Package ‘IRanges’

January 21, 2017

Title
Infrastructure for manipulating intervals on sequences

Description
Provides efficient low-level and highly reusable S4 classes for storing, manipulating and aggregating over annotated ranges of integers. Implements an algebra of range operations, including efficient algorithms for finding overlaps and nearest neighbors. Defines efficient list-like classes for storing, transforming and aggregating large grouped data, i.e., collections of atomic vectors and DataFrames.

Version
2.8.1

Encoding
UTF-8

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

Maintainer
Bioconductor Package Maintainer <maintainer@bioconductor.org>

biocViews
Infrastructure, DataRepresentation

Depends
R (>= 3.1.0), methods, utils, stats, BiocGenerics (>= 0.19.1), S4Vectors (>= 0.11.19)

Imports
stats4

LinkingTo
S4Vectors

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

License
Artistic-2.0

Collate
coverage-methods.R
slice-methods.R
setops-methods.R
nearest-methods.R
cbind-Rle-methods.R
tile-methods.R
bind-arrays.R
zzz.R

**NeedsCompilation** yes

### R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicList</td>
<td>3</td>
</tr>
<tr>
<td>AtomicList-utils</td>
<td>5</td>
</tr>
<tr>
<td>bind-arrays</td>
<td>7</td>
</tr>
<tr>
<td>CompressedHitsList-class</td>
<td>7</td>
</tr>
<tr>
<td>CompressedList-class</td>
<td>8</td>
</tr>
<tr>
<td>coverage-methods</td>
<td>9</td>
</tr>
<tr>
<td>DataFrame-utils</td>
<td>14</td>
</tr>
<tr>
<td>DataFrameList-class</td>
<td>15</td>
</tr>
<tr>
<td>extractList</td>
<td>17</td>
</tr>
<tr>
<td>findOverlaps-methods</td>
<td>19</td>
</tr>
<tr>
<td>GappedRanges-class</td>
<td>24</td>
</tr>
<tr>
<td>Grouping-class</td>
<td>24</td>
</tr>
<tr>
<td>Hits-class-leftovers</td>
<td>30</td>
</tr>
<tr>
<td>inter-range-methods</td>
<td>31</td>
</tr>
<tr>
<td>intra-range-methods</td>
<td>37</td>
</tr>
<tr>
<td>IRanges-class</td>
<td>43</td>
</tr>
<tr>
<td>IRanges-constructor</td>
<td>45</td>
</tr>
<tr>
<td>IRanges-utils</td>
<td>47</td>
</tr>
<tr>
<td>IRangesList-class</td>
<td>49</td>
</tr>
<tr>
<td>List-class-leftovers</td>
<td>50</td>
</tr>
<tr>
<td>MaskCollection-class</td>
<td>50</td>
</tr>
<tr>
<td>multisplit</td>
<td>53</td>
</tr>
<tr>
<td>NCList-class</td>
<td>53</td>
</tr>
<tr>
<td>nearest-methods</td>
<td>56</td>
</tr>
<tr>
<td>RangedData-class</td>
<td>59</td>
</tr>
<tr>
<td>RangedDataList-class</td>
<td>64</td>
</tr>
<tr>
<td>RangedSelection-class</td>
<td>64</td>
</tr>
<tr>
<td>Ranges-class</td>
<td>65</td>
</tr>
<tr>
<td>Ranges-comparison</td>
<td>69</td>
</tr>
<tr>
<td>RangesList-class</td>
<td>73</td>
</tr>
<tr>
<td>rdapply</td>
<td>75</td>
</tr>
<tr>
<td>read.Mask</td>
<td>75</td>
</tr>
<tr>
<td>reverse</td>
<td>78</td>
</tr>
<tr>
<td>Rle-class-leftovers</td>
<td>79</td>
</tr>
<tr>
<td>RleViews-class</td>
<td>80</td>
</tr>
<tr>
<td>RleViewsList-class</td>
<td>80</td>
</tr>
<tr>
<td>seqapply</td>
<td>82</td>
</tr>
<tr>
<td>setops-methods</td>
<td>82</td>
</tr>
<tr>
<td>slice-methods</td>
<td>85</td>
</tr>
<tr>
<td>Vector-class-leftovers</td>
<td>86</td>
</tr>
<tr>
<td>view-summarization-methods</td>
<td>88</td>
</tr>
<tr>
<td>Views-class</td>
<td>90</td>
</tr>
<tr>
<td>ViewsList-class</td>
<td>91</td>
</tr>
</tbody>
</table>

**Index** 93
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.

FactorList(..., compress = TRUE): Concatenates the factor objects in ... into a new FactorList. 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, "SimpleNormalIRangesList"): 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.

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

as(from, “RleViews”): Creates an RleViews where each view corresponds to an element of from. The subject is unlist(from).

as.matrix(x, col.names=NULL): Maps the elements of the list to rows of a matrix. The column mapping depends on whether there are inner names (either on the object or provided via col.names as a List object). If there are no inner names, each row is padded with NAs to reach the length of the longest element. If there are inner names, there is a column for each unique name and the mapping is by name. To provide inner names, the col.names argument should be a List, usually a CharacterList or FactorList (which is particularly efficient). If col.names is a character vector, it names the columns of the result, but does not imply inner names.

**Compare, Order, Tabulate**

The following methods are provided for element-wise comparison of 2 AtomicList objects, and ordering or tabulating of each list element of an AtomicList object: is.na, duplicated, unique, match, %in%, table, order, sort.

**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.

**Author(s)**

P. Aboyoun

**See Also**

• AtomicList-utils for common operations on AtomicList objects.

• List objects in the S4Vectors package for the parent class.

**Examples**

```r
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)

---

AtomicList-utils

Common operations on AtomicList objects

Description

Common operations on AtomicList objects.

Group Generics

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

- **Arith**: "+", "-", "+", "/", "/", "/%
- **Compare**: ">", ">", ">", ">
- **Logic**: "&", "}
- **Ops**: "Arith", "Compare", "Logic"
Math2 "round", "signif"
Summary "max", "min", "range", "prod", "sum", "any", "all"
Complex "Arg", "Conj", "Im", "Mod", "Re"

See S4groupGeneric for more details.

Other 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:

**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

The which.min and which.max functions have an extra argument, global=FALSE, which controls whether the returned subscripts are global (compatible with the unlisted form of the input) or local (compatible with the corresponding list element).

The rank method only supports tie methods "average", "first", "min" and "max".

Specialized Methods

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

Author(s)

P. Aboyoun

See Also

- AtomicList objects.

Examples

```r
## group generics
int1 <- c(1L, 2L, 3L, 5L, 2L, 8L)
int2 <- c(15L, 45L, 20L, 1L, 15L, 100L, 80L, 5L)
col1 <- IntegerList(one = int1, int2)
2 * col1
col1 + col1
col1 > 2
sum(col1) # equivalent to (but faster than) 'sapply(col1, sum)'
mean(col1) # equivalent to 'sapply(col1, mean)'
```
Bind arrays along their rows or columns

Description
Bind array-like objects with an arbitrary number of dimensions along their rows (arbind) or columns (acbind).

Usage
arbind(...)
acbind(...)

Arguments
... The array-like objects to bind.

Value
An array-like object, typically of the same class as the input objects if they all have the same class.

See Also
- rbind and cbind in the base package for the corresponding operations on matrix-like objects.
- The abind package on CRAN.

Examples
a1 <- array(1:60, c(3, 5, 4),
dimnames=list(NULL, paste0("M1y", 1:5), NULL))
a2 <- array(101:240, c(7, 5, 4),
dimnames=list(paste0("M2x", 1:7), paste0("M2y", 1:5), NULL))
a3 <- array(10001:10100, c(5, 5, 4),
dimnames=list(paste0("M3x", 1:5), NULL, paste0("M3z", 1:4)))
arbind(a1, a2, a3)

CompressedHitsList-class
CompressedHitsList objects

Description
An efficient representation of HitsList objects. See ?HitsList for more information about HitsList objects.

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

Author(s)

Michael Lawrence

See Also

HitsList objects.

Description

Like the SimpleList class defined in the S4Vectors package, the CompressedList class extends the List virtual class.

Details

Unlike the SimpleList class, CompressedList is virtual, that is, it cannot be instantiated. Many concrete (i.e. non-virtual) CompressedList subclasses are defined and documented in this package (e.g. CompressedIntegerList, CompressedCharacterList, CompressedRleList, etc...), as well as in other packages (e.g. GRangesList in the GenomicRanges package, GAlignmentsList in the GenomicAlignments package, etc...). It’s easy for developers to extend CompressedList to create a new CompressedList subclass and there is generally very little work involved to make this new subclass fully operational.

In a CompressedList object the list elements are concatenated together in a single vector-like object. The partitioning of this single vector-like object (i.e. the information about where each original list element starts and ends) is also kept in the CompressedList object. This internal representation is generally more memory efficient than SimpleList, especially if the object has many list elements (e.g. thousands or millions). Also it makes it possible to implement many basic list operations very efficiently.

Many objects like LogicalList, IntegerList, CharacterList, RleList, etc... exist in 2 flavors: CompressedList and SimpleList. Each flavor is incarnated by a concrete subclass: CompressedLogicalList and SimpleLogicalList for virtual class LogicalList, CompressedIntegerList and SimpleIntegerList for virtual class IntegerList, etc... It’s easy to switch from one representation to the other with as(x, "CompressedList") and as(x, "SimpleList"). Also the constructor function for those virtual classes have a switch that lets the user choose the representation at construction time e.g. CharacterList(..., compress=TRUE) or CharacterList(..., compress=FALSE). See below for more information.

Constructor

See the List man page in the S4Vectors package for a quick overview of how to construct List objects in general.

Unlike for SimpleList objects, there is no CompressedList constructor function.

However, many constructor functions for List objects have a switch that lets the user choose between the CompressedList and SimpleList representation at construction time. For example, a CompressedCharacterList object can be constructed with CharacterList(..., compress=TRUE).
Accessors

Same as for List objects. See the List man page in the S4Vectors package for more information.

Coercion

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

Subsetting

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

Looping and functional programming

Same as for List objects. See ?`List-utils` in the S4Vectors package for more information.

Displaying

When a CompressedList object is displayed, the "Compressed" prefix is removed from the real class name of the object. See classNameForDisplay in the S4Vectors package for more information about this.

See Also

• The List class defined and documented in the S4Vectors package for the parent class.
• The SimpleList class defined and documented in the S4Vectors package for an alternative to CompressedList.
• The CompressedIntegerList class for a CompressedList subclass example.

Examples

```r
## Displaying a CompressedList object:
x <- IntegerList(11:12, integer(0), 3:-2, compress=TRUE)
class(x)

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

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

---

Coverage of a set of ranges

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 Ge-nomicRanges 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.

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. Pagès 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

```r
## 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))
cov0 <- coverage(x)  # default
coverage(x, shift=7)  # shift
coverage(x, shift=7, width=27)  # width
coverage(x, shift=c(-4, 2))  # shift gets recycled
coverage(x, shift=c(-4, 2), width=12)
cov2 <- coverage(x, shift=-max(end(x)))
cov0[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 doesn't 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)
## D. COVERAGE OF AN IRangesList OBJECT

```r
x <- IRangesList(A=IRanges(3*(4:-1), width=1:3), B=IRanges(2:10, width=5))
cvg <- coverage(x)
cvg
```

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

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

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

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

---

### Description

Common operations on DataFrame objects.

### Splitting

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

### Author(s)

Michael Lawrence
DataFrameList-class

See Also

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

Examples

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

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

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

### 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.
DataFrameList-class

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 ....

Transformation

transform('._data', ...): Transforms a SplitDataFrame in a manner analogous to the base transform, where the columns are List objects adhering to the structure of _data.

Coercion

In the following code snippets, x is a DataFrameList.

as(from, "DataFrame"): Coerces a SplitDataFrameList to a DataFrame, which has a column for every column in from, except each column is a list with the same structure as from.

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. If from is a DataFrame, each row becomes an element in the list.

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.

as.data.frame(x, row.names = NULL, optional = FALSE, ..., value.name = "value", use.outer.mcols = FALSE): Coerces x to a data.frame. See as.data.frame on the List man page for details (?List).

Author(s)

Michael Lawrence

See Also

DataFrame, RangedData, which uses a DataFrameList to split the data by the spaces.
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** and **S4Vectors** packages define relist and split methods that operate on a Vector object and return a List object. Note that the split methods defined in **S4Vectors** delegate to the splitAsList function defined in **IRanges** and documented below.

Because relist and split both impose 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.

**Usage**

```r
## relist()
## --------
## S4 method for signature 'ANY,List'
relist(flesh, skeleton)

