------------------------------

## S4 method for signature 'List'
```

```
rbind(..., deparse.level=1L)
## S4 method for signature 'List'
cbind(..., deparse.level=1L)
```

Arguments

- `X, x` — A list, data.frame or List object.
- `FUN` — The function to be applied to each element of `X` (for `endoapply`) or for the elements in `...` (for `mendoapply`).
- `...` — For `lapply`, `sapply`, and `endoapply`, optional arguments to `FUN`.

  For `mendoapply`, a set of list, data.frame or List objects to compute over.

  For `pc`, one or more list-like objects.

  For `Map`, one or more List objects. (FIXME: Mixing List objects with ordinary lists doesn't seem to work properly at the moment.)
- `simplify, USE.NAMES` — See `?base::sapply` for a description of these arguments.
- `MoreArgs` — A list of other arguments to `FUN`.
- `i` — Index specifying the elements to replace. Can be anything supported by `` `[<-` ``.
- `f, init, right, accumulate, nomatch` — See `?base::Reduce` for a description of these arguments.
- `data` — A List object.
- `expr` — Expression to evaluate.

Details

Looping on List objects: Like the standard `lapply` function defined in the **base** package, the `lapply` method for List objects returns a list of the same length as `X`, with each element being the result of applying `FUN` to the corresponding element of `X`.

Like the standard `sapply` function defined in the **base** package, the `sapply` method for List objects is a user-friendly version of `lapply` by default returning a vector or matrix if appropriate.

`endoapply` and `mendoapply` perform the endomorphic equivalents of `lapply` and `mapply` by returning objects of the same class as the inputs rather than a list.

`revElements(x, i)` reverses the list elements in `x` specified by `i`. It's equivalent to, but faster than, doing `x[i] <- endoapply(x[i], rev)`.

`pc(...)` combines list-like objects in an element-wise fashion. It's similar to, but faster than, `mapply(c, ..., SIMPLIFY=FALSE)`. With the following differences:

1. `pc()` ignores the supplied objects that are NULL.
2. `pc()` does not recycle its arguments. All the supplied objects must have the same length.
3. If one of the supplied objects is a List object, then `pc()` returns a List object.
4. `pc()` always returns a homogenous list or List object, that is, an object where all the list elements have the same type.

Functional programming methods for List objects: The R base package defines some higher-order functions that are commonly found in Functional Programming Languages. See `?base::Reduce` for the details, and, in particular, for a description of their arguments. The **S4Vectors** package provides methods for List objects, so, in addition to be an ordinary vector or list, the `x` argument can also be a List object.

Evaluation of an expression in a List object: within evaluates expr within as.env(data) via eval(data). Similar to with, except assignments made during evaluation are taken as assignments into data, i.e., new symbols have their value appended to data, and assigning new values to existing symbols results in replacement.

Binding Lists into a matrix: There are methods for cbind and rbind that will bind multiple lists together into a basic list matrix. The usual geometric constraints apply. In the future, this might return a List (+ dimensions), but for now the return value is an ordinary list.

Value

endoapply returns an object of the same class as X, each element of which is the result of applying FUN to the corresponding element of X.

mendoapply returns an object of the same class as the first object specified in ..., each element of which is the result of applying FUN to the corresponding elements of ....

pc returns a list or List object of the same length as the input objects.

See ?base::Reduce for the value returned by the functional programming methods.

See ?base::within for the value returned by within.

cbind and rbind return a list matrix.

Author(s)

P. Aboyoun and H. Pagès

See Also

- The List class.
- base::lapply and base::mapply for the default lapply and mapply methods.
- base::Reduce for the default functional programming methods.
- base::within for the default within method.
- base::cbind and base::rbind for the default matrix binding methods.

Examples

```
a <- data.frame(x = 1:10, y = rnorm(10))
b <- data.frame(x = 1:10, y = rnorm(10))

endoapply(a, function(x) (x - mean(x))/sd(x))
mendoapply(function(e1, e2) (e1 - mean(e1)) * (e2 - mean(e2)), a, b)

x <- list(a=11:13, b=26:21, c=letters)
y <- list(-(5:1), c("foo", "bar"), 0.25)
pc(x, y)

library(IRanges)
x <- IntegerList(a=11:13, b=26:21, c=31:36, d=4:2)
y <- NumericList(-(5:1), 1:2, numeric(0), 0.25)
pc(x, y)

Reduce("+", x)

Filter(is.unsorted, x)
```

```
pos1 <- Position(is.unsorted, x)
stopifnot(identical(Find(is.unsorted, x), x[[pos1]]))

pos2 <- Position(is.unsorted, x, right=TRUE)
stopifnot(identical(Find(is.unsorted, x, right=TRUE), x[[pos2]]))

y <- x * 1000L
Map("c", x, y)

rbind(x, y)
cbind(x, y)
```

Pairs-class *Pairs objects*

Description

Pairs is a Vector that stores two parallel vectors (any object that can be a column in a DataFrame). It provides conveniences for performing binary operations on the vectors, as well as for converting between an equivalent list representation. Virtually all of the typical R vector operations should behave as expected.

A typical use case is representing the pairing from a findOverlaps call, for which findOverlapPairs is a shortcut.

Constructor

Pairs(first, second, ..., names = NULL, hits = NULL): Constructs a Pairs object by aligning the vectors first and second. The vectors must have the same length, unless hits is specified. Arguments in ... are combined as columns in the mcols of the result. The names argument specifies the names on the result. If hits is not NULL, it should be a Hits object that collates the elements in first and second to produce the corresponding pairs.

Accessors

In the code snippets below, x is a Pairs object.

names(x), names(x) <- value: get or set the names

first(x), first(x) <- value: get or set the first member of each pair

second(x), second(x) <- value: get or set the second member of each pair

Coercion

zipup(x): Interleaves the Pairs object x into a list, where each element is composed of a pair. The type of list depends on the type of the elements.

zipdown(x): The inverse of zipup(). Converts x, a list where every element is of length 2, to a Pairs object, by assuming that each element of the list represents a pair.

Subsetting

In the code snippets below, x is a Pairs object.

x[i]: Subset the Pairs object.

Author(s)

Michael Lawrence

See Also

- Hits-class, a typical way to define a pairing.
- `findOverlapPairs` in the **IRanges** package, which generates an instance of this class based on overlaps.
- setops-methods in the **IRanges** package, for set operations on Pairs objects.

Examples

