Package 'IRanges'

November 1, 2025

Title Foundation of integer range manipulation in Bioconductor

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.

biocViews Infrastructure, DataRepresentation

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

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

Version 2.44.0

License Artistic-2.0

Encoding UTF-8

Depends R (>= 4.0.0), methods, utils, stats, BiocGenerics (>= 0.53.2), S4Vectors (>= 0.47.6)

Imports stats4

LinkingTo S4Vectors

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

Collate thread-control.R range-squeezers.R Vector-class-leftovers.R DataFrameList-class.R DataFrameList-utils.R AtomicList-class.R AtomicList-utils.R Ranges-and-RangesList-classes.R IPosRanges-class.R IPosRanges-comparison.R IntegerRangesList-class.R IRanges-class.R IRanges-constructor.R makeIRangesFromDataFrame.R IRanges-utils.R Rle-class-leftovers.R IPos-class.R subsetting-utils.R Grouping-class.R Views-class.R RleViews-class.R RleViews-summarization.R extractList.R seqapply.R multisplit.R SimpleGrouping-class.R IRangesList-class.R IPosList-class.R ViewsList-class.R RleViewsList-class.R RleViewsList-utils.R RangedSelection-class.R MaskCollection-class.R read.Mask.R CompressedList-class.R CompressedList-comparison.R CompressedHitsList-class.R CompressedDataFrameList-class.R CompressedAtomicList-class.R

CompressedAtomicList-summarization.R CompressedGrouping-class.R
CompressedRangesList-class.R Hits-class-leftovers.R
NCList-class.R findOverlaps-methods.R windows-methods.R
intra-range-methods.R inter-range-methods.R reverse-methods.R
coverage-methods.R cvg-methods.R slice-methods.R
setops-methods.R nearest-methods.R cbind-Rle-methods.R
tile-methods.R extractListFragments.R zzz.R

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

git_branch RELEASE_3_22

git_last_commit 964a290

git_last_commit_date 2025-10-29

Repository Bioconductor 3.22

Date/Publication 2025-10-31

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

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

Contents

`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, "SimpleNormalIRang`
Creates a CompressedNormalIRangesList/SimpleNormalIRangesList instance from a LogicalList or logical RleList instance.

`as(from, "CharacterList")`, `as(from, "ComplexList")`, `as(from, "IntegerList")`, `as(from, "LogicalList")`,
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 `IntegerRangesList`.

Author(s)

P. Aboyoun

See Also

- AtomicList_utils for common operations on AtomicList objects.
- AtomicList_summarization for operations from the `Summary` group generic and other summarization operations on AtomicList objects.
- List objects in the **S4Vectors** package for the parent class.

Examples

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

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

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

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

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

## concatenating
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-summarization`

AtomicList summarization methods

Description

Operations from the `Summary` group generic and other summarization operations on AtomicList objects.

Details

The following summarization operations are supported on AtomicList objects:

- `any`, `all`, `min`, `max`, `range`, `sum`, `prod`
- `mean`, `var`, `cov`, `cor`, `sd`, `median`, `quantile`, `mad`, `IQR`
- `which.max`, `which.min`

See S4groupGeneric for more details.

Note that 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).

See Also

- AtomicList_utils for common operations on AtomicList objects.
- AtomicList objects.

Examples

```
x1 <- c(1L, 2L, 3L, 5L, 2L, 8L)
x2 <- c(15L, -5L, 80L, -1L, 15L, 79L, 20L, 5L)
il <- IntegerList(one=x1, x2)

sum(il)        # equivalent to 'sapply(il, sum)'
mean(il)       # equivalent to 'sapply(il, mean)'
range(il)      # equivalent to 'sapply(il, range)'
which.max(il)  # equivalent to 'sapply(il, which.max)'
```

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"

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

`Math2` "round", "signif"

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

See S4groupGeneric for more details.