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

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

## extractList()
## -------------
extractList(x, i)

## regroup()
## ---------
regroup(x, g)
```

**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).
**g**

A Grouping or an object coercible to one. For regroup, g groups the elements of x.

... Arguments to pass to methods.

### Details

relist, split, and extractList have in common that they return a list-like object 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.

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 (or `splitAsList`) can be seen as special cases 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 practice 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 not the transformation performed by `extractList` (in general).

The `regroup` function splits the elements of `unlist(x)` into a list according to the grouping g. Each element of `unlist(x)` inherits its group from its parent element of x. `regroup` is different from `relist` and `split`, because x is already grouped, and the goal is to combine groups.

### Value

The `relist` methods behave like `utils::relist` except that they return a List object. If skeleton has names, then they are propagated to the returned value.

`splitAsList` behaves like `base::split` except that the former returns a List object instead of an ordinary list.

`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 return a list-like object where the list elements have the same class as x. `relistToClass` gives the exact class of the returned object.
findOverlaps-methods

Author(s)
Hervé Pagès

See Also
• The unlist and relist functions in the base and utils packages, respectively.
• The split and unsplit functions in the base package.
• The split methods defined in the S4Vectors package.
• Vector, List, Rle, and DataFrame objects in the S4Vectors package. relistToClass is documented in the man page for List objects.
• Ranges objects.

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

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

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 Ranges, Views, RangesList, or ViewsList objects. See ?findOverlaps,GenomicRanges,GenomicRanges-method in the GenomicRanges package for methods that operate on GenomicRanges or GRangesList 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"),
    invert=FALSE,
    ...)  
mergeByOverlaps(query, subject, ...)
findOverlapPairs(query, subject, ...)

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

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

Arguments

query, subject  Each of them can be a Ranges, Views, RangesList, ViewsList, or RangedData object. In addition, 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 subject is omitted, query is queried against itself. In this case, and only this case, the drop.self and drop.redundant 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 drop.self is TRUE, all self matches are dropped. If drop.redundant 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 subject may be wider than the query.
findOverlaps-methods

select

If query is a Ranges or Views object: When select is "all" (the default), the results are returned as a Hits object. Otherwise the returned value is an integer vector parallel to query (i.e. same length) containing the first, last, or arbitrary overlapping interval in subject, with NA indicating intervals that did not overlap any intervals in subject.

If query is a RangesList, ViewsList, or RangedData object: When select is "all" (the default), the results are returned as a HitsList object. Otherwise the returned 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. When select != "all" && drop, an integer vector is returned containing indices that are offset to align with the unlisted query.

invert

If TRUE, keep only the query ranges that do not overlap the subject.

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

- drop: Supported only when query is a RangesList, ViewsList, or RangedData object. FALSE by default. See select argument above for the details.
- drop.sel, drop.redundant: When subject is omitted, the drop.sel and drop.redundant arguments (both FALSE by default) are allowed. See query and subject arguments above for the details.

x

Hits or HitsList 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").

Value

For findOverlaps: see select argument above.

For countOverlaps: the overlap hit count for each range in query using the specified findOverlaps parameters. For RangesList objects, it returns an IntegerList object.

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.

%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, subject) 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.

mergeByOverlaps computes the overlap between query and subject according to the arguments in ... . It then extracts the corresponding hits from each object and returns a DataFrame containing one column for the query and one for the subject, as well as any mcols that were present on either object. The query and subject columns are named by quoting and deparsing the corresponding argument.

findOverlapPairs is like mergeByOverlaps, except it returns a formal Pairs object that provides useful downstream conveniences, such as finding the intersection of the overlapping ranges with pintersect.
poverlaps compares query and subject in parallel (like e.g., pmin) and returns a logical vector indicating whether each pair of ranges overlaps. Integer vectors are treated as width-one ranges. When x is a Hits or HitsList object, ranges(x, query, subject) returns a Ranges (or RangesList) object of the same shape as 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. Same shape means same length when x is a Hits object, and same length and same elementNROWS when x is a HitsList object.

**Author(s)**

Michael Lawrence and Hervé Pagès

**References**


**See Also**

- Hits and HitsList objects in the S4Vectors package for representing a set of hits between 2 vector-like or list-like objects.
- findOverlaps, GenomicRanges, GenomicRanges-method in the GenomicRanges package for methods that operate on GRanges or GRangesList objects.
- The NCLList class and constructor.
- The Ranges, Views, RangesList, and ViewsList classes.
- The IntegerList and LogicalList classes.

**Examples**

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

## 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, subject, maxgap=1L)
```

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

## shortcut
findOverlaps(query, subject)
```
### one Ranges with itself

```r
findOverlaps(query)
```

### single points as query

```r
subject <- IRanges(c(1, 6, 13), c(4, 9, 14))
findOverlaps(c(3L, 7L, 10L), subject, select="first")
```

### alternative overlap types

```r
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")
```

```r
ov <- findOverlaps(query, subject, type="within", maxgap=1L)
```

### Using pairs to find intersection of overlapping ranges

```r
hits <- findOverlaps(query, subject)
p <- Pairs(query, subject, hits=hits)
pintersect(p)
```

### Shortcut

```r
p <- findOverlapPairs(query, subject)
pintersect(p)
```

### overlapsAny()

```r
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")
```
```r
grouping <- c("a","a","b","b")
qpartition <- factor(grouping)
qlist <- split(query, qpartition)
subject <- IRanges(c(1, 3, 5, 6), width=c(4, 4, 5, 4))
spartition <- factor(grouping)
slist <- split(subject, spartition)
```

```r
findOverlaps(qlist, slist, select="arbitrary")
query <- IRanges(c(1, 5, 3, 4), width=c(2, 2, 4, 6))
qupartition <- factor(c("a","a","b","b"))
qlist <- split(query, qpartition)
```
The Grouping class is a virtual class that formalizes the most general kind of grouping. More specific groupings (e.g. many-to-one groupings or block-groupings) are formalized via specific Grouping subclasses.

This man page documents the core Grouping API, and 3 important Grouping subclasses: ManyToOneGrouping, GroupingRanges, and Partitioning (the last one deriving from the 2 first).

**The core Grouping API**

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

Groups $G_i$ are indexed from 1 to $NG$ ($1 \leq i \leq NG$).

Objects $O_j$ are indexed from 1 to $NO$ ($1 \leq j \leq NO$).

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

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 ($NO$).

Going from groups to objects:

- $x[[i]]$: Returns the indices of the objects (the $j$'s) that belong to $G_i$. This provides the mapping from groups to objects.
- `grouplengths(x, i=NULL)`: Returns the number of objects in $G_i$. Works in a vectorized fashion (unlike $x[[i]]$). `grouplengths(x)` is equivalent to `grouplengths(x, seq_len(length(x)))`. If $i$ is not NULL, `grouplengths(x, i)` is equivalent to `sapply(i, function(ii) length(x[[ii]])).

Note to developers: Given that `length`, `names` and `[[` are expected to work on any Grouping object, those objects can be seen as `List` objects. More precisely, the Grouping class actually extends the `IntegerList` class. In particular, many other "list" operations like `as.list`, `elementNROWS`, and `unlist`, etc... should work out-of-the-box on any Grouping object.

**ManyToOneGrouping objects**

The ManyToOneGrouping class is a virtual subclass of Grouping for representing many-to-one groupings, that is, groupings where each object in the original collection of objects belongs to exactly one group.

The grouping of an empty collection of objects in an arbitrary number of (necessarily empty) groups is a valid ManyToOneGrouping object.

Note that, for a ManyToOneGrouping object, if $NG$ is 0 then $NO$ must also be 0.

The ManyToOneGrouping API extends the core Grouping API by adding a couple more operations for going from groups to objects:

- `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))`.

And also by adding operations for going from objects to groups:
tgroup(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.
tgroup(x) is equivalent to tgroup(x, seq_len(nobj(x))): both return the entire mapping
in an integer vector of length NO. If j is not NULL, tgroup(x, j) is equivalent to
y <- tgroup(x); y[j].

tgrouplength(x, j=NULL): Returns the number of objects that belong to the same group as O_j
(including O_j itself). Equivalent to grouplengths(x, tgroup(x, j)).

One important property of any ManyToOneGrouping object x is that unlist(as.list(x)) is al-
ways 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.

2 ManyToOneGrouping concrete subclasses: H2LGrouping, Dups and SimpleManyToOneGroup-
ing

DOCUMENT ME  Constructors:

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

ManyToOneGrouping(..., compress=TRUE): Collect ... into a ManyToOneGrouping. The ar-

guments will be coerced to integer vectors and combined into a list, unless there is a single
list argument, which is taken to be an integer list. The resulting integer list should have a
structure analogous to that of Grouping itself: each element represents a group in terms of the
subscripts of the members. If compress is TRUE, the representation uses a CompressedList,
otherwise a SimpleList.

ManyToManyGrouping objects

The ManyToManyGrouping class is a virtual subclass of Grouping for representing many-to-many
groupings, that is, groupings where each object in the original collection of objects belongs to any
number of groups.

Constructors:

ManyToManyGrouping(x, compress=TRUE): Collect ... into a ManyToManyGrouping. The ar-

guments will be coerced to integer vectors and combined into a list, unless there is a single
list argument, which is taken to be an integer list. The resulting integer list should have a
structure analogous to that of Grouping itself: each element represents a group in terms of the
subscripts of the members. If compress is TRUE, the representation uses a CompressedList,
otherwise a SimpleList.

GroupingRanges objects

The GroupingRanges class is a virtual subclass of Grouping for representing block-groupings, that is,
groupings where each group is a block of adjacent elements in the original collection of objects.
GroupingRanges objects support the Ranges API (e.g. start, end, width, etc...) in addition to the
Grouping API. See ?Ranges for a description of the Ranges API.

Partitioning objects

The Partitioning class is a virtual subclass of GroupingRanges for representing block-groupings
where the blocks fully cover the original collection of objects and don’t overlap. Since this makes
them many-to-one groupings, the Partitioning class is also a subclass of ManyToOneGrouping. An
additional constraint of Partitioning objects is that the blocks must be ordered by ascending position with respect to the original collection of objects.

The Partitioning virtual class itself has 3 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), and PartitioningMap which contains PartitioningByEnd and two additional slots to re-order and re-list the object to a related mapping.

Constructors:

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 elementNROWS(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 elementNROWS(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.

PartitioningMap(x=integer(), mapOrder=integer()): x is a list-like object or a sorted integer vector used to construct a PartitioningByEnd object. mapOrder numeric vector of the mapped order.

Returns a PartitioningMap object.

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

Coercions to Grouping objects

These types can be coerced to different derivatives of Grouping objects:

factor Analogous to calling split with the factor. Returns a ManyToOneGrouping if there are no NAs, otherwise a ManyToManyGrouping. If a factor is explicitly converted to a ManytoOneGrouping, then any NAs are placed in the last group.
A vector is effectively treated as a factor, but more efficiently. The order of the groups is not defined.

**FactorList** Same as the factor coercion, except using the interaction of every factor in the list. The interaction has an NA wherever any of the elements has one. Every element must have the same length.

**DataFrame** Effectively converted via a FactorList by coercing each column to a factor.

**grouping** Equivalent Grouping representation of the base R `grouping` object.

**Hits** Returns roughly the same object as `as(x, "List")`, except it is a ManyToManyGrouping, i.e., it knows the number of right nodes.

Author(s)

Hervé Pagès, Michael Lawrence

See Also

`IntegerList-class`, `Ranges-class`, `IRanges-class`, `successiveIRanges`, `cumsum`, `diff`

Examples

```r
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)
high2low <- c(NA, NA, 2, 2, NA, NA, NA, 6, NA, 1, 2, NA, 6, NA, NA, 2)
h2l <- H2LGrouping(high2low)
h2l

# The core Grouping API:
length(h2l) # same as 'length(h2l)' for H2LGrouping objects
h2l[[1]]
h2l[[2]]
h2l[[3]]
h2l[[4]]
h2l[[5]]
grouplengths(h2l) # same as 'unname(sapply(h2l, length))'
grouplengths(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 'grouplengths(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:

```r
x <- c(2, 7, 4, 7, 2, 8, 8, 4, 99)
x_high2low <- high2low(x)
x_high2low # same length as 'x'
dups2 <- Dups(x_high2low)
togroup(dups2)
duplicated(dups2)
togrouplength(dups2) # frequency for each element
table(x)
```

### C. Partitioning OBJECTS

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

```r
length(pbe1)
nobj(pbe1)
pbe1[[1]]
pbe1[[2]]
pbe1[[3]]
grouplengths(pbe1) # same as 'unname(sapply(pbe1, length))'
 # and 'width(pbe1)'
togroup(pbe1)
togrouplength(pbe1) # same as 'grouplengths(pbe1, togroup(pbe1))'
names(pbe1)
```

```r
start(pbe1)
end(pbe1)
width(pbe1)
```

```r
as.list(pbe1)
sapply(pbe1, length)
```

```r
names(pbe1)[3] <- "empty partition"
pbe1
```

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

```r
# 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))
```
mywidths <- c(4, 3, 0, 1, 7)

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))

Description

IMPORTANT NOTE - 4/29/2014: This man page is being refactored. Most of the things that used to be documented here have been moved to the man page for Hits objects located in the S4Vectors package.

Details

The as.data.frame method coerces a Hits object to a two column 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.

Coercion

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

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

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

as.list(x) : Coerces x to a list of integers, grouping the the right node hits for each left node.

as(x, "List") : Analogous to as.list(x).

as(x, "Grouping") : Returns roughly the same object as as(x, "List"), except it is a Many-ToManyGrouping, i.e., it knows the number of right nodes.

See Also

The Hits class defined and documented in the S4Vectors package.

Examples

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

hits <- findOverlaps(query, subject)
as.matrix(hits)
as.data.frame(hits)
as.table(hits) # hits per query
as.table(t(hits)) # hits per subject

## Turn a Hits object into an IntegerList object with one list element
## per element in the original query.
as(hits, "IntegerList")
as(hits, "List")  # same as as(hits, "IntegerList")

## Turn a Hits object into a PartitioningByEnd object that describes
## the grouping of hits by query.
as(hits, "PartitioningByEnd")
as(hits, "Partitioning")  # same as as(hits, "PartitioningByEnd")

## remapHits()
## -------------------------------
hits2 <- remapHits(hits,
    Rnodes.remapping=factor(c("e", "e", "d"), letters[1:5]))
hits2

hits3 <- remapHits(hits,
    Rnodes.remapping=c(5, 5, 4), new.nRnode=5)
hits3
stopifnot(identical(hits2, hits3))

---

### Description

Range-based transformations are grouped in 2 categories:

1. **Intra range transformations** (e.g. `shift()`) transform each range individually (and independently of the other ranges). They return an object parallel to the input object, that is, where the i-th range corresponds to the i-th range in the input. Those transformations are described in the `intra-range-methods` man page (see `?intra-range-methods`).