```
p <- Pairs(1:10, Rle(1L, 10), score=rnorm(10), names=letters[1:10])
identical(first(p), 1:10)
mcols(p)$score
p
as.data.frame(p)
z <- zipup(p)
first(p) <- Rle(1:10)
identical(zipdown(z), p)
```

`Rle-class` *Rle objects*

Description

The Rle class is a general container for storing an atomic vector that is stored in a run-length encoding format. It is based on the `rle` function from the base package.

Constructor

`Rle(values, lengths)`: This constructor creates an Rle instance out of an atomic vector or factor object `values` and an integer or numeric vector `lengths` with all positive elements that represent how many times each value is repeated. The length of these two vectors must be the same. `lengths` can be missing in which case `values` is turned into an Rle.

Getters

In the code snippets below, `x` is an Rle object:

`runLength(x)`: Returns the run lengths for `x`.

`runValue(x)`: Returns the run values for `x`.

`nrun(x)`: Returns the number of runs in `x`.

`start(x)`: Returns the starts of the runs for `x`.

`end(x)`: Returns the ends of the runs for `x`.

`width(x)`: Same as `runLength(x)`.

Setters

In the code snippets below, x is an Rle object:

`runLength(x) <- value`: Replaces x with a new Rle object using run values `runValue(x)` and run lengths `value`.

`runValue(x) <- value`: Replaces x with a new Rle object using run values `value` and run lengths `runLength(x)`.

Coercion

From atomic vector to Rle: In the code snippets below, `from` is an atomic vector:

`as(from, "Rle")`: This coercion creates an Rle instances out of an atomic vector `from`.

From Rle to other objects: In the code snippets below, x and `from` are Rle objects:

`as.vector(x, mode="any")`, `as(from, "vector")`: Creates an atomic vector based on the values contained in x. The vector will be coerced to the requested `mode`, unless `mode` is "any", in which case the most appropriate type is chosen.

`as.vectorORfactor(x)`: Creates an atomic vector or factor, based on the type of values contained in x. This is the most general way to decompress the Rle to a native R data structure.

`as.factor(x)`, `as(from, "factor")`: Creates a factor object based on the values contained in x.

`as.data.frame(x)`, `as(from, "data.frame")`: Creates a `data.frame` with a single column holding the result of `as.vector(x)`.

`decode(x)`: Converts an Rle to its native form, such as an atomic vector or factor. Calling `decode` on a non-Rle will return x by default, so it is generally safe for ensuring that an object is native.

General Methods

In the code snippets below, x is an Rle object:

`x[i, drop=getOption("dropRle", default=FALSE)]`: Subsets x by index i, where i can be positive integers, negative integers, a logical vector of the same length as x, an Rle object of the same length as x containing logical values, or an IRanges object. When `drop=FALSE` returns an Rle object. When `drop=TRUE`, returns an atomic vector.

`x[i] <- value`: Replaces elements in x specified by i with corresponding elements in `value`. Supports the same types for i as `x[i]`.

`x %in% table`: Returns a logical Rle representing set membership in `table`.

`append(x, values, after = length(x))`: Insert one Rle into another Rle.

`values` the Rle to insert.

`after` the subscript in x after which the values are to be inserted.

`c(x, ...)`: Combines a set of Rle objects.

`findRun(x, vec)`: Returns an integer vector indicating the run indices in Rle `vec` that are referenced by the indices in the integer vector x.

`head(x, n = 6L)`: If n is non-negative, returns the first n elements of x. If n is negative, returns all but the last `abs(n)` elements of x.

`is.na(x)`: Returns a logical Rle indicating with values are NA.

`is.unsorted(x, na.rm = FALSE, strictly = FALSE)`: Returns a logical value specifying if x is unsorted.

na.rm remove missing values from check.

strictly check for _strictly_ increasing values.

length(x): Returns the underlying vector length of x.

match(x, table, nomatch = NA_integer_, incomparables = NULL): Matches the values in x to table:

table the values to be matched against.

nomatch the value to be returned in the case when no match is found.

incomparables a vector of values that cannot be matched. Any value in x matching a value in this vector is assigned the nomatch value.

rep(x, times, length.out, each), rep.int(x, times): Repeats the values in x through one of the following conventions:

times Vector giving the number of times to repeat each element if of length length(x), or to repeat the whole vector if of length 1.

length.out Non-negative integer. The desired length of the output vector.

each Non-negative integer. Each element of x is repeated each times.

rev(x): Reverses the order of the values in x.

show(object): Prints out the Rle object in a user-friendly way.

order(..., na.last = TRUE, decreasing = FALSE, method = c("shell", "radix")): Returns a permutation which rearranges its first argument into ascending or descending order, breaking ties by further arguments. See order.

sort(x, decreasing = FALSE, na.last = NA): Sorts the values in x.

decreasing If TRUE, sort values in decreasing order. If FALSE, sort values in increasing order.

na.last If TRUE, missing values are placed last. If FALSE, they are placed first. If NA, they are removed.

subset(x, subset): Returns a new Rle object made of the subset using logical vector subset.

table(...): Returns a table containing the counts of the unique values. Supported arguments include useNA with values of 'no' and 'ifany'. Multiple Rle's must be combined with c() before calling table.

tabulate(bin, nbins = max(bin, 1L, na.rm = TRUE)): Just like tabulate, except optimized for Rle.

tail(x, n = 6L): If n is non-negative, returns the last n elements of x. If n is negative, returns all but the first abs(n) elements of x.

unique(x, incomparables = FALSE, ...): Returns the unique run values. The incomparables argument takes a vector of values that cannot be compared with FALSE being a special value that means that all values can be compared.

Set Operations

In the code snippets below, x and y are Rle object or some other vector-like object:

setdiff(x, y): Returns the unique elements in x that are not in y.

union(x, y): Returns the unique elements in either x or y.

intersect(x, y): Returns the unique elements in both x and y.

Author(s)

P. Aboyoun

See Also

Rle-utils, Rle-runstat, and aggregate for more operations on Rle objects.

rle

Vector-class

Examples

```
x <- Rle(10:1, 1:10)
x

runLength(x)
runValue(x)
nrun(x)

diff(x)
unique(x)
sort(x)
x[c(1,3,5,7,9)]
x > 4

x2 <- Rle(LETTERS[c(21:26, 25:26)], 8:1)
table(x2)

y <- Rle(c(TRUE,TRUE,FALSE,FALSE,TRUE,FALSE,TRUE,TRUE,TRUE))
y
as.vector(y)
rep(y, 10)
c(y, x > 5)
```

Rle-runstat *Fixed-width running window summaries*

Description

The runsum, runmean, runmed, runwtsum, runq functions calculate the sum, mean, median, weighted sum, and order statistic for fixed width running windows.

Usage

```
runsum(x, k, endrule = c("drop", "constant"), ...)

runmean(x, k, endrule = c("drop", "constant"), ...)

