Package 'IRanges'

October 8, 2014

Title Infrastructure for manipulating intervals on sequences

Description The package provides efficient low-level and highly reusable S4 classes for storing ranges of integers, RLE vectors (Run-Length Encoding), and, more generally, data that can be organized sequentially (formally defined as Vector objects), as well as views on these Vector objects. Efficient list-like classes are also provided for storing big collections of instances of the basic classes. All classes in the package use consistent naming and share the same rich and consistent ``Vector API" as much as possible.

Version 1.22.10

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

Maintainer Bioconductor Package Maintainer <maintainer@bioconductor.org>

biocViews Infrastructure, DataRepresentation

Depends R (>= 3.1.0), methods, utils, stats, BiocGenerics (>= 0.9.1)

Imports methods, utils, stats, BiocGenerics, stats4

Suggests XVector, GenomicRanges, BSgenome.Celegans.UCSC.ce2, RUnit

License Artistic-2.0

ExtraLicenses The following files in the 'src' directory are licensed for all use by Jim Kent, in a manner compatible with the Artistic 2.0 license: common.c/h, memalloc.c/h, localmem.c/h,hash.c/h, errabort.c/h, rbTree.c/h, dlist.c/h, errCatch.h

Collate S4-utils.R utils.R isConstant.R normarg-utils.R subsetting-utils.R int-utils.R str-utils.R compact_bitvector.R endoapply.R runstat.R Annotated-class.R Vector-class.R Vector-comparison.R List-class.R AtomicList-class.R Ranges-class.R Ranges-comparison.R IRanges-class.R IRanges-constructor.R IRanges-utils.R DataTable-API.R DataTable-stats.R Views-class.R Grouping-class.R SimpleList-class.R CompressedList-class.R Rle-class.R RleViews-class.R RleViews-utils.R extractList.R seqapply.R

multisplit.R AtomicList-impl.R List-comparison.R
DataFrame-class.R DataFrame-utils.R DataFrameList-class.R
DataFrameList-utils.R RangesList-class.R GappedRanges-class.R
ViewsList-class.R RleViewsList-class.R RleViewsList-utils.R
MaskCollection-class.R RangedData-class.R FilterRules-class.R
RDApplyParams-class.R RangedData-utils.R Hits-class.R
HitsList-class.R RangesMapping-class.R IntervalTree-class.R
IntervalTree-utils.R IntervalForest-class.R
RangedSelection-class.R read.Mask.R funprog-methods.R
intra-range-methods.R inter-range-methods.R reverse-methods.R
coverage-methods.R slice-methods.R setops-methods.R
findOverlaps-methods.R nearest-methods.R expand-methods.R
updateObject-methods.R classNameForDisplay-methods.R
tile-methods.R test_IRanges_package.R debug.R zzz.R

R topics documented:

`Annotated-class` *Annotated class*

Description

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

Details

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

Accessors

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

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

Author(s)

P. Aboyoun

See Also

`Vector` for example implementations

`AtomicList` *Lists of Atomic Vectors in Natural and Rle Form*

Description

An extension of `List` that holds only atomic vectors in either a natural or run-length encoded form.

Details

The lists of atomic vectors are `LogicalList`, `IntegerList`, `NumericList`, `ComplexList`, `CharacterList`, and `RawList`. There is also an `RleList` class for run-length encoded versions of these atomic vector types.

Each of the above mentioned classes is virtual with Compressed* and Simple* non-virtual representations.

Constructors

`LogicalList(..., compress = TRUE)`: Concatenates the `logical` vectors in `...` into a new `LogicalList`. If `compress`, the internal storage of the data is compressed.

`IntegerList(..., compress = TRUE)`: Concatenates the `integer` vectors in `...` into a new `IntegerList`. If `compress`, the internal storage of the data is compressed.

`NumericList(..., compress = TRUE)`: Concatenates the `numeric` vectors in `...` into a new `NumericList`. If `compress`, the internal storage of the data is compressed.

`ComplexList(..., compress = TRUE)`: Concatenates the `complex` vectors in `...` into a new `ComplexList`. If `compress`, the internal storage of the data is compressed.

`CharacterList(..., compress = TRUE)`: Concatenates the `character` vectors in `...` into a new `CharacterList`. If `compress`, the internal storage of the data is compressed.

`RawList(..., compress = TRUE)`: Concatenates the `raw` vectors in `...` into a new `RawList`. If `compress`, the internal storage of the data is compressed.

`RleList(..., compress = TRUE)`: Concatenates the run-length encoded atomic vectors in `...` into a new `RleList`. If `compress`, the internal storage of the data is compressed.

Coercion

`as(from, "CompressedSplitDataFrameList")`, `as(from, "SimpleSplitDataFrameList")`: Creates a CompressedSplitDataFrameList/SimpleSplitDataFrameList instance from an AtomicList instance.

`as(from, "IRangesList")`, `as(from, "CompressedIRangesList")`, `as(from, "SimpleIRangesList")`: Creates a CompressedIRangesList/SimpleIRangesList instance from a LogicalList or logical RleList instance. Note that the elements of this instance are guaranteed to be normal.

`as(from, "NormalIRangesList")`, `as(from, "CompressedNormalIRangesList")`, `as(from, "SimpleNormalIRangesl`
Creates a CompressedNormalIRangesList/SimpleNormalIRangesList instance from a LogicalList or logical RleList instance.

`as(from, "CharacterList")`, `as(from, "ComplexList")`, `as(from, "IntegerList")`, `as(from, "LogicalList")`, `as(from, "NumericList")`, `as(from, "RawList")`, `as(from, "RleList")`: Coerces an `AtomicList` `from` to another derivative of `AtomicList`.

`as(from, "AtomicList")`: If `from` is a vector, converts it to an `AtomicList` of the appropriate type.

Group Generics

AtomicList objects have support for S4 group generic functionality to operate within elements across objects:

`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.

Other Basic Methods

The AtomicList objects also support a large number of basic methods. Like the group generics above, these methods perform the corresponding operation on each element of the list separately. The methods are:

General `is.na`, `duplicated`, `unique`, `match`, `%in%`, `table`, `order`, `sort`

Logical `!`, `which`, `which.max`, `which.min`

Numeric `diff`, `pmax`, `pmax.int`, `pmin`, `pmin.int`, `mean`, `var`, `cov`, `cor`, `sd`, `median`, `quantile`, `mad`, `IQR`

Running Window `smoothEnds`, `runmed`. `runmean`, `runsum`, `runwtsum`, `runq`

Character `nchar`, `chartr`, `tolower`, `toupper`, `sub`, `gsub`

RleList Methods

RleList has a number of methods that are not shared by other AtomicList derivatives.

`runLength(x)`: Gets the run lengths of each element of the list, as an IntegerList.

`runValue(x)`, `runValue(x) <- value`: Gets or sets the run values of each element of the list, as an AtomicList.

`ranges(x)`: Gets the run ranges as a `RangesList`.

Specialized Methods

`drop(x)`: Checks if every element of `x` is of length one, and, if so, unlists `x`. Otherwise, an error is thrown.

`unstrsplit(x, sep="")`: A fast `sapply(x, paste0, collapse=sep)`. See ?`unstrsplit` for the details.

Author(s)

P. Aboyoun

See Also

`List` for the applicable methods.

Examples

```
int1 <- c(1L,2L,3L,5L,2L,8L)
int2 <- c(15L,45L,20L,1L,15L,100L,80L,5L)
collection <- IntegerList(int1, int2)

## names
names(collection) <- c("one", "two")
names(collection)
names(collection) <- NULL # clear names
names(collection)
names(collection) <- "one"
names(collection) # c("one", NA)

## extraction
collection[[1]] # range1
collection[["1"]] # NULL, does not exist
collection[["one"]] # range1
collection[[NA_integer_]] # NULL

## subsetting
collection[numeric()] # empty
collection[NULL] # empty
collection[] # identity
collection[c(TRUE, FALSE)] # first element
collection[2] # second element
collection[c(2,1)] # reversed
```

```
collection[-1] # drop first
collection$one

## replacement
collection$one <- int2
collection[[2]] <- int1

## combining
col1 <- IntegerList(one = int1, int2)
col2 <- IntegerList(two = int2, one = int1)
col3 <- IntegerList(int2)
append(col1, col2)
append(col1, col2, 0)
col123 <- c(col1, col2, col3)
col123

## revElements
revElements(col123)
revElements(col123, 4:5)

## group generics
2 * col1
col1 + col1
col1 > 2
sum(col1)  # equivalent to (but faster than) sapply(col1, sum)
mean(col1)  # equivalent to sapply(col1, mean)
```

`classNameForDisplay` *Provide a class name for displaying to users*

Description

Generic function to create a class name suitable for display to users. Current methods remove "Compressed" or "Simple" from the formal names of classes defined in IRanges.

Usage

```
classNameForDisplay(x)
```

Arguments

x	An instance of any class.

Value

A character vector of length 1, as returned by `class`.

Author(s)

Martin Morgan

Examples

```
classNameForDisplay(IntegerList())
class(IntegerList())
```

---

coverage-methods *Coverage of a set of ranges*

Description

For each position in the space underlying a set of ranges, counts the number of ranges that cover it.

Usage

```
coverage(x, shift=0L, width=NULL, weight=1L, ...)

## S4 method for signature Ranges
coverage(x, shift=0L, width=NULL, weight=1L,
            method=c("auto", "sort", "hash"))

## S4 method for signature RangesList
coverage(x, shift=0L, width=NULL, weight=1L,
            method=c("auto", "sort", "hash"))
```

Arguments

x	A Ranges, Views, or RangesList object. See ?coverage-methods in the **GenomicRanges** package for coverage methods for other objects.
shift	Specifies how much each range in x should be shifted before the coverage is computed. • If x is a Ranges or Views object: shift must be an integer or numeric vector parallel to x (will get recycled if necessary) and with no NAs. • If x is a RangesList object: shift must be a numeric vector or list-like object of the same length as x (will get recycled if necessary). If it's a numeric vector, it's first turned into a list with as.list. After recycling, each list element shift[[i]] must be an integer or numeric vector parallel to x[[i]] (will get recycled if necessary) and with no NAs. A positive shift value will shift the corresponding range in x to the right, and a negative value to the left.
width	Specifies the length of the returned coverage vector(s). • If x is a Ranges object: width must be NULL (the default), an NA, or a single non-negative integer. After being shifted, the ranges in x are always clipped on the left to keep only their positive portion i.e. their intersection with the [1, +inf) interval. If width is a single non-negative integer, then they're also clipped on the right to keep only their intersection with the [1, width] interval. In that case coverage returns a vector of length width. Otherwise, it returns a vector that extends to the last position in the underlying space covered by the shifted ranges.

- If x is a Views object: Same as for a Ranges object, except that, if width is NULL then it's treated as if it was length(subject(x)).
- If x is a RangesList object: width must be NULL or an integer vector parallel to x (i.e. with one element per list element in x). If not NULL, the vector must contain NAs or non-negative integers and it will get recycled to the length of x if necessary. If NULL, it is replaced with NA and recycled to the length of x. Finally width[i] is used to compute the coverage vector for x[[i]] and is therefore treated like explained above (when x is a Ranges object).

weight — Assigns a weight to each range in x.

- If x is a Ranges or Views object: weight must be an integer or numeric vector parallel to x (will get recycled if necessary).
- If x is a RangesList object: weight must be a numeric vector or list-like object of the same length as x (will get recycled if necessary). If it's a numeric vector, it's first turned into a list with as.list. After recycling, each list element weight[[i]] must be an integer or numeric vector parallel to x[[i]] (will get recycled if necessary).

If weight is an integer vector or list-like object of integer vectors, the coverage vector(s) will be returned as integer-Rle object(s). If it's a numeric vector or list-like object of numeric vectors, the coverage vector(s) will be returned as numeric-Rle object(s).

Alternatively, weight can also be specified as a single string naming a metadata column in x (i.e. a column in mcols(x)) to be used as the weight vector.

method — If method is set to "sort", then x is sorted previous to the calculation of the coverage. If method is set to hash, then x is hashed directly to a vector of length width without previous sorting.

The "hash" method is faster than the "sort" method when x is large (i.e. contains a lot of ranges). When x is small and width is big (e.g. x represents a small set of reads aligned to a big chromosome), then method="sort" is faster and uses less memory than method="hash".

Using method="auto" selects the best method based on length(x) and width.

... — Further arguments to be passed to or from other methods.

Value

If x is a Ranges or Views object: An integer- or numeric-Rle object depending on whether weight is an integer or numeric vector.

If x is a RangesList object: An RleList object with one coverage vector per list element in x, and with x names propagated to it. The i-th coverage vector can be either an integer- or numeric-Rle object, depending on the type of weight[[i]] (after weight has gone thru as.list and recycling, like described previously).

Author(s)

H. Pages and P. Aboyoun

See Also

- coverage-methods in the **GenomicRanges** package for more `coverage` methods.
- The `slice` function for slicing the Rle or RleList object returned by `coverage`.
- The Ranges, RangesList, Rle, and RleList classes.

Examples

```
## ---------------------------------------------------------------------
## A. COVERAGE OF AN IRanges OBJECT
## ---------------------------------------------------------------------
x <- IRanges(start=c(-2L, 6L, 9L, -4L, 1L, 0L, -6L, 10L),
             width=c( 5L, 0L, 6L,  1L, 4L, 3L,  2L,  3L))
coverage(x)
coverage(x, shift=7)
coverage(x, shift=7, width=27)
coverage(x, shift=c(-4, 2))  # shift gets recycled
coverage(x, shift=c(-4, 2), width=12)
coverage(x, shift=-max(end(x)))

coverage(restrict(x, 1, 10))
coverage(reduce(x), shift=7)
coverage(gaps(shift(x, 7), start=1, end=27))

## With weights:
coverage(x, weight=as.integer(10^(0:7)))  # integer-Rle
coverage(x, weight=c(2.8, -10))  # numeric-Rle, shift gets recycled

## ---------------------------------------------------------------------
## B. SOME MATHEMATICAL PROPERTIES OF THE coverage() FUNCTION
## ---------------------------------------------------------------------

## PROPERTY 1: The coverage vector is not affected by reordering the
## input ranges:
set.seed(24)
x <- IRanges(sample(1000, 40, replace=TRUE), width=17:10)
cvg0 <- coverage(x)
stopifnot(identical(coverage(sample(x)), cvg0))

## Of course, if the ranges are shifted and/or assigned weights, then
## this doesnt hold anymore, unless the shift and/or weight
## arguments are reordered accordingly.

## PROPERTY 2: The coverage of the concatenation of 2 Ranges objects x
## and y is the sum of the 2 individual coverage vectors:
y <- IRanges(sample(-20:280, 36, replace=TRUE), width=28)
stopifnot(identical(coverage(c(x, y), width=100),
                    coverage(x, width=100) + coverage(y, width=100)))

## Note that, because adding 2 vectors in R recycles the shortest to
## the length of the longest, the following is generally FALSE:
identical(coverage(c(x, y)), coverage(x) + coverage(y))  # FALSE
```

```
## It would only be TRUE if the 2 coverage vectors we add had the same
## length, which would only happen by chance. By using the same width
## value when we computed the 2 coverages previously, we made sure they
## had the same length.

## Because of properties 1 & 2, we have:
x1 <- x[c(TRUE, FALSE)]  # pick up 1st, 3rd, 5th, etc... ranges
x2 <- x[c(FALSE, TRUE)]  # pick up 2nd, 4th, 6th, etc... ranges
cvg1 <- coverage(x1, width=100)
cvg2 <- coverage(x2, width=100)
stopifnot(identical(coverage(x, width=100), cvg1 + cvg2))

## PROPERTY 3: Multiplying the weights by a scalar has the effect of
## multiplying the coverage vector by the same scalar:
weight <- runif(40)
cvg3 <- coverage(x, weight=weight)
stopifnot(all.equal(coverage(x, weight=-2.68 * weight), -2.68 * cvg3))

## Because of properties 1 & 2 & 3, we have:
stopifnot(identical(coverage(x, width=100, weight=c(5L, -11L)),
                    5L * cvg1 - 11L * cvg2))

## PROPERTY 4: Using the sum of 2 weight vectors produces the same
## result as using the 2 weight vectors separately and summing the
## 2 results:
weight2 <- 10 * runif(40) + 3.7
stopifnot(all.equal(coverage(x, weight=weight + weight2),
                    cvg3 + coverage(x, weight=weight2)))

## PROPERTY 5: Repeating any input range N number of times is
## equivalent to multiplying its assigned weight by N:
times <- sample(0:10L, length(x), replace=TRUE)
stopifnot(all.equal(coverage(rep(x, times), weight=rep(weight, times)),
                    coverage(x, weight=weight * times)))

## In particular, if weight is not supplied:
stopifnot(identical(coverage(rep(x, times)), coverage(x, weight=times)))

## PROPERTY 6: If none of the input range actually gets clipped during
## the "shift and clip" process, then:
##
##     sum(cvg) = sum(width(x) * weight)
##
stopifnot(sum(cvg3) == sum(width(x) * weight))

## In particular, if weight is not supplied:
stopifnot(sum(cvg0) == sum(width(x)))

## Note that this property is sometimes used in the context of a
## ChIP-Seq analysis to estimate "the number of reads in a peak", that
## is, the number of short reads that belong to a peak in the coverage
## vector computed from the genomic locations (a.k.a. genomic ranges)
```

```
## of the aligned reads. Because of property 6, the number of reads in
## a peak is approximately the area under the peak divided by the short
## read length.

## PROPERTY 7: If weight is not supplied, then disjoining or reducing
## the ranges before calling coverage() has the effect of "shaving" the
## coverage vector at elevation 1:
table(cvg0)
shaved_cvg0 <- cvg0
runValue(shaved_cvg0) <- pmin(runValue(cvg0), 1L)
table(shaved_cvg0)

stopifnot(identical(coverage(disjoin(x)), shaved_cvg0))
stopifnot(identical(coverage(reduce(x)), shaved_cvg0))

## ---------------------------------------------------------------------
## C. SOME SANITY CHECKS
## ---------------------------------------------------------------------
dummy.coverage <- function(x, shift=0L, width=NULL)
{
    y <- unlist(shift(x, shift))
    if (is.null(width))
        width <- max(c(0L, y))
    Rle(tabulate(y,  nbins=width))
}

check_real_vs_dummy <- function(x, shift=0L, width=NULL)
{
    res1 <- coverage(x, shift=shift, width=width)
    res2 <- dummy.coverage(x, shift=shift, width=width)
    stopifnot(identical(res1, res2))
}
check_real_vs_dummy(x)
check_real_vs_dummy(x, shift=7)
check_real_vs_dummy(x, shift=7, width=27)
check_real_vs_dummy(x, shift=c(-4, 2))
check_real_vs_dummy(x, shift=c(-4, 2), width=12)
check_real_vs_dummy(x, shift=-max(end(x)))

## With a set of distinct single positions:
x3 <- IRanges(sample(50000, 20000), width=1)
stopifnot(identical(sort(start(x3)), which(coverage(x3) != 0L)))

## ---------------------------------------------------------------------
## D. COVERAGE OF AN IRangesList OBJECT
## ---------------------------------------------------------------------
x <- IRangesList(A=IRanges(3*(4:-1), width=1:3), B=IRanges(2:10, width=5))
cvg <- coverage(x)
cvg

stopifnot(identical(cvg[[1]], coverage(x[[1]])))
stopifnot(identical(cvg[[2]], coverage(x[[2]])))
```

```
coverage(x, width=c(50, 9))
coverage(x, width=c(NA, 9))
coverage(x, width=9)  # width gets recycled

## Each list element in shift and weight gets recycled to the length
## of the corresponding element in x.
weight <- list(as.integer(10^(0:5)), -0.77)
cvg2 <- coverage(x, weight=weight)
cvg2  # 1st coverage vector is an integer-Rle, 2nd is a numeric-Rle

identical(mapply(coverage, x=x, weight=weight), as.list(cvg2))
```

DataFrame-class *External Data Frame*

Description

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

Details

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

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

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

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)).

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.

Constructor

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

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

Splitting and Combining

In the following code snippets, x is a DataFrame.

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

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

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

mstack(..., .index.var = "name"): Stacks the data frames passed as through ..., using .index.var as the index column name. See stack.

Aggregation

In the following code snippets, data is a DataFrame.

aggregate(x, data, FUN, ..., subset, na.action = na.omit): Aggregates the DataFrame data according to the formula x and the aggregating function FUN. See aggregate and its method for formula.

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.

Author(s)

Michael Lawrence

See Also

`DataTable`, `Vector`, and `RangedData`, which makes heavy use of this class.

Examples

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

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

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

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

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

```
# subsetting

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

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

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

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

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

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

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

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

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

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

## column replacement

df[["counts"]] <- counts
df[["counts"]]
df[[3]] <- score
df[["X"]]
```

```
df[[3]] <- NULL # deletion

## split

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

## rbind

do.call(rbind, as.list(swsplit))

## cbind

cbind(DataFrame(score), DataFrame(counts))
```

DataFrameList-class *List of DataFrames*

Description

Represents a list of DataFrame objects. The SplitDataFrameList class contains the additional restriction that all the columns be of the same name and type. Internally it is stored as a list of DataFrame objects and extends List.

Accessors

In the following code snippets, x is a DataFrameList.

dim(x): Get the two element integer vector indicating the number of rows and columns over the entire dataset.

dimnames(x): Get the list of two character vectors, the first holding the rownames (possibly NULL) and the second the column names.

columnMetadata(x): Get the DataFrame of metadata along the columns, i.e., where each column in x is represented by a row in the metadata. The metadata is common across all elements of x. Note that calling mcols(x) returns the metadata on the DataFrame elements of x.

columnMetadata(x) <- value: Set the DataFrame of metadata for the columns.

Subsetting

In the following code snippets, x is a SplitDataFrameList. In general x follows the conventions of SimpleList/CompressedList with the following addition:

x[i,j,drop]: If matrix subsetting is used, i selects either the list elements or the rows within the list elements as determined by the [method for SimpleList/CompressedList, j selects the columns, and drop is used when one column is selected and output can be coerced into an AtomicList or RangesList subclass.

x[i,j] <- value: If matrix subsetting is used, i selects either the list elements or the rows within the list elements as determined by the [<- method for SimpleList/CompressedList, j selects the columns and value is the replacement value for the selected region.

Constructor

`DataFrameList(...)`: Concatenates the `DataFrame` objects in `...` into a new `DataFrameList`.

`SplitDataFrameList(..., compress = TRUE, cbindArgs = FALSE)`: If `cbindArgs` is `FALSE`, the `...` arguments are coerced to `DataFrame` objects and concatenated to form the result. The arguments must have the same number and names of columns. If `cbindArgs` is `TRUE`, the arguments are combined as columns. The arguments must then be the same length, with each element of an argument mapping to an element in the result. If `compress = TRUE`, returns a `CompressedSplitDataFrameList`; else returns a `SimpleSplitDataFrameList`.

Combining

In the following code snippets, objects in `...` are of class `DataFrameList`.

`rbind(...)`: Creates a new `DataFrameList` containing the element-by-element row concatenation of the objects in `...`.

`cbind(...)`: Creates a new `DataFrameList` containing the element-by-element column concatenation of the objects in `...`.

Coercion

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

`as(from, "DataFrame")`: Coerces a `DataFrameList` to an `DataFrame` by combining the rows of the elements. This essentially unsplits the `DataFrame`. Every element of `x` must have the same columns.

`as(from, "SplitDataFrameList")`: By default, simply calls the `SplitDataFrameList` constructor on `from`. If `from` is a `List`, each element of `from` is passed as an argument to `SplitDataFrameList`, like calling `as.list` on a vector.

`as.data.frame(x, row.names=NULL, optional=FALSE, ...)`: Unsplits the `DataFrame` and coerces it to a `data.frame`, with the rownames specified in `row.names`. The `optional` argument is ignored.

`stack(x, index.var = "name")`: Unlists `x` and adds a column named `index.var` to the result, indicating the element of `x` from which each row was obtained.

Author(s)

Michael Lawrence

See Also

`DataFrame`, `RangedData`, which uses a `DataFrameList` to split the data by the spaces.

DataTable-API *The DataTable API*

Description

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

Accessors

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

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

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

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

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

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

Subsetting

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

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

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

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

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

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

select expression indicating columns to keep.

drop passed on to `[` indexing operator.

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

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

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

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

Combining

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

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

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

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

Looping

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

`aggregate(x, by, FUN, start = NULL, end = NULL, width = NULL, frequency = NULL, delta = NU`
Generates summaries on the specified windows and returns the result in a convenient form:

`by` An object with `start`, `end`, and `width` methods.

`FUN` The function, found via `match.fun`, to be applied to each window of `x`.

`start`, `end`, `width` the start, end, or width of the window. If by is missing, then must supply two of the three.

`frequency`, `delta` Optional arguments that specify the sampling frequency and increment within the window.

`...` Further arguments for `FUN`.