2. **Inter range transformations** (e.g. `reduce()`) transform all the ranges together as a set to produce a new set of ranges. They return an object that is generally NOT parallel to the input object. Those transformations are described below.

### Usage

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

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

## reduce()
```
## inter-range-methods

```r
# --------
reduce(x, drop.empty.ranges=FALSE, ...)

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

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

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

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

# disjoin(), isDisjoint(), and disjointBins()
# -------------------------------------------
disjoin(x, ...)

# S4 method for signature 'Ranges'
disjoin(x, with.revmap=FALSE)

# S4 method for signature 'RangesList'
disjoin(x, with.revmap=FALSE)

isDisjoint(x, ...)

disjointBins(x, ...)
```

### Arguments

- **x**  
  A `Ranges` or `RangesList` object for `range`, `disjoin`, `isDisjoint`, and `disjointBins`.  
  A `Ranges`, `Views`, or `RangesList` object for `reduce` and `gaps`.

- **...**  
  For `range`, additional `Ranges` or `RangesList` object 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 output to input ranges be stored in the returned object? If yes, then it is stored as metadata column `revmap` of type `IntegerList`.

- **with.inframe.attrib**  
  TRUE or FALSE. For internal use.

- **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 `Ranges` top-level element.
Details

Unless specified otherwise, when \( x \) is a \texttt{RangesList} object, any transformation described here is equivalent to applying the transformation to each \texttt{RangesList} top-level element separately.

\textbf{reduce:}
reduce first orders the ranges in \( x \) from left to right, then merges the overlapping or adjacent ones.

\textbf{range:}
range first combines \( x \) and the arguments in \ldots If the combined \texttt{IRanges} object contains at least 1 range, then range returns an \texttt{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), \texttt{IRanges()} is returned (i.e. an \texttt{IRanges} instance of length 0).

When passing more than 1 \texttt{RangesList} object to range(), they are first merged into a single \texttt{RangesList} object: by name if all objects have names, otherwise, if they are all of the same length, by position. Else, an exception is thrown.

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

\textbf{gaps:}
gaps returns the "normal" \texttt{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 \texttt{Views} object, then start=\texttt{NA} and end=\texttt{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.

\textbf{isDisjoint:}
A \texttt{Ranges} object \( x \) is considered to be "disjoint" if its ranges are non-overlapping. \texttt{isDisjoint} tests whether the object is "disjoint" or not.

Note that a "normal" \texttt{Ranges} object is always "disjoint" but the opposite is not true. See ?\texttt{isNormal} for more information about normal \texttt{Ranges} objects.

About empty ranges. \texttt{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 \( B \)). More precisely, single empty range \( A \) is considered to overlap with single range \( B \) iff

\[
\text{start}(B) < \text{start}(A) \quad \text{and} \quad \text{end}(A) < \text{end}(B)
\]

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

\[
\text{start}(B) < \text{start}(A) \leq \text{end}(B)
\]

and also equivalent to:

\[
\text{start}(B) \leq \text{end}(A) < \text{end}(B)
\]

Finally, it is also equivalent to:

\[
\text{pcompare}(A, B) == 2
\]

See '?\texttt{Ranges-comparison}' for the meaning of the codes returned by the \texttt{pcompare} function.

\textbf{disjoin:}
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.
**disjointBins:**
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.

**Value**
If x is a Ranges object:
- range, reduce, gaps, and disjoin return an IRanges instance.
- isDisjoint returns TRUE or FALSE.
- disjointBins returns an integer vector parallel to x, that is, where the i-th element corresponds to the i-th element in x.

If x is a Views object: reduce and gaps return a Views object on the same subject as x but with modified views.

If x is a RangesList object:
- range, reduce, gaps, and disjoin return a RangesList object parallel to x.
- isDisjoint returns a logical vector parallel to x.
- disjointBins returns an IntegerList object parallel to x.

**Author(s)**
H. Pagès, M. Lawrence, and P. Aboyoun

**See Also**
- intra-range-methods for intra range transformations.
- The Ranges, Views, RangesList, and MaskCollection classes.
- The inter-range-methods man page in the GenomicRanges package for inter range transformations of genomic ranges.
- setops-methods for set operations on IRanges objects.
- endoapply in the S4Vectors package.

**Examples**
```r
## 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)
```
```r
## On a RangesList object (XVector package required):
rang1 <- IRanges(start=c(1, 2, 3), end=c(5, 2, 8))
rang2 <- IRanges(start=c(15, 45, 20, 1), end=c(15, 100, 80, 5))
rang3 <- IRanges(start=c(-2, 6, 7), width=c(8, 0, 0))  # with empty ranges
collection <- IRangesList(one=rang1, rang2, rang3)
if (require(XVector)) {
  range(collection)
}
```r
ir11 <- IRangesList(a=IRanges(c(1, 2), c(4, 3)), b=IRanges(c(4, 6), c(10, 7)))
ir12 <- IRangesList(c=IRanges(c(0, 2), c(4, 5)), a=IRanges(c(4, 5), c(6, 7)))
range(ir11, ir12) # matched by names
names(ir12) <- NULL
range(ir11, ir12) # now by position

## ---------------------------------------------------------------------
## 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
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)

## ---------------------------------------------------------------------
## 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:

```r
res1 <- gaps(collection)
res2 <- IRangesList(one=gaps(range1), gaps(range2), gaps(range3))
res3 <- do.call(IRangesList, lapply(collection, gaps))
res4 <- endoapply(collection, gaps)
```

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

## On a MaskCollection object:

```r
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()

```r
## 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))
disjoin(ir, with.revmap=TRUE)
```

## On a RangesList object:

```r
disjoin(collection)
disjoin(collection, with.revmap=TRUE)
```

## isDisjoint()

```r
## 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:

```r
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:

```r
sapply(11:17, 
  function(i) pcompare(IRanges(i, width=0), IRanges(12, 15)))
```

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

## On a RangesList object:

```r
isDisjoint(collection)
```

```r
## disjointBins()
```
## Disjoint bins

```r
## 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)
```

### Description

Range-based transformations are grouped in 2 categories:

1. **Intra range transformations** (e.g. `shift()`) transform each range individually (and independently of the other ranges). They return an object parallel to the input object, that is, where the i-th range corresponds to the i-th range in the input. Those transformations are described below.

2. **Inter range transformations** (e.g. `reduce()`) transform all the ranges together as a set to produce a new set of ranges. They return an object that is generally **NOT** parallel to the input object. Those transformations are described in the `inter-range-methods` man page (see `?inter-range-methods`).

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.

### Usage

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

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

## resize()
resize(x, width, fix="start", 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)

## 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 resize and flank, either an integer vector or an IntegerList object containing the width of the resized or flanking 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).
- **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`.
- **...**: Additional arguments for methods.
- **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.
intra-range-methods

upstream, downstream

Single integer values \( \geq 0 \). 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.

bounds

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

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.

Details

Unless specified otherwise, when \( x \) is a RangesList object, any transformation described here is equivalent to applying the transformation to each RangesList top-level element separately.

shift:

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

narrow:

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, end=end, width=width) 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.

resize:

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

flank:

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:**

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 upstream 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 a GRanges object see `?promoters,GenomicRanges-method` in the GenomicRanges package.

**reflect:**

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:**

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

**threebands:**

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=TRUE)`.

**Author(s)**

H. Pagès, M. Lawrence, and 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 GenomicRanges package for intra range transformations of genomic ranges.
- `setops-methods` for set operations on IRanges objects.
- `endoapply` in the S4Vectors package.
Examples

```r
## ---------------------------------------------------------------------
## 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) # same as endoapply(collection, shift, shift=5)

## Sanity check:
res1 <- shift(collection, shift=5)
res2 <- endoapply(collection, shift, shift=5)
stopifnot(identical(res1, res2))

## ---------------------------------------------------------------------
## 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)

## resize()

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

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

### On a Ranges object:

```r
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:

```r
flank(collection)
flank(collection, width=10)
```

## promoters()

### On a Ranges object:

```r
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:

```r
promoters(collection, upstream=5, downstream=2)
```

## reflect()

### On a Ranges object:

```r
bounds <- IRanges(c(0, 5, 3), c(10, 6, 9))
reflect(ir3, bounds)
```

### On a Ranges object:

```r
reflect() does not yet support RangesList objects!
```

## restrict()

### On a Ranges object:

```r
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:

```r
restrict(collection, start=2, end=8)
```

## threebands()
## On a Ranges object:

```r
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.

---

### 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-class`.

### 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)

Hervé Pagès

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)
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

```r
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)
```

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 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)

Hervé Pagès

See Also

IRanges-class, narrow

Examples

```r
# 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())
```

```r
# 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, nchunk)

## 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).
nchunk A single positive integer. The number of chunks.
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

succesiveIRanges 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)

Hervé Pagès

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))
succesiveIRanges(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
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)
IMPORTANT NOTE - 9/4/2014: This man page is being refactored. Most of the things that used to be documented here have been moved to the man page for List objects located in the S4Vectors package.

Details

The only thing left here is the documentation of the stack method for List objects. In the code snippets below, x is a List 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.

See Also

• The List class defined and documented in the S4Vectors package.

Examples

starts <- IntegerList(c(1, 5), c(2, 8))
ends <- IntegerList(c(3, 8), c(5, 9))
rgl <- IRangesList(start=starts, end=ends)
rangleDataFrame <- stack(rgl, "space", "ranges")
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 \(\geq 1\) and \(\leq \text{width}(x)\). 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 \(\text{mask.width}\) (a single integer \(\geq 1\)) and masking the ranges of positions specified by \(\text{start}\), \(\text{end}\) and \(\text{width}\). See the IRanges constructor (?IRanges) for how \(\text{start}\), \(\text{end}\) and \(\text{width}\) 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 \(\geq 0\) and \(\leq \text{width}(x)\).
- `maskedratio(x)`: \(\text{maskedwidth}(x) / \text{width}(x)\)
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)

Hervé Pagès

See Also

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

Examples

```r
## Making a MaskCollection object:
m1 <- Mask(mask.width=29, start=c(11, 25, 28), width=c(5, 2, 2))
m2 <- Mask(mask.width=29, start=c(3, 10, 27), width=c(5, 8, 1))
m3 <- Mask(mask.width=29, start=c(7, 12), width=c(2, 4))
mymasks <- append(append(m1, m2), m3)
mymasks
length(mymasks)
width(mymasks)
collapse(mymasks)

## Names and descriptions:
names(mymasks) <- c("A", "B", "C")  # names should be short and unique...
mymasks
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]])
```
**multisplit**

```r
c Roths[[2]][[-3]]
append(Roths[[-2]], gaps(Roths[[2]]))
```

---

**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**

```r
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**

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

---

**NCLlist-class**

*Nested Containment List objects*

**Description**

The NCLlist class is a container for storing the Nested Containment List representation of a `Ranges` object. Preprocessing a `Ranges` object as a Nested Containment List allows efficient overlap-based operations like `findOverlaps`.

The NCLLists class is a container for storing a collection of NCLlist objects. An NCLLists object is typically the result of preprocessing each list element of a `RangesList` object as a Nested Containment List. Like with NCLlist, the NCLLists object can then be used for efficient overlap-based operations.

To preprocess a `Ranges` or `RangesList` object, simply call the NCLlist or NCLLists constructor function on it.
Usage

```r
NCList(x, circle.length=NA_integer_)
NCLists(x, circle.length=NA_integer_)
```

Arguments

- **x**: The `Ranges` or `RangesList` object to preprocess.
- **circle.length**: Use only if the space (or spaces if `x` is a `RangesList` object) on top of which the ranges in `x` are defined needs (need) to be considered circular. If that's the case, then use `circle.length` to specify the length(s) of the circular space(s).
  - For `NCList`, `circle.length` must be a single positive integer (or NA if the space is linear).
  - For `NCLists`, it must be an integer vector parallel to `x` (i.e. same length) and with positive or NA values (NAs indicate linear spaces).

Details

The `GenomicRanges` package also defines the `GNCList` constructor and class for preprocessing and representing a vector of genomic ranges as a data structure based on Nested Containment Lists.

Some important differences between the new `findOverlaps/countOverlaps` implementation based on Nested Containment Lists (BioC >= 3.1) and the old implementation based on Interval Trees (BioC < 3.1):

- With the new implementation, the hits returned by `findOverlaps` are not fully ordered (i.e. ordered by queryHits and subject Hits) anymore, but only partially ordered (i.e. ordered by queryHits only). Other than that, and except for the 2 particular situations mentioned below, the 2 implementations produce the same output. However, the new implementation is faster and more memory efficient.
- With the new implementation, either the query or the subject can be preprocessed with `NCList` for a `Ranges` object (replacement for `IntervalTree`), `NCLists` for a `RangesList` object (replacement for `IntervalForest`), and `GNCList` for a `GenomicRanges` object (replacement for `GIntervalTree`). However, for a one-time use, it is NOT advised to explicitly preprocess the input. This is because `findOverlaps` or `countOverlaps` will take care of it and do a better job at it (by preprocessing only what’s needed when it’s needed, and releasing memory as they go).
- With the new implementation, `countOverlaps` on `Ranges` or `GenomicRanges` objects doesn’t call `findOverlaps` in order to collect all the hits in a growing `Hits` object and count them only at the end. Instead, the counting happens at the C level and the hits are not kept. This reduces memory usage considerably when there is a lot of hits.
- When `minoverlap=0`, zero-width ranges are now interpreted as insertion points and considered to overlap with ranges that contain them. With the old algorithm, zero-width ranges were always ignored. This is the 1st situation where the new and old implementations produce different outputs.
- When using `select="arbitrary"`, the new implementation will generally not select the same hits as the old implementation. This is the 2nd situation where the new and old implementations produce different outputs.
- The new implementation supports preprocessing of a `GenomicRanges` object with ranges defined on circular sequences (e.g. on the mitochondrial chromosome). See `GNCList` in the `GenomicRanges` package for some examples.
NCList-class

- Objects preprocessed with NClist, NCLists, and GNCList are serializable (with save) for later use. Not a typical thing to do though, because preprocessing is very cheap (i.e. very fast and memory efficient).

Value

An NCList object for the NCList constructor and an NCLists object for the NCLists constructor.

Author(s)

Hervé Pagès

References


See Also

- The GNCList constructor and class defined in the GenomicRanges package.
- findOverlaps for finding/counting interval overlaps between two range-based objects.
- Ranges and RangesList objects.

Examples

```r
## The example below is for illustration purpose only and does NOT
## reflect typical usage. This is because, for a one-time use, it is
## NOT advised to explicitely preprocess the input for findOverlaps()
## or countOverlaps(). These functions will take care of it and do a
## better job at it (by preprocessing only what's needed when it's
## needed, and release memory as they go).