## S4 method for signature 'Rle'
smoothEnds(y, k = 3)

## S4 method for signature 'Rle'
runmed(x, k, endrule = c("median", "keep", "drop", "constant"),
       algorithm = NULL, print.level = 0)

runwtsum(x, k, wt, endrule = c("drop", "constant"), ...)

runq(x, k, i, endrule = c("drop", "constant"), ...)
```

Arguments

`x`	The data object.
`k`	An integer indicating the fixed width of the running window. Must be odd when `endrule != "drop"`.
`endrule`	A character string indicating how the values at the beginning and the end (of the data) should be treated. `"median"` see `runmed`; `"keep"` see `runmed`; `"drop"` do not extend the running statistics to be the same length as the underlying vectors; `"constant"` copies running statistic to the first values and analogously for the last ones making the smoothed ends *constant*.
`wt`	A numeric vector of length `k` that provides the weights to use.
`i`	An integer in [0, k] indicating which order statistic to calculate.
`...`	Additional arguments passed to methods. Specifically, `na.rm`. When `na.rm = TRUE`, the `NA` and `NaN` values are removed. When `na.rm = FALSE`, `NA` is returned if either `NA` or `NaN` are in the specified window.

Details

The `runsum`, `runmean`, `runmed`, `runwtsum`, and `runq` functions provide efficient methods for calculating the specified numeric summary by performing the looping in compiled code.

Value

An object of the same class as `x`.

Author(s)

P. Aboyoun and V. Obenchain

See Also

`runmed`, Rle-class, RleList-class

Examples

```
x <- Rle(1:10, 1:10)
runsum(x, k = 3)
runsum(x, k = 3, endrule = "constant")
runmean(x, k = 3)
runwtsum(x, k = 3, wt = c(0.25, 0.5, 0.25))
runq(x, k = 5, i = 3, endrule = "constant")

## Missing and non-finite values
x <- Rle(c(1, 2, NA, 0, 3, Inf, 4, NaN))
runsum(x, k = 2)
runsum(x, k = 2, na.rm = TRUE)
runmean(x, k = 2, na.rm = TRUE)
runwtsum(x, k = 2, wt =  c(0.25, 0.5), na.rm = TRUE)
runq(x, k = 2, i = 2, na.rm = TRUE) ## max value in window
```

```
## The .naive_runsum() function demonstrates the semantics of
## runsum(). This test ensures the behavior is consistent with
## base::sum().

.naive_runsum <- function(x, k, na.rm=FALSE)
    sapply(0:(length(x)-k),
        function(offset) sum(x[1:k + offset], na.rm=na.rm))

x0 <- c(1, Inf, 3, 4, 5, NA)
x <- Rle(x0)
target1 <- .naive_runsum(x0, 3, na.rm = TRUE)
target2 <- .naive_runsum(x, 3, na.rm = TRUE)
stopifnot(target1 == target2)
current <- as.vector(runsum(x, 3, na.rm = TRUE))
stopifnot(target1 == current)

## runmean() and runwtsum() :
x <- Rle(c(2, 1, NA, 0, 1, -Inf))
runmean(x, k = 3)
runmean(x, k = 3, na.rm = TRUE)
runwtsum(x, k = 3, wt = c(0.25, 0.50, 0.25))
runwtsum(x, k = 3, wt = c(0.25, 0.50, 0.25), na.rm = TRUE)

## runq() :
runq(x, k = 3, i = 1, na.rm = TRUE) ## smallest value in window
runq(x, k = 3, i = 3, na.rm = TRUE) ## largest value in window

## When na.rm = TRUE, it is possible the number of non-NA
## values in the window will be less than the 'i' specified.
## Here we request the 4th smallest value in the window,
## which tranlates to the value at the 4/5 (0.8) percentile.
x <- Rle(c(1, 2, 3, 4, 5))
runq(x, k=length(x), i=4, na.rm=TRUE)

## The same request on a Rle with two missing values
## finds the value at the 0.8 percentile of the vector
## at the new length of 3 after the NA's have been removed.
## This translates to round((0.8) * 3).
x <- Rle(c(1, 2, 3, NA, NA))
runq(x, k=length(x), i=4, na.rm=TRUE)
```

Rle-utils *Common operations on Rle objects*

Description

Common operations on Rle objects.

Group Generics

Rle objects have support for S4 group generic functionality:

`Arith` "+", "-", "*", "^", "%%", "%/%", "/"

`Compare` "==", ">", "<", "!=", "<=", ">="

`Logic` "&", "|"

`Ops` "Arith", "Compare", "Logic"

`Math` "abs", "sign", "sqrt", "ceiling", "floor", "trunc", "cummax", "cummin", "cumprod", "cumsum", "log", "log10", "log2", "log1p", "acos", "acosh", "asin", "asinh", "atan", "atanh", "exp", "expm1", "cos", "cosh", "sin", "sinh", "tan", "tanh", "gamma", "lgamma", "digamma", "trigamma"

`Math2` "round", "signif"

`Summary` "max", "min", "range", "prod", "sum", "any", "all"

`Complex` "Arg", "Conj", "Im", "Mod", "Re"

See S4groupGeneric for more details.

Summary

In the code snippets below, `x` is an Rle object:

`summary(object, ..., digits = max(3, getOption("digits") - 3))`: Summarizes the Rle object using an atomic vector convention. The `digits` argument is used for number formatting with `signif()`.

Logical Data Methods

In the code snippets below, `x` is an Rle object:

`!x`: Returns logical negation (NOT) of `x`.

`which(x)`: Returns an integer vector representing the `TRUE` indices of `x`.

Numerical Data Methods

In the code snippets below, `x` is an Rle object:

`diff(x, lag = 1, differences = 1`: Returns suitably lagged and iterated differences of `x`.

- `lag` An integer indicating which lag to use.
- `differences` An integer indicating the order of the difference.

`pmax(..., na.rm = FALSE)`, `pmax.int(..., na.rm = FALSE)`: Parallel maxima of the Rle input values. Removes `NA`s when `na.rm = TRUE`.

`pmin(..., na.rm = FALSE)`, `pmin.int(..., na.rm = FALSE)`: Parallel minima of the Rle input values. Removes `NA`s when `na.rm = TRUE`.

`which.max(x)`: Returns the index of the first element matching the maximum value of `x`.

`mean(x, na.rm = FALSE)`: Calculates the mean of `x`. Removes `NA`s when `na.rm = TRUE`.

`var(x, y = NULL, na.rm = FALSE)`: Calculates the variance of `x` or covariance of `x` and `y` if both are supplied. Removes `NA`s when `na.rm = TRUE`.

`cov(x, y, use = "everything")`, `cor(x, y, use = "everything")`: Calculates the covariance and correlation respectively of Rle objects `x` and `y`. The use argument is an optional character string giving a method for computing covariances in the presence of missing values. This must be (an abbreviation of) one of the strings "everything", "all.obs", "complete.obs", "na.or.complete", or "pairwise.complete.obs".

`sd(x, na.rm = FALSE)`: Calculates the standard deviation of `x`. Removes `NA`s when `na.rm = TRUE`.

`median(x, na.rm = FALSE)`: Calculates the median of `x`. Removes `NA`s when `na.rm = TRUE`.

`quantile(x, probs = seq(0, 1, 0.25), na.rm = FALSE, names = TRUE, type = 7, ...)`: Calculates the specified quantiles of `x`.

- `probs` A numeric vector of probabilities with values in [0,1].
- `na.rm` If `TRUE`, removes `NA`s from `x` before the quantiles are computed.
- `names` If `TRUE`, the result has names describing the quantiles.
- `type` An integer between 1 and 9 selecting one of the nine quantile algorithms detailed in `quantile`.
- `...` Further arguments passed to or from other methods.

`mad(x, center = median(x), constant = 1.4826, na.rm = FALSE, low = FALSE, high = FALSE)`: Calculates the median absolute deviation of `x`.

- `center` The center to calculate the deviation from.
- `constant` The scale factor.
- `na.rm` If `TRUE`, removes `NA`s from `x` before the mad is computed.
- `low` If `TRUE`, compute the 'lo-median'.
- `high` If `TRUE`, compute the 'hi-median'.

`IQR(x, na.rm = FALSE)`: Calculates the interquartile range of `x`.

- `na.rm` If `TRUE`, removes `NA`s from `x` before the IQR is computed.

`smoothEnds(y, k = 3)`: Smooth end points of an Rle `y` using subsequently smaller medians and Tukey's end point rule at the very end.

- `k` An integer indicating the width of largest median window; must be odd.

Character Data Methods

In the code snippets below, `x` is an Rle object:

`nchar(x, type = "chars", allowNA = FALSE)`: Returns an integer Rle representing the number of characters in the corresponding values of `x`.

- `type` One of `c("bytes", "chars", "width")`.
- `allowNA` Should `NA` be returned for invalid multibyte strings rather than throwing an error?

`substr(x, start, stop)`, `substring(text, first, last = 1000000L)`: Returns a character or factor Rle containing the specified substrings beginning at `start`/`first` and ending at `stop`/`last`.

`chartr(old, new, x)`: Returns a character or factor Rle containing a translated version of `x`.

- `old` A character string specifying the characters to be translated.
- `new` A character string specifying the translations.

`tolower(x)`: Returns a character or factor Rle containing a lower case version of `x`.

`toupper(x)`: Returns a character or factor Rle containing an upper case version of `x`.

`sub(pattern, replacement, x, ignore.case = FALSE, perl = FALSE, fixed = FALSE, useByt`
Returns a character or factor Rle containing replacements based on matches determined by regular expression matching. See `sub` for a description of the arguments.

`gsub(pattern, replacement, x, ignore.case = FALSE, perl = FALSE, fixed = FALSE, useBy`
Returns a character or factor Rle containing replacements based on matches determined by regular expression matching. See `gsub` for a description of the arguments.

`paste(..., sep = " ", collapse = NULL)`: Returns a character or factor Rle containing a concatenation of the values in `...`.

Factor Data Methods

In the code snippets below, `x` is an Rle object:

`levels(x)`, `levels(x) <- value`: Gets and sets the factor levels, respectively.

`nlevels(x)`: Returns the number of factor levels.

Author(s)

P. Aboyoun

See Also

Rle objects

S4groupGeneric

Examples