See AtomicList_summarization for operations from the `Summary` group generic and other summarization operations on AtomicList objects.

Other Methods

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`

Numeric `diff`, `pmax`, `pmax.int`, `pmin`, `pmin.int`

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

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

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

Since `ifelse` relies on non-standard evaluation for arguments that need to be in the generic signature, we provide `ifelse2`, which has eager but otherwise equivalent semantics.

Specialized Methods

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

Author(s)

P. Aboyoun

See Also

- AtomicList_summarization for operations from the `Summary` group generic and other summarization operations on AtomicList objects.
- AtomicList objects.

Examples

```
## 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
```

`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.

Author(s)

Michael Lawrence

See Also

HitsList objects.

`CompressedList-class` *CompressedList 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 derivatives provide the `compress` argument that lets the user choose between the CompressedList and SimpleList representations at construction time. For example, depending on whether the `compress` argument of the `CharacterList()` constructor is set to `TRUE` or `FALSE`, a CompressedCharacterList or SimpleCharacterList instance will be returned.

Finally let's mention that the most efficient way to construct a CompressedList derivative is with

```
relist(unlisted, partitioning)
```

where `unlisted` is a vector-like object and `partitioning` a PartitioningByEnd object describing a partitioning of `unlisted`. The cost of this relist operation is virtually zero because `unlisted` and `partitioning` get stored *as-is* in the returned object.

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

- List in the **S4Vectors** package for an introduction to List objects and their derivatives (CompressedList is a direct subclass of List which makes CompressedList objects List derivatives).
- The SimpleList class defined and documented in the **S4Vectors** package for an alternative to CompressedList.
- relist and extractList for efficiently constructing a List derivative from a vector-like object.
- The CompressedNumericList class for an example of a concrete CompressedList subclass.
- PartitioningByEnd objects. These objects are used inside CompressedList derivatives to keep track of the *partitioning* of the single vector-like object made of all the list elements concatenated together.

Examples

```
## Fastest way to construct a CompressedList object:
unlisted <- runif(12)
partitioning <- PartitioningByEnd(c(5, 5, 10, 12), names=LETTERS[1:4])
partitioning

x1 <- relist(unlisted, partitioning)
x1

stopifnot(identical(lengths(partitioning), lengths(x1)))

## Note that the class of the CompressedList derivative returned by
## relist() is determined by relistToClass():
relistToClass(unlisted)
stopifnot(relistToClass(unlisted) == class(x1))

## Displaying a CompressedList object:
x2 <- IntegerList(11:12, integer(0), 3:-2, compress=TRUE)
class(x2)

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

## This is controlled by internal helper classNameForDisplay():
classNameForDisplay(x2)
classNameForDisplay(x1)
```

---

coverage-methods *Coverage of a set of ranges*

Description

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

Usage

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

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

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

Arguments

x	A IntegerRanges, Views, or IntegerRangesList object. See ?`coverage-methods` in the **GenomicRanges** package for coverage methods for other objects.

`shift`, `weight` `shift` specifies how much each range in `x` should be shifted before the coverage is computed. A positive shift value will shift the corresponding range in `x` to the right, and a negative value to the left. NAs are not allowed.

`weight` assigns a weight to each range in `x`.

- If `x` is an IntegerRanges or Views object: each of these arguments must be an integer or numeric vector parallel to `x` (will get recycled if necessary). Alternatively, each of these arguments 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 `shift` (or `weight`) vector. Note that when `x` is an IPos object, each of these arguments can only be a single number.
- If `x` is an IntegerRangesList object: each of these arguments 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]]` (or `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).

`width` Specifies the length of the returned coverage vector(s).

- If `x` is an IntegerRanges 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 IntegerRanges object, except that, if `width` is `NULL` then it's treated as if it was `length(subject(x))`.
- If `x` is a IntegerRangesList 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 IntegerRanges object).