`simplify` A logical value specifying whether or not the result should be simplified to a vector or matrix if possible.

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

Utilities

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

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

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

Coercion

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

See Also

DataTable-stats for statistical functionality, like fitting regression models, data.frame

Examples

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

DataTable-stats *Statistical modeling with DataTable*

Description

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

Tabulation

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

See Also

`DataTable` for general manipulation, `DataFrame` for an implementation that mimics `data.frame`.

Examples

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

---

endoapply *Endomorphisms via application of a function over an object's elements*

Description

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

Usage

```
endoapply(X, FUN, ...)

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

Arguments

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 endoapply, optional arguments to FUN. For mendoapply, a set of list, data.frame or List objects to compute over.
MoreArgs	a list of other arguments to FUN.

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 ....

See Also

lapply, mapply

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)
```

---

expand *The expand method for uncompressing compressed data columns*

Description

Expand an object with compressed columns such that all compressed values are represented as separate rows.

Usage

```
## S4 method for signature DataFrame
expand(x, colnames, keepEmptyRows, ...)
```

Arguments

x	A DataFrame containing some columns that are compressed (e.g., CompressedCharacterList).
colnames	A character or numeric vector containing the names or indices of the compressed columns to expand. The order of expansion is controlled by the column order in this vector.
keepEmptyRows	A logical indicating if rows containing empty values in the specified colnames should be retained or dropped. When TRUE, empty values are set to NA and all rows are kept. When FALSE, rows with empty values in the colnames columns are dropped.
...	Arguments passed to other methods.

Value

A DataFrame that has been expanded row-wise to match the dimension of the uncompressed columns.

Author(s)

Herve Pages and Marc Carlson

See Also

DataFrame-class

Examples

```
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 the aa column only, and keep rows adjacent to empty values
expand(df, colnames="aa", keepEmptyRows=TRUE)

## expand the aa column only but do not keep rows
expand(df, colnames="aa", keepEmptyRows=FALSE)

## expand the aa and then the bb column, but
## keeping rows next to empty compressed values
expand(df, colnames=c("aa","bb"),  keepEmptyRows=TRUE)

## expand the bb and then the aa column, but dont keep rows adjacent to
## empty values from bb and aa
expand(df, colnames=c("aa","bb"), keepEmptyRows=FALSE)
```

extractList *Group elements of a vector-like object into a list-like object*

Description

relist and split are 2 common ways of grouping the elements of a vector-like object into a list-like object. The **IRanges** package defines relist and split methods that operate on a Vector object and return a List object.

Because relist and split both impose severe restrictions on the kind of grouping that they support (e.g. every element in the input object needs to go in a group and can only go in one group), the **IRanges** package introduces the extractList generic function for performing *arbitrary* groupings.

relist, split, and extractList have in common that they return a list-like value where each list element has the same class as the original vector-like object. Thus they need to be able to select the appropriate List concrete subclass to use for this returned value. This selection is performed by relistToClass and is based only on the class of the original object.

Usage

```
## relist()
## --------

## S4 method for signature ANY,List
relist(flesh, skeleton)
## S4 method for signature Vector,list
relist(flesh, skeleton)

## splitAsList() and split()
## -------------------------

splitAsList(x, f, drop=FALSE)

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

## extractList()
## -------------

extractList(x, i)

## relistToClass()
## ---------------------------

relistToClass(x)
```

Arguments

flesh, x	A vector-like object.

skeleton	A list-like object. Only the "shape" (i.e. element lengths) of skeleton matters. Its exact content is ignored.
f	An atomic vector or a factor (possibly in Rle form).
drop	Logical indicating if levels that do not occur should be dropped (if f is a factor).
i	A list-like object. Unlike for skeleton, the content here matters (see Details section below). Note that i can be a Ranges object (a particular type of list-like object), and, in that case, extractList is particularly fast (this is a common use case).

Details

By default, `extractList(x, i)` is equivalent to:

```
relist(x[unlist(i)], i)
```

An exception is made when x is a data-frame-like object. In that case x is subsetted along the rows, that is, `extractList(x, i)` is equivalent to:

```
relist(x[unlist(i), ], i)
```

This is more or less how the default method is implemented, except for some optimizations when i is a Ranges object.

relist and split can be seen as specialized versions of extractList:

```
relist(flesh, skeleton) is equivalent to
extractList(flesh, PartitioningByEnd(skeleton))

split(x, f) is equivalent to
extractList(x, split(seq_along(f), f))
```

It is good practise to use extractList only for cases not covered by relist or split. Whenever possible, using relist or split is preferred as they will always perform more efficiently. In addition their names carry meaning and are familiar to most R users/developers so they'll make your code easier to read/understand.

Note that the transformation performed by relist or split is always reversible (via unlist and unsplit, respectively), but the transformation performed by extractList is not.

Value

The relist method behaves like utils::relist except that it returns a List object. If skeleton has names, then they are propagated to the returned value.

splitAsList and the split method behave like base::split except that they return a List object. The difference between splitAsList and split is that the former always returns a List object while the latter can return an ordinary list (e.g. when x and f are ordinary vectors and/or factors).

extractList returns a list-like object parallel to i and with the same "shape" as i (i.e. same element lengths). If i has names, then they are propagated to the returned value.

All these functions (except relistToClass) return a list-like object where the list elements have the same class as x. relistToClass gives the exact class of the returned object.

Author(s)

H. Pages

See Also

- The unlist and relist functions in the **base** and **utils** packages, respectively.
- The split and unsplit functions in the **base** package.
- Vector and List objects.
- Ranges, Rle and DataFrame objects.

Examples

```
## On an Rle object:
x <- Rle(101:105, 6:2)
i <- IRanges(6:10, 16:12, names=letters[1:5])
extractList(x, i)

## On a DataFrame object:
df <- DataFrame(X=x, Y=LETTERS[1:20])
extractList(df, i)
```

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-class *Collection of Filter Rules*

Description

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

Details

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

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

Accessor methods

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

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

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

Constructor

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

Subsetting and Replacement

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

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

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

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

Combining

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

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

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

Evaluating

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

`evalSeparately(expr, envir = parent.frame(), enclos = if (is.list(envir) || is.pairlist(envir)) 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

`rdapply`, which accepts a `FilterRules` instance to filter each space before invoking the user function.

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) # NAs are dropped
active(filters) <- c("peaks", NA)
```

---

findOverlaps-methods *Finding overlapping ranges*

Description

Various methods for finding/counting interval overlaps between two "range-based" objects: a query and a subject.

NOTE: This man page describes the methods that operate on a query and a subject that are both either a Ranges, Views, RangesList, ViewsList, or RangedData object. (In addition, if the query is a Ranges object, the subject can be an IntervalTree object; if the query is a RangesList object, the subject can be a IntervalForest object. And if the subject is a Ranges object, the query can be an integer vector.)

See ?findOverlaps,GenomicRanges,GenomicRanges-method in the GenomicRanges package for methods that operate on GRanges or GRangesList objects. See also the ?GIntervalTree class and the ?findOverlaps,GenomicRanges,GIntervalTree-method method for finding overlaps with persistent IntervalForest objects.

Usage

```
findOverlaps(query, subject, maxgap=0L, minoverlap=1L,
             type=c("any", "start", "end", "within", "equal"),
             select=c("all", "first", "last", "arbitrary"), ...)

countOverlaps(query, subject, maxgap=0L, minoverlap=1L,
              type=c("any", "start", "end", "within", "equal"), ...)

overlapsAny(query, subject, maxgap=0L, minoverlap=1L,
            type=c("any", "start", "end", "within", "equal"), ...)
query %over% subject
query %within% subject
```

```
query %outside% subject

subsetByOverlaps(query, subject, maxgap=0L, minoverlap=1L,
                 type=c("any", "start", "end", "within", "equal"), ...)

## S4 method for signature Hits
ranges(x, query, subject)
```

Arguments

`query, subject`	Each of them can be a Ranges, Views, RangesList, ViewsList, or RangedData object. In addition, if `query` is a Ranges object, `subject` can be an IntervalTree object; if `query` is a RangesList object, then `subject` can be an IntervalForest object. And if `subject` is a Ranges object, `query` can be an integer vector to be converted to length-one ranges. If `query` is a RangesList or RangedData, `subject` must be a RangesList or RangedData. If both lists have names, each element from the subject is paired with the element from the query with the matching name, if any. Otherwise, elements are paired by position. The overlap is then computed between the pairs as described below. If `query` is unsorted, it is sorted first, so it is usually better to sort up-front, to avoid a sort with each `findOverlaps` call. If `subject` is omitted, `query` is queried against itself. In this case, and only this case, the `ignoreSelf` and `ignoreRedundant` arguments are allowed. By default, the result will contain hits for each range against itself, and if there is a hit from A to B, there is also a hit for B to A. If `ignoreSelf` is `TRUE`, all self matches are dropped. If `ignoreRedundant` is `TRUE`, only one of A->B and B->A is returned.
`maxgap, minoverlap`	Intervals with a separation of `maxgap` or less and a minimum of `minoverlap` overlapping positions, allowing for `maxgap`, are considered to be overlapping. `maxgap` should be a scalar, non-negative, integer. `minoverlap` should be a scalar, positive integer.
`type`	By default, any overlap is accepted. By specifying the `type` parameter, one can select for specific types of overlap. The types correspond to operations in Allen's Interval Algebra (see references). If `type` is `start` or `end`, the intervals are required to have matching starts or ends, respectively. While this operation seems trivial, the naive implementation using `outer` would be much less efficient. Specifying `equal` as the type returns the intersection of the `start` and `end` matches. If `type` is `within`, the query interval must be wholly contained within the subject interval. Note that all matches must additionally satisfy the `minoverlap` constraint described above. The `maxgap` parameter has special meaning with the special overlap types. For `start`, `end`, and `equal`, it specifies the maximum difference in the starts, ends or both, respectively. For `within`, it is the maximum amount by which the query may be wider than the subject.
`select`	When `select` is `"all"` (the default), the results are returned as a Hits object. When `select` is `"first"`, `"last"`, or `"arbitrary"` the results are returned as

an integer vector of length query containing the first, last, or arbitrary overlapping interval in subject, with NA indicating intervals that did not overlap any intervals in subject.

If select is "all", a Hits object is returned. For all other select the return value depends on the drop argument. When select != "all" && !drop, an IntegerList is returned, where each element of the result corresponds to a space in query. Whenselect != "all" && drop, an integer vector is returned containing indices that are offset to align with the unlisted query.

... Further arguments to be passed to or from other methods:

- drop: All methods accept the drop argument (FALSE by default). See select argument above for the details.
- ignoreSelf, ignoreRedundant: When subject is omitted, the ignoreSelf and ignoreRedundant arguments (both FALSE by default) are allowed. See query and subject arguments above for the details.

x Hits object returned by findOverlaps.

Details

A common type of query that arises when working with intervals is finding which intervals in one set overlap those in another.

The simplest approach is to call the findOverlaps function on a Ranges or other object with range information (aka "range-based object").

An IntervalTree object is a derivative of Ranges and stores its ranges as a tree that is optimized for overlap queries. Thus, for repeated queries against the same subject, it is more efficient to create an IntervalTree once for the subject using the IntervalTree constructor and then perform the queries against the IntervalTree instance. An IntervalForest object is a derivative of RangesList and stores its ranges as a set of trees optimizized for partitioned overlap queries. Again, for repeated queries against the same subject list, it is more efficient to create an IntervalForest once and then perform the queries against the IntervalForest instance.

Value

findOverlaps returns either a Hits object when select="all" (the default), or an integer vector when select is not "all". For RangesList objects it returns a HitsList-class object when select="all", or an IntegerList when select is not "all". When subject is an IntervalForest object, it returns a CompressedHitsList or CompressedIntegerList respectively.

countOverlaps returns the overlap hit count for each range in query using the specified findOverlaps parameters. For RangesList objects, it returns an IntegerList object. When subject is an IntervalForest it returns a CompressedIntegerList.

overlapsAny finds the ranges in query that overlap any of the ranges in subject. For Ranges or Views objects, it returns a logical vector of length equal to the number of ranges in query. For RangesList, RangedData, or ViewsList objects, it returns a LogicalList object, where each element of the result corresponds to a space in query. When subject is an IntervalForest object, it returns a CompressedLogicalList object.

%over% and %within% are convenience wrappers for the 2 most common use cases. Currently defined as %over% <- function(query, subject) overlapsAny(query, subject) and %within% <- function(query, su overlapsAny(query, subject, type="within"). %outside% is simply the inverse of %over%.

`subsetByOverlaps` returns the subset of `query` that has an overlap hit with a range in `subject` using the specified `findOverlaps` parameters.

`ranges(x, query, subject)` returns a `Ranges` of the same length as Hits object `x` holding the regions of intersection between the overlapping ranges in objects `query` and `subject`, which should be the same query and subject used in the call to `findOverlaps` that generated `x`.

Author(s)

Michael Lawrence with contributions by Hector Corrada Bravo

References

Allen's Interval Algebra: James F. Allen: Maintaining knowledge about temporal intervals. In: Communications of the ACM. 26/11/1983. ACM Press. S. 832-843, ISSN 0001-0782

See Also

- The Hits and HitsList classes for representing a set of hits between 2 vector-like objects.
- findOverlaps,GenomicRanges,GenomicRanges-method in the GenomicRanges package for methods that operate on GRanges or GRangesList objects.
- findOverlaps,GenomicRanges,GIntervalTree-method in the GenomicRanges package for methods that use IntervalForest objects to find overlaps.
- The `IntervalTree` class and constructor.
- The `IntervalForest` class and constructor.
- The Ranges, Views, RangesList, ViewsList, and RangedData classes.
- The IntegerList and LogicalList classes.

Examples

```
query <- IRanges(c(1, 4, 9), c(5, 7, 10))
subject <- IRanges(c(2, 2, 10), c(2, 3, 12))
tree <- IntervalTree(subject)

## ---------------------------------------------------------------------
## findOverlaps()
## ---------------------------------------------------------------------

## at most one hit per query
findOverlaps(query, tree, select = "first")
findOverlaps(query, tree, select = "last")
findOverlaps(query, tree, select = "arbitrary")

## overlap even if adjacent only
## (FIXME: the gap between 2 adjacent ranges should be still considered
## 0. So either we have an argument naming problem, or we should modify
## the handling of the maxgap argument so that the user would need to
## specify maxgap = 0L to obtain the result below.)
findOverlaps(query, tree, maxgap = 1L)
```

```
## shortcut
findOverlaps(query, subject)

query <- IRanges(c(1, 4, 9), c(5, 7, 10))
subject <- IRanges(c(2, 2), c(5, 4))
tree <- IntervalTree(subject)

## one Ranges with itself
findOverlaps(query)

## single points as query
subject <- IRanges(c(1, 6, 13), c(4, 9, 14))
findOverlaps(c(3L, 7L, 10L), subject, select = "first")

## alternative overlap types
query <- IRanges(c(1, 5, 3, 4), width=c(2, 2, 4, 6))
subject <- IRanges(c(1, 3, 5, 6), width=c(4, 4, 5, 4))

findOverlaps(query, subject, type = "start")
findOverlaps(query, subject, type = "start", maxgap = 1L)
findOverlaps(query, subject, type = "end", select = "first")
ov <- findOverlaps(query, subject, type = "within", maxgap = 1L)
ov

## ---------------------------------------------------------------------
## overlapsAny()
## ---------------------------------------------------------------------

overlapsAny(query, subject, type="start")
overlapsAny(query, subject, type="end")
query %over% subject    # same as overlapsAny(query, subject)
query %within% subject  # same as overlapsAny(query, subject,
                        #                     type="within")

## ---------------------------------------------------------------------
## "ranges" METHOD FOR Hits OBJECTS
## ---------------------------------------------------------------------

## extract the regions of intersection between the overlapping ranges
ranges(ov, query, subject)

## ---------------------------------------------------------------------
## using IntervalForest objects
## ---------------------------------------------------------------------
query <- IRanges(c(1, 4, 9), c(5, 7, 10))
qpartition <- factor(c("a","a","b"))
qlist <- split(query, qpartition)

subject <- IRanges(c(2, 2, 10), c(2, 3, 12))
spartition <- factor(c("a","a","b"))
slist <- split(subject, spartition)

forest <- IntervalForest(slist)
```

```
## at most one hit per query
findOverlaps(qlist, forest, select = "first")
findOverlaps(qlist, forest, select = "last")
findOverlaps(qlist, forest, select = "arbitrary")

query <- IRanges(c(1, 5, 3, 4), width=c(2, 2, 4, 6))
qpartition <- factor(c("a","a","b","b"))
qlist <- split(query, qpartition)

subject <- IRanges(c(1, 3, 5, 6), width=c(4, 4, 5, 4))
spartition <- factor(c("a","a","b","b"))
slist <- split(subject, spartition)
forest <- IntervalForest(slist)

overlapsAny(qlist, forest, type="start")
overlapsAny(qlist, forest, type="end")
qlist

subsetByOverlaps(qlist, forest)
countOverlaps(qlist, forest)
```

---

funprog-methods *Functional programming methods for List objects*

Description

The R base package defines some higher-order functions that are commonly found in Functional Programming Languages. See ?Reduce for the details, and, in particular, for a description of their arguments. The IRanges 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.

Usage

```
## 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_)
```

Arguments

f, init, right, accumulate, nomatch	See ?base::Reduce for a description of these arguments.
x	A List object.
...	One or more List objects. (FIXME: Mixing List objects with ordinary lists doesn't seem to work properly at the moment.)

Author(s)

P. Aboyoun

See Also

- The List class.
- The IntegerList class and constructor for an example of a List subclass.
- Reduce for a full description of what these functions do and what they return.

Examples

```
x <- IntegerList(a=1:3, b=16:11, c=22:21, d=31:36)
x

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)
```

GappedRanges-class *GappedRanges objects*

Description

The GappedRanges class is a vector-like container for storing a set of "gapped ranges".

Details

A "gapped range" is conceptually the union of 1 or more non-overlapping (and non-empty) ranges ordered from left to right. More precisely, a "gapped range" can be represented by a normal IRanges object of length >= 1. In particular normality here ensures that the individual ranges are non-empty and are separated by non-empty gaps. The start of a "gapped range" is the start of its first range. The end of a "gapped range" is the end of its last range. If we ignore the gaps, then a GappedRanges object can be seen as a Ranges object.

Constructor

No constructor function is provided for GappedRanges objects. The coercion methods described below can be used to create GappedRanges objects.

Coercion

`as(from, "GappedRanges")`: Turns a CompressedNormalIRangesList or CompressedIRangesList object into a GappedRanges object.

`as(from, "RangesList")`: Turns a GappedRanges object into a RangesList object (more precisely the result will be a CompressedNormalIRangesList object).

Accessor methods

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

`length(x)`: Returns the number of "gapped ranges" in `x`.

`start(x), end(x)`: Returns an integer vector of length `length(x)` containing the start and end (respectively) of each "gapped range" in `x`. See Details section above for the exact definitions of the start and end of a "gapped range".

`width(x)`: Defined as `end(x) - start(x) + 1L`.

`ngap(x)`: Returns an integer vector of length `length(x)` containing the number of gaps for each "gapped range" in `x`. Equivalent to `elementLengths(x) - 1L`.

`names(x)`: `NULL` or a character vector of length `length(x)`.

Subsetting and related operations

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

`x[i]`: Returns a new GappedRanges object made of the selected "gapped ranges". `i` can be a numeric, character or logical vector, or any of the types supported by the `[` method for CompressedNormalIRangesList objects.

`x[[i]]`: Returns the NormalIRanges object representing the i-th element in `x`. Equivalent to `as(from, "RangesList")[[i]]`. `i` can be a single numeric value or a single character string.

`elemenType(x)`: Returns the type of `x[[i]]` as a single string (always `"NormalIRanges"`). Note that the semantic of the `[[` method for GappedRanges objects is different from the semantic of the method for Ranges objects (the latter returns an integer vector).

`elementLengths(x)`: Semantically equivalent to

```
sapply(seq_len(length(x)), function(i) length(x[[i]]))
```

but much faster. Note that the semantic of the `elementLengths` method for GappedRanges objects is different from the semantic of the method for Ranges objects (the latter returns the `width` of the Ranges object).

Combining and related operations

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

`c(x, ...)`: Combine `x` and the GappedRanges objects in `...` together. The result is an object of the same class as `x`.

Author(s)

H. Pages

See Also

Ranges-class, CompressedNormalIRangesList-class

Examples

```
## The 3 following IRanges objects are normal. Each of them will be
## stored as a "gapped range" in the GappedRanges object gr.
ir1 <- IRanges(start=c(11, 21, 23), end=c(15, 21, 30))
ir2 <- IRanges(start=-2, end=15)
ir3 <- IRanges(start=c(-2, 21), end=c(10, 22))
irl <- IRangesList(ir1, ir2, ir3)