```
x <- Rle(10:1, 1:10)
x

sqrt(x)
x^2 + 2 * x + 1
range(x)
sum(x)
mean(x)

z <- c("the", "quick", "red", "fox", "jumps", "over", "the", "lazy", "brown", "dog")
z <- Rle(z, seq_len(length(z)))
chartr("a", "@", z)
toupper(z)
```

`shiftApply-methods` *Apply a function over subsequences of 2 vector-like objects*

Description

`shiftApply` loops and applies a function overs subsequences of vector-like objects `X` and `Y`.

Usage

```
shiftApply(SHIFT, X, Y, FUN, ..., OFFSET=0L, simplify=TRUE, verbose=FALSE)
```

Arguments

`SHIFT`	A non-negative integer vector of shift values.
`X, Y`	The vector-like objects to shift.
`FUN`	The function, found via `match.fun`, to be applied to each set of shifted vectors.
`...`	Further arguments for `FUN`.
`OFFSET`	A non-negative integer offset to maintain throughout the shift operations.
`simplify`	A logical value specifying whether or not the result should be simplified to a vector or matrix if possible.
`verbose`	A logical value specifying whether or not to print the `i` indices to track the iterations.

Details

Let i be the indices in SHIFT, X_i = window(X, 1 + OFFSET, length(X) - SHIFT[i]), and Y_i = window(Y, 1 + SHIFT[i], length(Y) - OFFSET). shiftApply calculates the set of FUN(X_i, Y_i, ...) values and returns the results in a convenient form.

See Also

- The window and aggregate methods for vector-like objects defined in the **S4Vectors** package.
- Vector and Rle objects.

Examples

```
set.seed(0)
lambda <- c(rep(0.001, 4500), seq(0.001, 10, length = 500),
            seq(10, 0.001, length = 500))
xRle <- Rle(rpois(1e7, lambda))
yRle <- Rle(rpois(1e7, lambda[c(251:length(lambda), 1:250)]))

cor(xRle, yRle)
shifts <- seq(235, 265, by=3)
corrs <- shiftApply(shifts, yRle, xRle, FUN=cor)

cor(xRle, yRle)
shiftApply(249:251, yRle, xRle,
           FUN=function(x, y) var(x, y) / (sd(x) * sd(y)))
```

SimpleList-class *SimpleList objects*

Description

The (non-virtual) SimpleList class extends the List virtual class.

Details

The SimpleList class is the simplest, most generic concrete implementation of the List abstraction. It provides an implementation that subclasses can easily extend.

In a SimpleList object the list elements are stored internally in an ordinary list.

Constructor

See the List man page for a quick overview of how to construct List objects in general.

The following constructor is provided for SimpleList objects:

SimpleList(...): Takes possibly named objects as elements for the new SimpleList object.

Accessors

Same as for List objects. See the List man page for more information.

Coercion

All the coercions documented in the List man page apply to SimpleList objects.

Subsetting

Same as for List objects. See the List man page for more information.

Looping and functional programming

Same as for List objects. See ?`List-utils` for more information.

Displaying

When a SimpleList object is displayed, the "Simple" prefix is removed from the real class name of the object. See classNameForDisplay for more information about this.

See Also

- List objects for the parent class.
- The CompressedList class defined in the **IRanges** package for a more efficient alternative to SimpleList.
- The SimpleIntegerList class defined in the **IRanges** package for a SimpleList subclass example.
- The DataFrame class for another SimpleList subclass example.

Examples

```
## Displaying a SimpleList object:
x1 <- SimpleList(a=letters, i=Rle(22:20, 4:2))
class(x1)