`method` If `method` is set to `"sort"`, then `x` is sorted previous to the calculation of the coverage. If `method` is set to `"hash"` or `"naive"`, 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"`.

The `"naive"` method is a slower version of the `"hash"` method that has the advantage of avoiding floating point artefacts in the no-coverage regions of the numeric-Rle object returned by `coverage()` when the weights are supplied as a numeric vector of type `double`. See "FLOATING POINT ARITHMETIC CAN BRING A SURPRISE" section in the Examples below for more information.

Using `method="auto"` selects between the `"sort"` and `"hash"` methods, picking the one that is predicted to be faster based on `length(x)` and `width`.

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

Value

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

If `x` is a IntegerRangesList 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`.
- IntegerRanges, IPos, IntegerRangesList, Rle, and RleList objects.

Examples

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

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

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

## ---------------------------------------------------------------------
## B. FLOATING POINT ARITHMETIC CAN BRING A SURPRISE
## ---------------------------------------------------------------------
## Please be aware that rounding errors in floating point arithmetic can
## lead to some surprising results when computing a weighted coverage:
y <- IRanges(c(4, 10), c(18, 15))
w1 <- 0.958
w2 <- 1e4
cvg <- coverage(y, width=100, weight=c(w1, w2))
cvg  # non-zero coverage at positions 19 to 100!
```

```
## This is an artefact of floating point arithmetic and the algorithm
## used to compute the weighted coverage. It can be observed with basic
## floating point arithmetic:
w1 + w2 - w2 - w1  # very small non-zero value!

## Note that this only happens with the "sort" and "hash" methods but
## not with the "naive" method:
coverage(y, width=100, weight=c(w1, w2), method="sort")
coverage(y, width=100, weight=c(w1, w2), method="hash")
coverage(y, width=100, weight=c(w1, w2), method="naive")

## These very small non-zero coverage values in the no-coverage regions
## of the numeric-Rle object returned by coverage() are not always
## present. But when they are, they can cause problems downstream or
## in unit tests. For example downstream code that relies on things
## like 'cvg != 0' to find regions with coverage won't work properly.
## This can be mitigated either by selecting the "naive" method (be aware
## that this can slow down things significantly) or by "cleaning" 'cvg'
## first e.g. with something like 'cvg <- round(cvg, digits)' where
## 'digits' is a carefully chosen number of digits:
cvg <- round(cvg, digits=3)

## Note that this rounding will also have the interesting side effect of
## reducing the memory footprint of the Rle object in general (because
## some runs might get merged into a single run as a consequence of the
## rounding).

## ---------------------------------------------------------------------
## C. COVERAGE OF AN IPos OBJECT
## ---------------------------------------------------------------------
pos_runs <- IRanges(c(1, 5, 9), c(10, 8, 15))
ipos <- IPos(pos_runs)
coverage(ipos)

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

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

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

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

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

```
## ---------------------------------------------------------------------
## E. 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 IntegerRanges
## 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 that 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))

## ---------------------------------------------------------------------
## F. SOME SANITY CHECKS
## ---------------------------------------------------------------------
dummy_coverage <- function(x, shift=0L, width=NULL)
{
    y <- IRanges:::unlist_as_integer(shift(x, shift))
    if (is.null(width))
        width <- max(c(0L, y))
    Rle(tabulate(y,  nbins=width))
}

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

```
check_real_vs_dummy(x, shift=-max(end(x)))

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

DataFrameList-class *List of DataFrames*

Description

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

Accessors

In the following code snippets, x is a DataFrameList.

dims(x): Get the two-column matrix indicating the number of rows and columns over the entire dataset.

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

In the following code snippets, x is a SplitDataFrameList.

commonColnames(x): Get the character vector of column names present in the individual DataFrames in x.

commonColnames(x) <- value: Set the column names of the DataFrames in x.

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 IntegerRangesList subclass.