gr <- as(irl, "GappedRanges")
gr

length(gr)
start(gr)
end(gr)
width(gr)
ngap(gr)
gr[-1]
gr[ngap(gr) >= 1]
gr[[1]]
as.integer(gr[[1]])
gr[[2]]
as.integer(gr[[2]])
as(gr, "RangesList")
start(as(gr, "RangesList"))  # not the same as start(gr)
```

Grouping-class *Grouping objects*

Description

In this man page, we call "grouping" the action of dividing a collection of NO objects into NG groups (some of which may be empty). The Grouping class and subclasses are containers for representing groupings.

The Grouping core API

Let's give a formal description of the Grouping core API:

Groups G_i are indexed from 1 to NG (1 <= i <= NG).

Objects O_j are indexed from 1 to NO (1 <= j <= NO).

Every object must belong to one group and only one.

Given that empty groups are allowed, NG can be greater than NO.

Grouping an empty collection of objects (NO = 0) is supported. In that case, all the groups are empty. And only in that case, NG can be zero too (meaning there are no groups).

If `x` is a Grouping object:

`length(x)`: Returns the number of groups (NG).

`names(x)`: Returns the names of the groups.

`nobj(x)`: Returns the number of objects (NO). Equivalent to `length(togroup(x))`.

Going from groups to objects:

`x[[i]]`: Returns the indices of the objects (the j's) that belong to G_i. The j's are returned in ascending order. This provides the mapping from groups to objects (one-to-many mapping).

`grouplength(x, i=NULL)`: Returns the number of objects in G_i. Works in a vectorized fashion (unlike `x[[i]]`). `grouplength(x)` is equivalent to `grouplength(x, seq_len(length(x)))`. If `i` is not NULL, `grouplength(x, i)` is equivalent to `sapply(i, function(ii) length(x[[ii]]))`.

`members(x, i)`: Equivalent to `x[[i]]` if `i` is a single integer. Otherwise, if `i` is an integer vector of arbitrary length, it's equivalent to `sort(unlist(sapply(i, function(ii) x[[ii]])))`.

`vmembers(x, L)`: A version of `members` that works in a vectorized fashion with respect to the `L` argument (`L` must be a list of integer vectors). Returns `lapply(L, function(i) members(x, i))`.

Going from objects to groups:

`togroup(x, j=NULL)`: Returns the index i of the group that O_j belongs to. This provides the mapping from objects to groups (many-to-one mapping). Works in a vectorized fashion. `togroup(x)` is equivalent to `togroup(x, seq_len(nobj(x)))`: both return the entire mapping in an integer vector of length NO. If `j` is not NULL, `togroup(x, j)` is equivalent to `y <- togroup(x); y[j]`.

`togrouplength(x, j=NULL)`: Returns the number of objects that belong to the same group as O_j (including O_j itself). Equivalent to `grouplength(x, togroup(x, j))`.

Given that length, names and [[are defined for Grouping objects, those objects can be considered List objects. In particular, as.list works out-of-the-box on them.

One important property of any Grouping object x is that unlist(as.list(x)) is always a permutation of seq_len(nobj(x)). This is a direct consequence of the fact that every object in the grouping belongs to one group and only one.

The H2LGrouping and Dups subclasses

DOCUMENT ME

The Partitioning subclass

A Partitioning container represents a block-grouping, i.e. a grouping where each group contains objects that are neighbors in the original collection of objects. More formally, a grouping x is a block-grouping iff togroup(x) is sorted in increasing order (not necessarily strictly increasing).

A block-grouping object can also be seen (and manipulated) as a Ranges object where all the ranges are adjacent starting at 1 (i.e. it covers the 1:NO interval with no overlap between the ranges).

Note that a Partitioning object is both: a particular type of Grouping object and a particular type of Ranges object. Therefore all the methods that are defined for Grouping and Ranges objects can also be used on a Partitioning object. See ?Ranges for a description of the Ranges API.

The Partitioning class is virtual with 2 concrete subclasses: PartitioningByEnd (only stores the end of the groups, allowing fast mapping from groups to objects), and PartitioningByWidth (only stores the width of the groups).

Constructors

H2LGrouping(high2low=integer()): [DOCUMENT ME]

Dups(high2low=integer()): [DOCUMENT ME]

PartitioningByEnd(x=integer(), NG=NULL, names=NULL): x must be either a list-like object or a sorted integer vector. NG must be either NULL or a single integer. names must be either NULL or a character vector of length NG (if supplied) or length(x) (if NG is not supplied).

Returns the following PartitioningByEnd object y:

- If x is a list-like object, then the returned object y has the same length as x and is such that width(y) is identical to elementLengths(x).
- If x is an integer vector and NG is not supplied, then x must be sorted (checked) and contain non-NA non-negative values (NOT checked). The returned object y has the same length as x and is such that end(y) is identical to x.
- If x is an integer vector and NG is supplied, then x must be sorted (checked) and contain values >= 1 and <= NG (checked). The returned object y is of length NG and is such that togroup(y) is identical to x.

If the names argument is supplied, it is used to name the partitions.

PartitioningByWidth(x=integer(), NG=NULL, names=NULL): x must be either a list-like object or an integer vector. NG must be either NULL or a single integer. names must be either NULL or a character vector of length NG (if supplied) or length(x) (if NG is not supplied).

Returns the following PartitioningByWidth object y:

- If x is a list-like object, then the returned object y has the same length as x and is such that width(y) is identical to elementLengths(x).
- If x is an integer vector and NG is not supplied, then x must contain non-NA non-negative values (NOT checked). The returned object y has the same length as x and is such that width(y) is identical to x.
- If x is an integer vector and NG is supplied, then x must be sorted (checked) and contain values >= 1 and <= NG (checked). The returned object y is of length NG and is such that togroup(y) is identical to x.

If the names argument is supplied, it is used to name the partitions.

Note that these constructors don't recycle their names argument (to remain consistent with what names<- does on standard vectors).

Author(s)

H. Pages

See Also

List-class, Ranges-class, IRanges-class, successiveIRanges, cumsum, diff

Examples

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

## ---------------------------------------------------------------------
## A. H2LGrouping OBJECTS
## ---------------------------------------------------------------------
high2low <- c(NA, NA, 2, 2, NA, NA, NA, 6, NA, 1, 2, NA, 6, NA, NA, 2)
h2l <- H2LGrouping(high2low)
h2l

## The Grouping core API:
length(h2l)
nobj(h2l)  # same as length(h2l) for H2LGrouping objects
h2l[[1]]
h2l[[2]]
h2l[[3]]
h2l[[4]]
h2l[[5]]
grouplength(h2l)  # same as unname(sapply(h2l, length))
grouplength(h2l, 5:2)
members(h2l, 5:2)  # all the members are put together and sorted
togroup(h2l)
togroup(h2l, 5:2)
togrouplength(h2l)  # same as grouplength(h2l, togroup(h2l))
togrouplength(h2l, 5:2)

## The List API:
as.list(h2l)
sapply(h2l, length)
```

```
## ---------------------------------------------------------------------
## B. Dups OBJECTS
## ---------------------------------------------------------------------
dups1 <- as(h2l, "Dups")
dups1
duplicated(dups1)  # same as duplicated(togroup(dups1))

### The purpose of a Dups object is to describe the groups of duplicated
### elements in a vector-like object:
x <- c(2, 77, 4, 4, 7, 2, 8, 8, 4, 99)
x_high2low <- high2low(x)
x_high2low  # same length as x
dups2 <- Dups(x_high2low)
dups2
togroup(dups2)
duplicated(dups2)
togrouplength(dups2)  # frequency for each element
table(x)

## ---------------------------------------------------------------------
## C. Partitioning OBJECTS
## ---------------------------------------------------------------------
pbe1 <- PartitioningByEnd(c(4, 7, 7, 8, 15), names=LETTERS[1:5])
pbe1  # the 3rd partition is empty

## The Grouping core API:
length(pbe1)
nobj(pbe1)
pbe1[[1]]
pbe1[[2]]
pbe1[[3]]
grouplength(pbe1)  # same as unname(sapply(pbe1, length)) and width(pbe1)
togroup(pbe1)
togrouplength(pbe1)  # same as grouplength(pbe1, togroup(pbe1))
names(pbe1)

## The Ranges core API:
start(pbe1)
end(pbe1)
width(pbe1)

## The List API:
as.list(pbe1)
sapply(pbe1, length)

## Replacing the names:
names(pbe1)[3] <- "empty partition"
pbe1

## Coercion to an IRanges object:
as(pbe1, "IRanges")

## Other examples:
```

```
PartitioningByEnd(c(0, 0, 19), names=LETTERS[1:3])
PartitioningByEnd()  # no partition
PartitioningByEnd(integer(9))  # all partitions are empty
x <- c(1L, 5L, 5L, 6L, 8L)
pbe2 <- PartitioningByEnd(x, NG=10L)
stopifnot(identical(togroup(pbe2), x))
pbw2 <- PartitioningByWidth(x, NG=10L)
stopifnot(identical(togroup(pbw2), x))

## ---------------------------------------------------------------------
## D. RELATIONSHIP BETWEEN Partitioning OBJECTS AND successiveIRanges()
## ---------------------------------------------------------------------
mywidths <- c(4, 3, 0, 1, 7)

## The 3 following calls produce the same ranges:
ir <- successiveIRanges(mywidths)  # IRanges instance.
pbe <- PartitioningByEnd(cumsum(mywidths))  # PartitioningByEnd instance.
pbw <- PartitioningByWidth(mywidths)  # PartitioningByWidth instance.
stopifnot(identical(as(ir, "PartitioningByEnd"), pbe))
stopifnot(identical(as(ir, "PartitioningByWidth"), pbw))
```

Hits-class *Set of hits between 2 vector-like objects*

Description

The Hits class stores a set of "hits" between the elements in one vector-like object (called the "query") and the elements in another (called the "subject"). Currently, Hits are used to represent the result of a call to findOverlaps, though other operations producing "hits" are imaginable.

Details

The as.matrix and as.data.frame methods coerce a Hits object to a two column matrix or data.frame with one row for each hit, where the value in the first column is the index of an element in the query and the value in the second column is the index of an element in the subject.

The as.table method counts the number of hits for each query element and outputs the counts as a table.

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

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 query index (first column) and subject index (second column).

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

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

`as.table(x)`: counts the number of hits for each query element in `x` and outputs the counts as a `table`.

`t(x)`: Interchange the query and subject in `x`, returns a transposed `Hits`.

`as.list(x)`: Returns a list with an element for each query, where each element contains the indices of the subjects that have a hit with the corresponding query.

`as(x, "List")`: Like `as.list`, above.

Subsetting

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

Accessors

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

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

`countQueryHits(x)`: Counts the number of hits for each query, returning an integer vector.

`countSubjectHits(x)`: Counts the number of hits for each subject, returning an integer vector.

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

`queryLength(x)`, `nrow(x)`: get the number of elements in the query

`subjectLength(x)`, `ncol(x)`: get the number of elements in the subject

Other operations

`remapHits(x, query.map=NULL, new.queryLength=NA, subject.map=NULL, new.subjectLe`
Remaps the hits in `x` thru a "query map" and/or a "subject map" map. The query hits are remapped thru the "query map", which is specified via the `query.map` and `new.queryLength` arguments. The subject hits are remapped thru the "subject map", which is specified via the `subject.map` and `new.subjectLength` arguments.

The "query map" is conceptually a function (in the mathematical sense) and is also known as the "mapping function". It must be defined on the 1..M interval and take values in the 1..N interval, where N is `queryLength(x)` and M is the value specified by the user via the `new.queryLength` 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 `query.map` can be NULL (identity map), or a vector of `queryLength(x)` non-NA integers that are >= 1 and <= `new.queryLength`, or a factor of length `queryLength(x)` with no NAs (a factor is treated as an integer vector, and, if missing, `new.queryLength` 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 apply to the "subject map".

`remapHits` returns a Hits object where all the query and subject hits (accessed with `queryHits` and `subjectHits`, respectively) 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).

Author(s)

Michael Lawrence

See Also

findOverlaps, which generates an instance of this class. setops-methods for set operations on Hits objects.

Examples

```
query <- IRanges(c(1, 4, 9), c(5, 7, 10))
subject <- IRanges(c(2, 2, 10), c(2, 3, 12))
tree <- IntervalTree(subject)
overlaps <- findOverlaps(query, tree)

as.matrix(overlaps)
as.data.frame(overlaps)

as.table(overlaps) # hits per query
as.table(t(overlaps)) # hits per subject

hits1 <- remapHits(overlaps, subject.map=factor(c("e", "e", "d"), letters[1:5]))
hits1
hits2 <- remapHits(overlaps, subject.map=c(5, 5, 4), new.subjectLength=5)
hits2
stopifnot(identical(hits1, hits2))
```

HitsList-class *List of Hits objects*

Description

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

Details

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

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

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

To transpose a HitsList 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.

When the HitsList object is the result of a call to findOverlaps on two RangesList objects, the actual regions of intersection between the overlapping ranges can be obtained with the ranges accessor.

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.

Accessors

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

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

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

ranges(x, query, subject): returns a RangesList holding the intersection of the ranges in the RangesList objects query and subject, which should be the same subject and query used in the call to findOverlaps that generated x. Eventually, we might store the query and subject inside x, in which case the arguments would be redundant.

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, which generates an instance of this class.

---

inter-range-methods *Inter range transformations of a Ranges, Views, RangesList, MaskCollection, or RangedData object*

Description

Except for `isDisjoint()` and `disjointBins()`, all the transformations described in this man page are *endomorphisms* that operate on a single "range-based" object, that is, they transform the ranges contained in the input object and return them in an object of the *same class* as the input object.

Range-based endomorphisms are grouped in 2 categories:

1. Intra range transformations like `shift()` that transform each range individually (and independently of the other ranges) and return an object of the *same length* as the input object. Those transformations are described in the intra-range-methods man page (see `?intra-range-methods`).
2. Inter range transformations like `reduce()` that transform all the ranges together as a set to produce a new set of ranges and return an object not necessarily of the same length as the input object. Those transformations are described in this man page.

Usage

```
## range()
## -------
## S4 method for signature Ranges
range(x, ..., na.rm=FALSE)

## S4 method for signature RangesList
range(x, ..., na.rm=FALSE)

## reduce()
## --------
reduce(x, ...)

## S4 method for signature Ranges
reduce(x, drop.empty.ranges=FALSE, min.gapwidth=1L,
       with.revmap=FALSE, with.mapping=FALSE, with.inframe.attrib=FALSE)

## S4 method for signature Views
reduce(x, drop.empty.ranges=FALSE, min.gapwidth=1L,
       with.revmap=FALSE, with.mapping=FALSE, with.inframe.attrib=FALSE)

## S4 method for signature RangesList
reduce(x, drop.empty.ranges=FALSE, min.gapwidth=1L,
       with.revmap=FALSE, with.mapping=FALSE, with.inframe.attrib=FALSE)

## S4 method for signature RangedData
reduce(x, by=character(), drop.empty.ranges=FALSE,
       min.gapwidth=1L, with.inframe.attrib=FALSE)

## gaps()
## ------
gaps(x, start=NA, end=NA)

## disjoin()
```

```
## ---------
disjoin(x, ...)

## isDisjoint()
## ------------
isDisjoint(x, ...)

## disjointBins()
## --------------
disjointBins(x, ...)
```

Arguments

x	A Ranges, Views, RangesList, MaskCollection, or RangedData object.
...	For range, additional Ranges or RangesList to consider.
na.rm	Ignored.
drop.empty.ranges	TRUE or FALSE. Should empty ranges be dropped?
min.gapwidth	Ranges separated by a gap of at least min.gapwidth positions are not merged.
with.revmap	TRUE or FALSE. Should the mapping from reduced to original ranges be stored in the returned object? If yes, then it is stored as metadata column "revmap" of type IntegerList.
with.mapping	Deprecated. Please use the with.revmap argument instead.
with.inframe.attrib	TRUE or FALSE. For internal use.
by	A character vector.
start, end	• If x is a Ranges or Views object: A single integer or NA. Use these arguments to specify the interval of reference i.e. which interval the returned gaps should be relative to. • If x is a RangesList object: Integer vectors containing the coordinate bounds for each RangesList top-level element.

Details

Here we start by describing how each transformation operates on a Ranges object x.

range first combines x and the arguments in .... If the combined IRanges object contains at least 1 range, then range returns an IRanges instance with a single range, from the minimum start to the maximum end of the combined object. Otherwise (i.e. if the combined object contains no range), IRanges() is returned (i.e. an IRanges instance of length 0).

If x is a RangedData object, then range returns a RangesList object resulting from calling range(ranges(x)), i.e. the bounds of the ranges in each space.

reduce first orders the ranges in x from left to right, then merges the overlapping or adjacent ones. If x is a RangedData object, reduce merges the ranges in each of the spaces after grouping by the by values columns and returns the result as a RangedData containing the reduced ranges and the by value columns.

gaps returns the "normal" Ranges object representing the set of integers that remain after the set of integers represented by x has been removed from the interval specified by the start and end arguments.

If x is a Views object, then start=NA and end=NA are interpreted as start=1 and end=length(subject(x)), respectively, so, if start and end are not specified, then gaps are extracted with respect to the entire subject.

disjoin returns a disjoint object, by finding the union of the end points in x. In other words, the result consists of a range for every interval, of maximal length, over which the set of overlapping ranges in x is the same and at least of size 1.

isDisjoint returns a logical value indicating whether the ranges x are disjoint (i.e. non-overlapping). isDisjoint handles empty ranges (a.k.a. zero-width ranges) as follow: single empty range A is considered to overlap with single range B iff it's contained in B without being on the edge of B (in which case it would be ambiguous whether A is contained in or adjacent to B). In other words, single empty range A is considered to overlap with single range B iff

```
start(B) < start(A) and end(A) < end(B)
```

Because A is an empty range it verifies end(A) = start(A) - 1 so the above is equivalent to:

```
start(B) < start(A) <= end(B)
```

and also equivalent to:

```
start(B) <= end(A) < end(B)
```

Finally, it is also equivalent to:

```
compare(A, B) == 2
```

See ?compare for the meaning of the codes returned by the compare function.

disjointBins segregates x into a set of bins so that the ranges in each bin are disjoint. Lower-indexed bins are filled first. The method returns an integer vector indicating the bin index for each range.

When x in a RangesList object, doing any of the transformation above is equivalent to applying the transformation to each RangesList top-level element separately.

For range, if there are additional RangesList objects in ..., they are merged into x by name, if all objects have names, otherwise, if they are all of the same length, by position. Else, an exception is thrown.

Author(s)

H. Pages, M. Lawrence, P. Aboyoun

See Also

- intra-range-methods for intra range transformations.
- The Ranges, Views, RangesList, MaskCollection, and RangedData classes.
- The inter-range-methods man page in the GenomicRanges package for methods that operate on GenomicRanges and other objects.
- setops-methods for set operations on IRanges objects.
- `solveUserSEW` for the SEW (Start/End/Width) interface.

Examples

```
## ---------------------------------------------------------------------
## range()
## ---------------------------------------------------------------------

## On a Ranges object:
x <- IRanges(start=c(-2, 6, 9, -4, 1, 0, -6, 3, 10),
             width=c( 5, 0, 6,  1, 4, 3,  2, 0,  3))
range(x)

## On a RangesList object (XVector package required):
range1 <- IRanges(start=c(1, 2, 3), end=c(5, 2, 8))
range2 <- IRanges(start=c(15, 45, 20, 1), end=c(15, 100, 80, 5))
range3 <- IRanges(start=c(-2, 6, 7), width=c(8, 0, 0))  # with empty ranges
collection <- IRangesList(one=range1, range2, range3)
if (require(XVector)) {
    range(collection)
}

irl1 <- IRangesList(a=IRanges(c(1,2),c(4,3)), b=IRanges(c(4,6),c(10,7)))
irl2 <- IRangesList(c=IRanges(c(0,2),c(4,5)), a=IRanges(c(4,5),c(6,7)))
range(irl1, irl2)  # matched by names
names(irl2) <- NULL
range(irl1, irl2)  # now by position

## On a RangedData object:
ranges <- IRanges(c(1,2,3),c(4,5,6))
score <- c(10L, 2L, NA)
rd <- RangedData(ranges, score)
range(rd)
rd2 <- RangedData(IRanges(c(5,2,0), c(6,3,1)))
range(rd, rd2)

## ---------------------------------------------------------------------
## reduce()
## ---------------------------------------------------------------------

## On a Ranges object:
reduce(x)
y <- reduce(x, with.revmap=TRUE)
mcols(y)$revmap  # an IntegerList
```

```
reduce(x, drop.empty.ranges=TRUE)
y <- reduce(x, drop.empty.ranges=TRUE, with.revmap=TRUE)
mcols(y)$revmap

## Use the mapping from reduced to original ranges to split the DataFrame
## of original metadata columns by reduced range:
ir0 <- IRanges(c(11:13, 2, 7:6), width=3)
mcols(ir0) <- DataFrame(id=letters[1:6], score=1:6)
ir <- reduce(ir0, with.revmap=TRUE)
ir
revmap <- mcols(ir)$revmap
revmap
relist(mcols(ir0)[unlist(revmap), ], revmap)  # a SplitDataFrameList

## On a RangesList object. These 4 are the same:
res1 <- reduce(collection)
res2 <- IRangesList(one=reduce(range1), reduce(range2), reduce(range3))
res3 <- do.call(IRangesList, lapply(collection, reduce))
res4 <- endoapply(collection, reduce)

stopifnot(identical(res2, res1))
stopifnot(identical(res3, res1))
stopifnot(identical(res4, res1))

reduce(collection, drop.empty.ranges=TRUE)

## On a RangedData object:
rd <- RangedData(
        RangesList(
          chrA=IRanges(start=c(1, 4, 6), width=c(3, 2, 4)),
          chrB=IRanges(start=c(1, 3, 6), width=c(3, 3, 4))),
        score=c(2, 7, 3, 1, 1, 1))
rd
reduce(rd)

## ---------------------------------------------------------------------
## gaps()
## ---------------------------------------------------------------------

## On a Ranges object:
x0 <- IRanges(start=c(-2, 6, 9, -4, 1, 0, -6, 10),
              width=c( 5, 0, 6,  1, 4, 3,  2,  3))
gaps(x0)
gaps(x0, start=-6, end=20)

## On a Views object:
subject <- Rle(1:-3, 6:2)
v <- Views(subject, start=c(8, 3), end=c(14, 4))
gaps(v)

## On a RangesList object. These 4 are the same:
res1 <- gaps(collection)
```

```
res2 <- IRangesList(one=gaps(range1), gaps(range2), gaps(range3))
res3 <- do.call(IRangesList, lapply(collection, gaps))
res4 <- endoapply(collection, gaps)

stopifnot(identical(res2, res1))
stopifnot(identical(res3, res1))
stopifnot(identical(res4, res1))

## On a MaskCollection object:
mask1 <- Mask(mask.width=29, start=c(11, 25, 28), width=c(5, 2, 2))
mask2 <- Mask(mask.width=29, start=c(3, 10, 27), width=c(5, 8, 1))
mask3 <- Mask(mask.width=29, start=c(7, 12), width=c(2, 4))
mymasks <- append(append(mask1, mask2), mask3)
mymasks
gaps(mymasks)

## ---------------------------------------------------------------------
## disjoin()
## ---------------------------------------------------------------------

## On a Ranges object:
ir <- IRanges(c(1, 1, 4, 10), c(6, 3, 8, 10))
disjoin(ir)  # IRanges(c(1, 4, 7, 10), c(3, 6, 8, 10))

## On a RangesList object:
disjoin(collection)

## ---------------------------------------------------------------------
## isDisjoint()
## ---------------------------------------------------------------------

## On a Ranges object:
isDisjoint(IRanges(c(2,5,1), c(3,7,3)))  # FALSE
isDisjoint(IRanges(c(2,9,5), c(3,9,6)))  # TRUE
isDisjoint(IRanges(1, 5))  # TRUE

## Handling of empty ranges:
x <- IRanges(c(11, 16, 11, -2, 11), c(15, 29, 10, 10, 10))
stopifnot(isDisjoint(x))

## Sliding an empty range along a non-empty range:
sapply(11:17,
       function(i) compare(IRanges(i, width=0), IRanges(12, 15)))

sapply(11:17,
       function(i) isDisjoint(c(IRanges(i, width=0), IRanges(12, 15))))

## On a RangesList object:
isDisjoint(collection)

## ---------------------------------------------------------------------
## disjointBins()
## ---------------------------------------------------------------------
```

```
## On a Ranges object:
disjointBins(IRanges(1, 5))  # 1L
disjointBins(IRanges(c(3, 1, 10), c(5, 12, 13)))  # c(2L, 1L, 2L)

## On a RangesList object:
disjointBins(collection)
```

`IntervalForest-class` *Interval Search Forests*

Description

Efficiently perform overlap queries with a set of interval trees.

Details

A common type of query that arises when working with intervals is finding which intervals in one set overlap those in another. An efficient family of algorithms for answering such queries is known as the Interval Tree. The `IntervalForest` class stores a set of Interval Trees corresponding to intervals that are partitioned into disjoint sets. The most efficient way to construct `IntervalForest` objects is to call the constructor below on a CompressedIRangesList object. See the IntervalTree class for the underlying Interval Tree data structure.

A canonical example of a compressed ranges list are `GenomicRanges` objects, where intervals are partitioned by their `seqnames`. See the GIntervalTree class to see the use of `IntervalForest` objects in this case.

The simplest approach for finding overlaps is to call the `findOverlaps` function on a RangesList object. See the man page of `findOverlaps-methods` for how to use this and other related functions.

Constructor

IntervalForest(rangesList): Creates an `IntervalForest` from the ranges list in `rangesList`, an object coercible to `CompressedIRangesList`.

Accessors

`length(x)`: Gets the number of ranges stored in the forest. This is a fast operation that does not bring the ranges into R.

`start(x)`: Get the starts of the ranges as a `CompressedIntegerList`.

`end(x)`: Get the ends of the ranges as `CompressedIntegerList`.

`x@partitioning`: The range partitioning of class `PartitioningByEnd`.

`names(x)`: Get the names of the range partitioning.

`elementLengths(x)`: The number of ranges in each partition.

Author(s)

Hector Corrada Bravo, Michael Lawrence

See Also

findOverlaps-methods for finding/counting interval overlaps between two compressed lists of "range-based" objects, RangesList, the parent of this class, CompressedHitsList, set of hits between 2 list-like objects, GIntervalTree, which uses IntervalForest objects.

Examples

```
query <- IRangesList(a=IRanges(c(1,4),c(5,7)),b=IRanges(9,10))
subject <- IRangesList(a=IRanges(c(2,2),c(2,3)),b=IRanges(10,12))
forest <- IntervalForest(subject)

findOverlaps(query, forest)
```

IntervalTree-class *Interval Search Trees*

Description

Efficiently perform overlap queries with an interval tree.

Details

A common type of query that arises when working with intervals is finding which intervals in one set overlap those in another. An efficient family of algorithms for answering such queries is known as the Interval Tree. This implementation makes use of the augmented tree algorithm from the reference below, but heavily adapts it for the use case of large, sorted query sets.

The simplest approach for finding overlaps is to call the findOverlaps function on a Ranges or other object with range information. See the man page of findOverlaps for how to use this and other related functions.

An IntervalTree object is a derivative of Ranges and stores its ranges as a tree that is optimized for overlap queries. Thus, for repeated queries against the same subject, it is more efficient to create an IntervalTree once for the subject using the constructor described below and then perform the queries against the IntervalTree instance.

Constructor

IntervalTree(ranges): Creates an IntervalTree from the ranges in ranges, an object coercible to IntervalTree, such as an IRanges object.

Coercion

as(from, "IRanges"): Imports the ranges in from, an IntervalTree, to an IRanges.

as(from, "IntervalTree"): Constructs an IntervalTree representing from, a Ranges object that is coercible to IRanges.

Accessors

`length(x)`: Gets the number of ranges stored in the tree. This is a fast operation that does not bring the ranges into R.

`start(x)`: Get the starts of the ranges.

`end(x)`: Get the ends of the ranges.

Notes on Time Complexity

The cost of constructing an instance of the interval tree is a `O(n*lg(n))`, which makes it about as fast as other types of overlap query algorithms based on sorting. The good news is that the tree need only be built once per subject; this is useful in situations of frequent querying. Also, in this implementation the data is stored outside of R, avoiding needless copying. Of course, external storage is not always convenient, so it is possible to coerce the tree to an instance of `IRanges` (see the Coercion section).

For the query operation, the running time is based on the query size `m` and the average number of hits per query `k`. The output size is then `max(mk,m)`, but we abbreviate this as `mk`. Note that when the `multiple` parameter is set to `FALSE`, `k` is fixed to 1 and drops out of this analysis. We also assume here that the query is sorted by start position (the `findOverlaps` function sorts the query if it is unsorted).