## The "Simple" prefix is removed from the real class name of the
## object:
x1

library(IRanges)
x2 <- IntegerList(11:12, integer(0), 3:-2, compress=FALSE)
class(x2)

## The "Simple" prefix is removed from the real class name of the
## object:
x2

## This is controlled by internal helper classNameForDisplay():
classNameForDisplay(x2)
```

---

split-methods *Divide a vector-like object into groups*

Description

`split` divides the data in a vector-like object `x` into the groups defined by `f`.

NOTE: This man page is for the `split` methods defined in the **S4Vectors** package. See `?base::split` for the default method (defined in the **base** package).

Usage

```
## S4 method for signature 'Vector,ANY'
split(x, f, drop=FALSE)

## S4 method for signature 'ANY,Vector'
split(x, f, drop=FALSE)

## S4 method for signature 'Vector,Vector'
split(x, f, drop=FALSE)

## S4 method for signature 'list,Vector'
split(x, f, drop=FALSE, ...)
```

Arguments

x, f	2 vector-like objects of the same length. f will typically be a factor, but not necessarily.
drop	Logical indicating if levels that do not occur should be dropped (if f is a factor).
...	Arguments passed to base::split (see Details below).

Details

The first 3 methods just delegate to the IRanges::splitAsList function defined in the **IRanges** package.

The method for list does:

```
split(x, as.vector(f), drop=drop, ...)
```

Value

All these methods behave like base::split except that the first 3 methods return a List object instead of an ordinary list.

See Also

- The split function in the **base** package.
- The splitAsList function in the **IRanges** package.
- Vector and List objects.
- Rle and DataFrame objects.

Examples

```
## On an Rle object:
x <- Rle(101:105, 6:2)
split(x, c("B", "B", "A", "B", "A"))

## On a DataFrame object:
groups <- c("group1", "group2")
DF <- DataFrame(
    a=letters[1:10],
    i=101:110,
    group=rep(factor(groups, levels=groups), c(3, 7))
```

```
)
split(DF, DF$group)
```

---

str-utils *Some utility functions to operate on strings*

Description

Some low-level string utilities that operate on ordinary character vectors. For more advanced string manipulations, see the **Biostrings** package.

Usage

```
unstrsplit(x, sep="")  # 'sep' default is "" (empty string)

strsplitAsListOfIntegerVectors(x, sep=",")  # 'sep' default is ","
```

Arguments

`x`	For `unstrsplit`: A list-like object where each list element is a character vector, or a character vector (identity). For `strsplitAsListOfIntegerVectors`: A character vector where each element is a string containing comma-separated decimal integer values.
`sep`	A single string containing the separator character. For `strsplitAsListOfIntegerVectors`, the separator must be a single-byte character.

Details

unstrsplit: `unstrsplit(x, sep)` is equivalent to (but much faster than) `sapply(x, paste0, collapse=sep)`. It's performing the reverse transformation of `strsplit( , fixed=TRUE)`, that is, if `x` is a character vector with no NAs and `sep` a single string, then `unstrsplit(strsplit(x, split=sep, fixed=TRUE), sep)` is identical to `x`. A notable exception to this though is when `strsplit` finds a match at the end of a string, in which case the last element of the output (which should normally be an empty string) is not returned (see `?strsplit` for the details).

strsplitAsListOfIntegerVectors: `strsplitAsListOfIntegerVectors` is similar to the `strsplitAsListOfInteger` function shown in the Examples section below, except that the former generally raises an error where the latter would have inserted an `NA` in the returned object. More precisely:

- The latter accepts NAs in the input, the former doesn't (raises an error).
- The latter introduces NAs by coercion (with a warning), the former doesn't (raises an error).
- The latter supports "inaccurate integer conversion in coercion" when the value to coerce is > INT_MAX (then it's coerced to INT_MAX), the former doesn't (raises an error).
- The latter coerces non-integer values (e.g. 10.3) to an int by truncating them, the former doesn't (raises an error).

When it fails, `strsplitAsListOfIntegerVectors` will print an informative error message. Finally, `strsplitAsListOfIntegerVectors` is faster and uses much less memory than `strsplitAsListOfIntegerVect`

Value

`unstrsplit` returns a character vector with one string per list element in `x`.

`strsplitAsListOfIntegerVectors` returns a list where each list element is an integer vector. There is one list element per string in `x`.

Author(s)

Hervé Pagès

See Also

- The strsplit function in the **base** package.

Examples

```
## ---------------------------------------------------------------------
## unstrsplit()
## ---------------------------------------------------------------------
x <- list(A=c("abc", "XY"), B=NULL, C=letters[1:4])
unstrsplit(x)
unstrsplit(x, sep=",")
unstrsplit(x, sep=" => ")

data(islands)
x <- names(islands)
y <- strsplit(x, split=" ", fixed=TRUE)
x2 <- unstrsplit(y, sep=" ")
stopifnot(identical(x, x2))

## But...
names(x) <- x
y <- strsplit(x, split="in", fixed=TRUE)
x2 <- unstrsplit(y, sep="in")
y[x != x2]
## In other words: strsplit() behavior sucks :-/

## ---------------------------------------------------------------------
## strsplitAsListOfIntegerVectors()
## ---------------------------------------------------------------------
x <- c("1116,0,-19",
       " +55291 , 2476,",
       "19184,4269,5659,6470,6721,7469,14601",
       "7778889, 426900, -4833,5659,6470,6721,7096",
       "19184 , -99999")