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

Constructor

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

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

Combining

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

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

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

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, with contributions from Aaron Lun

See Also

`DataFrame`

Examples

```
# Making a DataFrameList, which has different columns.
out <- DataFrameList(DataFrame(X=1, Y=2), DataFrame(A=1:2, B=3:4))
out[[1]]

# A more interesting SplitDataFrameList, which is guaranteed
```

```
# to have the same columns.
out <- SplitDataFrameList(DataFrame(X=1, Y=2), DataFrame(X=1:2, Y=3:4))
out[[1]]
out[,"X"]
out[,"Y"]

commonColnames(out)
commonColnames(out) <- c("x", "y")
out[[1]]

# We can also create these split objects using various split() functions:
out <- splitAsList(DataFrame(X=runif(100), Y=rpois(100, 5)),
    sample(letters, 100, replace=TRUE))
out[['a']]
```

`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.

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

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

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

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

## 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.
i	A list-like object. Unlike for skeleton, the content here matters (see Details section below). Note that i can be a IntegerRanges 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.

Details

Like split, relist and extractList have in common that they return a list-like object where all the list elements have the same class as the original vector-like object.

Methods that return a List derivative return an object of class relistToClass(x).

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 IntegerRanges object.

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

Note that the transformation performed by relist or split is always reversible (via unlist and unsplit, respectively), but 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.

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.

Author(s)

Hervé Pagès

See Also

- The `relistToClass` function and `split` methods defined in the **S4Vectors** package.
- The `unlist` and `relist` functions in the **base** and **utils** packages, respectively.
- The `split` and `unsplit` functions in the **base** package.
- PartitioningByEnd objects. These objects are used inside CompressedList derivatives to keep track of the *partitioning* of the single vector-like object made of all the list elements concatenated together.
- Vector, List, Rle, and DataFrame objects implemented in the **S4Vectors** package.
- IntegerRanges objects.

Examples

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

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

---

extractListFragments *Extract list fragments from a list-like object*

Description

Utilities for extracting *list fragments* from a list-like object.

Usage

```
extractListFragments(x, aranges, use.mcols=FALSE,
                     msg.if.incompatible=INCOMPATIBLE_ARANGES_MSG)

equisplit(x, nchunk, chunksize, use.mcols=FALSE)
```

Arguments

`x`	The list-like object from which to extract the list fragments. Can be any List derivative for `extractListFragments`. Can also be an ordinary list if `extractListFragments` is called with `use.mcols=TRUE`. Can be any List derivative that supports `relist()` for `equisplit`.
`aranges`	An IntegerRanges derivative containing the *absolute ranges* (i.e. the ranges *along* `unlist(x)`) of the list fragments to extract. The ranges in `aranges` must be compatible with the *cumulated length* of all the list elements in `x`, that is, `start(aranges)` and `end(aranges)` must be >= 1 and <= `sum(elementNROWS(x))`, respectively.

Also please note that only IntegerRanges objects that are disjoint and sorted are supported at the moment.

`use.mcols` Whether to propagate the metadata columns on `x` (if any) or not.

Must be `TRUE` or `FALSE` (the default). If set to `FALSE`, instead of having the metadata columns propagated from `x`, the object returned by `extractListFragments` has metadata columns `revmap` and `revmap2`, and the object returned by `equisplit` has metadata column `revmap`. Note that this is the default.

`msg.if.incompatible`

The error message to use if `aranges` is not compatible with the *cumulated length* of all the list elements in `x`.

`nchunk` The number of chunks. Must be a single positive integer.

`chunksize` The size of the chunks (last chunk might be smaller). Must be a single positive integer.

Details

A *list fragment* of list-like object `x` is a window in one of its list elements.

`extractListFragments` is a low-level utility that extracts list fragments from list-like object `x` according to the absolute ranges in `aranges`.

`equisplit` fragments and splits list-like object `x` into a specified number of partitions with equal (total) width. This is useful for instance to ensure balanced loading of workers in parallel evaluation. For example, if `x` is a GRanges object, each partition is also a GRanges object and the set of all partitions is returned as a GRangesList object.

