Package 'S4Vectors'

April 10, 2023

Title Foundation of vector-like and list-like containers in Bioconductor

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, Factor, and Hits) are implemented in the S4Vectors package itself (many more are implemented in the IRanges package and in other Bioconductor infrastructure packages).

biocViews Infrastructure, DataRepresentation

URL https://bioconductor.org/packages/S4Vectors

BugReports https://github.com/Bioconductor/S4Vectors/issues

Version 0.36.2

License Artistic-2.0

Encoding UTF-8

Depends R (>= 4.0.0), methods, utils, stats, stats4, BiocGenerics (>= 0.37.0)

Suggests IRanges, GenomicRanges, SummarizedExperiment, Matrix, DelayedArray, ShortRead, graph, data.table, RUnit, BiocStyle

Collate S4-utils.R show-utils.R utils.R normarg-utils.R bindROWS.R LLint-class.R isSorted.R subsetting-utils.R vector-utils.R integer-utils.R character-utils.R raw-utils.R eval-utils.R map_ranges_to_runs.R RectangularData-class.R Annotated-class.R DataFrame_OR_NULL-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 Factor-class.R List-class.R List-comparison.R splitAsList.R List-utils.R SimpleList-class.R HitsList-class.R DataFrame-class.R DataFrame-combine.R DataFrame-comparison.R DataFrame-utils.R DataFrameFactor-class.R TransposedDataFrame-class.R Pairs-class.R FilterRules-class.R stack-methods.R expand-methods.R aggregate-methods.R shiftApply-methods.R zzz.R

git_url https://git.bioconductor.org/packages/S4Vectors

git_branch RELEASE_3_16

git_last_commit 1671008

git_last_commit_date 2023-02-25

Date/Publication 2023-04-10

Author Hervé Pagès [aut, cre],
Michael Lawrence [aut],
Patrick Aboyoun [aut],
Aaron Lun [ctb]

Maintainer Hervé Pagès <hpages.on.github@gmail.com>

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 IntegerRangesList object to aggregate within the list elements rather than across them. When `by` is a IntegerRangesList 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 code snippet below, 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")
```

---

bindROWS *Combine objects along their ROWS or COLS*

Description

`bindROWS` and `bindCOLS` are low-level generic functions defined in the **S4Vectors** package for binding objects along their 1st or 2nd dimension. They are the workhorses behind higher-level operations like `c()`, `rbind()`, or `cbind()` on most vector-like or rectangular objects defined in Bioconductor.

They are not intended to be used directly by the end user.

Usage

```
bindROWS(x, objects=list(), use.names=TRUE, ignore.mcols=FALSE, check=TRUE)
bindCOLS(x, objects=list(), use.names=TRUE, ignore.mcols=FALSE, check=TRUE)
```

Arguments

x	An S4 object.
objects	A list of S4 objects to bind to x. They should typically (but not necessarily) have the same class as x.
use.names	Should the names on the input objects be propagated? By default they are.
ignore.mcols	Should the metadata columns on the input objects be ignored? By defaut they are not (i.e. they are propagated).
check	Should the result object be validated before being returned to the user? By default it is.

Value

An object of the same class as x.

Author(s)

Hervé Pagès

See Also

- The NROW and NCOL generic functions defined in the **BiocGenerics** package.

---

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

Description

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

Usage

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

## more to come...
```

Arguments

x	A list-like object where each list element is a character vector, or a character vector (identity).
sep	A single string containing the separator.

Details

unstrsplit(x, sep) is equivalent to (but much faster than) sapply(x, paste0, collapse=sep). It performs 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).

Value

A character vector with one string per list element in x.

Author(s)

Hervé Pagès

See Also

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

Examples

```
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 :-/
```

DataFrame-class *DataFrame objects*

Description

The DataFrame class extends the RectangularData virtual class 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 exceptions have to do with handling of the row names:

1. 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).
2. The row names are not required to be unique.
3. Subsetting by row names does not use partial matching.

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, stringsAsFactors)

Constructs a DataFrame in similar fashion to data.frame. Each argument in ... is coerced to a DataFrame and combined column-wise. The row names should be given in row.names; otherwise, they are inherited from the arguments, as in data.frame. Explicitly passing NULL to row.names ensures that there are no rownames. 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 [.

The stringsAsFactors argument is ignored. The coercion of column arguments to DataFrame determines whether strings become factors.

make_zero_col_DFrame(nrow)

Constructs a zero-column DFrame object with nrow rows. Intended for developers to use in other packages and typically not needed by the end user.

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.

`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.

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.

Author(s)

Michael Lawrence

See Also

- DataFrame-combine for combining DataFrame objects.
- DataFrame-utils for other common operations on DataFrame objects.
- TransposedDataFrame objects.
- RectangularData 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",] # no partial match (unlike with data.frame)

## 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
```

DataFrame-combine *Combine DataFrame objects along their rows or columns, or merge them*

Description

Various methods are provided to combine DataFrame objects along their rows or columns, or to merge them.

Details

In the code snippets below, all the input objects are expected to be DataFrame objects.

`rbind(...)`: Creates a new DataFrame object by aggregating the rows of the input objects. 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. The returned DataFrame object inherits its metadata and metadata columns from the first input object.

`cbind(...)`: Creates a new DataFrame object by aggregating the columns of the input objects. Very similar to `cbind.data.frame()`. The returned DataFrame object inherits its metadata from the first input object. The metadata columns of the returned DataFrame object are obtained by combining the metadata columns of the input object with `combineRows()`.

`combineRows(x, ...)`: `combineRows()` is a generic function documented in the man page for RectangularData objects (see `?RectangularData`). The method for DataFrame objects behaves as documented in that man page.

`combineCols(x, ..., use.names=TRUE)`: `combineCols()` is a generic function documented in the man page for RectangularData objects (see `?RectangularData`). The method for DataFrame objects behaves as documented in that man page.

`combineUniqueCols(x, ..., use.names=TRUE)`: This function is documented in the man page for RectangularData objects (see `?RectangularData`).

`merge(x, y, ...)`: Merges two DataFrame 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`.

Author(s)

Michael Lawrence, Hervé Pagès, and Aaron Lun

See Also

- DataFrame-utils for other common operations on DataFrame objects.
- DataFrame objects.
- TransposedDataFrame objects.
- RectangularData objects.
- `cbind` and `merge` in the **base** package.

Examples

```
## ---------------------------------------------------------------------
## rbind()
## ---------------------------------------------------------------------

x1 <- DataFrame(A=1:5, B=letters[1:5], C=11:15)
y1 <- DataFrame(B=c(FALSE, NA, TRUE), C=c(FALSE, NA, TRUE), A=101:103)
rbind(x1, y1)

x2 <- DataFrame(A=Rle(101:103, 3:1), B=Rle(51:52, c(1, 5)))
y2 <- DataFrame(A=runif(2), B=Rle(c("a", "b")))
rbind(x2, y2)

## ---------------------------------------------------------------------
## combineRows()
## ---------------------------------------------------------------------

y3 <- DataFrame(A=runif(2))
combineRows(x2, y3)

y4 <- DataFrame(B=Rle(c("a", "b")), C=runif(2))
combineRows(x2, y4)
combineRows(y4, x2)
combineRows(y4, x2, DataFrame(D=letters[1:3], B=301:303))