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

## Either the query or the subject of findOverlaps() can be preprocessed:

ppsubject <- NCList(subject)
hits1 <- findOverlaps(query, ppsubject)
hits1

ppquery <- NCList(query)
hits2 <- findOverlaps(ppquery, subject)
hits2

## Note that 'hits1' and 'hits2' contain the same hits but not in the
## same order.
stopifnot(identical(sort(hits1), sort(hits2)))
```
nearest-methods  

Finding the nearest range neighbor

Description

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

Usage

```r
## 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)

## S4 method for signature 'Pairs,missing'
distance(x, y)
```

Arguments

- **x**  The query Ranges instance, or (for distance()) a Pairs containing both the query (first) and subject (second).
- **subject**  The subject Ranges instance, within which the nearest neighbors are found. Can be missing, in which case x is also the subject.
- **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”.
- **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.
- **hits**  The hits between x and subject
- **...**  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.

Here is roughly how it proceeds, for a range \( x_i \) in x:
1. Find the ranges in subject that overlap \( x_i \). If a single range \( s_i \) in subject overlaps \( x_i \), \( s_i \) is returned as the nearest neighbor of \( x_i \). If there are multiple overlaps, one of the overlapping ranges is chosen arbitrarily.

2. If no ranges in subject overlap with \( x_i \), then the range in subject with the shortest distance from its end to the start \( x_i \) or its start to the end of \( x_i \) 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:

\[ x \text{ and } y \text{ overlap } \iff \text{ distance}(x, y) = 0 \]

Instead we say

\[ x \text{ and } y \text{ overlap } \implies \text{ distance}(x, y) = 0 \]

or

\[ x \text{ and } y \text{ overlap or are adjacent } \iff \text{ distance}(x, y) = 0 \]

- **selectNearest**: Selects the hits that have the minimum distance within those for each query range. Ties are possible and can be broken with `breakTies`.

**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 an elementMetadata column of the distance between the pair. Access distance with `mcols` accessor.

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

For selectNearest, a Hits object, sorted by query.

**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

```r
## ------------------------------------------
## 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**

IMPORTANT NOTE: Starting with BioC 3.4, RangedDataList objects are defunct in favor of GRangesList objects (the GRangesList class is defined in the GenomicRanges package).

Note that the use of RangedData and RangedDataList objects has been discouraged in favor of GRanges and GRangesList objects since BioC 2.12.

RangedData objects will be deprecated soon.

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.

**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.

**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(elementNROWS(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(elementNROWS(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(elementNROWS(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.

\[ \text{split}(x, f, \text{drop} = \text{FALSE}) \]: 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.

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

\[ \text{c}(x, \ldots, \text{recursive} = \text{FALSE}) \]: Combines x with arguments specified in \ldots, 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

An \text{lapply} method is provided to apply a function over the spaces of a RangedData:

\[ \text{lapply}(X, \text{FUN}, \ldots) \]: Applies FUN to each space in X with extra parameters in \ldots.

Author(s)

Michael Lawrence

See Also

\text{DataTable}, the parent of this class, with more utilities.

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]  # 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,] # row 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)
RangedSelection-class

RangedSelection objects are defunct!

RangedDataList-class

Lists of RangedData (DEFUNCT!)

Description

RangedDataList objects are defunct!

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.
**Ranges-class**

**Coercion**

\[ \text{as(from, "RangedSelection")}: \text{Coerces from to a RangedSelection object. Typically, from is a RangesList, the ranges of which become the ranges in the new RangedSelection.} \]

**Accessors**

In the code snippets below, \( x \) is always a RangedSelection.

\[ \text{ranges(x), ranges(x) <- value}: \text{Gets or sets the ranges, a RangesList, that select rows with overlapping intervals.} \]

\[ \text{colnames(x), colnames(x) <- value}: \text{Gets the names, a character vector, indicating the columns.} \]

**Author(s)**

Michael Lawrence

**Examples**

\[ \text{r1 <- RangesList(chr1 = IRanges(c(1, 5), c(3, 6)))} \]

\[ \text{RangedSelection(r1)} \]

\[ \text{as(r1, "RangedSelection") # same as above} \]

\[ \text{RangedSelection(r1, "score")} \]

---

**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: IRange, NCList, PartitioningByEnd.

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 IRange objects.

Note that many more operations on Ranges objects are described in other man pages of the IRange package. See for example the man page for intra range transformations (e.g. \texttt{shift()}), or the man page for inter range transformations (e.g. \texttt{reduce()})

**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):** \(\text{start}(x) + \text{width}(x) - 1\)

**mid(x):** returns the midpoint of the range, \(\text{start}(x) + \text{floor}(\text{width}(x) - 1)/2\).

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

**update(object, \ldots):** Convenience method for combining multiple modifications of object in one single call. For example \(\text{object} \leftarrow \text{update(object, start=\text{start(object)}-2L, end=\text{end(object)}+2L}\).

**tile(x, n, width, \ldots):** 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 IRangeList the same length as x. Ranges with a width less than the width argument are returned unchanged.

**slidingWindows(x, width, step=1L):** Generates sliding windows within each range of x, of width width, and starting every step positions. The return value is a IRangeList the same length as x. Ranges with a width less than the width argument are returned unchanged. If the sliding windows do not exactly cover x, the last window is partial.

**isEmpty(x):** Return a logical value indicating whether x is empty or not.

**as.matrix(x, \ldots):** Convert x into a 2-column integer matrix containing \text{start}(x) and \text{width}(x).

**as.data.frame(x, row.names=NULL, optional=FALSE, \ldots):** 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(\ldots) 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 \text{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 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 Nor-
malIRanges 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 TRUE or FALSE 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. Pagès and M. Lawrence

See Also

- IRanges objects (NormalIRanges objects are documented in the same man page).
- Ranges-comparison for comparing and ordering ranges.
- findOverlaps-methods for finding/counting overlapping ranges.
- intra-range-methods and inter-range-methods for intra range and inter range transformations of a Ranges object.
- coverage-methods for computing the coverage of a set of ranges.
- setops-methods for set operations on ranges.
- nearest-methods for finding the nearest range neighbor.

Examples

```r
## Basic manipulation
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))
```
### Ranges-comparison

Comparing and ordering ranges

#### Description

Methods for comparing and/or ordering Ranges objects.

#### Usage

```r
## match() & selfmatch()
## ---------------------
## S4 method for signature 'Ranges,Ranges'
match(x, table, nomatch=NA_integer_, incomparables=NULL,
      method=c("auto", "quick", "hash"))

## S4 method for signature 'Ranges'
selfmatch(x, method=c("auto", "quick", "hash"))

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

## Generalized parallel comparison of 2 Ranges objects
## ---------------------------------------------------
## S4 method for signature 'Ranges,Ranges'
pcompare(x, y)

rangeComparisonCodeToLetter(code)
```

#### Arguments

- `x, table, y`: 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`: For match and selfmatch: Use a Quicksort-based (method="quick") or a hash-based (method="hash") algorithm. The latter tends to give better performance, except maybe for some pathological input that we've not encountered so far. When method="auto" is specified, the most efficient algorithm will be used, that is, the hash-based algorithm if length(x) <= 2^29, otherwise the Quicksort-based algorithm.
  
  For order: The method argument is ignored.
One or more Ranges objects. The Ranges objects after the first one are used to break ties.

na.last Ignored.

decreasing TRUE or FALSE.

code A vector of codes as returned by pcompare.

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.

pcompare(x, y): Performs element-wise (aka "parallel") comparison of 2 Ranges objects of x and y, that is, returns an integer vector where the i-th element is a code describing how x[i] is qualitatively positioned with respect to y[i].

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]: .000000000... 2 i: x[i]: ......oooo.
  y[i]: ....oooo....          y[i]: .000000000.. 

-1 f: x[i]: ....oooo..... 1 h: x[i]: ......oooo.
  y[i]: ......oooo...         y[i]: ....oooo... 

 0 g: x[i]: ......oooo....
  y[i]: ......oooo... 

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 pcompare 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 pcompare. 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).

`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)**

Hervé Pagès

**See Also**

- The `Ranges` class.
- `Vector-comparison` in the `S4Vectors` package for general information about comparing, ordering, and tabulating vector-like objects.
- `GenomicRanges-comparison` in the `GenomicRanges` package for comparing and ordering genomic ranges.
- `findOverlaps` for finding overlapping ranges.
- `intra-range-methods` and `inter-range-methods` for intra and inter range transformations.
- `setops-methods` for set operations on `IRanges` objects.

**Examples**

```r
## A. ELEMENT-WISE (AKA "PARALLEL") COMPARISON OF 2 Ranges OBJECTS
x0 <- IRanges(1:11, width=4)
x0
```
y0 <- IRanges(6, 9)
pcompare(x0, y0)
pcompare(IRanges(4:6, width=6), y0)
pcompare(IRanges(6:8, width=2), y0)
pcompare(x0, y0) < 0  # equivalent to 'x0 < y0'
pcompare(x0, y0) == 0  # equivalent to 'x0 == y0'
pcompare(x0, y0) > 0  # equivalent to 'x0 > y0'

rangeComparisonCodeToLetter(-10:10)
rangeComparisonCodeToLetter(pcompare(x0, y0))

## Handling of zero-width ranges (a.k.a. empty ranges):
x1 <- IRanges(11:17, width=0)
x1
pcompare(x1, x1[4])
pcompare(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, pcompare()
## considers x1[2] and x1[6] to be *adjacent* to IRanges(12, 15), and
## thus returns codes -5 and 5:
pcompare(x1[2], IRanges(12, 15))  # -5
pcompare(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

## ---------------------------------------------------------------------
## 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
RangesList-class

## ---------------------------------------------------------------------
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(elementNROWS(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 ends, value can be an integer vector of length `sum(elementNROWS(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 widths, value can be an integer vector of length `sum(elementNROWS(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, ..., value.name = "value", use.outer.mcols = FALSE)`: Coerces x to a data.frame. See as.data.frame on the List man page for details (?List).
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.

In the following code snippets, from is something other than a RangesList:

as(from, "RangesList"): When from is a Ranges, analogous to `as.list` on a vector.

**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**

```r
## Basic manipulation
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), elementNROWS(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

---

### rdapply

**Applying over spaces (DEFUNCT!)**

#### Description

RDApplyParams objects and the `rdapply()` function are defunct!

---

### 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

```r
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.
read.Mask

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 constituting 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 constituting the returned mask. See `?IRanges-class` for more information about the names of an IRanges object.

See Also

MaskCollection-class, IRanges-class

Examples

```r
## 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
## hs_ref_chrY.agp.gz 5 KB 24/03/08 04:33:00 PM
##
## on May 9, 2008.

chrY_length <- 5772954
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 you're 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
}
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)
Description

IMPORTANT NOTE - 7/3/2014: This man page is being refactored. Most of the things that used to be documented here have been moved to the man page for Rle objects located in the S4Vectors package.

Coercion

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

\[
\text{as(from, "IRanges")}: \text{Creates an IRanges instance from a logical Rle. Note that this instance is guaranteed to be normal.}
\]
\[
\text{as(from, "NormalIRanges")}: \text{Creates a NormalIRanges instance from a logical Rle.}
\]

General Methods

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

\[
\text{split(x, f, drop=FALSE)}: \text{Splits x according to f to create a CompressedRleList object. If f is a list-like object then drop is ignored and f is treated as if it was rep(seq_len(length(f)), sapply(f, length)), so the returned object has the same shape as f (it also receives the names of f). Otherwise, if f is not a list-like object, empty list elements are removed from the returned object if drop is TRUE.}
\]
\[
\text{findRange(x, vec)}: \text{Returns an IRanges object representing the ranges in Rle vec that are referenced by the indices in the integer vector x.}
\]
\[
\text{splitRanges(x)}: \text{Returns a CompressedIRangesList object that contains the ranges for each of the unique run values.}
\]

See Also

The Rle class defined and documented in the S4Vectors package.

Examples

\[
x <- \text{Rle(10:1, 1:10)}
x
\]
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, rangesList = start).

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)
Description

unsplit method for List object and split<- method for Vector object.

Usage

## S4 method for signature 'List'
unsplit(value, f, drop = FALSE)
## S4 replacement method for signature 'Vector'
split(x, f, drop = FALSE, ...) <- value

Arguments

value The List object to unsplit.
f A factor or list of factors
drop Whether to drop empty elements from the returned list
x Like x
... Extra arguments to pass to FUN

Details

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.

Author(s)

Michael Lawrence

Description

Performs set operations on Ranges and RangesList objects.
Usage

## Vector-wise set operations
## --------------------------
## S4 method for signature 'Ranges,Ranges'
union(x, y)
## S4 method for signature 'Pairs,missing'
union(x, y, ...)

## S4 method for signature 'Ranges,Ranges'
intersect(x, y)
## S4 method for signature 'Pairs,missing'
intersect(x, y, ...)

## S4 method for signature 'Ranges,Ranges'
setdiff(x, y)
## S4 method for signature 'Pairs,missing'
setdiff(x, y, ...)

## Element-wise (aka "parallel") set operations
## --------------------------------------------
## S4 method for signature 'Ranges,Ranges'
punion(x, y, fill.gap=FALSE)
## S4 method for signature 'Pairs,missing'
punion(x, y, ...)

## S4 method for signature 'Ranges,Ranges'
pintersect(x, y, resolve.empty=c("none", "max.start", "start.x"))
## S4 method for signature 'Pairs,missing'
pintersect(x, y, ...)

## S4 method for signature 'Ranges,Ranges'
psetdiff(x, y)
## S4 method for signature 'Pairs,missing'
psetdiff(x, y, ...)

## S4 method for signature 'Ranges,Ranges'
pgap(x, y)

Arguments

x, y  Objects representing ranges.
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.)
...

The methods for Pairs objects pass any extra argument to the internal call to
punion(first(x), last(x), ...), pintersect(first(x), last(x), ...), etc...

Details

The union, intersect and setdiff methods for Ranges objects return a "normal" Ranges object 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 Ranges objects are defined. For these methods, x and y must have the same length (i.e. same number of ranges). They return a Ranges object parallel to x and y i.e. where the i-th range corresponds to the i-th range in x and y[i].

If x is a Pairs object, then y should be missing, and the operation is performed between the members of each pair.

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. Pagès 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.
- setops-methods in the GenomicRanges package for set operations on genomic ranges.
- findOverlaps-methods for finding/counting overlapping ranges.
- intra-range-methods and inter-range-methods for intra range and inter range transformations.
- Ranges and RangesList objects. In particular, normality of a Ranges object is discussed in the man page for Ranges objects.
- mendoapply in the S4Vectors package.

Examples