An upper bound for finding overlaps is `O(min(mk*lg(n),n+mk))`. The fastest interval tree algorithm known is bounded by `O(min(m*lg(n),n)+mk)` but is a lot more complicated and involves two auxillary trees. The lower bound is `Omega(lg(n)+mk)`, which is almost the same as for returning the answer, `Omega(mk)`. The average is of course somewhere in between.

This analysis informs the choice of which set of ranges to process into a tree, i.e. assigning one to be the subject and the other to be the query. Note that if `m > n`, then the running time is `O(m)`, and the total operation of complexity `O(n*lg(n) + m)` is better than if `m` and `n` were exchanged. Thus, for once-off operations, it is often most efficient to choose the smaller set to become the tree (but `k` also affects this). This is reinforced by the realization that if `mk` is about the same in either direction, the running time depends only on `n`, which should be minimized. Even in cases where a tree has already been constructed for one of the sets, it can be more efficient to build a new tree when the existing tree of size `n` is much larger than the query set of size `m`, roughly when `n > m*lg(n)`.

Author(s)

Michael Lawrence

References

Interval tree algorithm from: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. Introduction to Algorithms, second edition, MIT Press and McGraw-Hill. ISBN 0-262-53196-8

See Also

`findOverlaps` for finding/counting interval overlaps between two "range-based" objects, `Ranges`, the parent of this class, `Hits`, set of hits between 2 vector-like objects.

Examples

```
query <- IRanges(c(1, 4, 9), c(5, 7, 10))
subject <- IRanges(c(2, 2, 10), c(2, 3, 12))
tree <- IntervalTree(subject)

findOverlaps(query, tree)

## query and subject are easily interchangeable
query <- IRanges(c(1, 4, 9), c(5, 7, 10))
subject <- IRanges(c(2, 2), c(5, 4))
tree <- IntervalTree(subject)

t(findOverlaps(query, tree))
# the same as:
findOverlaps(subject, query)
```

---

intra-range-methods *Intra range transformations of a Ranges, Views, RangesList, or MaskCollection object*

Description

Except for `threebands()`, all the transformations described in this man page are *endomorphisms* that operate on a single "range-based" object, that is, they transform the ranges contained in the input object and return them in an object of the *same class* as the input object.

Range-based endomorphisms are grouped in 2 categories:

1. Intra range transformations like `shift()` that transform each range individually (and independently of the other ranges) and return an object of the *same length* as the input object. Those transformations are described in this man page.
2. Inter range transformations like `reduce()` that transform all the ranges together as a set to produce a new set of ranges and return an object not necessarily of the same length as the input object. Those transformations are described in the inter-range-methods man page (see `?inter-range-methods`).

Usage

```
## shift()
shift(x, shift=0L, use.names=TRUE)

## narrow()
narrow(x, start=NA, end=NA, width=NA, use.names=TRUE)

## flank()
flank(x, width, start=TRUE, both=FALSE, use.names=TRUE, ...)

## promoters()
```

```
promoters(x, upstream=2000, downstream=200, ...)

## reflect()
reflect(x, bounds, use.names=TRUE)

## resize()
resize(x, width, fix="start", use.names=TRUE, ...)

## restrict()
restrict(x, start=NA, end=NA, keep.all.ranges=FALSE, use.names=TRUE)

## threebands()
threebands(x, start=NA, end=NA, width=NA)
```

Arguments

x	A Ranges, Views, RangesList, or MaskCollection object.
shift	An integer vector containing the shift information. Recycled as necessary so that each element corresponds to a range in x. It can also be an IntegerList object if x is a RangesList object.
use.names	TRUE or FALSE. Should names be preserved?
start, end	• If x is a Ranges or Views object: A vector of integers for all functions except for flank. For restrict, the supplied start and end arguments must be vectors of integers, eventually with NAs, that specify the restriction interval(s). Recycled as necessary so that each element corresponds to a range in x. Same thing for narrow and threebands, except that here start and end must contain coordinates relative to the ranges in x. See the Details section below. For flank, start is a logical indicating whether x should be flanked at the start (TRUE) or the end (FALSE). Recycled as necessary so that each element corresponds to a range in x. • If x is a RangesList object: For flank, start must be either a logical vector or a LogicalList object indicating whether x should be flanked at the start (TRUE) or the end (FALSE). Recycled as necessary so that each element corresponds to a range in x. For narrow, start and end must be either an integer vector or an IntegerList object containing coordinates relative to the current ranges. For restrict, start and end must be either an integer vector or an IntegerList object (possibly containing NA's).
width	• If x is a Ranges or Views object: For narrow and threebands, a vector of integers, eventually with NAs. See the SEW (Start/End/Width) interface for the details (?solveUserSEW). For resize and flank, the width of the resized or flanking regions. Note that if both is TRUE, this is effectively doubled. Recycled as necessary so that each element corresponds to a range in x. • If x is a RangesList object: For flank and resize, either an integer vector or an IntegerList object containing the width of the flanking or resized regions. Recycled as necessary so that each element corresponds to a range in x. (Note for flank: if both is TRUE, this is effectively doubled.)

For narrow, either an integer vector or a IntegerList object containing the widths to narrow to. See the SEW (Start/End/Width) interface for the details (?solveUserSEW).

both — If TRUE, extends the flanking region width positions *into* the range. The resulting range thus straddles the end point, with width positions on either side.

bounds — An IRanges object to serve as the reference bounds for the reflection, see below.

fix
- If x is a Ranges or Views object: A character vector or character-Rle of length 1 or length(x) containing the values "start", "end", and "center" denoting what to use as an anchor for each element in x.
- If x is a RangesList object: A character vector of length 1, a CharacterList object, or a character-RleList object containing the values "start", "end", and "center" denoting what to use as an anchor for each element in x.

upstream, downstream

Single integer values >= 0L. upstream defines the number of nucleotides toward the 5' end and downstream defines the number toward the 3' end, relative to the transcription start site. Promoter regions are formed by merging the upstream and downstream ranges.

Default values for upstream and downstream were chosen based on our current understanding of gene regulation. On average, promoter regions in the mammalian genome are 5000 bp upstream and downstream of the transcription start site.

keep.all.ranges

TRUE or FALSE. Should ranges that don't overlap with the restriction interval(s) be kept? Note that "don't overlap" means that they end strictly before start - 1 or start strictly after end + 1. Ranges that end at start - 1 or start at end + 1 are always kept and their width is set to zero in the returned IRanges object.

... — Additional arguments for methods.

Details

Here we start by describing how each transformation operates on a Ranges object x.

shift shifts all the ranges in x by the amount specified by the shift argument.

narrow narrows the ranges in x i.e. each range in the returned Ranges object is a subrange of the corresponding range in x. The supplied start/end/width values are solved by a call to solveUserSEW(width(x), start=start, and therefore must be compliant with the rules of the SEW (Start/End/Width) interface (see ?solveUserSEW for the details). Then each subrange is derived from the original range according to the solved start/end/width values for this range. Note that those solved values are interpreted relatively to the original range.

flank generates flanking ranges for each range in x. If start is TRUE for a given range, the flanking occurs at the start, otherwise the end. The widths of the flanks are given by the width parameter. The widths can be negative, in which case the flanking region is reversed so that it represents a prefix or suffix of the range in x. The flank operation is illustrated below for a call of the form flank(x, 3, TRUE), where x indicates a range in x and - indicates the resulting flanking region:

```
---xxxxxxx
```

If start were FALSE:

```
xxxxxxx---
```

For negative width, i.e. flank(x, -3, FALSE), where * indicates the overlap between x and the result:

```
xxxx***
```

If both is TRUE, then, for all ranges in x, the flanking regions are extended *into* (or out of, if width is negative) the range, so that the result straddles the given endpoint and has twice the width given by width. This is illustrated below for flank(x, 3, both=TRUE):

```
---***xxxx
```

promoters generates promoter ranges for each range in x relative to the transcription start site (TSS), where TSS is start(x). The promoter range is expanded around the TSS according to the upsteam and downstream arguments. upstream represents the number of nucleotides in the 5' direction and downstream the number in the 3' direction. The full range is defined as, (start(x) - upstream) to (start(x) + downstream - 1). For documentation for using promoters on GenomicRanges objects see ?"promoters,GRanges-method".

reflect "reflects" or reverses each range in x relative to the corresponding range in bounds, which is recycled as necessary. Reflection preserves the width of a range, but shifts it such the distance from the left bound to the start of the range becomes the distance from the end of the range to the right bound. This is illustrated below, where x represents a range in x and [and] indicate the bounds:

```
[..xxx.....]
becomes
[.....xxx..]
```

restrict restricts the ranges in x to the interval(s) specified by the start and end arguments.

resize resizes the ranges to the specified width where either the start, end, or center is used as an anchor.

threebands extends the capability of narrow by returning the 3 ranges objects associated to the narrowing operation. The returned value y is a list of 3 ranges objects named "left", "middle" and "right". The middle component is obtained by calling narrow with the same arguments (except that names are dropped). The left and right components are also instances of the same class as x and they contain what has been removed on the left and right sides (respectively) of the original ranges during the narrowing.

Note that original object x can be reconstructed from the left and right bands with punion(y$left, y$right, fill.gap=T

When x in a RangesList object, doing any of the transformation above is equivalent to applying the transformation to each RangesList top-level element separately.

Author(s)

H. Pages, M. Lawrence, P. Aboyoun

See Also

- inter-range-methods for inter range transformations.
- The Ranges, Views, RangesList, and MaskCollection classes.
- The intra-range-methods man page in the XVector package for methods that operate on XVectorList objects.
- The intra-range-methods man page in the GenomicRanges package for methods that operate on GenomicRanges and other objects.
- setops-methods for set operations on IRanges objects.
- `solveUserSEW` for the SEW (Start/End/Width) interface.

Examples

```
## ---------------------------------------------------------------------
## shift()
## ---------------------------------------------------------------------

## On a Ranges object
ir1 <- successiveIRanges(c(19, 5, 0, 8, 5))
ir1
shift(ir1, shift=-3)

## On a RangesList object
range1 <- IRanges(start=c(1, 2, 3), end=c(5, 2, 8))
range2 <- IRanges(start=c(15, 45, 20, 1), end=c(15, 100, 80, 5))
range3 <- IRanges(start=c(-2, 6, 7), width=c(8, 0, 0))  # with empty ranges
collection <- IRangesList(one=range1, range2, range3)
shift(collection, shift=5)

## ---------------------------------------------------------------------
## narrow()
## ---------------------------------------------------------------------

## On a Ranges object
ir2 <- ir1[width(ir1) != 0]
narrow(ir2, start=4, end=-2)
narrow(ir2, start=-4, end=-2)
narrow(ir2, end=5, width=3)
narrow(ir2, start=c(3, 4, 2, 3), end=c(12, 5, 7, 4))

## On a RangesList object
narrow(collection[-3], start=2)
narrow(collection[-3], end=-2)

## On a MaskCollection object
mask1 <- Mask(mask.width=29, start=c(11, 25, 28), width=c(5, 2, 2))
mask2 <- Mask(mask.width=29, start=c(3, 10, 27), width=c(5, 8, 1))
```

```
mask3 <- Mask(mask.width=29, start=c(7, 12), width=c(2, 4))
mymasks <- append(append(mask1, mask2), mask3)
mymasks
narrow(mymasks, start=8)

## ---------------------------------------------------------------------
## flank()
## ---------------------------------------------------------------------

## On a Ranges object
ir3 <- IRanges(c(2,5,1), c(3,7,3))
flank(ir3, 2)
flank(ir3, 2, start=FALSE)
flank(ir3, 2, start=c(FALSE, TRUE, FALSE))
flank(ir3, c(2, -2, 2))
flank(ir3, 2, both = TRUE)
flank(ir3, 2, start=FALSE, both=TRUE)
flank(ir3, -2, start=FALSE, both=TRUE)

## On a RangesList object
flank(collection, width=10)

## ---------------------------------------------------------------------
## promoters()
## ---------------------------------------------------------------------

## On a Ranges object
ir4 <- IRanges(20:23, width=3)
promoters(ir4, upstream=0, downstream=0) ## no change
promoters(ir4, upstream=0, downstream=1) ## start value only
promoters(ir4, upstream=1, downstream=0) ## single upstream nucleotide

## On a RangesList object
promoters(collection, upstream=5, downstream=2)

## ---------------------------------------------------------------------
## reflect()
## ---------------------------------------------------------------------

## On a Ranges object
bounds <- IRanges(c(0, 5, 3), c(10, 6, 9))
reflect(ir3, bounds)

## reflect() does not yet support RangesList objects!

## ---------------------------------------------------------------------
## resize()
## ---------------------------------------------------------------------

## On a Ranges object
resize(ir2, 200)
resize(ir2, 2, fix="end")
```

```
## On a RangesList object
resize(collection, width=200)

## ---------------------------------------------------------------------
## restrict()
## ---------------------------------------------------------------------

## On a Ranges object
restrict(ir1, start=12, end=34)
restrict(ir1, start=20)
restrict(ir1, start=21)
restrict(ir1, start=21, keep.all.ranges=TRUE)

## On a RangesList object
restrict(collection, start=2, end=8)

## ---------------------------------------------------------------------
## threebands()
## ---------------------------------------------------------------------

## On a Ranges object
z <- threebands(ir2, start=4, end=-2)
ir2b <- punion(z$left, z$right, fill.gap=TRUE)
stopifnot(identical(ir2, ir2b))
threebands(ir2, start=-5)

## threebands() does not support RangesList objects.
```

IRanges-class *IRanges and NormalIRanges objects*

Description

The IRanges class is a simple implementation of the Ranges container where 2 integer vectors of the same length are used to store the start and width values. See the Ranges virtual class for a formal definition of Ranges objects and for their methods (all of them should work for IRanges objects).

Some subclasses of the IRanges class are: NormalIRanges, Views, etc...

A NormalIRanges object is just an IRanges object that is guaranteed to be "normal". See the Normality section in the man page for Ranges objects for the definition and properties of "normal" Ranges objects.

Constructor

See ?IRanges-constructor.

Coercion

`as(from, "IRanges")`: Creates an IRanges instance from a Ranges object, logical vector, or integer vector. When `from` is a logical vector, the resulting IRanges object contains the indices for the runs of `TRUE` values. When `from` is an integer vector, the elements are either singletons or "increase by 1" sequences.

`as(from, "NormalIRanges")`: Creates a NormalIRanges instance from a logical or integer vector. When `from` is an integer vector, the elements must be strictly increasing.

Combining

`c(x, ..., ignore.mcols=FALSE)` Combining `IRanges` objects is straightforward when they do not have any metadata columns. If only one of the `IRanges` object has metadata columns, then the corresponding metadata columns are attached to the other `IRanges` object and set to `NA`. When multiple `IRanges` object have their own `metadata columns`, the user must ensure that each such `linkS4class{DataFrame}` have identical layouts to each other (same columns defined), in order for the combination to be successful, otherwise an error will be thrown. The user can call `c(x, ..., ignore.mcols=TRUE)` in order to combine `IRanges` objects with differing sets of metadata columns, which will result in the combined object having NO metadata columns.

Methods for NormalIRanges objects

`max(x)`: The maximum value in the finite set of integers represented by `x`.

`min(x)`: The minimum value in the finite set of integers represented by `x`.

Author(s)

H. Pages

See Also

Ranges-class,

IRanges-constructor, IRanges-utils,

intra-range-methods for intra range transformations,

inter-range-methods for inter range transformations,

setops-methods

Examples

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

## ---------------------------------------------------------------------
## A. MANIPULATING IRanges OBJECTS
## ---------------------------------------------------------------------
## All the methods defined for Ranges objects work on IRanges objects.
## See ?Ranges for some examples.
## Also see ?IRanges-utils and ?setops-methods for additional
## operations on IRanges objects.
```

```
## Combining IRanges objects
ir1 <- IRanges(c(1, 10, 20), width=5)
mcols(ir1) <- DataFrame(score=runif(3))
ir2 <- IRanges(c(101, 110, 120), width=10)
mcols(ir2) <- DataFrame(score=runif(3))
ir3 <- IRanges(c(1001, 1010, 1020), width=20)
mcols(ir3) <- DataFrame(value=runif(3))
some.iranges <- c(ir1, ir2)
## all.iranges <- c(ir1, ir2, ir3) ## This will raise an error
all.iranges <- c(ir1, ir2, ir3, ignore.mcols=TRUE)
stopifnot(is.null(mcols(all.iranges)))

## ---------------------------------------------------------------------
## B. A NOTE ABOUT PERFORMANCE
## ---------------------------------------------------------------------
## Using an IRanges object for storing a big set of ranges is more
## efficient than using a standard R data frame:
N <- 2000000L  # nb of ranges
W <- 180L      # width of each range
start <- 1L
end <- 50000000L
set.seed(777)
range_starts <- sort(sample(end-W+1L, N))
range_widths <- rep.int(W, N)
## Instantiation is faster
system.time(x <- IRanges(start=range_starts, width=range_widths))
system.time(y <- data.frame(start=range_starts, width=range_widths))
## Subsetting is faster
system.time(x16 <- x[c(TRUE, rep.int(FALSE, 15))])
system.time(y16 <- y[c(TRUE, rep.int(FALSE, 15)), ])
## Internal representation is more compact
object.size(x16)
object.size(y16)
```

`IRanges-constructor` *The IRanges constructor and supporting functions*

Description

The `IRanges` function is a constructor that can be used to create IRanges instances.

`solveUserSEW0` and `solveUserSEW` are utility functions that solve a set of user-supplied start/end/width values.

Usage

```
## IRanges constructor:
IRanges(start=NULL, end=NULL, width=NULL, names=NULL)
```

```
## Supporting functions (not for the end user):
solveUserSEW0(start=NULL, end=NULL, width=NULL)
solveUserSEW(refwidths, start=NA, end=NA, width=NA,
             rep.refwidths=FALSE,
             translate.negative.coord=TRUE,
             allow.nonnarrowing=FALSE)
```

Arguments

`start, end, width`	For `IRanges` and `solveUserSEW0`: `NULL`, or vector of integers (eventually with NAs). For `solveUserSEW`: vector of integers (eventually with NAs).
`names`	A character vector or `NULL`.
`refwidths`	Vector of non-NA non-negative integers containing the reference widths.
`rep.refwidths`	`TRUE` or `FALSE`. Use of `rep.refwidths=TRUE` is supported only when `refwidths` is of length 1.
`translate.negative.coord, allow.nonnarrowing`	`TRUE` or `FALSE`.

IRanges constructor

Return the IRanges object containing the ranges specified by `start`, `end` and `width`. Input falls into one of two categories:

Category 1 `start`, `end` and `width` are numeric vectors (or NULLs). If necessary they are recycled to the length of the longest (NULL arguments are filled with NAs). After this recycling, each row in the 3-column matrix obtained by binding those 3 vectors together is "solved" i.e. NAs are treated as unknown in the equation `end = start + width - 1`. Finally, the solved matrix is returned as an IRanges instance.

Category 2 The `start` argument is a logical vector or logical Rle object and `IRanges(start)` produces the same result as `as(start, "IRanges")`. Note that, in that case, the returned IRanges instance is guaranteed to be normal.

Note that the `names` argument is never recycled (to remain consistent with what `names<-` does on standard vectors).

Supporting functions

`solveUserSEW0(start=NULL, end=NULL, width=NULL)`:

`solveUserSEW(refwidths, start=NA, end=NA, width=NA, rep.refwidths=FALSE, translat`
Use of `rep.refwidths=TRUE` is supported only when `refwidths` is of length 1. If `rep.refwidths=FALSE` (the default) then `start`, `end` and `width` are recycled to the length of `refwidths` (it's an error if one of them is longer than `refwidths`, or is of zero length while `refwidths` is not). If `rep.refwidths=TRUE` then `refwidths` is first replicated L times where L is the length of the longest of `start`, `end` and `width`. After this replication, `start`, `end` and `width` are recycled to the new length of `refwidths` (L) (it's an error if one of them is of zero length while L is != 0).

From now, refwidths, start, end and width are integer vectors of equal lengths. Each row in the 3-column matrix obtained by binding those 3 vectors together must contain at least one NA (otherwise an error is returned). Then each row is "solved" i.e. the 2 following transformations are performed (i is the indice of the row): (1) if translate.negative.coord is TRUE then a negative value of start[i] or end[i] is considered to be a -refwidths[i]-based coordinate so refwidths[i]+1 is added to it to make it 1-based; (2) the NAs in the row are treated as unknowns which values are deduced from the known values in the row and from refwidths[i].

The exact rules for (2) are the following. Rule (2a): if the row contains at least 2 NAs, then width[i] must be one of them (otherwise an error is returned), and if start[i] is one of them it is replaced by 1, and if end[i] is one of them it is replaced by refwidths[i], and finally width[i] is replaced by end[i] - start[i] + 1. Rule (2b): if the row contains only 1 NA, then it is replaced by the solution of the width[i] == end[i] - start[i] + 1 equation.

Finally, the set of solved rows is returned as an IRanges object of the same length as refwidths (after replication if rep.refwidths=TRUE).

Note that an error is raised if either (1) the set of user-supplied start/end/width values is invalid or (2) allow.nonnarrowing is FALSE and the ranges represented by the solved start/end/width values are not narrowing the ranges represented by the user-supplied start/end/width values.

Author(s)

H. Pages

See Also

IRanges-class, narrow

Examples

```
## ---------------------------------------------------------------------
## A. USING THE IRanges() CONSTRUCTOR
## ---------------------------------------------------------------------
IRanges(start=11, end=rep.int(20, 5))
IRanges(start=11, width=rep.int(20, 5))
IRanges(-2, 20)  # only one range
IRanges(start=c(2, 0, NA), end=c(NA, NA, 14), width=11:0)
IRanges()  # IRanges instance of length zero
IRanges(names=character())

## With logical input:
x <- IRanges(c(FALSE, TRUE, TRUE, FALSE, TRUE))  # logical vector input
isNormal(x)  # TRUE
x <- IRanges(Rle(1:30) %% 5 <= 2)  # logical Rle input
isNormal(x)  # TRUE

## ---------------------------------------------------------------------
## B. USING solveUserSEW()
## ---------------------------------------------------------------------
```

```
refwidths <- c(5:3, 6:7)
refwidths

solveUserSEW(refwidths)
solveUserSEW(refwidths, start=4)
solveUserSEW(refwidths, end=3, width=2)
solveUserSEW(refwidths, start=-3)
solveUserSEW(refwidths, start=-3, width=2)
solveUserSEW(refwidths, end=-4)

## The start/end/width arguments are recycled:
solveUserSEW(refwidths, start=c(3, -4, NA), end=c(-2, NA))

## Using rep.refwidths=TRUE:
solveUserSEW(10, start=-(1:6), rep.refwidths=TRUE)
solveUserSEW(10, end=-(1:6), width=3, rep.refwidths=TRUE)
```

IRanges-utils *IRanges utility functions*

Description

Utility functions for creating or modifying IRanges objects.

Usage

```
## Create an IRanges instance:
successiveIRanges(width, gapwidth=0, from=1)
breakInChunks(totalsize, chunksize)

## Turn a logical vector into a set of ranges:
whichAsIRanges(x)

## Coercion:
asNormalIRanges(x, force=TRUE)
```

Arguments

width	A vector of non-negative integers (with no NAs) specifying the widths of the ranges to create.
gapwidth	A single integer or an integer vector with one less element than the width vector specifying the widths of the gaps separating one range from the next one.
from	A single integer specifying the starting position of the first range.
totalsize	A single non-negative integer. The total size of the object to break.
chunksize	A single positive integer. The size of the chunks (last chunk might be smaller).
x	A logical vector for whichAsIRanges. An IRanges object for asNormalIRanges.
force	TRUE or FALSE. Should x be turned into a NormalIRanges object even if isNormal(x) is FALSE?

Details

successiveIRanges returns an IRanges instance containing the ranges that have the widths specified in the width vector and are separated by the gaps specified in gapwidth. The first range starts at position from. When gapwidth=0 and from=1 (the defaults), the returned IRanges can be seen as a partitioning of the 1:sum(width) interval. See ?Partitioning for more details on this.

whichAsIRanges returns an IRanges instance containing all of the ranges where x is TRUE.

If force=TRUE (the default), then asNormalIRanges will turn x into a NormalIRanges instance by reordering and reducing the set of ranges if necessary (i.e. only if isNormal(x) is FALSE, otherwise the set of ranges will be untouched). If force=FALSE, then asNormalIRanges will turn x into a NormalIRanges instance only if isNormal(x) is TRUE, otherwise it will raise an error. Note that when force=FALSE, the returned object is guaranteed to contain exactly the same set of ranges than x. as(x, "NormalIRanges") is equivalent to asNormalIRanges(x, force=TRUE).

Author(s)

H. Pages

See Also

Ranges-class, IRanges-class,

intra-range-methods for intra range transformations,

inter-range-methods for inter range transformations,

setops-methods, solveUserSEW, successiveViews

Examples

```
vec <- as.integer(c(19, 5, 0, 8, 5))

successiveIRanges(vec)

breakInChunks(600999, 50000)  # 13 chunks of size 50000 (last chunk is
                              # smaller).

whichAsIRanges(vec >= 5)

x <- IRanges(start=c(-2L, 6L, 9L, -4L, 1L, 0L, -6L, 10L),
             width=c( 5L, 0L, 6L,  1L, 4L, 3L,  2L,  3L))
asNormalIRanges(x)  # 3 non-empty ranges ordered from left to right and
                    # separated by gaps of width >= 1.