## ---------------------------------------------------------------------
## combineCols()
## ---------------------------------------------------------------------

X <- DataFrame(x=1)
Y <- DataFrame(y="A")
Z <- DataFrame(z=TRUE)

combineCols(X, Y, Z, use.names=FALSE)

Y <- DataFrame(y=LETTERS[1:2])
rownames(X) <- "foo"
rownames(Y) <- c("foo", "bar")
rownames(Z) <- "bar"

combineCols(X, Y, Z)
```

```
## ------------------------------------------------------------------------
## combineUniqueCols()
## ------------------------------------------------------------------------

X <- DataFrame(x=1)
Y <- DataFrame(y=LETTERS[1:2], dup=1:2)
Z <- DataFrame(z=TRUE, dup=2L)

rownames(X) <- "foo"
rownames(Y) <- c("foo", "bar")
rownames(Z) <- "bar"

combineUniqueCols(X, Y, Z)

Z$dup <- 3
combineUniqueCols(X, Y, Z)

## ------------------------------------------------------------------------
## merge()
## ------------------------------------------------------------------------

x6 <- DataFrame(key=c(155, 2, 33, 17, 2, 26, 1), aa=1:7)
y6 <- DataFrame(key=1:26, bb=LETTERS)
merge(x6, y6, by="key")
merge(x6, y6, by="key", all.x=TRUE)
```

DataFrame-comparison *DataFrame comparison methods*

Description

The DataFrame class provides methods to compare across rows of the DataFrame, including ordering and matching. Each DataFrame is effectively treated as a vector of rows.

Usage

```
## S4 method for signature 'DataFrame'
sameAsPreviousROW(x)

## S4 method for signature 'DataFrame,DataFrame'
match(x, table, nomatch = NA_integer_, incomparables = NULL, ...)

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

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

```
## S4 method for signature 'DataFrame,DataFrame'
e1 == e2

## S4 method for signature 'DataFrame,DataFrame'
e1 <= e2
```

Arguments

`x, table, y, e1, e2`	A DataFrame object.
`nomatch, incomparables`	See ?base::match.
`...`	For match, further arguments to pass to match. For order, one or more DataFrame objects.
`decreasing, na.last, method`	See ?base::order.

Details

The treatment of a DataFrame as a "vector of rows" is useful in many cases, e.g., when each row is a record that needs to be ordered or matched. The methods provided here allow the use of all methods described in ?Vector-comparison, including sorting, matching, de-duplication, and so on.

Careful readers will notice this behaviour differs from the usual semantics of a data.frame, which acts as a list-like vector of columns. This discrepancy rarely causes problems, as it is not particularly common to compare columns of a data.frame in the first place.

Note that a match method for DataFrame objects is explicitly defined to avoid calling the corresponding method for List objects, which would yield the (undesired) list-like semantics. The same rationale is behind the explicit definition of <= and == despite the availability of pcompare.

Value

For sameAsPreviousROW: see sameAsPreviousROW.

For match: see match.

For order: see order.

For pcompare, == and <=: see pcompare.

Author(s)

Aaron Lun

Examples

```
# Mocking up a DataFrame.
DF <- DataFrame(
    A=sample(LETTERS, 100, replace=TRUE),
    B=sample(5, 100, replace=TRUE)
)
```

```
# Matching:
match(DF, DF[1:10,])
selfmatch(DF)
unique(DF)

# Ordering, alone and with other vectors:
sort(DF)
order(DF, runif(nrow(DF)))

# Parallel comparison:
DF==DF
DF==DF[1,]
```

DataFrame-utils *Common operations on DataFrame objects*

Description

Common operations on DataFrame objects.

Splitting

In the code snippet below, x is a DataFrame object.

`split(x, f, drop = FALSE)`: Splits x into a SplitDataFrameList object, according to f, dropping elements corresponding to unrepresented levels if drop is TRUE.

Looping

In the code snippet below, x is a DataFrame object.

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

Subsetting based on NA content

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

`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.

Transforming

In the code snippet below, x is a DataFrame object.

`transform(`_data`, ...)`: adds or replaces columns based on expressions in `...`. See `transform`.

Statistical modeling with DataFrame

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

Tabulation:

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

Author(s)

Michael Lawrence

See Also

- `by` in the **base** package.
- `na.omit` in the **stats** package.
- `transform` in the **base** package.
- `xtabs` in the **stats** package.
- `splitAsList` in this package (**S4Vectors**).
- SplitDataFrameList objects in the **IRanges** package.
- DataFrame objects.

Examples

```
## split
sw <- DataFrame(swiss)
swsplit <- split(sw, sw[["Education"]])

## rbind & cbind
do.call(rbind, as.list(swsplit))
cbind(DataFrame(score), DataFrame(counts))

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

DataFrameFactor-class *DataFrameFactor objects*

Description

The DataFrameFactor class is a subclass of the Factor class where the levels are the rows of a DataFrame. It provides a few methods to mimic the behavior of an actual DataFrame while retaining the memory efficiency of the Factor structure.

Usage

```
DataFrameFactor(x, levels, index=NULL, ...)  # constructor function
```

Arguments

x, levels	DataFrame objects. At least one of x and levels must be specified. If index is NULL, both can be specified. When levels is specified, it must be a DataFrame with no duplicate rows (i.e. anyDuplicated(levels) must return 0). See ?Factor for more details.
index	NULL or an integer (or numeric) vector of valid positive indices (no NAs) into levels. See ?Factor for details.
...	Optional metadata columns.

Value

A DataFrameFactor object.

Accessors

DataFrameFactor objects support the same set of accessors as Factor objects. In addition, it mimics some aspects of the DataFrame interface. The general principle is that, for these methods, a DataFrameFactor x behaves like the expanded DataFrame unfactor(x).

- x$name will return column name from levels(x) and expand it according to the indices in x.
- x[i, j, ..., drop=TRUE] will return a new DataFrameFactor subsetted to entries i, where the levels are subsetted by column to contain only columns j. If the resulting levels only have one column and drop=TRUE, the expanded values of the column are returned directly.
- dim(x) will return the length of the DataFrameFactor and the number of columns in its levels.
- dimnames(x) will return the names of the DataFrameFactor and the column names in its levels.

Caution

The DataFrame-like methods implemented here are for convenience only. Users should not assume that the DataFrameFactor complies with other aspects of the DataFrame interface, due to fundamental differences between a DataFrame and the Factor parent class, e.g., in the interpretation of their "length". Outside of the methods listed above, the DataFrameFactor is not guaranteed to work as a drop-in replacement for a DataFrame - use unfactor(x) instead.

Author(s)

Aaron Lun

See Also

Factor objects for the parent class.

Examples

```
df <- DataFrame(X=sample(5, 100, replace=TRUE), Y=sample(c("A", "B"), 100, replace=TRUE))
dffac <- DataFrameFactor(df)
dffac

dffac$X
dffac[,c("Y", "X")]
dffac[1:10,"X"]
colnames(dffac)

# The usual Factor methods may also be used:
unfactor(dffac)
levels(dffac)
as.integer(dffac)
```

---

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, recursive = TRUE)
```

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.
recursive	If TRUE, expand each column recursively, with the result representing their cartesian product. If FALSE, expand all of the columns in parallel, which requires that they all share the same skeleton.