y <- strsplitAsListOfIntegerVectors(x)
y

## In normal situations (i.e. when the input is well-formed),
## strsplitAsListOfIntegerVectors() does actually the same as the
## function below but is more efficient (both in speed and memory
## footprint):
strsplitAsListOfIntegerVectors2 <- function(x, sep=",")
{
    tmp <- strsplit(x, sep, fixed=TRUE)
    lapply(tmp, as.integer)
}
y2 <- strsplitAsListOfIntegerVectors2(x)
stopifnot(identical(y, y2))
```

subsetting-utils *Subsetting utilities*

Description

Low-level utility functions and classes defined in the **S4Vectors** package to support subsetting of vector-like objects. They are not intended to be used directly.

Vector-class *Vector objects*

Description

The Vector virtual class serves as the heart of the S4Vectors package and has over 90 subclasses. It serves a similar role as vector in base R.

The Vector class supports the storage of *global* and *element-wise* metadata:

1. The *global* metadata annotates the object as a whole: this metadata is accessed via the `metadata` accessor and is represented as an ordinary list;
2. The *element-wise* metadata annotates individual elements of the object: this metadata is accessed via the `mcols` accessor (`mcols` stands for *metadata columns*) and is represented as a DataTable object (i.e. as an instance of a concrete subclass of DataTable, e.g. a DataFrame object), with a row for each element and a column for each metadata variable. Note that the element-wise metadata can also be `NULL`.

To be functional, a class that inherits from Vector must define at least a `length` and a `"["` method.

Accessors

In the following code snippets, `x` is a Vector object.

`length(x)`: Get the number of elements in `x`.

`lengths(x, use.names=TRUE)`: Get the length of each of the elements.
: Note: The `lengths` method for Vector objects is currently defined as an alias for `elementNROWS` (with addition of the `use.names` argument), so is equivalent to `sapply(x, NROW)`, not to `sapply(x, length)`. See `?BiocGenerics::lengths` in the **BiocGenerics** package for more information about this.

`NROW(x)`: Defined as `length(x)` for any Vector object that is *not* a DataTable object. If `x` is a DataTable object, then it's defined as `nrow(x)`.

`names(x)`, `names(x) <- value`: Get or set the names of the elements in the Vector.

`rename(x, value, ...)`: Replace the names of `x` according to a mapping defined by a named character vector, formed by concatenating `value` with any arguments in `...`. The names of the character vector indicate the source names, and the corresponding values the destination names. This also works on a plain old `vector`.

`nlevels(x)`: Returns the number of factor levels.

`mcols(x, use.names=FALSE)`, `mcols(x) <- value`: Get or set the metadata columns. If `use.names=TRUE` and the metadata columns are not `NULL`, then the names of `x` are propagated as the row names of the returned DataTable object. When setting the metadata columns, the supplied value must be `NULL` or a DataTable object holding element-wise metadata.

`elementMetadata(x, use.names=FALSE)`, `elementMetadata(x) <- value`, `values(x, use.names=FALSE)`, `values(x) <- value`: Alternatives to `mcols` functions. Their use is discouraged.

Coercion

`as(from, "data.frame")`, `as.data.frame(from)`: Coerces `from`, a `Vector`, to a `data.frame` by first coercing the `Vector` to a `vector` via `as.vector`. Note that many `Vector` derivatives do not support `as.vector`, so this coercion is possible only for certain types.

`as.env(x)`: Constructs an environment object containing the elements of `mcols(x)`.

Subsetting

In the code snippets below, `x` is a Vector object or regular R vector object. The R vector object methods for `window` are defined in this package and the remaining methods are defined in base R.

`x[i, drop=TRUE]`: If defined, returns a new Vector object made of selected elements `i`, which can be missing; an NA-free logical, numeric, or character vector; or a logical Rle object. The `drop` argument specifies whether or not to coerce the returned sequence to an ordinary vector.

`x[i] <- value`: Replacement version of `x[i]`.

Combining

In the code snippets below, `x` is a Vector object.

`c(x, ...)`: Combine `x` and the Vector objects in `...` together. Any object in `...` must belong to the same class as `x`, or to one of its subclasses, or must be `NULL`. The result is an object of the same class as `x`.

`append(x, values, after = length(x))`: Insert the `Vector` `values` onto `x` at the position given by `after`. `values` must have an `elementType` that extends that of `x`.

Displaying

FOR ADVANCED USERS OR DEVELOPERS Displaying of a Vector object is controlled by 2 internal helpers, `classNameForDisplay` and `showAsCell`.

For most objects `classNameForDisplay(x)` just returns `class(x)`. However, for some objects it can return the name of a parent class that is more suitable for display because it's simpler and as informative as the real class name. See SimpleList objects (defined in this package) and CompressedList objects (defined in the **IRanges** package) for examples of objects for which `classNameForDisplay` returns the name of a parent class.

`showAsCell(x)` produces a character vector *parallel* to `x` (i.e. with one string per vector element in `x`) that contains compact string representations of each elements in `x`.

Note that `classNameForDisplay` and `showAsCell` are generic functions so developers can implement methods to control how their own Vector extension gets displayed.

See Also

- Rle, Hits, IRanges and XRaw for example implementations.
- Vector-comparison for comparing, ordering, and tabulating vector-like objects.
- Vector-setops for set operations on vector-like objects.
- Vector-merge for merging vector-like objects.
- List for a direct Vector extension that serves a similar role as list in base R.
- extractList for grouping elements of a vector-like object into a list-like object.
- DataTable which is the type of objects returned by the `mcols` accessor.
- The Annotated class, which Vector extends.

Examples

```
showClass("Vector")  # shows (some of) the known subclasses
```

Vector-comparison *Compare, order, tabulate vector-like objects*

Description

Generic functions and methods for comparing, ordering, and tabulating vector-like objects.

Usage

```
## Element-wise (aka "parallel") comparison of 2 Vector objects
## ------------------------------------------------------------

pcompare(x, y)

## S4 method for signature 'Vector,Vector'
e1 == e2
## S4 method for signature 'Vector,ANY'
e1 == e2
## S4 method for signature 'ANY,Vector'
e1 == e2

## S4 method for signature 'Vector,Vector'
e1 <= e2
## S4 method for signature 'Vector,ANY'
e1 <= e2
## S4 method for signature 'ANY,Vector'
e1 <= e2

## S4 method for signature 'Vector,Vector'
e1 != e2
## S4 method for signature 'Vector,ANY'
e1 != e2
## S4 method for signature 'ANY,Vector'
e1 != e2

## S4 method for signature 'Vector,Vector'
e1 >= e2
## S4 method for signature 'Vector,ANY'
e1 >= e2
## S4 method for signature 'ANY,Vector'
e1 >= e2

## S4 method for signature 'Vector,Vector'
e1 < e2
## S4 method for signature 'Vector,ANY'
e1 < e2
## S4 method for signature 'ANY,Vector'
```

```
e1 < e2

## S4 method for signature 'Vector,Vector'
e1 > e2
## S4 method for signature 'Vector,ANY'
e1 > e2
## S4 method for signature 'ANY,Vector'
e1 > e2

## selfmatch()
## -----------

selfmatch(x, ...)

## duplicated() & unique()
## ----------------------

## S4 method for signature 'Vector'
duplicated(x, incomparables=FALSE, ...)

## S4 method for signature 'Vector'
unique(x, incomparables=FALSE, ...)

## %in%
## ----

## S4 method for signature 'Vector,Vector'
x %in% table
## S4 method for signature 'Vector,ANY'
x %in% table
## S4 method for signature 'ANY,Vector'
x %in% table

## findMatches() & countMatches()
## -----------------------------

findMatches(x, table, select=c("all", "first", "last"), ...)
countMatches(x, table, ...)