## More on normality:
example(IRanges-class)
isNormal(x16)                       # FALSE
if (interactive())
    x16 <- asNormalIRanges(x16)     # Error!
whichFirstNotNormal(x16)            # 57
isNormal(x16[1:56])                 # TRUE
xx <- asNormalIRanges(x16[1:56])
class(xx)
```

```
max(xx)
min(xx)
```

IRangesList-class *List of IRanges and NormalIRanges*

Description

IRangesList and NormalIRangesList objects for storing IRanges and NormalIRanges objects respectively.

Constructor

IRangesList(..., universe = NULL, compress = TRUE): The ... argument accepts either a comma-separated list of IRanges objects, or a single LogicalList / logical RleList object, or 2 elements named start and end each of them being either a list of integer vectors or an IntegerList object. When IRanges objects are supplied, each of them becomes an element in the new IRangesList, in the same order, which is analogous to the list constructor. If compress, the internal storage of the data is compressed.

Coercion

unlist(x): Unlists x, an IRangesList, by concatenating all of the ranges into a single IRanges instance. If the length of x is zero, an empty IRanges is returned.

Methods for NormalIRangesList objects

max(x): An integer vector containing the maximum values of each of the elements of x.

min(x): An integer vector containing the minimum values of each of the elements of x.

Author(s)

Michael Lawrence

See Also

RangesList, the parent of this class, for more functionality.

intra-range-methods and inter-range-methods for intra and inter range transformations of IRangesList objects.

setops-methods for set operations on IRangesList objects.

Examples

```
range1 <- IRanges(start=c(1,2,3), end=c(5,2,8))
range2 <- IRanges(start=c(15,45,20,1), end=c(15,100,80,5))
named <- IRangesList(one = range1, two = range2)
length(named) # 2
names(named) # "one" and "two"
named[[1]] # range1
unnamed <- IRangesList(range1, range2)
names(unnamed) # NULL

x <- IRangesList(start=list(c(1,2,3), c(15,45,20,1)),
                 end=list(c(5,2,8), c(15,100,80,5)))
as.list(x)
```

---

isConstant *Test if an atomic vector or array is constant*

Description

Generic function to test if an atomic vector or array is constant or not. Currently only methods for vectors or arrays of type integer or double are implemented.

Usage

```
isConstant(x)
```

Arguments

x	An atomic vector or array.

Details

Vectors of length 0 or 1 are always considered to be constant.

Value

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

Author(s)

H. Pages

See Also

duplicated, unique, all.equal, NA, is.finite

Examples

```
## ------------------------------------------------------------------------
## A. 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_))


## ------------------------------------------------------------------------
## B. 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 `elementLengths` 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 has `elementType = "integer"` and its validity method checks if this condition is TRUE.

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

Construction

`List` objects are typically constructed by calling the constructor of a concrete implementation, such as `RangesList` or `IntegerList`. A general and convenient way to convert any vector-like object into a `List` is to call `as(x, "List")`. This will typically yield an object from a subclass of `CompressedList`.

Accessors

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

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

`elementLengths(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.

Element extraction (list style)

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

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

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

Looping

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

`lapply(X, FUN, ...)`: 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`.

`sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)`: 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(X, FUN, ...)`: Similar to `lapply`, but performs an endomorphism, i.e. returns an object of `class(X)`.

`mendoapply(FUN, ..., MoreArgs = NULL)`: Similar to `mapply`, but performs an endomorphism across multiple objects, i.e. returns an object of `class(list(...)[[1]])`.

`revElements(x, i)`: A convenient way to do `x[i] <- endoapply(x[i], rev)`. There is a fast method for CompressedList objects, otherwise expect it to be rather slow.

Coercion

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

`as.env(x, enclos = parent.frame())`: Creates an environment from `x` with a symbol for each `names(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.

`as.list(x, ...)`, `as(from, "list")`: Turns `x` into a standard list.

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

`stack(x, index.var = "name", value.var = "value")`: As with `stack` on a `list`, 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.

Evaluating

In the code snippets below, `envir` and `data` are List objects.

`eval(expr, envir, enclos = parent.frame())`: Converts the List object specified in `envir` to an environment using `as.env`, with `enclos` as its parent, and then evaluates `expr` within that environment.

`with(data, expr, ...)`: Equivalent to `eval(quote(expr), data, ...)`.

`within(data, expr, ...)`: 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.

Author(s)

P. Aboyoun and H. Pages

See Also

- Vector objects for the parent class.
- The SimpleList and CompressedList classes for direct extensions of the List class.
- The IRanges class and constructor for an example of a concrete List subclass.
- extractList for grouping elements of a vector-like object into a list-like object.
- funprog-methods for using functional programming methods on List objects.

Examples

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

MaskCollection-class *MaskCollection objects*

Description

The MaskCollection class is a container for storing a collection of masks that can be used to mask regions in a sequence.

Details

In the context of the Biostrings package, a mask is a set of regions in a sequence that need to be excluded from some computation. For example, when calling `alphabetFrequency` or `matchPattern` on a chromosome sequence, you might want to exclude some regions like the centromere or the repeat regions. This can be achieved by putting one or several masks on the sequence before calling `alphabetFrequency` on it.

A MaskCollection object is a vector-like object that represents such set of masks. Like standard R vectors, it has a "length" which is the number of masks contained in it. But unlike standard R vectors, it also has a "width" which determines the length of the sequences it can be "put on". For example, a MaskCollection object of width 20000 can only be put on an XString object of 20000 letters.

Each mask in a MaskCollection object `x` is just a finite set of integers that are >= 1 and <= `width(x)`. When "put on" a sequence, these integers indicate the positions of the letters to mask. Internally, each mask is represented by a NormalIRanges object.

Basic accessor methods

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

`length(x)`: The number of masks in `x`.

`width(x)`: The common with of all the masks in `x`. This determines the length of the sequences that `x` can be "put on".

`active(x)`: A logical vector of the same length as `x` where each element indicates whether the corresponding mask is active or not.

`names(x)`: `NULL` or a character vector of the same length as `x`.

`desc(x)`: `NULL` or a character vector of the same length as `x`.

`nir_list(x)`: A list of the same length as `x`, where each element is a NormalIRanges object representing a mask in `x`.

Constructor

`Mask(mask.width, start=NULL, end=NULL, width=NULL)`: Return a single mask (i.e. a MaskCollection object of length 1) of width `mask.width` (a single integer >= 1) and masking the ranges of positions specified by `start`, `end` and `width`. See the `IRanges` constructor (`?IRanges`) for how `start`, `end` and `width` can be specified. Note that the returned mask is active and unnamed.

Other methods

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

`isEmpty(x)`: Return a logical vector of the same length as `x`, indicating, for each mask in `x`, whether it's empty or not.

`max(x)`: The greatest (or last, or rightmost) masked position for each mask. This is a numeric vector of the same length as `x`.

`min(x)`: The smallest (or first, or leftmost) masked position for each mask. This is a numeric vector of the same length as `x`.

`maskedwidth(x)`: The number of masked position for each mask. This is an integer vector of the same length as `x` where all values are >= 0 and <= `width(x)`.

`maskedratio(x)`: `maskedwidth(x) / width(x)`

Subsetting and appending

In the code snippets below, `x` and `values` are MaskCollection objects.

`x[i]`: Return a new MaskCollection object made of the selected masks. Subscript `i` can be a numeric, logical or character vector.

`x[[i, exact=TRUE]]`: Extract the mask selected by `i` as a NormalIRanges object. Subscript `i` can be a single integer or a character string.

`append(x, values, after=length(x))`: Add masks in `values` to `x`.

Other methods

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

`collapse(x)`: Return a MaskCollection object of length 1 obtained by collapsing all the active masks in `x`.

Author(s)

H. Pages

See Also

NormalIRanges-class, read.Mask, MaskedXString-class, `reverse`, `alphabetFrequency`, `matchPattern`

Examples

```
## Making a MaskCollection object:
mask1 <- Mask(mask.width=29, start=c(11, 25, 28), width=c(5, 2, 2))
mask2 <- Mask(mask.width=29, start=c(3, 10, 27), width=c(5, 8, 1))
mask3 <- Mask(mask.width=29, start=c(7, 12), width=c(2, 4))
mymasks <- append(append(mask1, mask2), mask3)
mymasks
length(mymasks)
width(mymasks)
```

```
collapse(mymasks)

## Names and descriptions:
names(mymasks) <- c("A", "B", "C")  # names should be short and unique...
mymasks
mymasks[c("C", "A")]  # ...to make subsetting by names easier
desc(mymasks) <- c("you can be", "more verbose", "here")
mymasks[-2]

## Activate/deactivate masks:
active(mymasks)["B"] <- FALSE
mymasks
collapse(mymasks)
active(mymasks) <- FALSE  # deactivate all masks
mymasks
active(mymasks)[-1] <- TRUE  # reactivate all masks except mask 1
active(mymasks) <- !active(mymasks)  # toggle all masks

## Other advanced operations:
mymasks[[2]]
length(mymasks[[2]])
mymasks[[2]][-3]
append(mymasks[-2], gaps(mymasks[2]))
```

`multisplit` *Split elements belonging to multiple groups*

Description

This is like `split`, except elements can belong to multiple groups, in which case they are repeated to appear in multiple elements of the return value.

Usage

```
multisplit(x, f)
```

Arguments

`x`	The object to split, like a vector.
`f`	A list-like object of vectors, the same length as `x`, where each element indicates the groups to which each element of `x` belongs.

Value

A list-like object, with an element for each unique value in the unlisted `f`, containing the elements in `x` where the corresponding element in `f` contained that value. Just try it.

Author(s)

Michael Lawrence

Examples

```
multisplit(1:3, list(letters[1:2], letters[2:3], letters[2:4]))
```

---

nearest-methods *Finding the nearest range neighbor*

Description

The nearest, precede, follow, distance and distanceToNearest methods for Ranges objects and subclasses.

Usage

```
## S4 method for signature Ranges,RangesORmissing
nearest(x, subject, select = c("arbitrary", "all"))

## S4 method for signature Ranges,RangesORmissing
precede(x, subject, select = c("first", "all"))

## S4 method for signature Ranges,RangesORmissing
follow(x, subject, select = c("last", "all"))

## S4 method for signature Ranges,RangesORmissing
distanceToNearest(x, subject, select = c("arbitrary", "all"))

## S4 method for signature Ranges,Ranges
distance(x, y)
```

Arguments

x	The query Ranges instance.
subject	The subject Ranges instance, within which the nearest neighbors are found. Can be missing, in which case x is also the subject.
y	For the distance method, a Ranges instance. Cannot be missing. If x and y are not the same length, the shortest will be recycled to match the length of the longest.
select	Logic for handling ties. By default, all the methods select a single interval (arbitrary for nearest,the first by order in subject for precede, and the last for follow). To get all matchings, as a Hits object, use "all".
...	Additional arguments for methods

Details

- nearest: The conventional nearest neighbor finder. Returns a integer vector containing the index of the nearest neighbor range in `subject` for each range in `x`. If there is no nearest neighbor (if `subject` is empty), NA's are returned.

  The algorithm is roughly as follows, for a range `xi` in `x`:

  1. Find the ranges in `subject` that overlap `xi`. If a single range `si` in `subject` overlaps `xi`, `si` is returned as the nearest neighbor of `xi`. If there are multiple overlaps, one of the overlapping ranges is chosen arbitrarily.
  2. If no ranges in `subject` overlap with `xi`, then the range in `subject` with the shortest distance from its end to the start `xi` or its start to the end of `xi` is returned.
- precede: For each range in `x`, `precede` returns the index of the interval in `subject` that is directly preceded by the query range. Overlapping ranges are excluded. `NA` is returned when there are no qualifying ranges in `subject`.
- follow: The opposite of `precede`, this function returns the index of the range in `subject` that a query range in `x` directly follows. Overlapping ranges are excluded. `NA` is returned when there are no qualifying ranges in `subject`.
- distanceToNearest: Returns the distance for each range in `x` to its nearest neighbor in `subject`.
- distance: Returns the distance for each range in `x` to the range in `y`.

  The `distance` method differs from others documented on this page in that it is symmetric; `y` cannot be missing. If `x` and `y` are not the same length, the shortest will be recycled to match the length of the longest. The `select` argument is not available for `distance` because comparisons are made in a pair-wise fashion. The return value is the length of the longest of `x` and `y`.

  The `distance` calculation changed in BioC 2.12 to accommodate zero-width ranges in a consistent and intuitive manner. The new distance can be explained by a *block* model where a range is represented by a series of blocks of size 1. Blocks are adjacent to each other and there is no gap between them. A visual representation of IRanges(4,7) would be

  ```
  +-----+-----+-----+-----+
     4     5     6     7
  ```

  The distance between two consecutive blocks is 0L (prior to Bioconductor 2.12 it was 1L). The new distance calculation now returns the size of the gap between two ranges.

  This change to distance affects the notion of overlaps in that we no longer say:

  x and y overlap <=> distance(x, y) == 0

  Instead we say

  x and y overlap => distance(x, y) == 0

  or

  x and y overlap or are adjacent <=> distance(x, y) == 0

Value

For `nearest`, `precede` and `follow`, an integer vector of indices in `subject`, or a `Hits` if `select="all"`.

For `distanceToNearest`, a `Hits` object with a column for the `query` index (queryHits), `subject` index (subjectHits) and `distance` between the pair.

For `distance`, an integer vector of distances between the ranges in `x` and `y`.

Author(s)

M. Lawrence

See Also

- The Ranges and Hits classes.
- The GenomicRanges and GRanges classes in the GenomicRanges package.
- `findOverlaps` for finding just the overlapping ranges.
- GenomicRanges methods for
  - `precede`
  - `follow`
  - `nearest`
  - `distance`
  - `distanceToNearest`

  are documented at ?`nearest-methods` or ?`precede,GenomicRanges,GenomicRanges-method`

Examples

```
## ----------------------------------------
## precede() and follow()
## ----------------------------------------
query <- IRanges(c(1, 3, 9), c(3, 7, 10))
subject <- IRanges(c(3, 2, 10), c(3, 13, 12))

precede(query, subject)     # c(3L, 3L, NA)
precede(IRanges(), subject) # integer()
precede(query, IRanges())   # rep(NA_integer_, 3)
precede(query)              # c(3L, 3L, NA)

follow(query, subject)      # c(NA, NA, 1L)
follow(IRanges(), subject)  # integer()
follow(query, IRanges())    # rep(NA_integer_, 3)
follow(query)               # c(NA, NA, 2L)

## ----------------------------------------
## nearest()
## ----------------------------------------
query <- IRanges(c(1, 3, 9), c(2, 7, 10))
subject <- IRanges(c(3, 5, 12), c(3, 6, 12))

nearest(query, subject) # c(1L, 1L, 3L)
nearest(query)          # c(2L, 1L, 2L)

## ----------------------------------------
## distance()
## ----------------------------------------
## adjacent
distance(IRanges(1,5), IRanges(6,10)) # 0L
## overlap
```

```
distance(IRanges(1,5), IRanges(3,7))  # 0L
## zero-width
sapply(-3:3, function(i) distance(shift(IRanges(4,3), i), IRanges(4,3)))
```

RangedData-class *Data on ranges*

Description

RangedData supports storing data, i.e. a set of variables, on a set of ranges spanning multiple spaces (e.g. chromosomes). Although the data is split across spaces, it can still be treated as one cohesive dataset when desired and extends DataTable. In order to handle large datasets, the data values are stored externally to avoid copying, and the rdapply function facilitates the processing of each space separately (divide and conquer).

Details

A RangedData object consists of two primary components: a RangesList holding the ranges over multiple spaces and a parallel SplitDataFrameList, holding the split data. There is also an universe slot for denoting the source (e.g. the genome) of the ranges and/or data.

There are two different modes of interacting with a RangedData. The first mode treats the object as a contiguous "data frame" annotated with range information. The accessors start, end, and width get the corresponding fields in the ranges as atomic integer vectors, undoing the division over the spaces. The [[and matrix-style [, extraction and subsetting functions unroll the data in the same way. [[<- does the inverse. The number of rows is defined as the total number of ranges and the number of columns is the number of variables in the data. It is often convenient and natural to treat the data this way, at least when the data is small and there is no need to distinguish the ranges by their space.

The other mode is to treat the RangedData as a list, with an element (a virtual Ranges/DataFrame pair) for each space. The length of the object is defined as the number of spaces and the value returned by the names accessor gives the names of the spaces. The list-style [subset function behaves analogously. The rdapply function provides a convenient and formal means of applying an operation over the spaces separately. This mode is helpful when ranges from different spaces must be treated separately or when the data is too large to process over all spaces at once.

Accessor methods

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

The following accessors treat the data as a contiguous dataset, ignoring the division into spaces:

Array accessors:

nrow(x): The number of ranges in x.

ncol(x): The number of data variables in x.

dim(x): An integer vector of length two, essentially c(nrow(x), ncol(x)).

rownames(x), rownames(x) <- value: Gets or sets the names of the ranges in x.

colnames(x), colnames(x) <- value: Gets the names of the variables in x.

`dimnames(x)`: A list with two elements, essentially `list(rownames(x), colnames(x))`.

`dimnames(x) <- value`: Sets the row and column names, where value is a list as described above.

`columnMetadata(x)`: Get the `DataFrame` of metadata along the value columns, i.e., where each column in `x` is represented by a row in the metadata. Note that calling `mcols(x)` returns the metadata on each space in `x`.

`columnMetadata(x) <- value`: Set the `DataFrame` of metadata for the columns.

`within(data, expr, ...)`: Evaluates `expr` within `data`, a `RangedData`. Any values assigned in `expr` will be stored as value columns in `data`, unless they match one of the reserved names: `ranges`, `start`, `end`, `width` and `space`. Behavior is undefined if any of the range symbols are modified inconsistently. Modifications to `space` are ignored.

Range accessors. The type of the return value depends on the type of `Ranges`. For `IRanges`, an integer vector. Regardless, the number of elements is always equal to `nrow(x)`.

`start(x), start(x) <- value`: Get or set the starts of the ranges. When setting the starts, `value` can be an integer vector of `length(sum(elementLengths(ranges(x))))` or an IntegerList object of length `length(ranges(x))` and names `names(ranges(x))`.

`end(x), end(x) <- value`: Get or set the ends of the ranges. When setting the ends, `value` can be an integer vector of `length(sum(elementLengths(ranges(x))))` or an IntegerList object of length `length(ranges(x))` and names `names(ranges(x))`.

`width(x), width(x) <- value`: Get or set the widths of the ranges. When setting the widths, `value` can be an integer vector of `length(sum(elementLengths(ranges(x))))` or an IntegerList object of length `length(ranges(x))` and names `names(ranges(x))`.

These accessors make the object seem like a list along the spaces:

`length(x)`: The number of spaces (e.g. chromosomes) in `x`.

`names(x)`, `names(x) <- value`: Get or set the names of the spaces (e.g. `"chr1"`). `NULL` or a character vector of the same length as `x`.

Other accessors:

`universe(x)`, `universe(x) <- value`: Get or set the scalar string identifying the scope of the data in some way (e.g. genome, experimental platform, etc). The universe may be `NULL`.

`ranges(x), ranges(x) <- value`: Gets or sets the ranges in `x` as a `RangesList`.

`space(x)`: Gets the spaces from `ranges(x)`.

`values(x), values(x) <- value`: Gets or sets the data values in `x` as a `SplitDataFrameList`.

`score(x), score(x) <- value`: Gets or sets the column representing a "score" in `x`, as a vector. This is the column named `score`, or, if this does not exist, the first column, if it is numeric. The get method return `NULL` if no suitable score column is found. The set method takes a numeric vector as its value.

Constructor

`RangedData(ranges = IRanges(), ..., space = NULL, universe = NULL)`: Creates a `RangedData` with the ranges in `ranges` and variables given by the arguments in `...`. See the constructor `DataFrame` for how the `...` arguments are interpreted.

If ranges is a Ranges object, the space argument is used to split of the data into spaces. If space is NULL, all of the ranges and values are placed into the same space, resulting in a single-space (length one) RangedData object. Otherwise, the ranges and values are split into spaces according to space, which is treated as a factor, like the f argument in split.

If ranges is a RangesList object, then the supplied space argument is ignored and its value is derived from ranges.

If ranges is not a Ranges or RangesList object, this function calls as(ranges, "RangedData") and returns the result if successful.

The universe may be specified as a scalar string by the universe argument.

Coercion

as.data.frame(x, row.names=NULL, optional=FALSE, ...): Copy the start, end, width of the ranges and all of the variables as columns in a data.frame. This is a bridge to existing functionality in R, but of course care must be taken if the data is large. Note that optional and ... are ignored.

as(from, "DataFrame"): Like as.data.frame above, except the result is an DataFrame and it probably involves less copying, especially if there is only a single space.

as(from, "RangedData"): Coerce from to a RangedData, according to the type of from:

Rle, RleList Converts each run to a range and stores the run values in a column named "score".

RleViewsList Creates a RangedData using the ranges given by the runs of subject(from) in each of the windows, with a value column score taken as the corresponding subject values.

Ranges Creates a RangedData with only the ranges in from; no data columns.

RangesList Creates a RangedData with the ranges in from. Also propagates the *inner* metadata columns of the RangesList (accessed with mcols(unlist(from))) to the data columns (aka values) of the RangedData. This makes it a *lossless* coercion and the exact reverse of the coercion from RangedData to RangesList.

data.frame **or** DataTable Constructs a RangedData, using the columns "start", "end", and, optionally, "space" columns in from. The other columns become data columns in the result. Any "width" column is ignored.

as(from, "RangesList"): Creates a CompressedIRangesList (a subclass of RangesList) made of the ranges in from. Also propagates the data columns (aka values) of the RangedData to the inner metadata columns of the RangesList. This makes it a *lossless* coercion and the exact reverse of the coercion from RangesList to RangedData.

as.env(x, enclos = parent.frame()): Creates an environment with a symbol for each variable in the frame, as well as a ranges symbol for the ranges. This is efficient, as no copying is performed.

Subsetting and Replacement

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

x[i]: Subsets x by indexing into its spaces, so the result is of the same class, with a different set of spaces. i can be numerical, logical, NULL or missing.

`x[i,j]`: Subsets `x` by indexing into its rows and columns. The result is of the same class, with a different set of rows and columns. The row index `i` can either treat `x` as a flat table by being a character, integer, or logical vector or treat `x` as a partitioned table by being a `RangesList`, `LogicalList`, or `IntegerList` of the same length as `x`.

`x[[i]]`: Extracts a variable from `x`, where `i` can be a character, numeric, or logical scalar that indexes into the columns. The variable is unlisted over the spaces.

For convenience, values of `"space"` and `"ranges"` are equivalent to `space(x)` and `unlist(ranges(x))` respectively.

`x$name`: similar to above, where `name` is taken literally as a column name in the data.

`x[[i]] <- value`: Sets value as column `i` in `x`, where `i` can be a character, numeric, or logical scalar that indexes into the columns. The length of `value` should equal `nrow(x)`. `x[[i]]` should be identical to `value` after this operation.

For convenience, `i="ranges"` is equivalent to `ranges(x) <- value`.

`x$name <- value`: similar to above, where `name` is taken literally as a column name in the data.

Splitting and Combining

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

`split(x, f, drop = FALSE)`: Split `x` according to `f`, which should be of length equal to `nrow(x)`. Note that `drop` is ignored here. The result is a `RangedDataList` where every element has the same length (number of spaces) but different sets of ranges within each space.

`rbind(...)`: Matches the spaces from the `RangedData` objects in `...` by name and combines them row-wise. In a way, this is the reverse of the `split` operation described above.

`c(x, ..., recursive = FALSE)`: Combines `x` with arguments specified in `...`, which must all be `RangedData` objects. This combination acts as if `x` is a list of spaces, meaning that the result will contain the spaces of the first concatenated with the spaces of the second, and so on. This function is useful when creating `RangedData` objects on a space-by-space basis and then needing to combine them.

Applying

There are two ways explicitly supported ways to apply a function over the spaces of a `RangedData`. The richest interface is `rdapply`, which is described in its own man page. The simpler interface is an `lapply` method:

`lapply(X, FUN, ...)`: Applies `FUN` to each space in `X` with extra parameters in `...`.

Author(s)

Michael Lawrence

See Also

DataTable, the parent of this class, with more utilities. The `rdapply` function for applying a function to each space separately.