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)

## Expand the aa and dd column in parallel:
df$dd <- relist(seq_along(unlist(aa)), aa)
expand(df, colnames=c("aa","dd"), recursive=FALSE)
```

Factor-class *Factor objects*

Description

The Factor class serves a similar role as factor in base R (a.k.a. ordinary factor) except that the levels of a Factor object can be *any vector-like object*, that is, they can be an ordinary vector or a Vector derivative, or even an ordinary factor or another Factor object.

A notable difference with ordinary factors is that Factor objects cannot contain NAs, at least for now.

Usage

```
Factor(x, levels, index=NULL, ...)  # constructor function
```

Arguments

x, levels	At least one of x and levels must be specified. If index is NULL, both can be specified. When levels is specified, it must be a *vector-like object* (see above) with no duplicates (i.e. anyDuplicated(levels) must return 0). When x and levels are both specified, they should typically have the same class (or, at least, match(x, levels) must work on them), and all the elements in x

must be represented in `levels` (i.e. the integer vector returned by `match(x, levels)` should contain no NAs). See Details section below.

`index` — `NULL` or an integer (or numeric) vector of valid positive indices (no NAs) into `levels`. See Details section below.

`...` — Optional metadata columns.

Details

There are 4 different ways to use the `Factor()` constructor function:

1. `Factor(x, levels)` (i.e. `index` is missing): In this case `match(x, levels)` is used internally to encode `x` as a Factor object. An error is returned if some elements in `x` cannot be matched to `levels` so it's important to make sure that all the elements in `x` are represented in `levels` when doing `Factor(x, levels)`.
2. `Factor(x)` (i.e. `levels` and `index` are missing): This is equivalent to `Factor(x, levels=unique(x))`.
3. `Factor(levels=levels, index=index)` (i.e. `x` is missing): In this case the encoding of the Factor object is supplied via `index`, that is, `index` must be an integer (or numeric) vector of valid positive indices (no NAs) into `levels`. This is the most efficient way to construct a Factor object.
4. `Factor(levels=levels)` (i.e. `x` and `index` are missing): This is a convenient way to construct a 0-length Factor object with the specified levels. In other words, it's equivalent to `Factor(levels=levels, index=integer(0))`.

Value

A Factor object.

Accessors

Factor objects support the same set of accessors as ordinary factors. That is:

- `length(x)` to get the length of Factor object `x`.
- `names(x)` and `names(x) <- value` to get and set the names of Factor object `x`.
- `levels(x)` and `levels(x) <- value` to get and set the levels of Factor object `x`.
- `nlevels(x)` to get the number of levels of Factor object `x`.
- `as.integer(x)` to get the encoding of Factor object `x`. Note that `length(as.integer(x))` and `names(as.integer(x))` are the same as `length(x)` and `names(x)`, respectively.

In addition, because Factor objects are Vector derivatives, they support the `mcols()` and `metadata()` getters and setters.

Decoding a Factor

`unfactor(x)` can be used to *decode* Factor object `x`. It returns an object of the same class as `levels(x)` and same length as `x`. Note that it is the analog of `as.character()` on ordinary factors, with the notable difference that `unfactor(x)` propagates the names on `x`.

For convenience, `unfactor(x)` also works on ordinary factor `x`.

`unfactor()` supports extra arguments `use.names` and `ignore.mcols` to control whether the names and metadata columns on the Factor object to decode should be propagated or not. By default they are propagated, that is, the default values for `use.names` and `ignore.mcols` are `TRUE` and `FALSE`, respectively.

Coercion

From vector or Vector to Factor: coercion of a vector-like object `x` to Factor is supported via `as(x, "Factor")` and is equivalent to `Factor(x)`. There are 2 IMPORTANT EXCEPTIONS to this:

1. If `x` is an ordinary factor, `as(x, "Factor")` returns a Factor with the same levels, encoding, and names, as `x`. Note that after coercing an ordinary factor to Factor, going back to factor again (with `as.factor()`) restores the original object with no loss.
2. If `x` is a Factor object, `as(x, "Factor")` is either a no-op (when `x` is a Factor *instance*), or a demotion to Factor (when `x` is a Factor derivative like GRangesFactor).

From Factor to integer: `as.integer(x)` is supported on Factor object `x` and returns its encoding (see Accessors section above).

From Factor to factor: `as.factor(x)` is supported on Factor object `x` and returns an ordinary factor where the levels are `as.character(levels(x))`.

From Factor to character: `as.character(x)` is supported on Factor object `x` and is equivalent to `unfactor(as.factor(x))`, which is also equivalent to `as.character(unfactor(x))`.

Subsetting

A Factor object can be subsetted with `[`, like an ordinary factor.

Concatenation

2 or more Factor objects can be concatenated with `c()`. Note that, unlike with ordinary factors, `c()` on Factor objects preserves the class i.e. it returns a Factor object. In other words, `c()` acts as an *endomorphism* on Factor objects.

The levels of `c(x, y)` are obtained by appending to `levels(x)` the levels in `levels(y)` that are "new" i.e. that are not already in `levels(x)`.

`append()`, which is implemented on top of `c()`, also works on Factor objects.

Comparing & Ordering

Factor objects support comparing (e.g. `==`, `!=`, `<=`, `<`, `match()`) and ordering (e.g. `order()`, `sort()`, `rank()`) operations. All these operations behave like they would on the *unfactored* versions of their operands.

For example `F1 <= F2`, `match(F1, F2)`, and `sort(F1)`, are equivalent to `unfactor(F1) <= unfactor(F2)`, `match(unfactor(F1), unfactor(F2))`, and `sort(unfactor(F1))`, respectively.

Author(s)

Hervé Pagès, with contributions from Aaron Lun

See Also

- factor in base R.
- GRangesFactor objects in the **GenomicRanges** package.
- IRanges objects in the **IRanges** package.
- Vector objects for the parent class.
- anyDuplicated in the **BiocGenerics** package.

Examples

```
showClass("Factor")  # Factor extends Vector

## ---------------------------------------------------------------------
## CONSTRUCTOR & ACCESSORS
## ---------------------------------------------------------------------
library(IRanges)
set.seed(123)
ir0 <- IRanges(sample(5, 8, replace=TRUE), width=10,
               names=letters[1:8], ID=paste0("ID", 1:8))

## Use explicit levels:
ir1 <- IRanges(1:6, width=10)
F1 <- Factor(ir0, levels=ir1)
F1
length(F1)
names(F1)
levels(F1)  # ir1
nlevels(F1)
as.integer(F1)  # encoding

## If we don't specify the levels, they'll be set to unique(ir0):
F2 <- Factor(ir0)
F2
length(F2)
names(F2)
levels(F2)  # unique(ir0)
nlevels(F2)
as.integer(F2)

## ---------------------------------------------------------------------
## DECODING
## ---------------------------------------------------------------------
unfactor(F1)

stopifnot(identical(ir0, unfactor(F1)))
stopifnot(identical(ir0, unfactor(F2)))

unfactor(F1, use.names=FALSE)
unfactor(F1, ignore.mcols=TRUE)

## ---------------------------------------------------------------------
```

```
## COERCION
## ------------------------------------------------------------------------
F2b <- as(ir0, "Factor")  # same as Factor(ir0)
stopifnot(identical(F2, F2b))

as.factor(F2)
as.factor(F1)

as.character(F1)  # same as unfactor(as.factor(F1)),
                  # and also same as as.character(unfactor(F1))