```r
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))
```
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, or any object coercible to an Rle object.

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 ?viewerumization-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

```r
## 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)
```

Description

IMPORTANT NOTE - 4/29/2014: This man page is being refactored. Most of the things that used to be documented here have been moved to the man page for Vector objects located in the S4Vectors package.

Evaluation

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

```r
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.
```
Convenience wrappers for common subsetting operations

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.

```r
window(x, start=NA, end=NA, width=NA): 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.

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. 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.

```r
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.

```r
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.
```

Coercion

```r
as.list(x): coerce a Vector to a list, where the `i`th element of the result corresponds to `x[i].
```

See Also

The Vector class defined and documented in the S4Vectors package.
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**

```r
## 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)
```
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.
ViewsList-class

Other methods

trim(x, use.names=TRUE): Equivalent to restrict(x, start=1L, end=length(subject(x)), keep.all.ranges).

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(width, gapwidth, from)). See ?successiveIRanges for a description of the width, gapwidth and from arguments.

Author(s)

Hervé Pagès

See Also

IRanges-class, Vector-class, IRanges-utils, XVector.

Some direct subclasses of the Views class: RleViews-class, XIntegerViews-class, XDoubleViews-class, XStringViews-class.

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.
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. Pagès

See Also

List-class, RleViewsList-class.
findOverlaps.

Examples

showClass("ViewsList")
Index

!,,CompressedList-method

(CompressedList-class), 8

*Topic arith

view-summarization-methods, 88

*Topic classes

AtomicList, 3
CompressedHitsList-class, 7
CompressedList-class, 8
DataFrameList-class, 15
GappedRanges-class, 24
Grouping-class, 24
Hits-class-leftovers, 30
IRanges-class, 43
IRangesList-class, 49
List-class-leftovers, 50
MaskCollection-class, 50
NCLList-class, 53
RangedData-class, 59
RangedDataList-class, 64
RangedSelection-class, 65
Ranges-class, 69
Ranges-comparison, 69
RangesList-class, 73
rdapply, 75
reverse, 78
Rle-class-leftovers, 79
RleViews-class, 80
RleViewsList-class, 80
slice-methods, 85
Vector-class-leftovers, 86
view-summarization-methods, 88
Views-class, 90
ViewsList-class, 91

*Topic utilities

coverage-methods, 9
inter-range-methods, 31
intra-range-methods, 37
IRanges-constructor, 45
IRanges-utils, 47
nearest-methods, 56
setops-methods, 82

[,,CompressedSplitDataFrameList-method
(DataFrameList-class), 15
[,,RangedData-method (RangedData-class), 59
[,,SimpleSplitDataFrameList-method
(DataFrameList-class), 15
[<-,SplitDataFrameList-method
(DataFrameList-class), 15
[[],RangedData-method
(DataFrameList-class), 15

[[],RangedData-method
(RangedData-class), 59
INDEX 95

CharacterList, 8, 38
CharacterList (AtomicList), 3
CharacterList-class (AtomicList), 3
cchartr, ANY, ANY, CompressedCharacterList-method
(AtomicList-utils), 5
cchartr, ANY, ANY, CompressedRleList-method
(AtomicList-utils), 5
cchartr, ANY, ANY, SimpleCharacterList-method
(AtomicList-utils), 5
cchartr, ANY, ANY, SimpleRleList-method
(AtomicList-utils), 5
class:AtomicList (AtomicList), 3
class:ComplexList (AtomicList), 3
class:CompressedAtomicList
(AtomicList), 3
class:CompressedCharacterList
(AtomicList), 3
class:CompressedComplexList
(AtomicList), 3
class:CompressedFactorList
(AtomicList), 3
class:CompressedGrouping
(Grouping-class), 24
class:CompressedHitsList
(CompressedHitsList-class), 7
class:CompressedIntegerList
(AtomicList), 3
class:CompressedIRangesList
(IRangesList-class), 49
class:CompressedList
(CompressedList-class), 8
class:CompressedLogicalList
(AtomicList), 3
class:CompressedManyToManyGrouping
(Grouping-class), 24
class:CompressedManyToOneGrouping
(Grouping-class), 24
class:CompressedNormalIRangesList
(IRangesList-class), 49
class:CompressedNumericList
(AtomicList), 3
class:CompressedRleList (AtomicList), 3
class:Dups (Grouping-class), 24
class:FactorList (AtomicList), 3
class:GappedRanges
(GappedRanges-class), 24
class:Grouping (Grouping-class), 24
class:GroupingIRanges (Grouping-class),
  24
class:GroupingRanges (Grouping-class), 24
class:H2LGrouping (Grouping-class), 24
class:IntegerList (AtomicList), 3
class:IRanges (IRanges-class), 43
class:IRangesList (IRangesList-class), 49
class:LogicalList (AtomicList), 3
class:ManyToOneGrouping
(Grouping-class), 24
class:ManyToManyGrouping
(Grouping-class), 24
class:MaskCollection
(MaskCollection-class), 50
class:NCLList (NCLList-class), 53
class:NCLLists (NCLList-class), 53
class:NormalIRanges (IRanges-class), 43
class:NormalIRangesList
(IRangesList-class), 49
class:NumericList (AtomicList), 3
class:Partitioning (Grouping-class), 24
class:PartitioningByEnd
(Grouping-class), 24
class:PartitioningByWidth
(Grouping-class), 24
class:PartitioningMap (Grouping-class), 24
class:RangedData (RangedData-class), 59
class:Ranges (Ranges-class), 65
class:RangesList-class
(RangesList-class), 73
class:RangesORmissing
(nearest-methods), 56
class:RawList (AtomicList), 3
class:RleList (AtomicList), 3
class:RleViews (RleViews-class), 80
class:SimpleAtomicList (AtomicList), 3
class:SimpleComplexList (AtomicList), 3
class:SimpleFactorList (AtomicList), 3
class:SimpleGrouping (Grouping-class),
  24
class:SimpleIntegerList (AtomicList), 3
class:SimpleManyToManyGrouping
(Grouping-class), 24
class:SimpleManyToOneGrouping
(Grouping-class), 24
class:SimpleNormalIRangesList
(IRangesList-class), 49
coerce, SortedByQueryHits, IRanges-method
(Hits-class-leftovers), 30
coerce, SortedByQueryHits, List-method
(Hits-class-leftovers), 30
coerce, SortedByQueryHits, Partitioning-method
(Hits-class-leftovers), 30
coerce, SortedByQueryHits, PartitioningByEnd-method
(Hits-class-leftovers), 30
coerce, SortedByQueryHits, Ranges-method
(DataFrameList-class), 15
coerce, SplitDataFrameList, DataFrame-method
(DataFrameList-class), 15
coerce, vector, AtomicList-method
(AtomicList), 3
coerce, vector, CompressedCharacterList-method
(AtomicList), 3
coerce, vector, CompressedComplexList-method
(AtomicList), 3
coerce, vector, CompressedIntegerList-method
(AtomicList), 3
coerce, vector, CompressedLogicalList-method
(AtomicList), 3
coerce, vector, CompressedNumericList-method
(AtomicList), 3
coerce, vector, CompressedRawList-method
(AtomicList), 3
coerce, vector, CompressedRleList-method
(AtomicList), 3
collapse (MaskCollection-class), 50
collapse, MaskCollection-method
(colnames, CompressedSplitDataFrameList-method
(DataFrameList-class), 15
collapse, MaskCollection-method
(colnames, DataFrameList-method
(DataFrameList-class), 15
collapse, NormalIRanges-method
(colnames, DataFrameList-method
(DataFrameList-class), 15
collapse, Ranges-method
(colnames, DataFrameList-method
(DataFrameList-class), 15
collapse, Ranges-method
(colnames, DataFrameList-method
(DataFrameList-class), 15
collapse, Views, IRanges-method
(colnames, NormalIRanges-method
(Views-class), 90
collapse, Views, Ranges-method
(Views-class), 90
collapse, Views, NormalIRanges-method
(Views-class), 90
colnames, CompressedSplitDataFrameList-method
(DataFrameList-class), 15
colnames, DataFrameList-method
(DataFrameList-class), 15
colnames, RangedData-method
(DataFrameList-class), 15
colnames, RangedSelection-method
(DataFrameList-class), 64
colnames, SimpleSplitDataFrameList-method
(DataFrameList-class), 15
colSums, 89
colSums, 89
columnMetadata (DataFrameList-class), 15
columnMetadata, CompressedSplitDataFrameList-method
(DataFrameList-class), 15
columnMetadata, DataFrameList-method
(DataFrameList-class), 15
columnMetadata, RangedData-method
(DataFrameList-class), 59
columnMetadata, SimpleSplitDataFrameList-method
(DataFrameList-class), 15
columnMetadata, RangedSelection-method
(DataFrameList-class), 64
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
columnMetadata<- (DataFrameList-class),
Complex, CompressedAtomicList-method
(AtomicList-utils), 5
Complex, CompressedAtomicList-method
(AtomicList-utils), 5
Complex, SimpleAtomicList-method
(AtomicList-utils), 5
ComplexList (AtomicList), 3
ComplexList-class (AtomicList), 3
CompressedAtomicList (AtomicList), 3
CompressedAtomicList-class
(AtomicList), 3
CompressedCharacterList, 8
CompressedCharacterList (AtomicList), 3
INDEX

CompressedCharacterList-class (AtomicList), 3
CompressedComplexList (AtomicList), 3
CompressedComplexList-class (AtomicList), 3
CompressedFactorList (AtomicList), 3
CompressedFactorList-class (AtomicList), 3
CompressedGrouping-class (Grouping-class), 24
CompressedHitsList (CompressedHitsList-class), 7
CompressedHitsList-class (AtomicList), 7
CompressedIntegerList (AtomicList), 3
CompressedIntegerList-class (AtomicList), 3
CompressedIRangesList, 3, 74, 79
CompressedIRangesList-iRangesList-class, 49
CompressedIRangesList-class (iRangesList-class), 49
CompressedList (CompressedList-class), 8
CompressedList-class, 8
CompressedLogicalList, 8
CompressedLogicalList-class (AtomicList), 3
CompressedLogicalList-class (AtomicList), 3
CompressedManyToManyGrouping-class (Grouping-class), 24
CompressedManyToOneGrouping-class (Grouping-class), 24
CompressedNormalIRangesList, 3, 74
CompressedNormalIRangesList-iRangesList-class, 49
CompressedNormalIRangesList-class (iRangesList-class), 49
CompressedNumericList (AtomicList), 3
CompressedNumericList-class (AtomicList), 3
CompressedRawList (AtomicList), 3
CompressedRawList-class (AtomicList), 3
CompressedRleList, 8, 79
CompressedRleList (AtomicList), 3
CompressedRleList-class (AtomicList), 3
CompressedSplitDataFrameList, 3, 14
CompressedSplitDataFrameList-class (DataFrameList-class), 15
cor, AtomicList, AtomicList-method (AtomicList-utils), 5
countOverlaps, 54
countOverlaps (findOverlaps-methods), 19
countOverlaps, RangedData, RangedData-method (findOverlaps-methods), 19
countOverlaps, RangedData, RangesList-method (findOverlaps-methods), 19
countOverlaps, Ranges, Ranges-method (findOverlaps-methods), 19
countOverlaps, RangesList, RangedData-method (findOverlaps-methods), 19
countOverlaps, RangesList, RangesList-method (findOverlaps-methods), 19
countOverlaps, Vector, missing-method (findOverlaps-methods), 19
countOverlaps, Vector, Vector-method (findOverlaps-methods), 19
countOverlaps, Vector, ViewsList-method (findOverlaps-methods), 19
countOverlaps, ViewsList, Vector-method (findOverlaps-methods), 19