Examples

```
ranges <- IRanges(c(1,2,3),c(4,5,6))
filter <- c(1L, 0L, 1L)
score <- c(10L, 2L, NA)

## constructing RangedData instances

## no variables
rd <- RangedData()
rd <- RangedData(ranges)
ranges(rd)
## one variable
rd <- RangedData(ranges, score)
rd[["score"]]
## multiple variables
rd <- RangedData(ranges, filter, vals = score)
rd[["vals"]] # same as rd[["score"]] above
rd$vals
rd[["filter"]]
rd <- RangedData(ranges, score + score)
rd[["score...score"]] # names made valid
## use a universe
rd <- RangedData(ranges, universe = "hg18")
universe(rd)

## split some data over chromosomes

range2 <- IRanges(start=c(15,45,20,1), end=c(15,100,80,5))
both <- c(ranges, range2)
score <- c(score, c(0L, 3L, NA, 22L))
filter <- c(filter, c(0L, 1L, NA, 0L))
chrom <- paste("chr", rep(c(1,2), c(length(ranges), length(range2))), sep="")

rd <- RangedData(both, score, filter, space = chrom, universe = "hg18")
rd[["score"]] # identical to score
rd[1][["score"]] # identical to score[1:3]

## subsetting

## list style: [i]

rd[numeric()] # these three are all empty
rd[logical()]
rd[NULL]
rd[] # missing, full instance returned
rd[FALSE] # logical, supports recycling
rd[c(FALSE, FALSE)] # same as above
rd[TRUE] # like rd[]
rd[c(TRUE, FALSE)]
rd[1] # numeric index
rd[c(1,2)]
rd[-2]
```

```
## matrix style: [i,j]

rd[,NULL] # no columns
rd[NULL,] # no rows
rd[,1]
rd[,1:2]
rd[,"filter"]
rd[1,] # now by the rows
rd[c(1,3),]
rd[1:2, 1] # row and column
rd[c(1:2,1,3),1] ## repeating rows

## dimnames

colnames(rd)[2] <- "foo"
colnames(rd)
rownames(rd) <- head(letters, nrow(rd))
rownames(rd)

## space names

names(rd)
names(rd)[1] <- "chr1"

## variable replacement

count <- c(1L, 0L, 2L)
rd <- RangedData(ranges, count, space = c(1, 2, 1))
## adding a variable
score <- c(10L, 2L, NA)
rd[["score"]] <- score
rd[["score"]] # same as score
## replacing a variable
count2 <- c(1L, 1L, 0L)
rd[["count"]] <- count2
## numeric index also supported
rd[[2]] <- score
rd[[2]] # gets score
## removing a variable
rd[[2]] <- NULL
ncol(rd) # is only 1
rd$score2 <- score

## combining/splitting

rd <- RangedData(ranges, score, space = c(1, 2, 1))
c(rd[1], rd[2]) # equal to rd
rd2 <- RangedData(ranges, score)
unlist(split(rd2, c(1, 2, 1))) # same as rd

## applying
```

```
lapply(rd, [[, 1) # get first column in each space
```

RangedDataList-class *Lists of RangedData*

Description

A formal list of RangedData objects. Extends and inherits all its methods from List. One use case is to group together all of the samples from an experiment generating data on ranges.

Constructor

RangedDataList(...): Concatenates the RangedData objects in ... into a new RangedDataList.

Other methods

stack(x, index.var = "name"): Concantenates the elements of x into a RangedData, with a column named by index.var that groups the records by their original element in x.

Author(s)

Michael Lawrence

See Also

RangedData, the element type of this List.

Examples

```
ranges <- IRanges(c(1,2,3),c(4,5,6))
a <- RangedData(IRanges(c(1,2,3),c(4,5,6)), score = c(10L, 2L, NA))
b <- RangedData(IRanges(c(1,2,4),c(4,7,5)), score = c(3L, 5L, 7L))
RangedDataList(sample1 = a, sample2 = b)
```

RangedSelection-class *Selection of ranges and columns*

Description

A RangedSelection represents a query against a table of interval data in terms of ranges and column names. The ranges select any table row with an overlapping interval. Note that the intervals are always returned, even if no columns are selected.

Details

Traditionally, tabular data structures have supported the `subset` function, which allows one to select a subset of the rows and columns from the table. In that case, the rows and columns are specified by two separate arguments. As querying interval data sources, especially those external to R, such as binary indexed files and databases, is increasingly common, there is a need to encapsulate the row and column specifications into a single data structure, mostly for the sake of interface cleanliness. The `RangedSelection` class fills that role.

Constructor

`RangedSelection(ranges = RangesList(), colnames = character())`: Constructors a `RangedSelection` with the given `ranges` and `colnames`.

Coercion

`as(from, "RangedSelection")`: Coerces `from` to a `RangedSelection` object. Typically, `from` is a `RangesList`, the ranges of which become the ranges in the new `RangedSelection`.

Accessors

In the code snippets below, `x` is always a `RangedSelection`.

`ranges(x)`, `ranges(x) <- value`: Gets or sets the ranges, a `RangesList`, that select rows with overlapping intervals.

`colnames(x)`, `colnames(x) <- value`: Gets the names, a `character` vector, indicating the columns.

Author(s)

Michael Lawrence

Examples

```
rl <- RangesList(chr1 = IRanges(c(1, 5), c(3, 6)))

RangedSelection(rl)
as(rl, "RangedSelection") # same as above

RangedSelection(rl, "score")
```

Ranges-class *Ranges objects*

Description

The Ranges virtual class is a general container for storing a set of integer ranges.

Details

A Ranges object is a vector-like object where each element describes a "range of integer values".

A "range of integer values" is a finite set of consecutive integer values. Each range can be fully described with exactly 2 integer values which can be arbitrarily picked up among the 3 following values: its "start" i.e. its smallest (or first, or leftmost) value; its "end" i.e. its greatest (or last, or rightmost) value; and its "width" i.e. the number of integer values in the range. For example the set of integer values that are greater than or equal to -20 and less than or equal to 400 is the range that starts at -20 and has a width of 421. In other words, a range is a closed, one-dimensional interval with integer end points and on the domain of integers.

The starting point (or "start") of a range can be any integer (see `start` below) but its "width" must be a non-negative integer (see `width` below). The ending point (or "end") of a range is equal to its "start" plus its "width" minus one (see `end` below). An "empty" range is a range that contains no value i.e. a range that has a null width. Depending on the context, it can be interpreted either as just the empty *set* of integers or, more precisely, as the position *between* its "end" and its "start" (note that for an empty range, the "end" equals the "start" minus one).

The length of a Ranges object is the number of ranges in it, not the number of integer values in its ranges.

A Ranges object is considered empty iff all its ranges are empty.

Ranges objects have a vector-like semantic i.e. they only support single subscript subsetting (unlike, for example, standard R data frames which can be subsetted by row and by column).

The Ranges class itself is a virtual class. The following classes derive directly from the Ranges class: IRanges and IntervalTree.

Methods

In the code snippets below, `x`, `y` and `object` are Ranges objects. Not all the functions described below will necessarily work with all kinds of Ranges objects but they should work at least for IRanges objects.

Note that many more operations on Ranges objects are described in other man pages of the IRanges package. See for example the man page for intra range transformations (e.g. `shift()`, see `?intra-range-methods`), or the man page for inter range transformations (e.g. `reduce()`, see `?inter-range-methods`), or the man page for `findOverlaps` methods (see `?findOverlaps-methods`), or the man page for RangesList objects where the `split` method for Ranges objects is documented.

`length(x)`: The number of ranges in `x`.

`start(x)`: The start values of the ranges. This is an integer vector of the same length as `x`.

`width(x)`: The number of integer values in each range. This is a vector of non-negative integers of the same length as `x`.

`end(x)`: `start(x) + width(x) - 1L`

`mid(x)`: returns the midpoint of the range, `start(x) + floor((width(x) - 1)/2)`.

`names(x)`: `NULL` or a character vector of the same length as `x`.

`update(object, ...)`: Convenience method for combining multiple modifications of `object` in one single call. For example `object <- update(object, start=start(object)-2L,` is equivalent to `start(object) <- start(object)-2L; end(object) <- end(object)+2L`.

`tile(x, n, width, ...)`: Splits each range in `x` into subranges as specified by `n` (number of ranges) or `width`. Only one of `n` or `width` can be specified. The return value is a `IRangesList` the same length as `x`. Ranges with a width less than the `width` argument are returned unchanged.

`isEmpty(x)`: Return a logical value indicating whether `x` is empty or not.

`as.matrix(x, ...)`: Convert `x` into a 2-column integer matrix containing `start(x)` and `width(x)`. Extra arguments (`...`) are ignored.

`as.data.frame(x, row.names=NULL, optional=FALSE, ...)`: Convert `x` into a standard R data frame object. `row.names` must be `NULL` or a character vector giving the row names for the data frame, and `optional` and any additional argument (`...`) is ignored. See `?as.data.frame` for more information about these arguments.

`as.integer(x)`: Convert `x` into an integer vector, by converting each range into the integer sequence formed by `from:to` and concatenating them together.

`unlist(x, recursive = TRUE, use.names = TRUE)`: Similar to `as.integer(x)` except can add names to elements.

`x[[i]]`: Return integer vector `start(x[i]):end(x[i])` denoted by `i`. Subscript `i` can be a single integer or a character string.

`x[i]`: Return a new Ranges object (of the same type as `x`) made of the selected ranges. `i` can be a numeric vector, a logical vector, `NULL` or missing. If `x` is a NormalIRanges object and `i` a positive numeric subscript (i.e. a numeric vector of positive values), then `i` must be strictly increasing.

`rep(x, times, length.out, each)`: 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 Ranges elements 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.

`c(x, ...)`: Combine `x` and the Ranges objects in `...` together. Any object in `...` must belong to the same class as `x`, or to one of its subclasses, or must be `NULL`. The result is an object of the same class as `x`. NOTE: Only works for IRanges (and derived) objects for now.

`x * y`: The arithmetic operation `x * y` is for centered zooming. It symmetrically scales the width of `x` by `1/y`, where `y` is a numeric vector that is recycled as necessary. For example, `x * 2` results in ranges with half their previous width but with approximately the same midpoint. The ranges have been "zoomed in". If `y` is negative, it is equivalent to `x * (1/abs(y))`. Thus, `x * -2` would double the widths in `x`. In other words, `x` has been "zoomed out".

`x + y`: Expands the ranges in `x` on either side by the corresponding value in the numeric vector `y`.

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

Normality

A Ranges object `x` is implicitly representing an arbitrary finite set of integers (that are not necessarily consecutive). This set is the set obtained by taking the union of all the values in all the ranges in `x`.

This representation is clearly not unique: many different Ranges objects can be used to represent the same set of integers. However one and only one of them is guaranteed to be "normal".

By definition a Ranges object is said to be "normal" when its ranges are: (a) not empty (i.e. they have a non-null width); (b) not overlapping; (c) ordered from left to right; (d) not even adjacent (i.e. there must be a non empty gap between 2 consecutive ranges).

Here is a simple algorithm to determine whether `x` is "normal": (1) if `length(x) == 0`, then `x` is normal; (2) if `length(x) == 1`, then `x` is normal iff `width(x) >= 1`; (3) if `length(x) >= 2`, then `x` is normal iff:

```
start(x)[i] <= end(x)[i] < start(x)[i+1] <= end(x)[i+1]
```

for every `1 <= i < length(x)`.

The obvious advantage of using a "normal" Ranges object to represent a given finite set of integers is that it is the smallest in terms of of number of ranges and therefore in terms of storage space. Also the fact that we impose its ranges to be ordered from left to right makes it unique for this representation.

A special container (NormalIRanges) is provided for holding a "normal" IRanges object: a NormalIRanges object is just an IRanges object that is guaranteed to be "normal".

Here are some methods related to the notion of "normal" Ranges:

`isNormal(x)`: Return a logical value indicating whether `x` is "normal" or not.

`whichFirstNotNormal(x)`: Return `NA` if `x` is normal, or the smallest valid indice `i` in `x` for which `x[1:i]` is not "normal".

Author(s)

H. Pages and M. Lawrence

See Also

Ranges-comparison, intra-range-methods, inter-range-methods, IRanges-class, IRanges-utils, setops-methods, RangedData-class, IntervalTree-class, `update`, `as.matrix`, `as.data.frame`, `rep`

Examples

```
x <- IRanges(start=c(2:-1, 13:15), width=c(0:3, 2:0))
x
length(x)
start(x)
width(x)
end(x)
isEmpty(x)
as.matrix(x)
as.data.frame(x)

## Subsetting:
x[4:2]                  # 3 ranges
x[-1]                   # 6 ranges
x[FALSE]                # 0 range
```

```
x0 <- x[width(x) == 0]  # 2 ranges
isEmpty(x0)

## Use the replacement methods to resize the ranges:
width(x) <- width(x) * 2 + 1
x
end(x) <- start(x)            # equivalent to width(x) <- 0
x
width(x) <- c(2, 0, 4)
x
start(x)[3] <- end(x)[3] - 2  # resize the 3rd range
x

## Name the elements:
names(x)
names(x) <- c("range1", "range2")
x
x[is.na(names(x))]  # 5 ranges
x[!is.na(names(x))]  # 2 ranges

ir <- IRanges(c(1,5), c(3,10))
ir*1 # no change
ir*c(1,2) # zoom second range by 2X
ir*-2 # zoom out 2X
```

Ranges-comparison *Comparing and ordering ranges*

Description

Methods for comparing and/or ordering Ranges objects.

Usage

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

## S4 method for signature Ranges,Ranges
compare(x, y)

rangeComparisonCodeToLetter(code)

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

## S4 method for signature Ranges,Ranges
match(x, table, nomatch=NA_integer_, incomparables=NULL,
      method=c("auto", "quick", "hash"), match.if.overlap=FALSE)
```

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

## S4 method for signature Ranges
selfmatch(x,
          method=c("auto", "quick", "hash"), match.if.overlap=FALSE)

## order() and related methods
## ---------------------------

## S4 method for signature Ranges
order(..., na.last=TRUE, decreasing=FALSE)

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

Arguments

x, y, table	Ranges objects.
nomatch	The value to be returned in the case when no match is found. It is coerced to an integer.
incomparables	Not supported.
method	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 been able to determine 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.
match.if.overlap	For match: This argument is defunct in BioC 2.14. Please use findOverlaps(x, table, select="first if you need to do match(x, table, match.if.overlap=TRUE). For selfmatch: This argument is defunct in BioC 2.14.
...	One or more Ranges objects. The additional Ranges objects are used to break ties.
na.last	Ignored.
decreasing	TRUE or FALSE.
ties.method	A character string specifying how ties are treated. Only "first" is supported for now.
code	A vector of codes as returned by compare.

Details

Two ranges are considered equal iff they share the same start and width. Note that with this definition, 2 empty ranges are generally not equal (they need to share the same start to be considered equal). This means that, when it comes to comparing ranges, an empty range is interpreted as a

position between its end and start. For example, a typical usecase is comparison of insertion points defined along a string (like a DNA sequence) and represented as empty ranges.

Ranges are ordered by starting position first, and then by width. This way, the space of ranges is totally ordered. On a Ranges object, `order`, `sort`, and `rank` are consistent with this order.

`compare(x, y)`: Performs "generalized range-wise comparison" of `x` and `y`, that is, returns an integer vector where the i-th element is a code describing how the i-th element in `x` is qualitatively positioned relatively to the i-th element in `y`.

Here is a summary of the 13 predefined codes (and their letter equivalents) and their meanings:

```
-6 a: x[i]: .oooo.......        6 m: x[i]: .......oooo.
      y[i]: .......oooo.             y[i]: .oooo.......

-5 b: x[i]: ..oooo......        5 l: x[i]: ......oooo..
      y[i]: ......oooo..             y[i]: ..oooo......

-4 c: x[i]: ...oooo.....        4 k: x[i]: .....oooo...
      y[i]: .....oooo...             y[i]: ...oooo.....

-3 d: x[i]: ...oooooo...        3 j: x[i]: .....oooo...
      y[i]: .....oooo...             y[i]: ...oooooo...

-2 e: x[i]: ..oooooooo..        2 i: x[i]: ....oooo....
      y[i]: ....oooo....             y[i]: ..oooooooo..

-1 f: x[i]: ...oooo.....        1 h: x[i]: ...oooooo...
      y[i]: ...oooooo...             y[i]: ...oooo.....

                0 g: x[i]: ...oooooo...
                     y[i]: ...oooooo...
```

Note that this way of comparing ranges is a refinement over the standard ranges comparison defined by the `==`, `!=`, `<=`, `>=`, `<` and `>` operators. In particular a code that is `< 0`, `= 0`, or `> 0`, corresponds to `x[i] < y[i]`, `x[i] == y[i]`, or `x[i] > y[i]`, respectively.

The `compare` method for Ranges objects is guaranteed to return predefined codes only but methods for other objects (e.g. for GenomicRanges objects) can return non-predefined codes. Like for the predefined codes, the sign of any non-predefined code must tell whether `x[i]` is less than, or greater than `y[i]`.

`rangeComparisonCodeToLetter(x)`: Translate the codes returned by `compare`. The 13 predefined codes are translated as follow: -6 -> a; -5 -> b; -4 -> c; -3 -> d; -2 -> e; -1 -> f; 0 -> g; 1 -> h; 2 -> i; 3 -> j; 4 -> k; 5-> l; 6 -> m. Any non-predefined code is translated to X. The translated codes are returned in a factor with 14 levels: a, b, ..., l, m, X.

`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 range in `table` (or `nomatch` if there is no matching range) for each range in `x`.

`selfmatch(x, method=c("auto", "quick", "hash"))`: Equivalent to, but more efficient than, `match(x, x, method=method)`.

`duplicated(x, fromLast=FALSE, method=c("auto", "quick", "hash"))`: Determines which elements of `x` are equal to elements with smaller subscripts, and returns a logical vector indicating which elements are duplicates. `duplicated(x)` is equivalent to, but more efficient than, `duplicated(as.data.frame(x))` on a Ranges object. See `duplicated` in the **base** package for more details.

`unique(x, fromLast=FALSE, method=c("auto", "quick", "hash"))`: Removes duplicate ranges from `x`. `unique(x)` is equivalent to, but more efficient than, `unique(as.data.frame(x))` on a Ranges object. See `unique` in the **base** package for more details.

`x %in% table`: A shortcut for finding the ranges in `x` that match any of the ranges in `table`. Returns a logical vector of length equal to the number of ranges in `x`.

`findMatches(x, table, method=c("auto", "quick", "hash"))`: An enhanced version of `match` that returns all the matches in a Hits object.

`countMatches(x, table, method=c("auto", "quick", "hash"))`: Returns an integer vector of the length of `x` containing the number of matches in `table` for each element in `x`.

`order(...)`: Returns a permutation which rearranges its first argument (a Ranges object) into ascending order, breaking ties by further arguments (also Ranges objects). See `order` in the **BiocGenerics** package for more information.

`sort(x)`: Sorts `x`. See `sort` in the **base** package for more details.

`rank(x, na.last=TRUE, ties.method=c("average", "first", "random", "max", "min"))`: Returns the sample ranks of the ranges in `x`. See `rank` in the **base** package for more details.

Author(s)

H. Pages

See Also

- The Ranges class.
- GenomicRanges-comparison in the **GenomicRanges** package for comparing and ordering genomic ranges.
- intra-range-methods and inter-range-methods for intra and inter range transformations.
- setops-methods for set operations on IRanges objects.
- `findOverlaps` for finding overlapping ranges.
- Vector-comparison for comparing, ordering, and tabulating vector-like objects.

Examples

```
## ---------------------------------------------------------------------
## A. ELEMENT-WISE (AKA "PARALLEL") COMPARISON OF 2 Ranges OBJECTS
## ---------------------------------------------------------------------
x0 <- IRanges(1:11, width=4)
x0
y0 <- IRanges(6, 9)
compare(x0, y0)
compare(IRanges(4:6, width=6), y0)
compare(IRanges(6:8, width=2), y0)
```

```
compare(x0, y0) < 0   # equivalent to x0 < y0
compare(x0, y0) == 0  # equivalent to x0 == y0
compare(x0, y0) > 0   # equivalent to x0 > y0

rangeComparisonCodeToLetter(-10:10)
rangeComparisonCodeToLetter(compare(x0, y0))

## Handling of zero-width ranges (a.k.a. empty ranges):
x1 <- IRanges(11:17, width=0)
x1
compare(x1, x1[4])
compare(x1, IRanges(12, 15))

## Note that x1[2] and x1[6] are empty ranges on the edge of non-empty
## range IRanges(12, 15). Even though -1 and 3 could also be considered
## valid codes for describing these configurations, compare()
## considers x1[2] and x1[6] to be *adjacent* to IRanges(12, 15), and
## thus returns codes -5 and 5:
compare(x1[2], IRanges(12, 15))  # -5
compare(x1[6], IRanges(12, 15))  #  5

x2 <- IRanges(start=c(20L, 8L, 20L, 22L, 25L, 20L, 22L, 22L),
              width=c( 4L, 0L, 11L,  5L,  0L,  9L,  5L,  0L))
x2

which(width(x2) == 0)  # 3 empty ranges
x2[2] == x2[2]  # TRUE
x2[2] == x2[5]  # FALSE
x2 == x2[4]
x2 >= x2[3]

## ---------------------------------------------------------------------
## B. match(), selfmatch(), %in%, duplicated(), unique()
## ---------------------------------------------------------------------
table <- x2[c(2:4, 7:8)]
match(x2, table)

x2 %in% table

duplicated(x2)
unique(x2)

## ---------------------------------------------------------------------
## C. findMatches(), countMatches()
## ---------------------------------------------------------------------
findMatches(x2, table)
countMatches(x2, table)

x2_levels <- unique(x2)
countMatches(x2_levels, x2)

## ---------------------------------------------------------------------
## D. order() AND RELATED METHODS
```

```
## ------------------------------------------------------------------
order(x2)
sort(x2)
rank(x2, ties.method="first")
```

RangesList-class *List of Ranges*

Description

An extension of List that holds only Ranges objects. Useful for storing ranges over a set of spaces (e.g. chromosomes), each of which requires a separate Ranges object. As a Vector, RangesList may be annotated with its universe identifier (e.g. a genome) in which all of its spaces exist.

Accessors

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

All of these accessors collapse over the spaces:

`start(x)`, `start(x) <- value`: Get or set the starts of the ranges. When setting the starts, value can be an integer vector of `length(sum(elementLengths(x)))` or an IntegerList object of length `length(x)` and names `names(x)`.

`end(x)`, `end(x) <- value`: Get or set the ends of the ranges. When setting the starts, value can be an integer vector of `length(sum(elementLengths(x)))` or an IntegerList object of length `length(x)` and names `names(x)`.

`width(x)`, `width(x) <- value`: Get or set the widths of the ranges. When setting the starts, value can be an integer vector of `length(sum(elementLengths(x)))` or an IntegerList object of length `length(x)` and names `names(x)`.

`space(x)`: Gets the spaces of the ranges as a character vector. This is equivalent to `names(x)`, except each name is repeated according to the length of its element.

These accessors are for the universe identifier:

`universe(x)`: gets the name of the universe as a single string, if one has been specified, NULL otherwise.

`universe(x) <- value`: sets the name of the universe to value, a single string or NULL.

Constructor

`RangesList(..., universe = NULL)`: Each Ranges in ... becomes an element in the new RangesList, in the same order. This is analogous to the list constructor, except every argument in ... must be derived from Ranges. The universe is specified by the universe parameter, which should be a single string or NULL, to leave unspecified.

Coercion

In the code snippets below, x and from are a RangesList object.

- as.data.frame(x, row.names = NULL, optional = FALSE): Coerces x to a data.frame. Essentially the same as calling data.frame(space=rep(names(x), elementLengths(x)), as.data
- as(from, "SimpleIRangesList"): Coerces from, to a SimpleIRangesList, requiring that all Ranges elements are coerced to internal IRanges elements. This is a convenient way to ensure that all Ranges have been imported into R (and that there is no unwanted overhead when accessing them).
- as(from, "CompressedIRangesList"): Coerces from, to a CompressedIRangesList, requiring that all Ranges elements are coerced to internal IRanges elements. This is a convenient way to ensure that all Ranges have been imported into R (and that there is no unwanted overhead when accessing them).
- as(from, "SimpleNormalIRangesList"): Coerces from, to a SimpleNormalIRangesList, requiring that all Ranges elements are coerced to internal NormalIRanges elements.
- as(from, "CompressedNormalIRangesList"): Coerces from, to a CompressedNormalIRangesList, requiring that all Ranges elements are coerced to internal NormalIRanges elements.

Arithmetic Operations

Any arithmetic operation, such as x + y, x * y, etc, where x is a RangesList, is performed identically on each element. Currently, Ranges supports only the * operator, which zooms the ranges by a numeric factor.

Author(s)

Michael Lawrence

See Also

List, the parent of this class, for more functionality.

Examples