## On an ordinary factor 'f', 'as(f, "Factor")' and 'Factor(f)' are
## NOT the same:
f <- factor(sample(letters, 500, replace=TRUE), levels=letters)
as(f, "Factor")  # same levels as 'f'
Factor(f)        # levels **are** 'f'!

stopifnot(identical(f, as.factor(as(f, "Factor"))))

## ------------------------------------------------------------------------
## CONCATENATION
## ------------------------------------------------------------------------
ir3 <- IRanges(c(5, 2, 8:6), width=10)
F3 <- Factor(levels=ir3, index=2:4)
F13 <- c(F1, F3)
F13
levels(F13)

stopifnot(identical(c(unfactor(F1), unfactor(F3)), unfactor(F13)))

## ------------------------------------------------------------------------
## COMPARING & ORDERING
## ------------------------------------------------------------------------
F1 == F2   # same as unfactor(F1) == unfactor(F2)

order(F1)  # same as order(unfactor(F1))
order(F2)  # same as order(unfactor(F2))

## The levels of the Factor influence the order of the table:
table(F1)
table(F2)
```

`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`.

Concatenation

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

`x & y`: Appends the rules in `y` to the rules in `x`.

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

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

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(envir)) parent.frame() else 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`.

`as(x, "DataFrame")`: Creates a DataFrame by combining the result of `as.matrix(x)` with `mcols(x)`.

`as.data.frame(x)`: Attempts to coerce the result of `as(x, "DataFrame")` to a `data.frame`.

Subsetting

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

`x[i]`: Return a new Hits object made of the elements selected by `i`.

`x[i, j]`: Like the above, but allow the user to conveniently subset the metadata columns thru `j`.

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

See `?`[`` in this package (the **S4Vectors** package) for more information about subsetting Vector derivatives and for an important note about the `x[i, j]` form.

Concatenation

`c(x, ..., ignore.mcols=FALSE)`: Concatenate Hits object x and the Hits objects in `...` together. See `?c` in this package (the **S4Vectors** package) for more information about concatenating Vector derivatives.

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, new.nRnode=NA)`: 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"), rank)`: 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. If `rank` is not missing, it should be a formula specifying an alternative ranking according to its terms (see `rank`).

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="first", nodup=TRUE)

selectHits(hits, select="last")
selectHits(hits, select="last", nodup=TRUE)

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("auto", "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 IntegerRangesList 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 IntegerRangesList) 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 IntegerRangesList 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 IntegerRangesList objects.

Examples

```
hits <- Hits(rep(1:20, each=5), 100:1, 20, 100)
hlist <- splitAsList(hits, 1:5)
hlist
hlist[[1]]
hlist[[2]]

## Some sanity checks:

hits1 <- Hits(c(4, 4, 15, 15), c(1, 2, 3, 4), 20, 4)
hits2 <- Hits(c(4, 4, 15, 15), c(1, 2, 3, 4), 20, 4, sort.by.query=TRUE)

fA <- c(1, 1, 2, 2)
hlist1A <- split(hits1, fA)
hlist2A <- split(hits2, fA)
stopifnot(identical(as(hlist1A, "SortedByQueryHitsList"), hlist2A))
stopifnot(identical(hlist1A, as(hlist2A, "HitsList")))

fB <- c(1, 2, 1, 2)
hlist1B <- split(hits1, fB)
hlist2B <- split(hits2, fB)
stopifnot(identical(as(hlist1B, "SortedByQueryHitsList"), hlist2B))
stopifnot(identical(hlist1B, as(hlist2B, "HitsList")))
```

---

integer-utils *Some utility functions to operate on integer vectors*

Description

Some low-level utility functions to operate on ordinary integer vectors.

Usage

```
isSequence(x, of.length=length(x))

toListOfIntegerVectors(x, sep=",")

## more to come...
```

Arguments

x	For `isSequence()`: An integer vector. For `toListOfIntegerVectors()`: A character vector where each element is a string containing comma-separated integers in decimal representation. Alternatively `x` can be a list of raw vectors, in which case it's treated like if it was `sapply(x, rawToChar)`.
of.length	The expected length of the integer sequence.
sep	The separator represented as a single-letter string.

Details

`isSequence()` returns `TRUE` or `FALSE` depending on whether `x` is identical to `seq_len(of.length)` or not.

`toListOfIntegerVectors()` is a fast and memory-efficient implementation of

```
lapply(strsplit(x, sep, fixed=TRUE), as.integer)
```

but, unlike the above code, it will raise an error if the input contains NAs or strings that don't represent integer values.

Value

A list *parallel* to `x` where each list element is an integer vector.

Author(s)

Hervé Pagès

See Also

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

Examples

```
## ---------------------------------------------------------------------
## isSequence()
## ---------------------------------------------------------------------
isSequence(1:5)              # TRUE
isSequence(5:1)              # FALSE
```

```
isSequence(0:5)               # FALSE
isSequence(integer(0))        # TRUE
isSequence(1:5, of.length=5)  # TRUE (the expected length)
isSequence(1:5, of.length=6)  # FALSE (not the expected length)

## ---------------------------------------------------------------------
## toListOfIntegerVectors()
## ---------------------------------------------------------------------

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

y <- toListOfIntegerVectors(x)
y

## When it doesn't choke on an NA or string that doesn't represent
## an integer value, toListOfIntegerVectors() is equivalent to
## the function below but is faster and more memory-efficient:
toListOfIntegerVectors2 <- function(x, sep=",")
{
    lapply(strsplit(x, sep, fixed=TRUE), as.integer)
}
y2 <- toListOfIntegerVectors2(x)
stopifnot(identical(y, y2))
```

`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 and derivatives are typically constructed using one of the following methods:

Use of a constructor function: Many constructor functions are provided in **S4Vectors** and other Bioconductor packages for List objects and derivatives e.g. `List()`, `IntegerList()`, `RleList()`, `IntegerRangesList()`, `GRangesList()`, etc...

Which one to use depends on the particular type of List derivative one wishes to construct e.g. use `IntegerList()` to get an IntegerList object, `RleList()` to get an RleList object, etc...

Note that the name of a constructor function is always the name of a valid class. See the man page of a particular constructor function for the details.

Coercion to List or to a List subclass: Many coercion methods are defined in **S4Vectors** and other Bioconductor packages to turn all kinds of objects into List objects.

One general and convenient way to convert any vector-like object `x` into a List is to call `as(x, "List")`. This will yield an object from a subclass of List. Note that this subclass will typically extend CompressedList but not necessarily (see `?CompressedList` in the **IRanges** package for more information about CompressedList objects).

However, if a specific type of List derivative is desired (e.g. CompressedGRangesList), then coercing explicitly to that class is preferrable as it is more robust and more readable.

Use of `splitAsList()`, `relist()`, or `extractList()`: `splitAsList()` behaves like `base::split()` except that it returns a List derivative instead of an ordinary list. See `?splitAsList` for more information.

The `relist()` methods for List objects and derivatives, as well as the `extractList()` function, are defined in the **IRanges** package. They provide very efficient ways to construct a List derivative from the vector-like object passed to their first argument (`flesh` for `relist()` and `x` for `extractList()`). See `?extractList` in the **IRanges** package for more information.

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.outer.mcols=FALSE, group_name.as.factor=FALSE, ...)`: 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 IntegerRangesList. 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.

Author(s)

P. Aboyoun and H. Pagès

See Also

- splitAsList for splitting a vector-like object into a List object.
- relist and extractList in the **IRanges** package for efficiently constructing a List derivative from a vector-like object.
- 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 some examples of List derivatives.

Examples