Value

An object of the same class as `x` for `extractListFragments`.

An object of class `relistToClass(x)` for `equisplit`.

Author(s)

Hervé Pagès

See Also

- IRanges and IRangesList objects.
- Partitioning objects.
- IntegerList objects.
- `breakInChunks` from breaking a vector-like object in chunks.
- GRanges and GRangesList objects defined in the **GenomicRanges** package.
- List objects defined in the **S4Vectors** package.
- intra-range-methods and inter-range-methods for *intra range* and *inter range* transformations.

Examples

```
## ---------------------------------------------------------------------
## A. extractListFragments()
## ---------------------------------------------------------------------

x <- IntegerList(a=101:109, b=5:-5)
```

```
x

aranges <- IRanges(start=c(2, 4, 8, 17, 17), end=c(3, 6, 14, 16, 19))
aranges
extractListFragments(x, aranges)

x2 <- IRanges(c(1, 101, 1001, 10001), width=c(10, 5, 0, 12),
              names=letters[1:4])
mcols(x2)$label <- LETTERS[1:4]
x2

aranges <- IRanges(start=13, end=20)
extractListFragments(x2, aranges)
extractListFragments(x2, aranges, use.mcols=TRUE)

aranges2 <- PartitioningByWidth(c(3, 9, 13, 0, 2))
extractListFragments(x2, aranges2)
extractListFragments(x2, aranges2, use.mcols=TRUE)

x2b <- as(x2, "IntegerList")
extractListFragments(x2b, aranges2)

x2c <- as.list(x2b)
extractListFragments(x2c, aranges2, use.mcols=TRUE)

## ---------------------------------------------------------------------
## B. equisplit()
## ---------------------------------------------------------------------

## equisplit() first calls breakInChunks() internally to create a
## PartitioningByWidth object that contains the absolute ranges of the
## chunks, then calls extractListFragments() on it 'x' to extract the
## fragments of 'x' that correspond to these absolute ranges. Finally
## the IRanges object returned by extractListFragments() is split into
## an IRangesList object where each list element corresponds to a chunk.
equisplit(x2, nchunk=2)
equisplit(x2, nchunk=2, use.mcols=TRUE)

equisplit(x2, chunksize=5)

library(GenomicRanges)
gr <- GRanges(c("chr1", "chr2"), IRanges(1, c(100, 1e5)))
equisplit(gr, nchunk=2)
equisplit(gr, nchunk=1000)

## ---------------------------------------------------------------------
## C. ADVANCED extractListFragments() EXAMPLES
## ---------------------------------------------------------------------

## === D1. Fragment list-like object into length 1 fragments ===

## First we construct a Partitioning object where all the partitions
## have a width of 1:
x2_cumlen <- nobj(PartitioningByWidth(x2))  # Equivalent to
                                            # length(unlist(x2)) except
                                            # that it doesn't unlist 'x2'
                                            # so is much more efficient.
```

```
aranges1 <- PartitioningByEnd(seq_len(x2_cumlen))
aranges1

## Then we use it to fragment 'x2':
extractListFragments(x2, aranges1)
extractListFragments(x2b, aranges1)
extractListFragments(x2c, aranges1, use.mcols=TRUE)

## === D2. Fragment a Partitioning object ===

partitioning2 <- PartitioningByEnd(x2b)  # same as PartitioningByEnd(x2)
extractListFragments(partitioning2, aranges2)

## Note that when the 1st arg is a Partitioning derivative, then
## swapping the 1st and 2nd elements in the call to extractListFragments()
## doesn't change the returned partitioning:
extractListFragments(aranges2, partitioning2)

## ---------------------------------------------------------------------
## D. SANITY CHECKS
## ---------------------------------------------------------------------

## If 'aranges' is 'PartitioningByEnd(x)' or 'PartitioningByWidth(x)'
## and 'x' has no zero-length list elements, then
## 'extractListFragments(x, aranges, use.mcols=TRUE)' is a no-op.
check_no_ops <- function(x) {
  aranges <- PartitioningByEnd(x)
  stopifnot(identical(
    extractListFragments(x, aranges, use.mcols=TRUE), x
  ))
  aranges <- PartitioningByWidth(x)
  stopifnot(identical(
    extractListFragments(x, aranges, use.mcols=TRUE), x
  ))
}

check_no_ops(x2[lengths(x2) != 0])
check_no_ops(x2b[lengths(x2b) != 0])
check_no_ops(x2c[lengths(x2c) != 0])
check_no_ops(gr)
```