```
range1 <- IRanges(start=c(1,2,3), end=c(5,2,8))
range2 <- IRanges(start=c(15,45,20,1), end=c(15,100,80,5))
named <- RangesList(one = range1, two = range2)
length(named) # 2
start(named) # same as start(c(range1, range2))
names(named) # "one" and "two"
named[[1]] # range1
unnamed <- RangesList(range1, range2)
names(unnamed) # NULL

# edit the width of the ranges in the list
edited <- named
width(edited) <- rep(c(3,2), elementLengths(named))
edited
```

```
# same as list(range1, range2)
as.list(RangesList(range1, range2))

# coerce to data.frame
as.data.frame(named)

# set the universe
universe(named) <- "hg18"
universe(named)
RangesList(range1, range2, universe = "hg18")

## zoom in 2X
collection <- RangesList(one = range1, range2)
collection * 2
```

RangesMapping-class *Mapping of ranges to another sequence*

Description

The map generic converts a set of ranges to the equivalent ranges on another sequence, through some sort of alignment between sequences, and outputs a RangesMapping object. There are three primary components of that object: the transformed ranges, the space (destination sequence) for the ranges, and the hits, a Hits object of the same length that matches each input range to a destination sequence (useful when the alignment is one/many to many). The pmap function is simpler: it treats the two inputs as parallel vectors, maps each input range via the corresponding alignment, and returns the mapped ranges. There is one result per input element, instead of the many-to-many result from map.

Usage

```
map(from, to, ...)
pmap(from, to, ...)
```

Arguments

from	Typically an object containing ranges to map.
to	Typically an object representing an alignment.
...	Arguments to pass to methods

Value

A RangesMapping object, as documented here.

RangesMapping Accessors

ranges(x): Gets the mapped ranges.

space(x): Gets the destination spaces (sequence names).

hits(x): Gets the matching between the input ranges and the destination sequences (of which there may be more than one).

dim(x): Same as dim(hits(x)).

length(x): Same as length(hits(x)).

subjectHits(x): Same as subjectHits(hits(x)).

queryHits(x): Same as queryHits(hits(x)).

RangesMapping Coercion

as(from, "RangedData"): Converts a RangesMapping into a RangedData. The ranges/space in the RangedData are the ranges/space of from, and the values result from the coercion of the hits to a DataFrame.

Author(s)

Michael Lawrence

See Also

Methods on the generic map, which generates an instance of this class, are defined in other packages, like GenomicRanges.

---

rdapply *Applying over spaces*

Description

The rdapply function applies a user function over the spaces of a RangedData. The parameters to rdapply are collected into an instance of RDApplyParams, which is passed as the sole parameter to rdapply.

Usage

```
rdapply(x, ...)
```

Arguments

x	The RDApplyParams instance, see below for how to make one.
...	Additional arguments for methods

Details

The rdapply function is an attempt to facilitate the common operation of performing the same operation over each space (e.g. chromosome) in a RangedData. To facilitate a wide array of such tasks, the function takes a large number of options. The RDApplyParams class is meant to help manage this complexity. In particular, it facilitates experimentation through its support for incremental changes to parameter settings.

There are two RangedData settings that are required: the user function object and the RangedData over which it is applied. The rest of the settings determine what is actually passed to the user function and how the return value is processed before relaying it to the user. The following is the description and rationale for each setting.

rangedData **REQUIRED**. The RangedData instance over which applyFun is applied.

applyFun **REQUIRED**. The user function to be applied to each space in the RangedData. The function must expect the RangedData as its first parameter and also accept the parameters specified in applyParams.

applyParams The list of additional parameters to pass to applyFun. Usually empty.

filterRules The instance of FilterRules that is used to filter each subset of the RangedData passed to the user function. This is an efficient and convenient means for performing the same operation over different subsets of the data on a space-by-space basis. In particular, this avoids the need to store subsets of the entire RangedData. A common workflow is to invoke rdapply with one set of active filters, enable different filters, reinvoke rdapply, and compare the results.

simplify A scalar logical (TRUE or FALSE) indicating whether the list to be returned from rdapply should be simplified as by sapply. Defaults to FALSE.

reducerFun The function that is used to convert the list that would otherwise be returned from rdapply to something more convenient. The function should take the list as its first parameter and also accept the parameters specified in reducerParams. This is an alternative to the primitive behavior of the simplify option (so simplify must be FALSE if this option is set). The aim is to orthogonalize the applyFun operation (i.e. the statistics) from the data structure of the result.

reducerParams A list of additional parameters to pass to reducerFun. Can only be set if reducerFun is set. Usually empty.

iteratorFun The function used for applying over the RangedData. By default, this is lapply, but it could also be a specialized function, like mclapply.

Value

By default a list holding the result of each invocation of the user function, but see details.

Constructing an RDApplyParams object

RDApplyParams(rangedData, applyFun, applyParams, filterRules, simplify, reducerFun, reducerParar
Constructs a RDApplyParams object with each setting specified by the argument of the same name. See the Details section for more information.

Accessors

In the following code snippets, x is an RDApplyParams object.

rangedData(x), rangedData(x) <- value: Get or set the RangedData instance over which applyFun is applied.

applyFun(x), applyFun(x) <- value: Get or set the user function to be applied to each space in the RangedData.

applyParams(x), applyParams(x) <- value: Get or set the list of additional parameters to pass to applyFun.

filterRules(x), filterRules(x) <- value: Get or set the instance of FilterRules that is used to filter each subset of the RangedData passed to the user function.

simplify(x), simplify(x) <- value: Get or set a a scalar logical (TRUE or FALSE) indicating whether the list to be returned from rdapply should be simplified as by sapply.

reducerFun(x), reducerFun(x) <- value: Get or set the function that is used to convert the list that would otherwise be returned from rdapply to something more convenient.

reducerParams(x), reducerParams(x) <- value: Get or set a list of additional parameters to pass to reducerFun.

iteratorFun(x), iteratorFun(x) <- value: Get or set the function used for applying over the RangedData.

Author(s)

Michael Lawrence

See Also

RangedData, FilterRules

Examples

```
ranges <- IRanges(c(1,2,3),c(4,5,6))
score <- c(2L, 0L, 1L)
rd <- RangedData(ranges, score, space = c("chr1","chr2","chr1"))

## a single function
countrows <- function(rd) nrow(rd)
params <- RDApplyParams(rd, countrows)
rdapply(params) # list(chr1 = 2L, chr2 = 1L)

## with a parameter
params <- RDApplyParams(rd, function(rd, x) nrow(rd)*x, list(x = 2))
rdapply(params) # list(chr1 = 4L, chr2 = 2L)

## add a filter
cutoff <- 0
rules <- FilterRules(filter = score > cutoff)
params <- RDApplyParams(rd, countrows, filterRules = rules)
rdapply(params) # list(chr1 = 2L, chr2 = 0L)
```

```
rules <- FilterRules(list(fun = function(rd) rd[["score"]] < 2),
                     filter = score > cutoff)
params <- RDApplyParams(rd, countrows, filterRules = rules)
rdapply(params) # list(chr1 = 1L, chr2 = 0L)
active(filterRules(params))["filter"] <- FALSE
rdapply(params) # list(chr1 = 1L, chr2 = 1L)

## simplify
params <- RDApplyParams(rd, countrows, simplify = TRUE)
rdapply(params) # c(chr1 = 2L, chr2 = 1L)

## reducing
params <- RDApplyParams(rd, countrows, reducerFun = unlist,
                        reducerParams = list(use.names = FALSE))
rdapply(params) ## c(2L, 1L)
```

`read.Mask` *Read a mask from a file*

Description

`read.agpMask` and `read.gapMask` extract the AGAPS mask from an NCBI "agp" file or a UCSC "gap" file, respectively.

`read.liftMask` extracts the AGAPS mask from a UCSC "lift" file (i.e. a file containing offsets of contigs within sequences).

`read.rmMask` extracts the RM mask from a RepeatMasker .out file.

`read.trfMask` extracts the TRF mask from a Tandem Repeats Finder .bed file.

Usage

```
read.agpMask(file, seqname="?", mask.width=NA, gap.types=NULL, use.gap.types=FALSE)
read.gapMask(file, seqname="?", mask.width=NA, gap.types=NULL, use.gap.types=FALSE)
read.liftMask(file, seqname="?", mask.width=NA)
read.rmMask(file, seqname="?", mask.width=NA, use.IDs=FALSE)
read.trfMask(file, seqname="?", mask.width=NA)
```

Arguments

`file`	Either a character string naming a file or a connection open for reading.
`seqname`	The name of the sequence for which the mask must be extracted. If no sequence is specified (i.e. `seqname="?"`) then an error is raised and the sequence names found in the file are displayed. If the file doesn't contain any information for the specified sequence, then a warning is issued and an empty mask of width `mask.width` is returned.
`mask.width`	The width of the mask to return i.e. the length of the sequence this mask will be put on. See ?`MaskCollection-class` for more information about the width of a MaskCollection object.

gap.types — NULL or a character vector containing gap types. Use this argument to filter the assembly gaps that are to be extracted from the "agp" or "gap" file based on their type. Most common gap types are "contig", "clone", "centromere", "telomere", "heterochromatin", "short_arm" and "fragment". With gap.types=NULL, all the assembly gaps described in the file are extracted. With gap.types="?", an error is raised and the gap types found in the file for the specified sequence are displayed.

use.gap.types — Whether or not the gap types provided in the "agp" or "gap" file should be used to name the ranges constituing the returned mask. See ?IRanges-class for more information about the names of an IRanges object.

use.IDs — Whether or not the repeat IDs provided in the RepeatMasker .out file should be used to name the ranges constituing the returned mask. See ?IRanges-class for more information about the names of an IRanges object.

See Also

MaskCollection-class, IRanges-class

Examples

```
## ---------------------------------------------------------------------
## A. Extract a mask of assembly gaps ("AGAPS" mask) with read.agpMask()
## ---------------------------------------------------------------------
## Note: The hs_b36v3_chrY.agp file was obtained by downloading,
## extracting and renaming the hs_ref_chrY.agp.gz file from
##
##   ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/Assembled_chromosomes/
##     hs_ref_chrY.agp.gz      5 KB  24/03/08  04:33:00 PM
##
## on May 9, 2008.

chrY_length <- 57772954
file1 <- system.file("extdata", "hs_b36v3_chrY.agp", package="IRanges")
mask1 <- read.agpMask(file1, seqname="chrY", mask.width=chrY_length,
                      use.gap.types=TRUE)
mask1
mask1[[1]]

mask11 <- read.agpMask(file1, seqname="chrY", mask.width=chrY_length,
                       gap.types=c("centromere", "heterochromatin"))
mask11[[1]]

## ---------------------------------------------------------------------
## B. Extract a mask of assembly gaps ("AGAPS" mask) with read.liftMask()
## ---------------------------------------------------------------------
## Note: The hg18liftAll.lft file was obtained by downloading,
## extracting and renaming the liftAll.zip file from
##
##   http://hgdownload.cse.ucsc.edu/goldenPath/hg18/bigZips/
##     liftAll.zip             03-Feb-2006 11:35  5.5K
##
```

```
## on May 8, 2008.

file2 <- system.file("extdata", "hg18liftAll.lft", package="IRanges")
mask2 <- read.liftMask(file2, seqname="chr1")
mask2
if (interactive()) {
    ## contigs 7 and 8 for chrY are adjacent
    read.liftMask(file2, seqname="chrY")

    ## displays the sequence names found in the file
    read.liftMask(file2)

    ## specify an unknown sequence name
    read.liftMask(file2, seqname="chrZ", mask.width=300)
}

## ---------------------------------------------------------------------
## C. Extract a RepeatMasker ("RM") or Tandem Repeats Finder ("TRF")
##    mask with read.rmMask() or read.trfMask()
## ---------------------------------------------------------------------
## Note: The ce2chrM.fa.out and ce2chrM.bed files were obtained by
## downloading, extracting and renaming the chromOut.zip and
## chromTrf.zip files from
##
##   http://hgdownload.cse.ucsc.edu/goldenPath/ce2/bigZips/
##     chromOut.zip            21-Apr-2004 09:05  2.6M
##     chromTrf.zip            21-Apr-2004 09:07  182K
##
## on May 7, 2008.

## Before you can extract a mask with read.rmMask() or read.trfMask(), you
## need to know the length of the sequence that youre going to put the
## mask on:
if (interactive()) {
    library(BSgenome.Celegans.UCSC.ce2)
    chrM_length <- seqlengths(Celegans)[["chrM"]]

    ## Read the RepeatMasker .out file for chrM in ce2:
    file3 <- system.file("extdata", "ce2chrM.fa.out", package="IRanges")
    RMmask <- read.rmMask(file3, seqname="chrM", mask.width=chrM_length)
    RMmask

    ## Read the Tandem Repeats Finder .bed file for chrM in ce2:
    file4 <- system.file("extdata", "ce2chrM.bed", package="IRanges")
    TRFmask <- read.trfMask(file4, seqname="chrM", mask.width=chrM_length)
    TRFmask
    desc(TRFmask) <- paste(desc(TRFmask), "[period<=12]")
    TRFmask

    ## Put the 2 masks on chrM:
    chrM <- Celegans$chrM
    masks(chrM) <- RMmask  # this would drop all current masks, if any
    masks(chrM) <- append(masks(chrM), TRFmask)
```

```
    chrM
}
```

---

reverse *reverse*

Description

A generic function for reversing vector-like or list-like objects. This man page describes methods for reversing a character vector, a Views object, or a MaskCollection object. Note that `reverse` is similar to but not the same as `rev`.

Usage

```
reverse(x, ...)
```

Arguments

`x`	A vector-like or list-like object.
`...`	Additional arguments to be passed to or from methods.

Details

On a character vector or a Views object, `reverse` reverses each element individually, without modifying the top-level order of the elements. More precisely, each individual string of a character vector is reversed.

Value

An object of the same class and length as the original object.

See Also

reverse-methods, Views-class, MaskCollection-class, `endoapply`, `rev`

Examples

```
## On a character vector:
reverse(c("Hi!", "How are you?"))
rev(c("Hi!", "How are you?"))

## On a Views object:
v <- successiveViews(Rle(c(-0.5, 12.3, 4.88), 4:2), 1:4)
v
reverse(v)
rev(v)

## On a MaskCollection object:
mask1 <- Mask(mask.width=29, start=c(11, 25, 28), width=c(5, 2, 2))
```

```
mask2 <- Mask(mask.width=29, start=c(3, 10, 27), width=c(5, 8, 1))
mask3 <- Mask(mask.width=29, start=c(7, 12), width=c(2, 4))
mymasks <- append(append(mask1, mask2), mask3)
reverse(mymasks)
```

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.

Constructors

`Rle(values)`: This constructor creates an Rle instances out of an atomic vector `values`.

`Rle(values, lengths)`: This constructor creates an Rle instances 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.

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

Accessors

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)`.

Replacers

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

In the code snippets below, x and from are Rle objects:

- as.vector(x, mode="any"), as(from, "vector"): Creates an atomic vector based on the values contained in x. The vector will be coerced to the requested mode, unless mode is "any", in which case the most appropriate type is chosen.
- as.vectorORfactor(x): Creates an atomic vector or factor, based on the type of values contained in x. This is the most general way to decompress the Rle to a native R data structure.
- as.logical(x), as(from, "logical"): Creates a logical vector based on the values contained in x.
- as.integer(x), as(from, "integer"): Creates an integer vector based on the values contained in x.
- as.numeric(x), as(from, "numeric"): Creates a numeric vector based on the values contained in x.
- as.complex(x), as(from, "complex"): Creates a complex vector based on the values contained in x.
- as.character(x), as(from, "character"): Creates a character vector based on the values contained in x.
- as.raw(x), as(from, "raw"): Creates a raw vector based on the values contained in x.
- 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).
- as(from, "IRanges"): Creates an IRanges instance from a logical Rle. Note that this instance is guaranteed to be normal.
- as(from, "NormalIRanges"): Creates a NormalIRanges instance from a logical Rle.

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.

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.

`aggregate(x, by, FUN, start = NULL, end = NULL, width = NULL, frequency = NULL, delta = NU`
Generates summaries on the specified windows and returns the result in a convenient form:

by An object with start, end, and width methods.

FUN The function, found via match.fun, to be applied to each window of x.

start, end, width the start, end, or width of the window. If by is missing, then must supply two of the three.

frequency, delta Optional arguments that specify the sampling frequency and increment within the window.

... Further arguments for FUN.

simplify A logical value specifying whether or not the result should be simplified to a vector or matrix if possible.

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

values the Rle to insert.

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

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

`findRange(x, vec)`: Returns an IRanges object representing the ranges in Rle vec that are referenced by the indices in the integer vector x.

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

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

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

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

na.rm remove missing values from check.

strictly check for _strictly_ increasing values.

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

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

table the values to be matched against.

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

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

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

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

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

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

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

`shiftApply(SHIFT, X, Y, FUN, ..., OFFSET = 0L, simplify = TRUE, verbose = FALSE)`: 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)`. Calculates the set of `FUN(X_i, Y_i, ...)` values and return the results in a convenient form:

`SHIFT` A non-negative integer vector of shift values.

`X, Y` The Rle 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.

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

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

`split(x, f, drop=FALSE)`: Splits `x` according to `f` to create a CompressedRleList object. If `f` is a list-like object then `drop` is ignored and `f` is treated as if it was `rep(seq_len(length(f)), sapply(f, length))`, so the returned object has the same shape as `f` (it also receives the names of `f`). Otherwise, if `f` is not a list-like object, empty list elements are removed from the returned object if `drop` is `TRUE`.

`splitRanges(x)`: Returns a CompressedIRangesList object that contain the ranges for each of the unique run values.

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

`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()`.

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

`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.

`window(x, start=NA, end=NA, width=NA, frequency=NULL, delta=NULL, ...)`: Extract the subsequence window from `x` specified by:

`start`, `end`, `width` The start, end, or width of the window. Two of the three are required.

`frequency`, `delta` Optional arguments that specify the sampling frequency and increment within the window.

`window(x, start=NA, end=NA, width=NA) <- value`: Replace the subsequence window specified on the left (i.e. the subsequence in `x` specified by `start`, `end` and `width`) by `value`. `value` must either be of class Rle, belong to a subclass of Rle, or be coercible to Rle or a subclass of Rle. The elements of `value` are repeated to create an Rle with the same number of elements as the width of the subsequence window it is replacing.

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`.

`ifelse(x, yes, no)`: For each element of `x`, returns the corresponding element in `yes` if `TRUE`, otherwise the element in `no`. `yes` and `no` may be `Rle` objects or anything else coercible to a vector.

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.

`runmean(x, k, endrule = c("drop", "constant"), na.rm = FALSE)`: Calculates the means for fixed width running windows across `x`.

`k` An integer indicating the fixed width of the running window. Must be odd when `endrule == "constant"`.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

`"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*;

`na.rm` A logical indicating if NA and NaN values should be removed.

`runmed(x, k, endrule = c("median", "keep", "drop", "constant"))`: Calculates the medians for fixed width running windows across `x`.

`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.

`"keep"` keeps the first and last k_2 values at both ends, where k_2 is the half-bandwidth `k2 = k %/% 2`, i.e., `y[j] = x[j]` for $j \in \{1, \ldots, k_2; n - k_2 + 1, \ldots, n\}$ $j = 1, .., k2 and (n - k2 + 1), .., n$;

`"constant"` copies the running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

"median" the default, smooths the ends by using symmetrical medians of subsequently smaller bandwidth, but for the very first and last value where Tukey's robust end-point rule is applied, see smoothEnds.

runsum(x, k, endrule = c("drop", "constant"), na.rm = FALSE): Calculates the sums for fixed width running windows across x.

k An integer indicating the fixed width of the running window. Must be odd when endrule == "constant".

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

"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*;

na.rm A logical indicating if NA and NaN values should be removed.

runwtsum(x, k, wt, endrule = c("drop", "constant"), na.rm = FALSE): Calculates the sums for fixed width running windows across x.

k An integer indicating the fixed width of the running window. Must be odd when endrule == "constant".

wt A numeric vector of length k that provides the weights to use.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

"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*;

na.rm A logical indicating if NA and NaN values should be removed.

runq(x, k, i, endrule = c("drop", "constant")): Calculates the order statistic for fixed width running windows across x.

k An integer indicating the fixed width of the running window. Must be odd when endrule == "constant".

i An integer indicating which order statistic to calculate.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

"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*;

na.rm A logical indicating if NA and NaN values should be removed.

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 = FAL`
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 = F`
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.

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`, Vector-class, S4groupGeneric, IRanges-class

Examples

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

runLength(x)
runValue(x)
```

```
nrun(x)

diff(x)
unique(x)
sort(x)
sqrt(x)
x^2 + 2 * x + 1
x[c(1,3,5,7,9)]
window(x, 4, 14)
range(x)
sum(x)
mean(x)
x > 4
aggregate(x, x > 4, mean)
aggregate(x, FUN = mean, start = 1:(length(x) - 50), end = 51:length(x))

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)

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

## ---------------------------------------------------------------------
## runsum, runmean, runwtsum, and runq functions
## ---------------------------------------------------------------------

## 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 NAs 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)
```

`RleViews-class` *The RleViews class*

Description

The RleViews class is the basic container for storing a set of views (start/end locations) on the same Rle object.

Details

An RleViews object contains a set of views (start/end locations) on the same Rle object called "the subject vector" or simply "the subject". Each view is defined by its start and end locations: both are integers such that start <= end. An RleViews object is in fact a particular case of a Views object (the RleViews class contains the Views class) so it can be manipulated in a similar manner: see ?`Views` for more information. Note that two views can overlap and that a view can be "out of limits" i.e. it can start before the first element of the subject or/and end after its last element.

Author(s)

P. Aboyoun

See Also

Views-class, Rle-class, view-summarization-methods

Examples

```
subject <- Rle(rep(c(3L, 2L, 18L, 0L), c(3,2,1,5)))
myViews <- Views(subject, 3:0, 5:8)
myViews
subject(myViews)
length(myViews)
start(myViews)
end(myViews)
width(myViews)
myViews[[2]]

set.seed(0)
vec <- Rle(sample(0:2, 20, replace = TRUE))
vec
Views(vec, vec > 0)
```

`RleViewsList-class` *List of RleViews*

Description

An extension of ViewsList that holds only RleViews objects. Useful for storing coverage vectors over a set of spaces (e.g. chromosomes), each of which requires a separate RleViews object.

Details

For more information on methods available for RleViewsList objects consult the man pages for ViewsList-class and view-summarization-methods.

Constructor

`RleViewsList(..., rleList, rangesList, universe = NULL)`: Either `...` or the `rleList`/`rangesList` couplet provide the RleViews for the list. If `...` is provided, each of these arguments must be RleViews objects. Alternatively, `rleList` and `rangesList` accept Rle and Ranges objects respectively that are meshed together for form the RleViewsList. The universe is specified by the `universe` parameter, which should be a single string or NULL, to leave unspecified.

`Views(subject, start=NULL, end=NULL, width=NULL, names=NULL)`: Same as `RleViewsList(rleList = subject, r`

Coercion

In the code snippets below, `from` is an RleViewsList object:

`as(from, "IRangesList")`: Creates a `CompressedIRangesList` object containing the view locations in `from`.

`as(from, "CompressedIRangesList")`: Creates a `CompressedIRangesList` object containing the view locations in `from`.

`as(from, "SimpleIRangesList")`: Creates a `SimpleIRangesList` object containing the view locations in `from`.

Author(s)

P. Aboyoun

See Also

ViewsList-class, view-summarization-methods

Examples

```
## Rle objects
subject1 <- Rle(c(3L,2L,18L,0L), c(3,2,1,5))
set.seed(0)
subject2 <- Rle(c(0L,5L,2L,0L,3L), c(8,5,2,7,4))

## Views
rleViews1 <- Views(subject1, 3:0, 5:8)
rleViews2 <- Views(subject2, subject2 > 0)

## RleList and RangesList objects
rleList <- RleList(subject1, subject2)
rangesList <- IRangesList(IRanges(3:0, 5:8), IRanges(subject2 > 0))

## methods for construction
method1 <- RleViewsList(rleViews1, rleViews2)
method2 <- RleViewsList(rleList = rleList, rangesList = rangesList)
identical(method1, method2)

## calculation over the views
viewSums(method1)
```

---

runstat *Fixed width running window summaries across vector-like objects*

Description

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

Usage

```
runsum(x, k, endrule = c("drop", "constant"), ...)
runmean(x, k, endrule = c("drop", "constant"), ...)
runwtsum(x, k, wt, endrule = c("drop", "constant"), ...)
runq(x, k, i, endrule = c("drop", "constant"), ...)
```

Arguments

x	The data object.
k	An integer indicating the fixed width of the running window. Must be odd when endrule == "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.
endrule	A character string indicating how the values at the beginning and the end (of the data) should be treated. "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*;
...	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, 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
```

score *Score accessor and setter*

Description

Gets and sets the score of an object.

Usage

```
score(x, ...)
score(x, ...) <- value
```

Arguments

x	An object to get or set the score value of.
value	A new score value.
...	Additional arguments.

seqapply *Apply function and cast to Vector*

Description

The seqapply family of functions behaves much like the existing lapply family, except the return value is cast to a Vector subclass. This facilitates constraining computation to the Vector framework across iteration and (for seqsplit) splitting.

Usage

```
## The seqapply family:
seqapply(X, FUN, ...)
mseqapply(FUN, ..., MoreArgs = NULL, USE.NAMES = TRUE)
tseqapply(X, INDEX, FUN = NULL, ...)
seqsplit(x, f, drop = FALSE)
seqby(data, INDICES, FUN, ...)