```
showClass("List")  # shows only the known subclasses define in this package

## ---------------------------------------------------------------------
## A. CONSTRUCTION
## ---------------------------------------------------------------------
x <- sample(500, 20)
y0 <- splitAsList(x, x %% 4)
y0

levels <- paste0("G", 1:10)
f1 <- factor(sample(levels, length(x), replace=TRUE), levels=levels)
y1 <- splitAsList(x, f1)
y1

f2 <- factor(sample(levels, 26, replace=TRUE), levels=levels)
y2 <- splitAsList(letters, f2)
y2

library(IRanges)  # for the NumericList() constructor and the
                  # coercion to CompressedCharacterList

NumericList(A=runif(10), B=NULL, C=runif(3))

## Another way to obtain 'splitAsList(letters, f2)' but using
## 'splitAsList()' should be preferred as it is a lot more efficient:
y2b <- as(split(letters, f2), "CompressedCharacterList")  # inefficient!
stopifnot(identical(y2, y2b))

## ---------------------------------------------------------------------
## B. SUBSETTING
## ---------------------------------------------------------------------
## Single-bracket and double-bracket subsetting behave like on ordinary
## lists:
y1[c(10, 1, 2, 2)]
y1[c(-10, -1, -2)]
```

```
y1[c(TRUE, FALSE)]
y1[c("G8", "G1")]
head(y1)
tail(y1, n=3)
y1[[2]]     # note the difference with y1[2]
y1[["G2"]]  # note the difference with y1["G2"]

y0[["3"]]
y0[[3]]

## In addition to all the forms of subscripting supported by ordinary
## lists, List objects and derivatives accept a subscript that is a
## list-like object. This form of subsetting is called "list-style
## subsetting":
i <- list(4:3, -2, 1)    # ordinary list
y1[i]
i <- y1 >= 200           # LogicalList object
y1[i]

## List-style subsetting also works with an RleList or IntegerRangesList
## subscript:
i <- RleList(y1 >= 200)  # RleList object
y1[i]
i <- IRangesList(RleList(y1 >= 200))  # IRangesList object
y1[i]

## ---------------------------------------------------------------------
## C. THE "UNLIST -> TRANFORM -> RELIST" IDIOM
## ---------------------------------------------------------------------
## The "unlist -> transform -> relist" idiom is a very efficient way to
## apply the same simple transformation to all the **inner elements** of
## a list-like object (i.e. to all the elements of its list elements).
## The result is another list-like object with the same shape as the
## original object (but not necessarily the same class):
relist(sqrt(unlist(y1)), y1)
relist(toupper(unlist(y2)), y2)

## However note that sqrt(), toupper(), and many other base functions,
## can be used directly on a List derivative. This is because the IRanges
## package defines methods for these functions that know how to handle
## List objects:
sqrt(y1)     # same as 'relist(sqrt(unlist(y1)), y1)'
toupper(y2)  # same as 'relist(toupper(unlist(y2)), y2)'
```

`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, pc and Map, one or more list-like objects.
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.
deparse.level	See ?base::rbind for a description of this argument.

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 an ordinary 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(...) combine list-like objects by concatenating them 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)
```

`LLint-class` *LLint vectors*

Description

The LLint class is a container for storing a vector of *large integers* (i.e. long long int values at the C level).

Usage

```
LLint(length=0L)
as.LLint(x)
is.LLint(x)
```

Arguments

`length`	A non-negative number (i.e. integer, double, or LLint value) specifying the desired length.
`x`	Object to be coerced or tested.

Details

LLint vectors aim to provide the same functionality as integer vectors in base R but their values are stored as long long int values at the C level vs int values for integer vectors. Note that on Intel platforms long long int values are 64-bit and int values 32-bit only. Therefore LLint vectors can hold values in the +/-9.223e18 range (approximately) vs +/-2.147e9 only for integer vectors.

NAs are supported and the `NA_LLint_` constant is predefined for convenience as `as(NA, "LLint")`.

Names are not supported for now.

Coercions from/to logical, integer, double, and character are supported.

Operations from the `Arith`, `Compare` and `Summary` groups are supported.

More operations coming soon...

Author(s)

Hervé Pagès

See Also

- integer vectors in base R.
- The Arith, Compare and Summary group generics in the **methods** package.