countOverlaps, ViewsList, ViewsList-method (findOverlaps-methods), 19
cov, AtomicList, AtomicList-method (AtomicList-utils), 5
coverage, 86
cov, AtomicList, AtomicList-method (AtomicList-utils), 5
coverage-methods, 9
coverage, RangedData-method (coverage-methods), 9
coverage, Ranges-method (coverage-methods), 9
coverage, RangesList-method (coverage-methods), 9
cov, AtomicList, AtomicList-method (AtomicList-utils), 5
cummax, CompressedAtomicList-method (AtomicList-utils), 5
cummin, CompressedAtomicList-method (AtomicList-utils), 5
cumprod, CompressedAtomicList-method (AtomicList-utils), 5
cumsum, 28
cumsum, CompressedAtomicList-method (AtomicList-utils), 5

dataframe, 14–16, 19, 59–61
dataframe-utils, 14
dataFrameList (DataFrameList-class), 15
dataFrameList-class, 15
dataTable, 15, 59, 62
desc (MaskCollection-class), 50
desc, MaskCollection-method (MaskCollection-class), 50
desc<- (MaskCollection-class), 50
desc<-,MaskCollection-method
(MaskCollection-class), 50
diff,28
diff,CompressedIntegerList-method
(AtomicList-utils), 5
diff,CompressedNumericList-method
(AtomicList-utils), 5
diff,CompressedRleList-method
(AtomicList-utils), 5
diff,IntegerList-method
(AtomicList-utils), 5
diff,NumericList-method
(AtomicList-utils), 5
diff,RleList-method
(AtomicList-utils), 5
dim,DataFrameList-method
(DataFrameList-class), 15
dimnames,DataFrameList-method
(DataFrameList-class), 15
dimnames<-,DataFrameList-method
(DataFrameList-class), 15
disjoin (inter-range-methods), 31
disjoin,CompressedIRangesList-method
(inter-range-methods), 31
disjoin,NormalIRanges-method
(inter-range-methods), 31
disjoin,Ranges-method
(inter-range-methods), 31
disjoin,RangesList-method
(inter-range-methods), 31
disjointBins (inter-range-methods), 31
disjointBins,NormalIRanges-method
(inter-range-methods), 31
disjointBins,Ranges-method
(inter-range-methods), 31
disjointBins,RangesList-method
(inter-range-methods), 31
distance (nearest-methods), 56
distance,Pairs,missing-method
(nearest-methods), 56
distance,Ranges-method
(nearest-methods), 56
distanceToNearest,Ranges-method
(nearest-methods), 56
distanceToNearest,RangesORmissing-method
(nearest-methods), 56
drop,AtomicList-method (AtomicList), 3
duplicated,71
duplicated,CompressedAtomicList-method
(AtomicList), 3
duplicated,CompressedList-method
(AtomicList), 3
duplicated,Dups-method
(AtomicList), 3
duplicated,Dups-method
(Grouping-class), 24
dupes (Grouping-class), 24
dupe-class (Grouping-class), 24
elementNROWS,CompressedList-method
(CompressedList-class), 8
elementNROWS,GappedRanges-method
(GappedRanges-class), 24
elementNROWS,NCLists-method
(NCList-class), 53
elementNROWS,RangedData-method
(RangedData-class), 59
elementNROWS,Ranges-method
(Ranges-class), 65
elementNROWS,Views-method
(Views-class), 90
dend,26
dend,CompressedIRangesList-method
(IRangesList-class), 49
dend,GappedRanges-method
(GappedRanges-class), 24
dend,NCList-method (NCList-class), 53
dend,NCLists-method (NCList-class), 53
dend,PartitioningByEnd-method
(Grouping-class), 24
dend,PartitioningByWidth-method
(Grouping-class), 24
dend,RangedData-method
(RangedData-class), 59
dend,Ranges-method (Ranges-class), 65
dend,RangesList-method
(RangesList-class), 73
dend,SimpleViewsList-method
(ViewsList-class), 91
dend,Views-method (Views-class), 90
dend<-(Ranges-class), 65
dend<-,IRanges-method (IRanges-class), 43
dend<-,RangedData-method
(RangedData-class), 59
dend<-,RangedList-method
(RangedList-class), 73
dend<-,Views-method (Views-class), 90
dendoapply,34,40,78
dendoapply,CompressedList-method
(CompressedList-class), 8
dendoapply,RangedData-method
(RangedData-class), 59
eval (Vector-class-leftovers), 86
eval,expression,Vector-method
(Vector-class-leftovers), 86
eval,language,Vector-method
(Vector-class-leftovers), 86
extractlist,17
extractList, ANY, ANY-method
(extractList), 17
extractList, ANY-method(extractList), 17
FactorList (AtomicList), 3
FactorList-class (AtomicList), 3
filterRules, RDApplyParams-method
(rdapply), 75
filterRules<-(rdapply), 75
filterRules<-, RDApplyParams-method
(rdapply), 75
findOverlapPairs
(findOverlaps-methods), 19
findOverlaps, 53–55, 58, 71, 91, 92
findOverlaps (findOverlaps-methods), 19
findOverlaps, ANY, Pairs-method
(findOverlaps-methods), 19
findOverlaps, GenomicRanges, GenomicRanges-method,
22
findOverlaps, integer, Ranges-method
(findOverlaps-methods), 19
findOverlaps, Pairs, ANY-method
(findOverlaps-methods), 19
findOverlaps, Pairs, missing-method
(findOverlaps-methods), 19
findOverlaps, Pairs, Pairs-method
(findOverlaps-methods), 19
findOverlaps, RangedData, RangedData-method
(findOverlaps-methods), 19
findOverlaps, RangedData, RangesList-method
(findOverlaps-methods), 19
findOverlaps, Ranges, Ranges-method
(findOverlaps-methods), 19
findOverlaps, RangesList, RangedData-method
(findOverlaps-methods), 19
findOverlaps, RangesList, RangesList-method
(findOverlaps-methods), 19
findOverlaps, Vector, missing-method
(findOverlaps-methods), 19
findOverlaps, Vector, Views-method
(findOverlaps-methods), 19
findOverlaps, Vector, ViewsList-method
(findOverlaps-methods), 19
findOverlaps, Views, Vector-method
(findOverlaps-methods), 19
findOverlaps, Views, Views-method
(findOverlaps-methods), 19
findOverlaps, ViewsList, Vector-method
(findOverlaps-methods), 19
findOverlaps, ViewsList, ViewsList-method
(findOverlaps-methods), 19
findOverlaps-methods, 19, 68, 84
findRange (Rle-class-leftovers), 79
findRange, Rle-method
(Rle-class-leftovers), 79
flank (intra-range-methods), 37
flank, CompressedIRangesList-method
(intra-range-methods), 37
flank, Ranges-method
(intra-range-methods), 37
flank, RangesList-method
(intra-range-methods), 37
follow (nearest-methods), 56
follow, Ranges, RangesORmissing-method
(nearest-methods), 56
from, CompressedHitsList-method
(CompressedHitsList-class), 7
GAlignmentsList, 8
GappedRanges (GappedRanges-class), 24
GappedRanges-class, 24
gaps (inter-range-methods), 31
gaps, CompressedIRangesList-method
(inter-range-methods), 31
gaps, MaskCollection-method
(inter-range-methods), 31
gaps, Ranges-method
(inter-range-methods), 31
gaps, RangesList-method
(inter-range-methods), 31
gaps, Views-method
(inter-range-methods), 31
GenomicRanges, 19, 54, 58, 70
GenomicRanges-comparison, 71
GNList, 54, 55
IRanges, 22, 40, 58, 59
IRangesList, 8, 19, 22, 59
Grouping, 18
Grouping (Grouping-class), 24
grouping, 28
Grouping-class, 24
GroupingIRanges (Grouping-class), 24
GroupingIRanges-class (Grouping-class),
24
GroupingRanges (Grouping-class), 24
GroupingRanges-class (Grouping-class),
24
grouplength (Grouping-class), 24
grouplengths (Grouping-class), 24
grouplengths, CompressedGrouping-method
(Grouping-class), 24
grouplengths, Grouping-method
(Grouping-class), 24
grouplengths, GroupingRanges-method
(Grouping-class), 24
grouplengths,H2LGrouping-method
(Grouping-class), 24

grouprank (Grouping-class), 24
grouprank,H2LGrouping-method
(Grouping-class), 24
gsub,ANY,ANY,CompressedCharacterList-method
(AtomicList-utils), 5
gsub,ANY,ANY,CompressedRleList-method
(AtomicList-utils), 5
gsub,ANY,ANY,SimpleCharacterList-method
(AtomicList-utils), 5
gsub,ANY,ANY,SimpleRleList-method
(AtomicList-utils), 5

H2LGrouping (Grouping-class), 24
H2LGrouping-class (Grouping-class), 24
high2low (Grouping-class), 24
high2low,ANY-method (Grouping-class), 24
high2low,H2LGrouping-method
(Grouping-class), 24

Hits, 21, 22, 30, 54, 57, 58, 71
Hits-class-leftovers, 30
Hits-examples (Hits-class-leftovers), 30
HitsList, 7, 8, 21, 22

IntegerList, 8, 21, 22, 25, 32, 34, 38, 61
IntegerList (AtomicList), 3
IntegerList-class, 28
IntegerList-class (AtomicList), 3
inter-range-methods, 31, 34, 37, 40, 44, 48, 49, 68, 71, 84
intersect (setops-methods), 82
intersect,CompressedAtomicList,CompressedRleList-method
(AtomicList-utils), 5
intersect,CompressedIRangesList,CompressedIRangesList-method
(setops-methods), 82
intersect,Pairs,missing-method
(setops-methods), 82
intersect,Ranges,Ranges-method
(setops-methods), 82
intersect,RangesList,RangesList-method
(setops-methods), 82
intra-range-methods, 31, 34, 37, 40, 44, 48, 49, 68, 71, 84
IQR,AtomicList-method
(AtomicList-utils), 5
IRanges, 33, 34, 39, 40, 45–49, 51, 60, 66–68, 71, 76, 79, 86, 89, 90
IRanges (IRanges-constructor), 45
IRanges-class, 28, 43, 46, 48, 76, 91
IRanges-constructor, 44, 45
IRanges-utils, 44, 47, 91
IRangesList, 49, 80, 89

IRangesList (IRangesList-class), 49
IRangesList-class, 49
is.na,CompressedList-method
(CompressedList-class), 8
isDisjoint (inter-range-methods), 31
isDisjoint,NormalIRanges-method
(inter-range-methods), 31
isDisjoint,Ranges-method
(inter-range-methods), 31
isDisjoint,RangesList-method
(inter-range-methods), 31
isEmpty,NormalIRanges-method
(IRanges-class), 43
isEmpty,Ranges-method (Ranges-class), 65
isNormal (Ranges-class), 65
isNormal,CompressedIRangesList-method
(IRangesList-class), 49
isNormal,IRanges-method
(IRanges-class), 43
isNormal,NormalIRanges-method
(IRanges-class), 43
isNormal,Ranges-method (Ranges-class), 65
isNormal,RangesList-method
(RangesList-class), 73
isNormal,SimpleIRangesList-method
(IRangesList-class), 49
iteratorFun (rdapply), 75
iteratorFun, RDApplyParams-method
(rdapply), 75
iteratorFun<-(rdapply), 75
iteratorFun<-, RDApplyParams-method
(rdapply), 75

lapply,CompressedList-method
(AtomicList), 3
lapply,CompressedList-method
(CompressedList-class), 8
lapply,RangedData-method
(RangedData-class), 59
length,CompressedList-method
(CompressedList-class), 8
length,GappedRanges-method
(GappedRanges-class), 24
length,H2LGrouping-method
(Grouping-class), 24
length,MaskCollection-method
(MaskCollection-class), 50
length,NCLists-method (NCList-class), 53
length,NCLists-method (NCList-class), 53
length,PartitioningByEnd-method
(Grouping-class), 24
INDEX

length, RangedData-method (RangedData-class), 59
length, Ranges-method (Ranges-class), 65
length, Views-method (Views-class), 90
length<-, H2LGrouping-method (Grouping-class), 24
List, 3, 4, 8, 9, 15, 17–19, 25, 50, 73, 74, 82, 89, 91, 92
list, 49, 73
list-class, 92
list-class-leftovers, 50
LogicalList, 8, 21, 22, 38, 61
LogicalList (AtomicList), 3
LogicalList-class (AtomicList), 3
low2high (Grouping-class), 24
low2high, H2LGrouping-method (Grouping-class), 24
mad, AtomicList-method (AtomicList-utils), 5
ManyToManyGrouping (Grouping-class), 24
ManyToManyGrouping-class (Grouping-class), 24
ManyToOneGrouping (Grouping-class), 24
ManyToOneGrouping-class (Grouping-class), 24
mapOrder (Grouping-class), 24
mapOrder, PartitioningMap-method (Grouping-class), 24
Mask (MaskCollection-class), 50
MaskCollection, 34, 38, 40, 76, 78
MaskCollection (MaskCollection-class), 50
MaskCollection-class, 50, 76, 78
MaskCollection-show_frame (MaskCollection-class), 50
maskedRatio (MaskCollection-class), 50
maskedRatio, MaskCollection-method (MaskCollection-class), 50
maskedWidth (MaskCollection-class), 50
maskedWidth, MaskCollection-method (MaskCollection-class), 50
MaskedXString-class, 52
match, CompressedList, vector-method (CompressedList-class), 8
match, Ranges, Ranges-method (Ranges-comparison), 69
matchPattern, 51, 52