## Reverse seqsplit():
## S4 method for signature List
unsplit(value, f, drop = FALSE)
## S4 replacement method for signature Vector
split(x, f, drop = FALSE, ...) <- value
```

Arguments

X	The object over which to iterate, usually a vector or Vector
x	Like X
data	Like X
FUN	The function that is applied to each element of X
MoreArgs	Additional arguments to FUN that are treated like scalars
USE.NAMES	Whether the return values should inherit names from one of the arguments
INDEX	A list of factors to split X into subsets, each of which is passed in a separate invocation of FUN
INDICES	Like INDEX, except a single factor need not be in a list.
f	A factor or list of factors
drop	Whether to drop empty elements from the returned list
...	Extra arguments to pass to FUN
value	The List object to unsplit.

Details

The functions in the seqapply family should be used just like their base equivalent:

seqapply => lapply

mseqapply => mapply

tseqapply => tapply

seqsplit => split

seqby => by

The only difference is that the result is cast to a Vector object. The casting logic simply looks for a common class from which all returned values inherit. It then checks for the existence of a function of the form ClassList where Class is the name of the class. If such a function is not found, the search proceeds up the hierarchy of classes. An error is thrown when hierarchy is exhausted. If ClassList is found, it is called with the list of return values as its only argument, under the assumption that a Vector-derived instance will be constructed.

unsplit unlists value, where the order of the returned vector is as if value were originally created by splitting that vector on the factor f.

split(x, f, drop = FALSE) <- value: Virtually splits x by the factor f, replaces the elements of the resulting list with the elements from the list value, and restores x to its original form. Note that this works for any Vector, even though split itself is not universally supported.

Value

A List object for the functions in the seqapply family.

Author(s)

Michael Lawrence

Examples

```
starts <- IntegerList(c(1, 5), c(2, 8))
ends <- IntegerList(c(3, 8), c(5, 9))
rangesList <- mseqapply(IRanges, starts, ends)
rangeDataFrame <- stack(rangesList, "space", "ranges")
dataFrameList <- seqsplit(rangeDataFrame, rangeDataFrame$space)
starts <- seqapply(dataFrameList[,"ranges"], start)
```

---

setops-methods *Set operations on IRanges, RangesList, and Hits objects*

Description

Performs set operations on IRanges objects.

Usage

```
## Vector-wise operations:
## S4 method for signature IRanges,IRanges
union(x, y,...)
## S4 method for signature IRanges,IRanges
intersect(x, y,...)
## S4 method for signature IRanges,IRanges
setdiff(x, y,...)

## Element-wise (aka "parallel") operations:
## S4 method for signature IRanges,IRanges
punion(x, y, fill.gap=FALSE, ...)
## S4 method for signature IRanges,IRanges
pintersect(x, y, resolve.empty=c("none", "max.start", "start.x"), ...)
## S4 method for signature IRanges,IRanges
psetdiff(x, y, ...)
## S4 method for signature IRanges,IRanges
pgap(x, y, ...)
```

Arguments

x, y	IRanges objects.
fill.gap	Logical indicating whether or not to force a union by using the rule start = min(start(x), start(y)),
resolve.empty	One of "none", "max.start", or "start.x" denoting how to handle ambiguous empty ranges formed by intersections. "none" - throw an error if an ambiguous empty range is formed, "max.start" - associate the maximum start value with any ambiguous empty range, and "start.x" - associate the start value of x with any ambiguous empty range. (See Details section below for the definition of an ambiguous range.)
...	Further arguments to be passed to or from other methods.

Details

The `union`, `intersect` and `setdiff` methods for IRanges objects return a "normal" IRanges object (of the same class as `x`) representing the union, intersection and (asymmetric!) difference of the sets of integers represented by `x` and `y`.

`punion`, `pintersect`, `psetdiff` and `pgap` are generic functions that compute the element-wise (aka "parallel") union, intersection, (asymmetric!) difference and gap between each element in `x` and its corresponding element in `y`. Methods for IRanges objects are defined. For these methods, `x` and `y` must have the same length (i.e. same number of ranges) and they return an IRanges instance of the same length as `x` and `y` where each range represents the union/intersection/difference/gap of/between the corresponding ranges in `x` and `y`.

By default, `pintersect` will throw an error when an "ambiguous empty range" is formed. An ambiguous empty range can occur three different ways: 1) when corresponding non-empty ranges elements `x` and `y` have an empty intersection, 2) if the position of an empty range element does not fall within the corresponding limits of a non-empty range element, or 3) if two corresponding empty range elements do not have the same position. For example if empty range element [22,21] is intersected with non-empty range element [1,10], an error will be produced; but if it is intersected with the range [22,28], it will produce [22,21]. As mentioned in the Arguments section above, this behavior can be changed using the `resolve.empty` argument.

Author(s)

H. Pages and M. Lawrence

See Also

`pintersect` is similar to `narrow`, except the end points are absolute, not relative. `pintersect` is also similar to `restrict`, except ranges outside of the restriction become empty and are not discarded.

union,

Ranges-class,

intra-range-methods for intra range transformations,

inter-range-methods for inter range transformations,

IRanges-class, IRanges-utils

Examples

```
x <- IRanges(c(1, 5, -2, 0, 14), c(10, 9, 3, 11, 17))
subject <- Rle(1:-3, 6:2)
y <- Views(subject, start=c(14, 0, -5, 6, 18), end=c(20, 2, 2, 8, 20))

## Vector-wise operations:
union(x, ranges(y))
union(ranges(y), x)

intersect(x, ranges(y))
intersect(ranges(y), x)
```

```
setdiff(x, ranges(y))
setdiff(ranges(y), x)

## Element-wise (aka "parallel") operations:
try(punion(x, ranges(y)))
punion(x[3:5], ranges(y)[3:5])
punion(x, ranges(y), fill.gap=TRUE)
try(pintersect(x, ranges(y)))
pintersect(x[3:4], ranges(y)[3:4])
pintersect(x, ranges(y), resolve.empty="max.start")
psetdiff(ranges(y), x)
try(psetdiff(x, ranges(y)))
start(x)[4] <- -99
end(y)[4] <- 99
psetdiff(x, ranges(y))
pgap(x, ranges(y))

## On RangesList objects:
irl1 <- IRangesList(a = IRanges(c(1,2),c(4,3)), b = IRanges(c(4,6),c(10,7)))
irl2 <- IRangesList(c = IRanges(c(0,2),c(4,5)), a = IRanges(c(4,5),c(6,7)))
union(irl1, irl2)
intersect(irl1, irl2)
setdiff(irl1, irl2)
```

`SimpleList-class` *Simple and Compressed List Classes*

Description

The (non-virtual) SimpleList and (virtual) CompressedList classes extend the List virtual class.

Details

The SimpleList and CompressedList classes provide an implementation that subclasses can easily extend. The underlying storage in a SimpleList subclass is a list object. The underlying storage in a CompressedList object is a virtually partitioned vector-like object. For more information on the available methods, see the List man page.

Constructor

`List` objects are typically constructed by calling the constructor of a concrete implementation, such as `RangesList` or `IntegerList`. The simplest, most generic implementation is `SimpleList`, which has the following constructor:

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

Calling `as(x, "List")` will convert a vector-like object into a `List`, usually a `CompressedList`. To explicitly request a `SimpleList` derivative, call `as(x, "SimpleList")`

Coercion

In the following code snippets, x is a SimpleList or CompressedList object.

`as.list(x)`: Copies the elements of x into a new R list object.

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

Subsetting

In the following code snippets, x is a SimpleList or CompressedList object.

`x[i]`: In addition to normal usage, the `i` parameter can be a `RangesList`, logical `RleList`, `LogicalList`, or `IntegerList` object to perform subsetting within the list elements rather than across them.

`x[i] <- value`: In addition to normal usage, the `i` parameter can be a `RangesList`, logical `RleList`, `LogicalList`, or `IntegerList` object to perform subsetting within the list elements rather than across them.

Looping

In the following code snippets, x is a SimpleList or CompressedList object.

`aggregate(x, by, FUN, start = NULL, end = NULL, width = NULL, frequency = NULL, delta = NU`
In addition to normal usage, the `by` parameter can be a `RangesList` to aggregate within the list elements rather than across them. When `by` is a `RangesList`, the output is either a `SimpleAtomicList` object, if possible, or a `SimpleList` object, if not.

Author(s)

P. Aboyoun

See Also

List, AtomicList and RangesList for example implementations

Examples

```
SimpleList(a = letters, ranges = IRanges(1:10, 1:10))
```

slice-methods *Slice a vector-like or list-like object*

Description

slice is a generic function that creates views on a vector-like or list-like object that contain the elements that are within the specified bounds.

Usage

```
slice(x, lower=-Inf, upper=Inf, ...)

## S4 method for signature Rle
slice(x, lower=-Inf, upper=Inf,
      includeLower=TRUE, includeUpper=TRUE, rangesOnly=FALSE)

## S4 method for signature RleList
slice(x, lower=-Inf, upper=Inf,
      includeLower=TRUE, includeUpper=TRUE, rangesOnly=FALSE)
```

Arguments

x	An Rle or RleList object for the methods described here.
lower, upper	The lower and upper bounds for the slice.
includeLower, includeUpper	Logical indicating whether or not the specified boundary is open or closed.
rangesOnly	A logical indicating whether or not to drop the original data from the output.
...	Additional arguments to be passed to specific methods.

Details

slice is useful for finding areas of absolute maxima (peaks), absolute minima (troughs), or fluctuations within specified limits. One or more view summarization methods can be used on the result of slice. See ?link{view-summarization-methods}

Value

The method for Rle objects returns an RleViews object if rangesOnly=FALSE or an IRanges object if rangesOnly=TRUE.

The method for RleList objects returns an RleViewsList object if rangesOnly=FALSE or an IRangesList object if rangesOnly=TRUE.

Author(s)

P. Aboyoun

See Also

- view-summarization-methods for summarizing the views returned by `slice`.
- slice-methods in the **XVector** package for more `slice` methods.
- `coverage` for computing the coverage across a set of ranges.
- The Rle, RleList, RleViews, and RleViewsList classes.

Examples

```
## Views derived from coverage
x <- IRanges(start=c(1L, 9L, 4L, 1L, 5L, 10L),
             width=c(5L, 6L, 3L, 4L, 3L,  3L))
cvg <- coverage(x)
slice(cvg, lower=2)
slice(cvg, lower=2, rangesOnly=TRUE)
```

---

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

Description

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

Usage

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

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

Arguments

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

Details

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

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

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

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

Value

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

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

Author(s)

H. Pages

See Also

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

Examples

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

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

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

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

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

y <- strsplitAsListOfIntegerVectors(x)
y

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

`updateObject-methods` *Update an object of a class defined in the IRanges package to its current class definition*

Description

The IRanges package provides an extensive collection of `updateObject` methods for updating almost any instance of a class defined in the package.

Usage

```
## Showing usage of method defined for IntegerList objects only (usage
## is the same for all methods).

## S4 method for signature IntegerList
updateObject(object, ..., verbose=FALSE)
```

Arguments

`object`	Object to be updated. Many (but not all) IRanges classes are supported. If no specific method is available for the object, then the default method (defined in the BiocGenerics package) is used. See ?`updateObject` for a description of the default method.
`..., verbose`	See ?`updateObject`.

Value

Returns a valid instance of `object`.

Author(s)

The Bioconductor Dev Team

See Also

`updateObject`

`Vector-class` *Vector objects*

Description

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

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

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

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

Accessors

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

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

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

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

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

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

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

`with(x, expr)`: Evaluates `expr` within `as.env(x)` via `eval(x)`.

`eval(expr, envir, enclos=parent.frame())`: Evaluates `expr` within `envir`, where `envir` is coerced to an environment with `as.env(envir, enclos)`. The `expr` is first processed with `bquote`, such that any escaped symbols are directly resolved in the calling frame.

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

Subsetting

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

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

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

`window(x, start=NA, end=NA, width=NA, frequency=NULL, delta=NULL, ...)`: Extract the subsequence window from the Vector object using:

`start`, `end`, `width` The start, end, or width of the window. Two of the three are required.

`frequency`, `delta` Optional arguments that specify the sampling frequency and increment within the window.

In general, this is more efficient than using `"["` operator.

`window(x, start=NA, end=NA, width=NA) <- value`: Replace the subsequence window specified on the left (i.e. the subsequence in `x` specified by `start`, `end` and `width`) by `value`. `value` must either be of class `class(x)`, belong to a subclass of `class(x)`, or be coercible to `class(x)` or a subclass of `class(x)`. The elements of `value` are repeated to create a Vector with the same number of elements as the width of the subsequence window it is replacing.

`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)`, `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.

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

Combining

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

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

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

`mstack(..., .index.var = "name")`: A variant of `stack`, where the list is taken as the list of arguments in `...`, each of which should be a `Vector` or `vector` (mixing the two will not work).

Looping

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

`tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)`: Like the standard `tapply` function defined in the base package, the `tapply` method for Vector objects applies a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

`shiftApply(SHIFT, X, Y, FUN, ..., OFFSET = 0L, simplify = TRUE, verbose = FALSE)`: 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)`. Calculates the set of `FUN(X_i, Y_i, ...)` values and return the results in a convenient form:

`SHIFT` A non-negative integer vector of shift values.

`X`, `Y` The Vector or R vector 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.

`aggregate(x, by, FUN, start = NULL, end = NULL, width = NULL, frequency = NULL, delta = NU`
Generates summaries on the specified windows and returns the result in a convenient form:

`by` An object with `start`, `end`, and `width` methods.

`FUN` The function, found via `match.fun`, to be applied to each window of x.

`start`, `end`, `width` the start, end, or width of the window. If by is missing, then must supply two of the three.

`frequency`, `delta` Optional arguments that specify the sampling frequency and increment within the window.

... Further arguments for `FUN`.

`simplify` A logical value specifying whether or not the result should be simplified to a vector or matrix if possible.

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)`.

Author(s)

P. Aboyoun

See Also

- Rle and XRaw for example implementations.
- Vector-comparison for comparing, ordering, and tabulating vector-like objects.
- List for a direct extension that serves a similar role as list in base R.
- extractList for grouping elements of a vector-like object into a list-like object.
- DataTable which is the type of objects returned by the `mcols` accessor.
- Annotated 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
## ------------------------------------------------------------

compare(x, y)

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

## S4 method for signature Vector,Vector
```

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

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

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

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

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

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

selfmatch(x, ...)

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

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

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

```
## %in%
## ----

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

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

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

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

## S4 method for signature Vector
sort(x, decreasing=FALSE, ...)

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

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

Arguments

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

`incomparables`
: The `duplicated` method for Vector objects does NOT support this argument.

  The `unique` method for Vector objects, which is implemented on top of `duplicated`, propagates this argument to its call to `duplicated`.

  See `?base::duplicated` and `?base::unique` for more information about this argument.

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

`decreasing`
: See `?base::sort`.

`...`
: 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 compare(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 compare(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 compare(x, y) returns a zero-length integer vector.

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

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

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

Value

For compare: see Details section above.

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

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

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

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

For sort: see ?BiocGenerics::sort.

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

Note

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

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

```
setMethod("==", c("Vector", "Vector"),
    function(e1, e2) { compare(e1, e2) == 0L }
)
setMethod("<=", c("Vector", "Vector"),
    function(e1, e2) { compare(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 compare works the expected way. If compare is not implemented, then it's enough to implement == and <= methods to have the 4 remaining operators (!=, >=, <, and >) work out-of-the-box.
2. The **IRanges** package provides no compare method for Vector objects. Specific compare methods need to be implemented for specific Vector subclasses (e.g. for Ranges objects). These specific methods must obey the rules described in the Details section above.
3. The duplicated, unique, and %in% methods for Vector objects are implemented on top of selfmatch, duplicated, and match, respectively, so they work out-of-the-box on Vector objects for which selfmatch, duplicated, and match work the expected way.
4. Also the default findMatches and countMatches methods are implemented on top of match and selfmatch so they work out-of-the-box on Vector objects for which those things work the expected way.
5. However, since selfmatch itself is also implemented on top of match, then having match work the expected way is actually enough to get selfmatch, duplicated, unique, %in%, findMatches, and countMatches work out-of-the-box on Vector objects.
6. The sort method for Vector objects is implemented on top of order, so it works out-of-the-box on Vector objects for which order works the expected way.
7. The table method for Vector objects is implemented on top of selfmatch, order, and as.character, so it works out-of-the-box on a Vector object for which those things work the expected way.

8. The **IRanges** package provides no `match` or `order` methods for Vector objects. Specific methods need to be implemented for specific Vector subclasses (e.g. for Ranges objects).

Author(s)

H. Pages

See Also

- The Vector class.
- Ranges-comparison for comparing and ordering ranges.
- `==` and `%in%` in the **base** package, and `BiocGenerics::match`, `BiocGenerics::duplicated`, `BiocGenerics::unique`, `BiocGenerics::order`, `BiocGenerics::sort`, `BiocGenerics::rank` in the **BiocGenerics** package for general information about the comparison/ordering operators and functions.
- The Hits class.
- `BiocGeneric::table` in the **BiocGenerics** package.

Examples

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

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

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

## See ?Ranges-comparison for more examples (using Ranges objects).

## ---------------------------------------------------------------------
## B. FOR DEVELOPPERS: HOW TO IMPLEMENT THE BINARY COMPARISON OPERATORS
##    FOR YOUR Vector SUBCLASS
## ---------------------------------------------------------------------

## The answer is: dont implement them. Just implement compare() 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 "compare" method for Raw objects.

setMethod("compare", 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(".-+"))))
```

`view-summarization-methods` *Summarize views on a vector-like object with numeric values*

Description

`viewApply` applies a function on each view of a Views or ViewsList object.

`viewMins`, `viewMaxs`, `viewSums`, `viewMeans` calculate respectively the minima, maxima, sums, and means of the views in a Views or ViewsList object.

Usage

```
viewApply(X, FUN, ..., simplify = TRUE)

viewMins(x, na.rm=FALSE)
## S4 method for signature Views
min(x, ..., na.rm = FALSE)

viewMaxs(x, na.rm=FALSE)
## S4 method for signature Views
max(x, ..., na.rm = FALSE)

viewSums(x, na.rm=FALSE)
## S4 method for signature Views
sum(x, ..., na.rm = FALSE)

viewMeans(x, na.rm=FALSE)
## S4 method for signature Views
mean(x, ...)

viewWhichMins(x, na.rm=FALSE)
## S4 method for signature Views
which.min(x)
```

```
viewWhichMaxs(x, na.rm=FALSE)
## S4 method for signature Views
which.max(x)

viewRangeMins(x, na.rm=FALSE)

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

Arguments

X	A Views object.
FUN	The function to be applied to each view in X.
...	Additional arguments to be passed on.
simplify	A logical value specifying whether or not the result should be simplified to a vector or matrix if possible.
x	An RleViews or RleViewsList object.
na.rm	Logical indicating whether or not to include missing values in the results.

Details

The viewMins, viewMaxs, viewSums, and viewMeans functions provide efficient methods for calculating the specified numeric summary by performing the looping in compiled code.

The viewWhichMins, viewWhichMaxs, viewRangeMins, and viewRangeMaxs functions provide efficient methods for finding the locations of the minima and maxima.

Value

For all the functions in this man page (except viewRangeMins and viewRangeMaxs): A numeric vector of the length of x if x is an RleViews object, or a List object of the length of x if it's an RleViewsList object.

For viewRangeMins and viewRangeMaxs: An IRanges object if x is an RleViews object, or an IRangesList object if it's an RleViewsList object.

Note

For convenience, methods for min, max, sum, mean, which.min and which.max are provided as wrappers around the corresponding view* functions (which might be deprecated at some point).

Author(s)

P. Aboyoun

See Also

- The slice function for slicing an Rle or RleList object.
- view-summarization-methods in the **XVector** package for more view summarization methods.
- The RleViews and RleViewsList classes.
- The which.min and colSums functions.

Examples

```
## Views derived from coverage
x <- IRanges(start=c(1L, 9L, 4L, 1L, 5L, 10L),
             width=c(5L, 6L, 3L, 4L, 3L,  3L))
cvg <- coverage(x)
cvg_views <- slice(cvg, lower=2)

viewApply(cvg_views, diff)

viewMins(cvg_views)
viewMaxs(cvg_views)

viewSums(cvg_views)
viewMeans(cvg_views)

viewWhichMins(cvg_views)
viewWhichMaxs(cvg_views)

viewRangeMins(cvg_views)
viewRangeMaxs(cvg_views)
```

`Views-class` *Views objects*

Description

The Views virtual class is a general container for storing a set of views on an arbitrary Vector object, called the "subject".

Its primary purpose is to introduce concepts and provide some facilities that can be shared by the concrete classes that derive from it.

Some direct subclasses of the Views class are: RleViews, XIntegerViews (defined in the XVector package), XStringViews (defined in the Biostrings package), etc...

Constructor

`Views(subject, start=NULL, end=NULL, width=NULL, names=NULL)`: This constructor is a generic function with dispatch on argument `subject`. Specific methods must be defined for the subclasses of the Views class. For example a method for XString subjects is defined in the Biostrings package that returns an XStringViews object. There is no default method.

The treatment of the `start`, `end` and `width` arguments is the same as with the `IRanges` constructor, except that, in addition, `Views` allows `start` to be a Ranges object. With this feature, `Views(subject, IRanges(my_starts, my_ends, my_widths, my_names))` and `Views(subject, my_starts, my_ends, my_widths, my_names)` are equivalent (except when `my_starts` is itself a Ranges object).

Coercion

In the code snippets below, `from` is a Views object:

`as(from, "IRanges")`: Creates an `IRanges` object containing the view locations in `from`.

Accessor-like methods

All the accessor-like methods defined for `IRanges` objects work on Views objects. In addition, the following accessors are defined for Views objects:

`subject(x)`: Return the subject of the views.

Subsetting

`x[i]`: Select the views specified by `i`.

`x[[i]]`: Extracts the view selected by `i` as an object of the same class as `subject(x)`. Subscript `i` can be a single integer or a character string. The result is the subsequence of `subject(x)` defined by `window(subject(x), start=start(x)[i], end=end(x)[i])` or an error if the view is "out of limits" (i.e. `start(x)[i] < 1` or `end(x)[i] > length(subject(x))`).

Combining

`c(x, ..., ignore.mcols=FALSE)`: Combine `Views` objects. They must have the same subject.

Other methods

`trim(x, use.names=TRUE)`: Equivalent to `restrict(x, start=1L, end=length(subject(x)), keep.all.ranges=TRU`

`subviews(x, start=NA, end=NA, width=NA, use.names=TRUE)`: `start`, `end`, and `width` arguments must be vectors of integers, eventually with NAs, that contain coordinates relative to the current ranges. Equivalent to `trim(narrow(x, start=start, end=end, width=width, use.names=use.names))`.

`successiveViews(subject, width, gapwidth=0, from=1)`: Equivalent to `Views(subject, successiveIRanges(widt`
See `?successiveIRanges` for a description of the `width`, `gapwidth` and `from` arguments.

Author(s)

H. Pages

See Also

IRanges-class, Vector-class, IRanges-utils, XVector.

Some direct subclasses of the Views class: RleViews-class, XIntegerViews-class, XDoubleViews-class, XStringViews-class.

`findOverlaps`.

Examples

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

## Create a set of 4 views on an XInteger subject of length 10:
subject <- Rle(3:-6)
v1 <- Views(subject, start=4:1, end=4:7)

## Extract the 2nd view:
v1[[2]]

## Some views can be "out of limits"
v2 <- Views(subject, start=4:-1, end=6)
trim(v2)
subviews(v2, end=-2)

## See ?XIntegerViews-class in the XVector package for more examples.
```

ViewsList-class *List of Views*

Description

An extension of List that holds only Views objects.

Details

ViewsList is a virtual class. Specialized subclasses like e.g. RleViewsList are useful for storing coverage vectors over a set of spaces (e.g. chromosomes), each of which requires a separate RleViews object.

As a Vector subclass, ViewsList may be annotated with its universe identifier (e.g. a genome) in which all of its spaces exist.

As a List subclass, ViewsList inherits all the methods available for List objects. It also presents an API that is very similar to that of Views, where operations are vectorized over the elements and generally return lists.

Author(s)

P. Aboyoun and H. Pages

See Also

List-class, RleViewsList-class.

findOverlaps.

Examples

```
showClass("ViewsList")
```

Index