`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 IntegerRanges and IntegerRangesList derivatives. See ?`findOverlaps,GenomicRanges,GenomicRanges-method` in the **GenomicRanges** package for methods that operate on GenomicRanges or GRangesList objects.

Usage

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

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

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

subsetByOverlaps(x, ranges, maxgap=-1L, minoverlap=0L,
                 type=c("any", "start", "end", "within", "equal"),
                 invert=FALSE,
                 ...)

overlapsRanges(query, subject, hits=NULL, ...)

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

mergeByOverlaps(query, subject, ...)

findOverlapPairs(query, subject, ...)
```

Arguments

`query`, `subject`, `x`, `ranges`

Each of them can be an IntegerRanges (e.g. IRanges, Views) or IntegerRangesList (e.g. IRangesList, ViewsList) derivative. In addition, if `subject` or `ranges` is an IntegerRanges object, `query` or `x` can be an integer vector to be converted to length-one ranges.

If `query` (or `x`) is an IntegerRangesList object, then `subject` (or `ranges`) must also be an IntegerRangesList object.

If both arguments are list-like objects with names, each list element from the 2nd argument is paired with the list element from the 1st argument with the matching name, if any. Otherwise, list 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`

A single integer >= -1.

If type is set to "any", maxgap is interpreted as the maximum *gap* that is allowed between 2 ranges for the ranges to be considered as overlapping. The *gap* between 2 ranges is the number of positions that separate them. The *gap* between 2 adjacent ranges is 0. By convention when one range has its start or end strictly inside the other (i.e. non-disjoint ranges), the *gap* is considered to be -1.

If type is set to anything else, maxgap has a special meaning that depends on the particular type. See type below for more information.

minoverlap

A single non-negative integer.

Only ranges with a minimum of minoverlap overlapping positions are considered to be overlapping.

When type is "any", at least one of maxgap and minoverlap must be set to its default value.

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. 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. If maxgap is set to -1 (the default), it's replaced internally by 0.

select

If query is an IntegerRanges derivative: 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 an IntegerRangesList derivative: 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 ranges in x that do *not* overlap ranges.

hits

The Hits or HitsList object returned by findOverlaps, or NULL. If NULL then hits is computed by calling findOverlaps(query, subject, ...) internally (the extra arguments passed to overlapsRanges are passed to findOverlaps).

...

Further arguments to be passed to specific methods. For example:

- drop: Supported only when query is an IntegerRangesList derivative. FALSE by default. See select argument above for the details.
- drop.self, drop.redundant: When subject is omitted, the drop.self and drop.redundant arguments (both FALSE by default) are allowed. See query and subject arguments above for the details.
- nthread: EXPERIMENTAL! Sets the maximum number of threads to use.

The implementation of findOverlaps and family supports multithreading on systems where OpenMP is available e.g. on most Linux systems. Because this feature is still experimental, only one thread is used by default at the moment. However, this can be changed by setting the nthread argument to the maximum number of threads that findOverlaps and family are allowed to use.
On systems where OpenMP is not available (e.g. on macOS), setting nthread has no effect.
Note that nthread is only supported when the input objects are IntegerRanges derivatives at the moment. Support for other input types will be added soon.

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 IntegerRanges 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 IntegerRangesList objects, it returns an IntegerList object.

overlapsAny finds the ranges in query that overlap any of the ranges in subject. For IntegerRanges derivatives, it returns a logical vector of length equal to the number of ranges in query. For IntegerRangesList derivatives, 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 x that has an overlap hit with a range in ranges using the specified findOverlaps parameters.

When hits is a Hits (or HitsList) object, overlapsRanges(query, subject, hits) returns a IntegerRanges (or IntegerRangesList) object of the *same shape* as hits 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 hits. *Same shape* means same length when hits is a Hits object, and same length and same elementNROWS when hits is a HitsList object.

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.

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.

Author(s)

Hervé Pagès and Michael Lawrence

References

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

See Also

- 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 NCList class and constructor.
- The IntegerRanges, Views, IntegerRangesList, and ViewsList classes.
- The IntegerList and LogicalList classes.

Examples

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

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

findOverlaps(query, subject)

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

## including adjacent ranges in the result
findOverlaps(query, subject, maxgap=0L)

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

## one IRanges object with itself
findOverlaps(query)

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

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

findOverlaps(query, subject, type="start")
findOverlaps(query, subject, type="start", maxgap=1L)
findOverlaps(query, subject, type="end", select="first")
```