Examples

```
## A long long int uses 8 bytes (i.e. 64 bits) in C:
.Machine$sizeof.longlong

## --------------------------------------------------------------------
## SIMPLE EXAMPLES
## --------------------------------------------------------------------

LLint()
LLint(10)

as.LLint(3e9)
as.LLint("3000000000")

x <- as.LLint(1:10 * 111111111)
x * x
5 * x   # result as vector of doubles (i.e. 'x' coerced to double)
5L * x  # result as LLint vector (i.e. 5L coerced to LLint vector)
max(x)
min(x)
range(x)
sum(x)

x <- as.LLint(1:20)
prod(x)
x <- as.LLint(1:21)
prod(x)  # result is out of LLint range (+/-9.223e18)
prod(as.numeric(x))

x <- as.LLint(1:75000)
sum(x * x * x) == sum(x) * sum(x)

## Note that max(), min() and range() *always* return an LLint vector
## when called on an LLint vector, even when the vector is empty:
max(LLint())  # NA with no warning
min(LLint())  # NA with no warning

## This differs from how max(), min() and range() behave on an empty
## integer vector:
max(integer())  # -Inf with a warning
min(integer())  #  Inf with a warning

## --------------------------------------------------------------------
## GOING FROM STRINGS TO INTEGERS
## --------------------------------------------------------------------

## as.integer() behaves like as.integer(as.double()) on a character
```

```
## vector. With the following consequence:
s <- "-2.9999999999999999"
as.integer(s)   # -3

## as.LLint() converts the string *directly* to LLint, without
## coercing to double first:
as.LLint(s)  # decimal part ignored

## ------------------------------------------------------------------------
## GOING FROM DOUBLE-PRECISION VALUES TO INTEGERS AND VICE-VERSA
## ------------------------------------------------------------------------

## Be aware that a double-precision value is not guaranteed to represent
## exactly an integer > 2^53. This can cause some surprises:
2^53 == 2^53 + 1  # TRUE, yep!

## And therefore:
as.LLint(2^53) == as.LLint(2^53 + 1)  # also TRUE

## This can be even more disturbing when passing a big literal integer
## value because the R parser will turn it into a double-precision value
## before passing it to as.LLint():
x1 <- as.LLint(9007199254740992)  # same as as.LLint(2^53)
x1
x2 <- as.LLint(9007199254740993)  # same as as.LLint(2^53 + 1)
x2
x1 == x2  # still TRUE

## However, no precision is lost if a string literal is used instead:
x1 <- as.LLint("9007199254740992")
x1
x2 <- as.LLint("9007199254740993")
x2
x1 == x2  # FALSE
x2 - x1

d1 <- as.double(x1)
d2 <- as.double(x2)  # warning!
d1 == d2  # TRUE

## ------------------------------------------------------------------------
## LLint IS IMPLEMENTED AS AN S4 CLASS
## ------------------------------------------------------------------------

class(LLint(10))
typeof(LLint(10))        # S4
storage.mode(LLint(10))  # S4
is.vector(LLint(10))     # FALSE
is.atomic(LLint(10))     # FALSE

## This means that an LLint vector cannot go in an ordinary data
## frame:
## Not run:
```

```
data.frame(id=as.LLint(1:5))  # error!

## End(Not run)
## A DataFrame needs to be used instead:
DataFrame(id=as.LLint(1:5))

## ---------------------------------------------------------------------
## SANITY CHECKS
## ---------------------------------------------------------------------

x <- as.integer(c(0, 1, -1, -3, NA, -99))
y <- as.integer(c(-6, NA, -4:3, 0, 1999, 6:10, NA))
xx <- as.LLint(x)
yy <- as.LLint(y)

## Operations from "Arith" group:
stopifnot(identical(x + y, as.integer(xx + yy)))
stopifnot(identical(as.LLint(y + x), yy + xx))
stopifnot(identical(x - y, as.integer(xx - yy)))
stopifnot(identical(as.LLint(y - x), yy - xx))
stopifnot(identical(x * y, as.integer(xx * yy)))
stopifnot(identical(as.LLint(y * x), yy * xx))
stopifnot(identical(x / y, xx / yy))
stopifnot(identical(y / x, yy / xx))
stopifnot(identical(x %/% y, as.integer(xx %/% yy)))
stopifnot(identical(as.LLint(y %/% x), yy %/% xx))
stopifnot(identical(x %% y, as.integer(xx %% yy)))
stopifnot(identical(as.LLint(y %% x), yy %% xx))
stopifnot(identical(x ^ y, xx ^ yy))
stopifnot(identical(y ^ x, yy ^ xx))

## Operations from "Compare" group:
stopifnot(identical(x == y, xx == yy))
stopifnot(identical(y == x, yy == xx))
stopifnot(identical(x != y, xx != yy))
stopifnot(identical(y != x, yy != xx))
stopifnot(identical(x <= y, xx <= yy))
stopifnot(identical(y <= x, yy <= xx))
stopifnot(identical(x >= y, xx >= yy))
stopifnot(identical(y >= x, yy >= xx))
stopifnot(identical(x < y, xx < yy))
stopifnot(identical(y < x, yy < xx))
stopifnot(identical(x > y, xx > yy))
stopifnot(identical(y > x, yy > xx))

## Operations from "Summary" group:
stopifnot(identical(max(y), as.integer(max(yy))))
stopifnot(identical(max(y, na.rm=TRUE), as.integer(max(yy, na.rm=TRUE))))
stopifnot(identical(min(y), as.integer(min(yy))))
stopifnot(identical(min(y, na.rm=TRUE), as.integer(min(yy, na.rm=TRUE))))
stopifnot(identical(range(y), as.integer(range(yy))))
stopifnot(identical(range(y, na.rm=TRUE), as.integer(range(yy, na.rm=TRUE))))
stopifnot(identical(sum(y), as.integer(sum(yy))))
```

```
stopifnot(identical(sum(y, na.rm=TRUE), as.integer(sum(yy, na.rm=TRUE))))
stopifnot(identical(prod(y), as.double(prod(yy))))
stopifnot(identical(prod(y, na.rm=TRUE), as.double(prod(yy, na.rm=TRUE))))
```

`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)
```

RectangularData-class *RectangularData objects*

Description

RectangularData is a virtual class with no slots to be extended by classes that aim at representing objects with a 2D rectangular shape.

Some examples of RectangularData extensions are:

- The DataFrame class defined in this package (**S4Vectors**).
- The DelayedMatrix class defined in the **DelayedArray** package.
- The SummarizedExperiment and Assays classes defined in the **SummarizedExperiment** package.

Details

Any object that belongs to a class that extends RectangularData is called a *RectangularData derivative*.

Users should be able to access and manipulate RectangularData derivatives via the *standard 2D API* defined in base R, that is, using things like `dim()`, `nrow()`, `ncol()`, `dimnames()`, the 2D form of `[` (`x[i, j]`), `rbind()`, `cbind()`, etc...

Not all RectangularData derivatives will necessarily support the full 2D API but they must support at least `dim()`, `nrow(x)`, `ncol(x)`, `NROW(x)`, and `NCOL(x)`. And of course, `dim()` must return an integer vector of length 2 on any of these objects.

Developers who implement RectangularData extensions should also make sure that they support low-level operations `bindROWS()` and `bindCOLS()`.

Accessors

In the following code snippets, x is a RectangularData derivative. Not all RectangularData derivatives will support all these accessors.

`dim(x)`: Length two integer vector defined as `c(nrow(x), ncol(x))`. Must work on any RectangularData derivative.

`nrow(x)`, `ncol(x)`: Get the number of rows and columns, respectively. Must work on any RectangularData derivative.

`NROW(x)`, `NCOL(x)`: Same as `nrow(x)` and `ncol(x)`, respectively. Must work on any RectangularData derivative.

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

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

Subsetting

In the code snippets below, x is a RectangularData derivative.

`x[i, j, drop=TRUE]`: Return a new RectangularData derivative of the same class as x made of the selected rows and columns.

For single row and/or column selection, the `drop` argument specifies whether or not to "drop the dimensions" of the result. More precisely, when `drop=TRUE` (the default), a single row or column is returned as a vector-like object (of length/NROW equal to `ncol(x)` if a single row, or equal to `nrow(x)` if a single column).

Not all RectangularData derivatives support the `drop` argument. For example DataFrame and DelayedMatrix objects support it (only for a single column selection for DataFrame objects), but SummarizedExperiment objects don't (`drop` is ignored for these objects and subsetting always returns a SummarizedExperiment derivative of the same class as x).

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

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

`subset(x, subset, select, drop=FALSE)`: Return a new RectangularData derivative 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.

Combining

In the code snippets below, all the input objects are expected to be RectangularData derivatives.

`rbind(...)`: Creates a new RectangularData derivative by aggregating the rows of the input objects.

`cbind(...)`: Creates a new RectangularData derivative by aggregating the columns of the input objects.

`combineRows(x, ...)`: Creates a new RectangularData derivative (of the same class as `x`) by aggregating the rows of the input objects. Unlike `rbind()`, `combineRows()` will handle cases involving differences in the column names of the input objects by adding the missing columns to them, and filling these columns with `NA`s. The column names of the returned object are a union of the column names of the input objects.

Behaves like an *endomorphism* with respect to its first argument i.e. returns an object of the same class as `x`.

Finally note that this is a generic function with methods defined for DataFrame objects and other RectangularData derivatives.

`combineCols(x, ..., use.names=TRUE)`: Creates a new RectangularData derivative (of the same class as `x`) by aggregating the columns of the input objects. Unlike `cbind()`, `combineCols()` will handle cases involving differences in the number of rows of the input objects.

If `use.names=TRUE`, all objects are expected to have non-`NULL`, non-duplicated row names. These row names do not have to be the same, or even shared, across the input objects. Missing rows in any individual input object are filled with `NA`s, such that the row names of the returned object are a union of the row names of the input objects.

If `use.names=FALSE`, all objects are expected to have the same number of rows, and this function behaves the same as `cbind()`. The row names of the returned object is set to `rownames(x)`. Differences in the row names between input objects are ignored.

Behaves like an *endomorphism* with respect to its first argument i.e. returns an object of the same class as `x`.

Finally note that this is a generic function with methods defined for DataFrame objects and other RectangularData derivatives.

`combineUniqueCols(x, ..., use.names=TRUE)`: Same as `combineCols()`, but this function will attempt to collapse multiple columns with the same name across the input objects into a single column in the output. This guarantees that the column names in the output object are always unique. The only exception is for unnamed columns, which are not collapsed. The function works on any rectangular objects for which `combineCols()` works.

When `use.names=TRUE`, collapsing is only performed if the duplicated column has identical values for the shared rows in the input objects involved. Otherwise, the contents of the later input object is simply ignored with a warning. Similarly, if `use.names=FALSE`, the duplicated columns must be identical for all rows in the affected input objects.

Behaves like an *endomorphism* with respect to its first argument i.e. returns an object of the same class as `x`.

Finally note that this function is implemented on top of `combineCols()` and is expected to work on any RectangularData derivatives for which `combineCols()` works.

Author(s)

Hervé Pagès and Aaron Lun

See Also

- DataFrame for a RectangularData extension that mimics `data.frame` objects from base R.
- DataFrame-combine for `combineRows()`, `combineCols()`, and `combineUniqueCols()` examples involving DataFrame objects.
- data.frame objects in base R.

Examples

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

`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.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`.

`c(x, ..., ignore.mcols=FALSE)`: Concatenate Rle object `x` and the Rle objects in `...` together. See `?c` in this package (the **S4Vectors** package) for more information about concatenating Vector derivatives.

`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.

`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 which values are `NA`.

`is.finite(x)`: Returns a logical Rle indicating which values are finite.

`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("auto", "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 concatenated 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,y	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.
algorithm,print.level	See ?stats::runmed for a description of these arguments.
...	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, useBytes = FALSE)`: 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, useBytes = FALSE)`: 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.

`droplevels(x)`: Drops unused 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)))
```