## sort()
## ------

## S4 method for signature 'Vector'
sort(x, decreasing=FALSE, na.last = NA, by)

## table()
## -------

## S4 method for signature 'Vector'
table(...)
```

Arguments

`x, y, e1, e2, table`
: Vector-like objects.

`incomparables`
: The `duplicated` method for Vector objects does NOT support this argument.
  The `unique` method for Vector objects, which is implemented on top of `duplicated`, propagates this argument to its call to `duplicated`.
  See `?base::duplicated` and `?base::unique` for more information about this argument.

`select`
: Only `select="all"` is supported at the moment. Note that you can use `match` if you want to do `select="first"`. Otherwise you're welcome to request this on the Bioconductor mailing list.

`decreasing, na.last`
: See `?base::sort`.

`by`
: A formula referencing the metadata columns by which to sort, e.g., `~ x + y` sorts by column "x", breaking ties with column "y".

`...`
: A Vector object for `table` (the `table` method for Vector objects can only take one input object).
  Otherwise, extra arguments supported by specific methods. In particular:
  - The default `selfmatch` method, which is implemented on top of `match`, propagates the extra arguments to its call to `match`.
  - The `duplicated` method for Vector objects, which is implemented on top of `selfmatch`, accepts extra argument `fromLast` and propagates the other extra arguments to its call to `selfmatch`. See `?base::duplicated` for more information about this argument.
  - The `unique` method for Vector objects, which is implemented on top of `duplicated`, propagates the extra arguments to its call to `duplicated`.
  - The default `findMatches` and `countMatches` methods, which are implemented on top of `match` and `selfmatch`, propagate the extra arguments to their calls to `match` and `selfmatch`.
  - The `sort` method for Vector objects, which is implemented on top of `order`, only accepts extra argument `na.last` and propagates it to its call to `order`.

Details

Doing `pcompare(x, y)` on 2 vector-like objects `x` and `y` of length 1 must return an integer less than, equal to, or greater than zero if the single element in `x` is considered to be respectively less than, equal to, or greater than the single element in `y`. If `x` or `y` have a length != 1, then they are typically expected to have the same length so `pcompare(x, y)` can operate element-wise, that is, in that case it returns an integer vector of the same length as `x` and `y` where the i-th element is the result of compairing `x[i]` and `y[i]`. If `x` and `y` don't have the same length and are not zero-length vectors, then the shortest is first recycled to the length of the longest. If one of them is a zero-length vector then `pcompare(x, y)` returns a zero-length integer vector.

`selfmatch(x, ...)` is equivalent to `match(x, x, ...)`. This is actually how the default method is implemented. However note that `selfmatch(x, ...)` will typically be more efficient than `match(x, x, ...)` on vector-like objects for which a specific `selfmatch` method is implemented.

`findMatches` is an enhanced version of `match` which, by default (i.e. if `select="all"`), returns all the matches in a Hits object.

`countMatches` returns an integer vector of the length of `x` containing the number of matches in `table` for each element in `x`.

Value

For `pcompare`: see Details section above.

For `selfmatch`: an integer vector of the same length as `x`.

For `duplicated`, `unique`, and `%in%`: see `?BiocGenerics::duplicated`, `?BiocGenerics::unique`, and `?`%in%``.

For `findMatches`: a Hits object by default (i.e. if `select="all"`).

For `countMatches`: an integer vector of the length of `x` containing the number of matches in `table` for each element in `x`.

For `sort`: see `?BiocGenerics::sort`.

For `table`: a 1D array of integer values promoted to the `"table"` class. See `?BiocGeneric::table` for more information.

Note

The following notes are for developers who want to implement comparing, ordering, and tabulating methods for their own Vector subclass:

1. The 6 traditional binary comparison operators are: `==`, `!=`, `<=`, `>=`, `<`, and `>`. The **S4Vectors** package provides the following methods for these operators:

```
setMethod("==", c("Vector", "Vector"),
    function(e1, e2) { pcompare(e1, e2) == 0L }
)
setMethod("<=", c("Vector", "Vector"),
    function(e1, e2) { pcompare(e1, e2) <= 0L }
)
setMethod("!=", c("Vector", "Vector"),
    function(e1, e2) { !(e1 == e2) }
)
setMethod(">=", c("Vector", "Vector"),
    function(e1, e2) { e2 <= e1 }
)
setMethod("<", c("Vector", "Vector"),
    function(e1, e2) { !(e2 <= e1) }
)
setMethod(">", c("Vector", "Vector"),
    function(e1, e2) { !(e1 <= e2) }
)
```

   With these definitions, the 6 binary operators work out-of-the-box on Vector objects for which `pcompare` works the expected way. If `pcompare` is not implemented, then it's enough to implement `==` and `<=` methods to have the 4 remaining operators (`!=`, `>=`, `<`, and `>`) work out-of-the-box.

2. The **S4Vectors** package provides no `pcompare` method for Vector objects. Specific `pcompare` methods need to be implemented for specific Vector subclasses (e.g. for Hits and Ranges objects). These specific methods must obey the rules described in the Details section above.

3. The `duplicated`, `unique`, and `%in%` methods for Vector objects are implemented on top of `selfmatch`, `duplicated`, and `match`, respectively, so they work out-of-the-box on Vector objects for which `selfmatch`, `duplicated`, and `match` work the expected way.

4. Also the default `findMatches` and `countMatches` methods are implemented on top of `match` and `selfmatch` so they work out-of-the-box on Vector objects for which those things work the expected way.
5. However, since `selfmatch` itself is also implemented on top of `match`, then having `match` work the expected way is actually enough to get `selfmatch`, `duplicated`, `unique`, `%in%`, `findMatches`, and `countMatches` work out-of-the-box on Vector objects.
6. The `sort` method for Vector objects is implemented on top of `order`, so it works out-of-the-box on Vector objects for which `order` works the expected way.
7. The `table` method for Vector objects is implemented on top of `selfmatch`, `order`, and `as.character`, so it works out-of-the-box on a Vector object for which those things work the expected way.
8. The **S4Vectors** package provides no `match` or `order` methods for Vector objects. Specific methods need to be implemented for specific Vector subclasses (e.g. for Hits and Ranges objects).

Author(s)

Hervé Pagès

See Also

- The Vector class.
- Hits-comparison for comparing and ordering hits.
- Vector-setops for set operations on vector-like objects.
- Vector-merge for merging vector-like objects.
- Ranges-comparison in the **IRanges** package for comparing and ordering ranges.
- `==` and `%in%` in the **base** package, and `BiocGenerics::match`, `BiocGenerics::duplicated`, `BiocGenerics::unique`, `BiocGenerics::order`, `BiocGenerics::sort`, `BiocGenerics::rank` in the **BiocGenerics** package for general information about the comparison/ordering operators and functions.
- The Hits class.
- `BiocGeneric::table` in the **BiocGenerics** package.

Examples

```
## ---------------------------------------------------------------------
## A. SIMPLE EXAMPLES
## ---------------------------------------------------------------------

y <- c(16L, -3L, -2L, 15L, 15L, 0L, 8L, 15L, -2L)
selfmatch(y)

x <- c(unique(y), 999L)
findMatches(x, y)
countMatches(x, y)

## See ?`Ranges-comparison` for more examples (on Ranges objects). You
## might need to load the IRanges package first.

## ---------------------------------------------------------------------
## B. FOR DEVELOPERS: HOW TO IMPLEMENT THE BINARY COMPARISON OPERATORS
```

```
##    FOR YOUR Vector SUBCLASS
## ---------------------------------------------------------------------

## The answer is: don't implement them. Just implement pcompare() and the
## binary comparison operators will work out-of-the-box. Here is an
## example:

## (1) Implement a simple Vector subclass.

setClass("Raw", contains="Vector", representation(data="raw"))

setMethod("length", "Raw", function(x) length(x@data))

setMethod("[", "Raw",
    function(x, i, j, ..., drop) { x@data <- x@data[i]; x }
)

x <- new("Raw", data=charToRaw("AB.x0a-BAA+C"))
stopifnot(identical(length(x), 12L))
stopifnot(identical(x[7:3], new("Raw", data=charToRaw("-a0x."))))