Math, CompressedAtomicList-method (AtomicList-utils), 5
Math, SimpleAtomicList-method (AtomicList-utils), 5
Math2, CompressedAtomicList-method (AtomicList-utils), 5
Math2, SimpleAtomicList-method (AtomicList-utils), 5
max, CompressedNormalIRangesList-method (IRangesList-class), 49
max, MaskCollection-method (MaskCollection-class), 50
max, NormalIRanges-method (IRanges-class), 43
max, SimpleNormalIRangesList-method (IRangesList-class), 49
max, Views-method (view-summarization-methods), 88
mean, AtomicList-method (AtomicList-utils), 5
mean, CompressedIntegerList-method (AtomicList-utils), 5
mean, CompressedLogicalList-method (AtomicList-utils), 5
mean, CompressedNumericList-method (AtomicList-utils), 5
mean, CompressedRleList-method (AtomicList-utils), 5
mean, Views-method (view-summarization-methods), 88
median, AtomicList-method (AtomicList-utils), 5
median, CompressedAtomicList-method (AtomicList-utils), 5
median, CompressedLogicalList-method (AtomicList-utils), 5
members (Grouping-class), 24
members, H2LGrouping-method (Grouping-class), 24
members, ManyToManyGrouping-method (Grouping-class), 24
mendoapply, 84
mendoapply, CompressedList-method (CompressedList-class), 8
merge, missing, RangesList-method (RangesList-class), 73
merge, RangesList, missing-method (RangesList-class), 73
merge, RangesList, RangesList-method (RangesList-class), 73
mergeByOverlaps (findOverlaps-methods), 19
mid (Ranges-class), 65
mid, Ranges-method (Ranges-class), 65
min, CompressedNormalIRangesList-method (IRangesList-class), 49
min, MaskCollection-method
(MaskCollection-class), 50
min, NormalIRanges-method
(IRanges-class), 43
min, SimpleNormalIRangesList-method
(IRangesList-class), 49
min, Views-method
(view-summarization-methods), 88
mstack (Vector-class-leftovers), 86
mstack, DataFrame-method
(DataFrame-utils), 14
mstack, Vector-method
(Vector-class-leftovers), 86
mstack, vector-method
(Vector-class-leftovers), 86
multisplit, 53
names, CompressedList-method
(CompressedList-class), 8
names, GappedRanges-method
(GappedRanges-class), 24
names, IRanges-method
(IRanges-class), 43
names, MaskCollection-method
(MaskCollection-class), 50
names, NCList-method
(NCList-class), 53
names, NCLists-method
(NCList-class), 53
names, Partitioning-method
(Grouping-class), 24
names, RangedData-method
(RangedData-class), 59
names, Views-method
(Views-class), 90
names<-, CompressedList-method
(CompressedList-class), 8
names<-, GappedRanges-method
(GappedRanges-class), 24
names<-, IRanges-method
(IRanges-class), 43
names<-, MaskCollection-method
(MaskCollection-class), 50
names<-, Partitioning-method
(Grouping-class), 24
names<-, RangedData-method
(RangedData-class), 59
names<-, Views-method
(Views-class), 90
narrow, (intra-range-methods), 46, 84
narrow, CompressedIRangesList-method
(intra-range-methods), 37
narrow, MaskCollection-method
(intra-range-methods), 37
narrow, NormalIRanges-method
(intra-range-methods), 43
narrow, Ranges-method
(intra-range-methods), 37
narrow, RangesList-method
(intra-range-methods), 37
ncol, CompressedSplitDataFrameList-method
(DataFrameList-class), 15
ncol, DataFrameList-method
(DataFrameList-class), 15
ncol, RangedData-method
(RangedData-class), 59
ncol, SimpleSplitDataFrameList-method
(DataFrameList-class), 15
ngap (GappedRanges-class), 24
ngap, GappedRanges-method
(GappedRanges-class), 24
nlnode, CompressedHitsList-method
(CompressedHitsList-class), 7
nobj (Grouping-class), 24
nobj, BaseManyToManyGrouping-method
(Grouping-class), 24
nobj, CompressedManyToOneGrouping-method
(Grouping-class), 24
nobj, H2LGrouping-method
(Grouping-class), 24
nobj, ManyToManyGrouping-method
(Grouping-class), 24
nobj, PartitioningByEnd-method
(Grouping-class), 24
NormalIRanges, 48, 49, 51, 52, 67, 68, 79
NormalIRanges (IRanges-class), 43
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalIRanges-class</td>
<td>52</td>
</tr>
<tr>
<td>NormalIRanges-class (IRanges-class)</td>
<td>43</td>
</tr>
<tr>
<td>NormalIRangesList</td>
<td>49</td>
</tr>
<tr>
<td>NormalIRangesList-class (IRangesList-class)</td>
<td>49</td>
</tr>
<tr>
<td>nRnode, CompressedHitsList-method</td>
<td>7</td>
</tr>
<tr>
<td>nrow, DataFrameList-method</td>
<td>15</td>
</tr>
<tr>
<td>NumericList</td>
<td>3</td>
</tr>
<tr>
<td>NumericList-class (AtomicList)</td>
<td>3</td>
</tr>
<tr>
<td>Ops, atomic, AtomicList-method</td>
<td>5</td>
</tr>
<tr>
<td>Ops, atomic, CompressedAtomicList-method (AtomicList-utils)</td>
<td>5</td>
</tr>
<tr>
<td>Ops, atomic, SimpleAtomicList-method (AtomicList-utils)</td>
<td>5</td>
</tr>
<tr>
<td>Ops, AtomicList, atomic-method (AtomicList-utils)</td>
<td>5</td>
</tr>
<tr>
<td>Ops, CompressedAtomicList, atomic-method (AtomicList-utils)</td>
<td>5</td>
</tr>
<tr>
<td>Ops, CompressedAtomicList, CompressedAtomicList-method</td>
<td>5</td>
</tr>
<tr>
<td>Ops, CompressedAtomicList, SimpleAtomicList-method</td>
<td>5</td>
</tr>
<tr>
<td>Ops, CompressedIRangesList, numeric-method (intra-range-methods)</td>
<td>37</td>
</tr>
<tr>
<td>Ops, Ranges, ANY-method (intra-range-methods)</td>
<td>37</td>
</tr>
<tr>
<td>Ops, Ranges, numeric-method (intra-range-methods)</td>
<td>37</td>
</tr>
<tr>
<td>Ops, SimpleAtomicList, numeric-method (intra-range-methods)</td>
<td>37</td>
</tr>
<tr>
<td>Ops, SimpleAtomicList, atomic-method (AtomicList-utils)</td>
<td>5</td>
</tr>
<tr>
<td>Ops, SimpleAtomicList, CompressedAtomicList-method</td>
<td>5</td>
</tr>
<tr>
<td>Ops, SimpleAtomicList, SimpleAtomicList-method</td>
<td>5</td>
</tr>
<tr>
<td>order, CompressedAtomicList-method (AtomicList-utils)</td>
<td>5</td>
</tr>
<tr>
<td>order, List-method (AtomicList)</td>
<td>3</td>
</tr>
<tr>
<td>order, Ranges-method (Ranges-comparison)</td>
<td>69</td>
</tr>
<tr>
<td>overlapsAny (findOverlaps-methods)</td>
<td>19</td>
</tr>
<tr>
<td>overlapsAny, RangedData, RangedData-method (findOverlaps-methods)</td>
<td>19</td>
</tr>
<tr>
<td>overlapsAny, RangedData, RangesList-method (findOverlaps-methods)</td>
<td>19</td>
</tr>
<tr>
<td>overlapsAny, RangesList, RangedData-method (findOverlaps-methods)</td>
<td>19</td>
</tr>
<tr>
<td>overlapsAny, RangesList, RangesData-method (findOverlaps-methods)</td>
<td>19</td>
</tr>
<tr>
<td>overlapsAny, RangesList, RangesList-method (findOverlaps-methods)</td>
<td>19</td>
</tr>
<tr>
<td>overlapsAny, Vector, missing-method (findOverlaps-methods)</td>
<td>19</td>
</tr>
<tr>
<td>overlapsAny, Vector, Vector-method (findOverlaps-methods)</td>
<td>19</td>
</tr>
<tr>
<td>overlapsAny, Vector, ViewsList-method (findOverlaps-methods)</td>
<td>19</td>
</tr>
<tr>
<td>overlapsAny, ViewsList, Vector-method (findOverlaps-methods)</td>
<td>19</td>
</tr>
<tr>
<td>overlapsAny, ViewsList, ViewsList-method (findOverlaps-methods)</td>
<td>19</td>
</tr>
<tr>
<td>Pairs</td>
<td>21, 56, 83, 84</td>
</tr>
<tr>
<td>Partitioning (Grouping-class)</td>
<td>24</td>
</tr>
<tr>
<td>Partitioning-class (Grouping-class)</td>
<td>24</td>
</tr>
<tr>
<td>PartitioningByEnd</td>
<td>66</td>
</tr>
<tr>
<td>PartitioningByEnd (Grouping-class)</td>
<td>24</td>
</tr>
<tr>
<td>PartitioningByEnd-class</td>
<td>24</td>
</tr>
<tr>
<td>PartitioningByHeight (Grouping-class)</td>
<td>24</td>
</tr>
<tr>
<td>PartitioningHeight (Grouping-class)</td>
<td>24</td>
</tr>
<tr>
<td>PartitioningMap (Grouping-class)</td>
<td>24</td>
</tr>
<tr>
<td>PartitioningMap-class (Grouping-class)</td>
<td>24</td>
</tr>
<tr>
<td>paste, CompressedAtomicList-method (AtomicList-utils)</td>
<td>5</td>
</tr>
<tr>
<td>pcompare</td>
<td>33</td>
</tr>
<tr>
<td>pcompare (Ranges-comparison)</td>
<td>69</td>
</tr>
<tr>
<td>pcompare, Ranges, Ranges-method (Ranges-comparison)</td>
<td>69</td>
</tr>
<tr>
<td>pgap (setops-methods)</td>
<td>82</td>
</tr>
<tr>
<td>pgap, Ranges, Ranges-method (setops-methods)</td>
<td>82</td>
</tr>
<tr>
<td>pintersect, 21</td>
<td>82</td>
</tr>
<tr>
<td>pintersect (setops-methods)</td>
<td>82</td>
</tr>
<tr>
<td>pintersect, Pairs, missing-method (setops-methods)</td>
<td>82</td>
</tr>
<tr>
<td>pintersect, Ranges, Ranges-method (setops-methods)</td>
<td>82</td>
</tr>
<tr>
<td>pmax, IntegerList-method</td>
<td>5</td>
</tr>
<tr>
<td>pmax, NumericList-method</td>
<td>5</td>
</tr>
<tr>
<td>pmax, RleList-method (AtomicList-utils)</td>
<td>5</td>
</tr>
<tr>
<td>pmax, int, IntegerList-method</td>
<td>5</td>
</tr>
</tbody>
</table>
pmax.int,NumericList-method (AtomicList-utils), 5
pmax.int,RleList-method (AtomicList-utils), 5
pmin,IntegerList-method (AtomicList-utils), 5
pmin,NumericList-method (AtomicList-utils), 5
pmin,RleList-method (AtomicList-utils), 5
pmin.int,IntegerList-method (AtomicList-utils), 5
pmin.int,NumericList-method (AtomicList-utils), 5
pmin.int,RleList-method (AtomicList-utils), 5
poverlaps (findOverlaps-methods), 19
poverlaps,integer,Ranges-method (findOverlaps-methods), 19
poverlaps,Ranges,integer-method (findOverlaps-methods), 19
poverlaps,Ranges,Ranges-method (findOverlaps-methods), 19
precede (nearest-methods), 56
precede,Ranges,RangesORmissing-method (nearest-methods), 56
promoters (intra-range-methods), 37
promoters,CompressedIRangesList-method (intra-range-methods), 37
promoters,Ranges-method (intra-range-methods), 37
promoters,RangesList-method (intra-range-methods), 37
promoters,Views-method (intra-range-methods), 37
psetdiff (setops-methods), 82
psetdiff,Pairs,missing-method (setops-methods), 82
psetdiff,Ranges,Ranges-method (setops-methods), 82
punion (setops-methods), 82
punion,Pairs,missing-method (setops-methods), 82
punion,Ranges,Ranges-method (setops-methods), 82
quantile,AtomicList-method (AtomicList-utils), 5
range (inter-range-methods), 31
range,CompressedIntegerList-method (AtomicList-utils), 5
range,CompressedLogicalList-method (AtomicList-utils), 5
range,CompressedNumericList-method (AtomicList-utils), 5
range,RangedData-method (inter-range-methods), 31
range,Ranges-method (inter-range-methods), 31
range,RangesList-method (inter-range-methods), 31
rangeComparisonCodeToLetter (Ranges-comparison), 69
RangedData, 15, 16, 20, 21, 33
RangedData (RangedData-class), 59
rangedData (rdapply), 75
RangedData-class, 59
rangedData<-(rdapply), 75
RangedData<-,RDApplyParams-method (rdapply), 75
RangedDataList, 62
RangedDataList (RangedDataList-class), 64
RangedDataList-class, 64
RangedSelection (RangedSelection-class), 64
RangedSelection-class, 64
Ranges, 10, 11, 17–22, 26, 32–34, 38–40, 43, 53–56, 58–61, 69–71, 73, 82, 84, 90
Ranges (Ranges-class), 65
ranges (Views-class), 90
ranges,CompressedRleList-method (AtomicList), 3
ranges,Hits-method (findOverlaps-methods), 19
ranges,HitsList-method (findOverlaps-methods), 19
ranges,NCList-method (NCList-class), 53
ranges,NCLists-method (NCList-class), 53
ranges,RangedData-method (RangedData-class), 59
ranges,RangedSelection-method (RangedSelection-class), 64
ranges,Rle-method (Rle-class-leftovers), 79
ranges,RleList-method (AtomicList), 3
ranges,SimpleViewsList-method (ViewsList-class), 91
ranges,Views-method (Views-class), 90
INDEX

Ranges-class, 28, 44, 48, 65
Ranges-comparison, 68, 69
ranges<-(Views-class), 90
ranges<-, RangedData-method
  (RangedData-class), 59