`show-utils` *Display utilities*

Description

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

`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)
```

---

splitAsList *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, ...)

splitAsList(x, f, drop=FALSE, ...)

relistToClass(x)
```

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).
...	Extra arguments passed to any of the first 3 split() methods will be passed to splitAsList() (see Details below). Extra arguments passed to the last split() method will be passed to base::split() (see Details below). Extra arguments passed to splitAsList() will be passed to the specific method selected by method dispatch.

Details

The first 3 split() methods just delegate to splitAsList().

The last split() method just does:

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

splitAsList() is an S4 generic function. It is the workhorse behind the first 3 split() methods above. It behaves like base::split() except that it returns a List derivative instead of an ordinary list. The exact class of this List derivative depends only on the class of x and can be obtained independently with relistToClass(x).

Note that 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))
```

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

Value

splitAsList() and the first 3 split() methods behave like base::split() except that they return a List derivative (of class relistToClass(x)) instead of an ordinary list. Like with base::split(), all the list elements in this object have the same class as x.

See Also

- The split function in the **base** package.
- The relist methods and extractList generic function defined 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)

## Use splitAsList() if you need to split an ordinary vector into a
## List object:
split(letters, 1:2)        # ordinary list
splitAsList(letters, 1:2)  # List object
```

---

stack-methods *Stack objects*

Description

The **S4Vectors** package defines stack methods for List and matrix objects.

It also introduces mstack(), a variant of stack where the list is taken as the list of arguments in ....

Usage

```
## S4 method for signature 'List'
stack(x, index.var="name", value.var="value", name.var=NULL)

## S4 method for signature 'matrix'
stack(x, row.var=names(dimnames(x))[1L],
      col.var=names(dimnames(x))[2L],
      value.var="value")

mstack(..., .index.var="name")
```

Arguments

x	A List derivative (for the stack method for List objects), or a matrix (for the stack method for matrix objects).
index.var, .index.var	A single string specifying the column name for the index (source name) column.
value.var	A single string specifying the column name for the values.
name.var	TODO
row.var, col.var	TODO
...	The objects to stack. Each of them should be a Vector or vector (mixing the two will not work).

Details

As with stack on a list, stack on a List derivative constructs a DataFrame with two columns: one for the unlisted values, the other indicating the name of the element from which each value was obtained. index.var specifies the column name for the index (source name) column and value.var specifies the column name for the values.

[TODO: Document stack() method for matrix objects.]

See Also

- stack in the **utils** package.
- List and DataFrame objects.

Examples

```
library(IRanges)
starts <- IntegerList(c(1, 5), c(2, 8))
ends <- IntegerList(c(3, 8), c(5, 9))
rgl <- IRangesList(start=starts, end=ends)
rangeDataFrame <- stack(rgl, "space", "ranges")
```

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.

TransposedDataFrame-class *TransposedDataFrame objects*

Description

The TransposedDataFrame class is a container for representing a transposed DataFrame object, that is, a rectangular data container where the rows are the variables and the columns the observations.

A typical situation for using a TransposedDataFrame object is when one needs to store a DataFrame object in the `assay()` component of a SummarizedExperiment object but the rows in the DataFrame object should correspond to the samples and the columns to the features. In this case the DataFrame object must first be transposed so that the variables in it run "horizontally" instead of "vertically". See the Examples section at the bottom of this man page for an example.

Details

TransposedDataFrame objects are constructed by calling `t()` on a DataFrame object.

Like for a DataFrame object, or, more generally, for a data-frame-like object, the length of a TransposedDataFrame object is its number of variables. However, *unlike* for a data-frame-like object, its length is also its number of rows, not its number of columns. For this reason, a TransposedDataFrame object is NOT considered to be a data-frame-like object.

Author(s)

Hervé Pagès

See Also

- DataFrame objects.
- SummarizedExperiment objects in the **SummarizedExperiment** package.

Examples

```
## A DataFrame object with 3 variables:
df <- DataFrame(aa=101:126, bb=letters, cc=Rle(c(TRUE, FALSE), 13),
                row.names=LETTERS)
dim(df)
length(df)
df$aa

tdf <- t(df)
tdf
dim(tdf)
length(tdf)
tdf$aa

t(tdf)  # back to 'df'
stopifnot(identical(df, t(tdf)))

tdf$aa <- 0.05 * tdf$aa

x1 <- DataFrame(A=1:5, B=letters[1:5], C=11:15)
y1 <- DataFrame(B=c(FALSE, NA, TRUE), C=c(FALSE, NA, TRUE), A=101:103)
cbind(t(x1), t(y1))
stopifnot(identical(t(rbind(x1, y1)), cbind(t(x1), t(y1))))

## A TransposedDataFrame object can be used in the assay() component of a
## SummarizedExperiment object if the transposed layout is needed i.e. if
## the rows and columns of the original DataFrame object need to be treated
## as the samples and features (in this order) of the SummarizedExperiment
## object:
library(SummarizedExperiment)
se1 <- SummarizedExperiment(df)
se1
assay(se1)  # the 3 variables run "vertically"

se2 <- SummarizedExperiment(tdf)
se2
assay(se2)  # the 3 variables run "horizontally"
```

`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 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)`.