## (2) Implement a "pcompare" method for Raw objects.

setMethod("pcompare", c("Raw", "Raw"),
    function(x, y) {as.integer(x@data) - as.integer(y@data)}
)

stopifnot(identical(which(x == x[1]), c(1L, 9L, 10L)))
stopifnot(identical(x[x < x[5]], new("Raw", data=charToRaw(".-+"))))
```

Vector-merge *Merge vector-like objects*

Description

A merge method for vector-like objects.

Usage

```
## S4 method for signature 'Vector,Vector'
merge(x, y, ..., all=FALSE, all.x=NA, all.y=NA, sort=TRUE)
```

Arguments

x, y, ...	Vector-like objects, typically all of the same class and typically not list-like objects (even though some list-like objects like Ranges and DNAStringSet are supported). Duplicated elements in each object are removed with a warning.
all	TRUE or FALSE. Whether the vector elements in the result should be the union (when all=TRUE) or intersection (when all=FALSE) of the vector elements in x, y, ...
all.x, all.y	To be used only when merging 2 objects (binary merge). Both all.x and all.y must be single logicals. If any of them is NA, then it's set to the value of all.

Setting both of them to TRUE or both of them to FALSE is equivalent to setting all to TRUE or to FALSE, respectively (see above).

If all.x is TRUE and all.y is FALSE then the vector elements in the result will be the unique elements in x. If all.x is FALSE and all.y is TRUE then the vector elements in the result will be the unique elements in y.

sort Whether to sort the merged result.

Details

This merge method acts much like merge.data.frame, except for 3 important differences:

1. The matching is based on the vector values, not arbitrary columns in a table.
2. Self merging is a no-op if sort=FALSE (or object already sorted) and if the object has no duplicates.
3. This merge method accepts an arbitrary number of vector-like objects (n-ary merge).

If some of the objects to merge are list-like objects not supported by the method described here, then the merging is simply done by calling base::merge() on the objects. This might succeed or not...

Value

A vector-like object of the same class as the input objects (if they all have the same class) containing the merged vector values and metadata columns.

See Also

- The Vector class.
- Vector-comparison for comparing and ordering vector-like objects.
- Vector-setops for set operations on vector-like objects.

Examples

```
library(GenomicRanges)
x <- GRanges(c("chr1:1-1000", "chr2:2000-3000"),
             score=c(0.45, 0.1), a1=c(5L, 7L), a2=c(6, 8))
y <- GRanges(c("chr2:150-151", "chr1:1-10", "chr2:2000-3000"),
             score=c(0.7, 0.82, 0.1), b1=c(0L, 5L, 1L), b2=c(1, -2, 1))
merge(x, y)
merge(x, y, all=TRUE)
merge(x, y, all.x=TRUE)
merge(x, y, all.y=TRUE)

## Shared metadata columns must agree:
mcols(x)$score[2] <- 0.11
#merge(x, y)  # error!

## NAs agree with anything:
mcols(x)$score[2] <- NA
merge(x, y)
```

Vector-setops *Set operations on vector-like objects*

Description

Perform set operations on Vector objects.

Usage

```
## S4 method for signature 'Vector,Vector'
union(x, y)

## S4 method for signature 'Vector,Vector'
intersect(x, y)

## S4 method for signature 'Vector,Vector'
setdiff(x, y)

## S4 method for signature 'Vector,Vector'
setequal(x, y)
```

Arguments

`x, y`	Vector-like objects.

Details

The `union`, `intersect`, and `setdiff` methods for Vector objects return a Vector object containing respectively the union, intersection, and (asymmetric!) difference of the 2 sets of vector elements in `x` and `y`. The `setequal` method for Vector objects checks for *set equality* between `x` and `y`.

They're defined as follow:

```
setMethod("union", c("Vector", "Vector"),
    function(x, y) unique(c(x, y))
)
setMethod("intersect", c("Vector", "Vector"),
    function(x, y) unique(x[x %in% y])
)
setMethod("setdiff", c("Vector", "Vector"),
    function(x, y) unique(x[!(x %in% y)])
)
setMethod("setequal", c("Vector", "Vector"),
    function(x, y) all(x %in% y) && all(y %in% x)
)
```

so they work out-of-the-box on Vector objects for which `c`, `unique`, and `%in%` are defined.

Value

`union` returns a Vector object obtained by appending to `x` the elements in `y` that are not already in `x`.

`intersect` returns a Vector object obtained by keeping only the elements in `x` that are also in `y`.

`setdiff` returns a Vector object obtained by dropping from `x` the elements that are in `y`.

`setequal` returns `TRUE` if `x` and `y` contain the same *sets* of vector elements and `FALSE` otherwise.

`union`, `intersect`, and `setdiff` propagate the names and metadata columns of their first argument (`x`).

Author(s)

Hervé Pagès

See Also

- Vector-comparison for comparing and ordering vector-like objects.
- Vector-merge for merging vector-like objects.
- Vector objects.
- `BiocGenerics::union`, `BiocGenerics::intersect`, and `BiocGenerics::setdiff` in the **BiocGenerics** package for general information about these generic functions.

Examples

```
## See ?`Hits-setops` for some examples.
```

---

zip-methods *Convert between parallel vectors and lists*

Description

The `zipup` and `zipdown` functions convert between two parallel vectors and a list of doublets (elements of length 2). The metaphor, borrowed from Python's `zip`, is that of a zipper. The `zipup` function interleaves the elements of the parallel vectors into a list of doublets. The inverse operation is `zipdown`, which returns a `Pairs` object.

Usage

```
zipup(x, y, ...)
zipdown(x, ...)
```

Arguments

`x,y`	For `zipup`, any vector-like object. For `zipdown`, a doublet list.
`...`	Arguments passed to methods.

Value

For `zipup`, a list-like object, where every element is of length 2. For `zipdown`, a `Pairs` object.

See Also

- Pairs objects.

Examples

```
z <- zipup(1:10, Rle(1L, 10))
pairs <- zipdown(z)
```

Index