ranges<-, RangedSelection-method
  (RangedSelection-class), 64
ranges<-, Views-method (Views-class), 90
RangesList, 10, 11, 19–22, 32–34, 38–40, 49,
  53–55, 59–61, 65, 66, 82, 84
RangesList (RangesList-class), 73
RangesList-class, 73
RangesORmissing (nearest-methods), 56
RangesORmissing-class
  (nearest-methods), 56
rank, 71
rank, CompressedAtomicList-method
  (AtomicList-utils), 5
rank, List-method (AtomicList), 3
RawList (AtomicList), 3
RawList-class (AtomicList), 3
rbind, 7
rbind, DataFrameList-method
  (DataFrameList-class), 15
rbind, RangedData-method
  (RangedData-class), 59
rdapply, 75
rdapply, RDApplyParams-method (rdapply), 75
RDApplyParams (rdapply), 75
RDApplyParams-class (rdapply), 75
read.agpMask (read.Mask), 75
read.gapMask (read.Mask), 75
read.liftMask (read.Mask), 75
read.Mask, 52, 75
read.rmMask (read.Mask), 75
read.trfMask (read.Mask), 75
reduce, 37
reduce, (inter-range-methods), 31
reduce, CompressedIRangesList-method
  (inter-range-methods), 31
reduce, RangedData-method
  (inter-range-methods), 31
reduce, Ranges-method
  (inter-range-methods), 31
reduce, RangesList-method
  (inter-range-methods), 31
reduce, Views-method
  (inter-range-methods), 31
reducerFun (rdapply), 75
reducerFun, RDApplyParams-method
  (rdapply), 75
reducerFun<-, RDApplyParams-method
  (rdapply), 75
reducerParams (rdapply), 75
reducerParams, RDApplyParams-method
  (rdapply), 75
reducerParams<-, (rdapply), 75
reducerParams<-, RDApplyParams-method
  (rdapply), 75
reflect (intra-range-methods), 37
reflect, Ranges-method
  (intra-range-methods), 37
regroup (extractList), 17
relist, 19
relist, ANY, List-method (extractList), 17
relist, ANY, PartitioningByEnd-method
  (extractList), 17
relist, Vector, list-method
  (extractList), 17
relistToClass, 18, 19
relistToClass, data.frame-method
  (data.frame-utils), 14
relistToClass, DataFrame-method
  (data.frame-utils), 14
rep, Vector-method
  (Vector-class-leftovers), 86
rep.int, Vector-method
  (Vector-class-leftovers), 86
resize (inter-range-methods), 37
resize, CompressedIRangesList-method
  (inter-range-methods), 37
resize, IntervalList-method
  (inter-range-methods), 37
resize, Ranges-method
  (inter-range-methods), 37
resize, RangesList-method
  (inter-range-methods), 37
restrict, 84
restrict (intra-range-methods), 37
restrict, CompressedIRangesList-method
  (intra-range-methods), 37
restrict, Ranges-method
  (intra-range-methods), 37
restrict, RangesList-method
  (intra-range-methods), 37
rev, 78
rev, Vector-method
  (Vector-class-leftovers), 86
revElements, CompressedList-method
  (CompressedList-class), 8
reverse, 52, 78
reverse, character-method (reverse), 78
<table>
<thead>
<tr>
<th>Index Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SplitDataFrameList</td>
<td>15</td>
</tr>
<tr>
<td>SplitDataFrameList-class</td>
<td>15</td>
</tr>
<tr>
<td>splitRanges (Rle-class-leftovers)</td>
<td>79</td>
</tr>
<tr>
<td>splitRanges, Rle-method</td>
<td>79</td>
</tr>
<tr>
<td>splitRanges, vectorORfactor-method</td>
<td>79</td>
</tr>
<tr>
<td>stack, 14, 50, 87</td>
<td></td>
</tr>
<tr>
<td>stack, DataFrameList-method</td>
<td>15</td>
</tr>
<tr>
<td>stack, List-method</td>
<td>50</td>
</tr>
<tr>
<td>stack, RangedDataList-method</td>
<td>64</td>
</tr>
<tr>
<td>start, 26</td>
<td></td>
</tr>
<tr>
<td>start, CompressedIRangesList-method</td>
<td>49</td>
</tr>
<tr>
<td>start, GappedRanges-method</td>
<td>24</td>
</tr>
<tr>
<td>start, IRanges-method</td>
<td>43</td>
</tr>
<tr>
<td>start, NCLlist-method</td>
<td>53</td>
</tr>
<tr>
<td>start, NCLLists-method</td>
<td>53</td>
</tr>
<tr>
<td>start, PartitioningByEnd-method</td>
<td>24</td>
</tr>
<tr>
<td>start, PartitioningByWidth-method</td>
<td>24</td>
</tr>
<tr>
<td>start, RangedData-method</td>
<td>59</td>
</tr>
<tr>
<td>start, Ranges-method</td>
<td>65</td>
</tr>
<tr>
<td>start, RangesList-method</td>
<td>73</td>
</tr>
<tr>
<td>start, SimpleViewsList-method</td>
<td>91</td>
</tr>
<tr>
<td>start&lt;-, (Ranges-class)</td>
<td>65</td>
</tr>
<tr>
<td>start&lt;-, IRanges-method</td>
<td>43</td>
</tr>
<tr>
<td>start&lt;-, RangedData-method</td>
<td>59</td>
</tr>
<tr>
<td>start&lt;-, RangesList-method</td>
<td>73</td>
</tr>
<tr>
<td>start&lt;-, Views-method</td>
<td>90</td>
</tr>
<tr>
<td>sub, ANY, ANY, CompressedCharacterList-method</td>
<td>5</td>
</tr>
<tr>
<td>sub, ANY, ANY, CompressedRleList-method</td>
<td>5</td>
</tr>
<tr>
<td>sub, ANY, ANY, SimpleCharacterList-method</td>
<td>5</td>
</tr>
<tr>
<td>sub, ANY, ANY, SimpleRleList-method</td>
<td>5</td>
</tr>
<tr>
<td>subject (Views-class)</td>
<td>90</td>
</tr>
<tr>
<td>subject, SimpleRleViewsList-method</td>
<td></td>
</tr>
<tr>
<td>subject, SimpleRleViewsList-method</td>
<td></td>
</tr>
<tr>
<td>subject, Views-method</td>
<td>90</td>
</tr>
<tr>
<td>subset, 64</td>
<td></td>
</tr>
<tr>
<td>subset, Vector-method</td>
<td></td>
</tr>
<tr>
<td>subsetByOverlaps</td>
<td></td>
</tr>
<tr>
<td>subsetByOverlaps, RangedData, RangedData-method</td>
<td></td>
</tr>
<tr>
<td>subsetByOverlaps, RangedData, RangesList-method</td>
<td></td>
</tr>
<tr>
<td>subsetByOverlaps, RangesList, RangedData-method</td>
<td></td>
</tr>
<tr>
<td>subsetByOverlaps, Vector, Vector-method</td>
<td></td>
</tr>
<tr>
<td>subviews (Views-class)</td>
<td>90</td>
</tr>
<tr>
<td>subviews, Views-method</td>
<td>90</td>
</tr>
<tr>
<td>successiveIRanges</td>
<td>28</td>
</tr>
<tr>
<td>successiveIRanges, IRanges-utils</td>
<td>47</td>
</tr>
<tr>
<td>successiveViews</td>
<td>48</td>
</tr>
<tr>
<td>sum, CompressedIntegerList-method</td>
<td>5</td>
</tr>
<tr>
<td>sum, CompressedLogicalList-method</td>
<td>5</td>
</tr>
<tr>
<td>sum, CompressedNumericList-method</td>
<td>5</td>
</tr>
<tr>
<td>sum, Views-method</td>
<td></td>
</tr>
<tr>
<td>Summary, AtomicList-method</td>
<td>5</td>
</tr>
<tr>
<td>Summary, CompressedIRangesList-method</td>
<td>5</td>
</tr>
<tr>
<td>Summary, CompressedRleList-method</td>
<td>5</td>
</tr>
<tr>
<td>Summary, Views-method</td>
<td></td>
</tr>
<tr>
<td>table, AtomicList-method</td>
<td>3</td>
</tr>
<tr>
<td>tapply, 87</td>
<td></td>
</tr>
<tr>
<td>tapply, ANY, Vector-method</td>
<td></td>
</tr>
<tr>
<td>threebands (intra-range-methods)</td>
<td>37</td>
</tr>
<tr>
<td>threebands, IRanges-method</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

tile (Ranges-class), 65
tile, Ranges-method (Ranges-class), 65
to, CompressedHitsList-method (CompressedHitsList-class), 7
togroup (Grouping-class), 24
togroup, ANY-method (Grouping-class), 24
togroup, H2LGrouping-method (Grouping-class), 24
togroup, ManyToOneGrouping-method (Grouping-class), 24
togroup, Partitioning-method (Grouping-class), 24
togrouplength (Grouping-class), 24
togrouplength, ManyToOneGrouping-method (Grouping-class), 24
togrouprank (Grouping-class), 24
tolower, CompressedCharacterList-method (AtomicList-utils), 5
tolower, CompressedRleList-method (AtomicList-utils), 5
tolower, SimpleCharacterList-method (AtomicList-utils), 5
tolower, SimpleRleList-method (AtomicList-utils), 5
toupper, CompressedCharacterList-method (AtomicList-utils), 5
toupper, CompressedRleList-method (AtomicList-utils), 5
toupper, SimpleCharacterList-method (AtomicList-utils), 5
toupper, SimpleRleList-method (AtomicList-utils), 5
transform, 16
transform, SplitDataFrameList-method (DataFrameList-class), 15
trim (Views-class), 90
trim, Views-method (Views-class), 90
union (setops-methods), 82
union, CompressedIRangesList, CompressedIRangesList-method (setops-methods), 82
union, Pairs, missing-method (setops-methods), 82
union, Ranges, Ranges-method (setops-methods), 82
union, RangesList, RangesList-method (setops-methods), 82
unique, 71
unique, CompressedList-method (AtomicList), 3
unique, RleList-method (AtomicList), 3
universe (RangesList-class), 73
universe, RangedData-method (RangedData-class), 59
universe, Ranges-method (Ranges-class), 73
universe, ViewsList-method (ViewsList-class), 91
universe<- (RangesList-class), 73
universe<-, RangedData-method (RangedData-class), 59
universe<-, RangesList-method (Ranges-class), 73
universe<-, ViewsList-method (ViewsList-class), 91
unlist, 19
unlist, CompressedList-method (CompressedList-class), 8
unlist, IRangesList-method (IRangesList-class), 49
unlist, RangedDataList-method (RangedDataList-class), 64
unlist, Ranges-method (Ranges-class), 65
unlist, SimpleFactorList-method (AtomicList), 3
unlist, SimpleNormalIRangesList-method (IRangesList-class), 49
unlist, SimpleRleList-method (AtomicList), 3
unsplit, 19
unsplit, List-method (seqapply), 82
unstrsplit, 6
unstrsplit, CharacterList-method (AtomicList-utils), 5
unstrsplit, RleList-method (AtomicList-utils), 5
update, IRanges-method (IRanges-class), 43
update, Ranges-method (Ranges-class), 65
values, RangedData-method (RangedData-class), 59
var, AtomicList, AtomicList-method (AtomicList-utils), 5
var, AtomicList, missing-method (AtomicList-utils), 5
Vector, 15, 17, 19, 82, 86, 87, 90, 91
Vector-class, 91
Vector-class-leftovers, 86
Vector-comparison, 71
view-summarization-methods, 80, 81, 86, 88, 89
viewApply (view-summarization-methods), 88
viewApply, RleViews-method (view-summarization-methods), 88
viewApply, RleViewsList-method (view-summarization-methods), 88
viewApply, Views-method (view-summarization-methods), 88
viewMaxs (view-summarization-methods), 88
viewMaxs, RleViews-method (view-summarization-methods), 88
viewMaxs, RleViewsList-method (view-summarization-methods), 88
viewMeans (view-summarization-methods), 88
viewMeans, RleViews-method (view-summarization-methods), 88
viewMeans, RleViewsList-method (view-summarization-methods), 88
viewMins (view-summarization-methods), 88
viewMins, RleViews-method (view-summarization-methods), 88
viewMins, RleViewsList-method (view-summarization-methods), 88
viewRangeMaxs (view-summarization-methods), 88
viewRangeMaxs, RleViews-method (view-summarization-methods), 88
viewRangeMaxs, RleViewsList-method (view-summarization-methods), 88
viewRangeMins (view-summarization-methods), 88
viewRangeMins, RleViews-method (view-summarization-methods), 88
viewRangeMins, RleViewsList-method (view-summarization-methods), 88
viewSums (view-summarization-methods), 88
viewSums, RleViews-method (view-summarization-methods), 88
viewSums, RleViewsList-method (view-summarization-methods), 88
viewWhichMaxs (view-summarization-methods), 88
viewWhichMaxs, RleViews-method (view-summarization-methods), 88
viewWhichMaxs, RleViewsList-method (view-summarization-methods), 88
viewWhichMins (view-summarization-methods), 88
viewWhichMins, RleViews-method (view-summarization-methods), 88
viewWhichMins, RleViewsList-method (view-summarization-methods), 88
vmembers (Grouping-class), 24
vmembers, H2LGrouping-method (Grouping-class), 24
vmembers, ManyToOneGrouping-method (Grouping-class), 24
which, CompressedLogicalList-method (AtomicList-utils), 5
which, CompressedRleList-method (AtomicList-utils), 5
which, SimpleLogicalList-method (AtomicList-utils), 5
which, SimpleRleList-method (AtomicList-utils), 5
which.max, CompressedRleList-method (AtomicList-utils), 5