`NROW(x)`: Equivalent to either `nrow(x)` or `length(x)`, depending on whether `x` has dimensions (i.e. `dim(x)` is not `NULL`) or not (i.e. `dim(x)` is `NULL`).

`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`.

`unname(x)`: removes the names from `x`, if any.

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

`mcols(x, use.names=TRUE)`, `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 DataFrame object. When setting the metadata columns, the supplied value must be `NULL` or a DataFrame 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.

`x[i]`: When supported, return a new Vector object of the same class as `x` made of the elements selected by `i`. `i` can be missing; an NA-free logical, numeric, or character vector or factor (as ordinary vector or Rle object); or a IntegerRanges object.

`x[i, j]`: Like the above, but allow the user to conveniently subset the metadata columns thru `j`.

NOTE TO DEVELOPERS: A Vector subclass with a true 2-D semantic (e.g. SummarizedExperiment) needs to overwrite the `"["` method for Vector objects. This means that code intended to operate on an arbitrary Vector derivative `x` should not use this feature as there is no guarantee that `x` supports it. For this reason this feature should preferrably be used *interactively* only.

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

Convenience wrappers for common subsetting operations

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

`subset(x, subset, select, drop=FALSE, ...)`: Return a new Vector object made of the subset using logical vector `subset`, where missing values are taken as FALSE. TODO: Document `select`, `drop`, and `...`.

`window(x, start=NA, end=NA, width=NA)`: Extract the subsequence from `x` that corresponds to the window defined by `start`, `end`, and `width`. At most 2 of `start`, `end`, and `width` can be set to a non-`NA` value, which must be a non-negative integer. More precisely:

- If `width` is set to `NA`, then `start` or `end` or both can be set to `NA`. In this case `start=NA` is equivalent to `start=1` and `end=NA` is equivalent to `end=length(x)`.
- If `width` is set to a non-negative integer value, then one of `start` or `end` must be set to a non-negative integer value and the other one to `NA`.

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

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

`rev(x)`: Return a new Vector object made of the original elements in the reverse order.

`rep(x, times, length.out, each)` and `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.

Concatenation

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

`c(x, ..., ignore.mcols=FALSE)`: Concatenate `x` and the Vector objects in `...` together. Any object in `...` should belong to the same class as `x` or to one of its subclasses. If not, then an attempt will be made to coerce it with `as(object, class(x), strict=FALSE)`. `NULL`s are accepted and ignored. The result of the concatenation is an object of the same class as `x`.

Handling of the metadata columns:

- If only one of the Vector objects has metadata columns, then the corresponding metadata columns are attached to the other Vector objects and set to `NA`.

- When multiple Vector objects have their own metadata columns, the user must ensure that each such DataFrame have identical layouts to each other (same columns defined), in order for the concatenation to be successful, otherwise an error will be thrown.
- The user can call `c(x, ..., ignore.mcols=FALSE)` in order to concatenate Vector objects with differing sets of metadata columns, which will result in the concatenated object having NO metadata columns.

IMPORTANT NOTE: Be aware that calling `c` with named arguments (e.g. `c(a=x, b=y)`) tends to break method dispatch so please make sure that `args` is an *unnamed* list when using `do.call(c, args)` to concatenate a list of objects together.

`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`.

`expand.grid(...)`: Find cartesian product of every vector in `...` and return a data.frame, each column of which corresponds to an argument. See `expand.grid`.

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.
- Factor for a direct Vector extension that serves a similar role as factor in base R.
- 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.
- DataFrame which is the type of object 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

## sameAsPreviousROW()
## -------------------

sameAsPreviousROW(x)

## match()
## -------

## S4 method for signature 'Vector,Vector'
match(x, table, nomatch = NA_integer_,
    incomparables = NULL, ...)

## 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)

## rank()
## ------

## S4 method for signature 'Vector'
rank(x, na.last = TRUE, ties.method = c("average",
        "first", "last", "random", "max", "min"), by)

## xtfrm()
## -------

## S4 method for signature 'Vector'
xtfrm(x)

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

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

Arguments

x, y, e1, e2, table	Vector-like objects.
nomatch	See ?base::match.
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 for these generics. The match method for Vector objects does support this argument, see ?base::match for details.
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.
ties.method	See ?base::rank.
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 ANY method is implemented. However note that the default selfmatch(x, ...) for Vector x will typically be more efficient than match(x, x, ...), and can be made even more so if a specific selfmatch method is implemented for a given subclass.

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 sameAsPreviousROW: a logical vector of length equal to x, indicating whether each entry of x is equal to the previous entry. The first entry is always FALSE for a non-zero-length x.

For match and 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 `xtfrm`: see `?base::xtfrm`.

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.

Subclass comparison methods can be split into various categories. The first category *must* be implemented for each subclass, as these do not have sensible defaults for arbitrary Vector objects:

- The **S4Vectors** package provides no `order` method for Vector objects. So calling `order` on a Vector derivative for which no specific `order` method is defined will use `base::order`, which calls `xtfrm`, with in turn calls `order`, which calls `xtfrm`, and so on. This infinite recursion of S4 dispatch eventually results in an error about reaching the stack limit.

  To avoid this behavior, a specialized `order` method needs to be implemented for specific Vector subclasses (e.g. for Hits and IntegerRanges objects).

- `sameAsPreviousROW` is default implemented on top of the == method, so will work out-of-the-box on Vector objects for which == works as expected. However, == is default implemented on top of `pcompare`, which itself has a default implementation that relies on `sameAsPreviousROW`! This again leads to infinite recursion and an error about the stack limit.

  To avoid this behavior, a specialized `sameAsPreviousROW` method must be implemented for specific Vector subclasses.

The second category contains methods that have default implementations provided for all Vector objects and their derivatives. These methods rely on the first category to provide sensible default behaviour without further work from the developer. However, it is often the case that greater efficiency can be achieved for a specific data structure by writing a subclass-specific version of these methods.

- The `pcompare` method for Vector objects is implemented on top of `order` and `sameAsPreviousROW`, and so will work out-of-the-box on Vector derivatives for which `order` and `sameAsPreviousROW` work as expected.
- The `xtfrm` method for Vector objects is also implemented on top of `order` and `sameAsPreviousROW`, and so will also work out-of-the-box on Vector derivatives for which `order` and `sameAsPreviousROW` work as expected.
- `selfmatch` is itself implemented on top of `xtfrm` (indirectly, via `grouping`) so it will work out-of-the-box on Vector objects for which `xtfrm` works as expected.
- The `match` method for Vector objects is implemented on top of `selfmatch`, so works out-of-the-box on Vector objects for which `selfmatch` works as expected.

(A careful reader may notice that `xtfrm` and `order` could be swapped between categories to achieve the same effect. Similarly, `sameAsPreviousROW` and `pcompare` could also be swapped. The exact

categorization of these methods is left to the discretion of the developer, though this is mostly academic if both choices are specialized.)

The third category also contains methods that have default implementations, but unlike the second category, these defaults are straightforward and generally do not require any specialization for efficiency purposes.

- 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.

- 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.
- 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.
- 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.
- 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.

Author(s)

Hervé Pagès, with contributions from Aaron Lun

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.
- IntegerRanges-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 ?`IntegerRanges-comparison` for more examples (on IntegerRanges
## 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 IntegerRanges 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