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

## Using pairs to find intersection of overlapping ranges
hits <- findOverlaps(query, subject)
p <- Pairs(query, subject, hits=hits)
pintersect(p)

## Shortcut
p <- findOverlapPairs(query, subject)
pintersect(p)

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

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

## ---------------------------------------------------------------------
## overlapsRanges()
## ---------------------------------------------------------------------

## Extract the regions of intersection between the overlapping ranges:
overlapsRanges(query, subject, ov)

## ---------------------------------------------------------------------
## Using IntegerRangesList objects
## ---------------------------------------------------------------------

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

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

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

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

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

overlapsAny(qlist, slist, type="start")
overlapsAny(qlist, slist, type="end")
qlist
```

```
subsetByOverlaps(qlist, slist)
countOverlaps(qlist, slist)
```

`Grouping-class` *Grouping objects*

Description

We call *grouping* an arbitrary mapping from a collection of NO objects to a collection of NG groups, or, more formally, a bipartite graph between integer sets [1, NO] and [1, NG]. Objects mapped to a given group are said to belong to, or to be assigned to, or to be in that group. Additionally, the objects in each group are ordered. So for example the 2 following groupings are considered different:

```
Grouping 1: NG = 3, NO = 5
            group   objects
                1 : 4, 2
                2 :
                3 : 4

Grouping 2: NG = 3, NO = 5
            group   objects
                1 : 2, 4
                2 :
                3 : 4
```

There are no restriction on the mapping e.g. any object can be mapped to 0, 1, or more groups, and can be mapped twice to the same group. Also some or all the groups can be empty.

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 <= i <= NG).

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

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

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

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

2 ManyToOneGrouping concrete subclasses: H2LGrouping, Dups and SimpleManyToOneGrouping

[DOCUMENT ME]

Constructors:

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

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

`ManyToOneGrouping(..., compress=TRUE)`: Collect `...` into a `ManyToOneGrouping`. The arguments 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 arguments 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 IntegerRanges API (e.g. `start`, `end`, `width`, etc...) in addition to the Grouping API. See `?IntegerRanges` for a description of the IntegerRanges 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.

vector 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, IntegerRanges-class, IRanges-class, successiveIRanges, cumsum, diff

Examples

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

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

## The core Grouping API:
length(h2l)
nobj(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:
x <- c(2, 77, 4, 4, 7, 2, 8, 8, 4, 99)
x_high2low <- high2low(x)
x_high2low  # same length as 'x'
dups2 <- Dups(x_high2low)
dups2
togroup(dups2)
duplicated(dups2)
togrouplength(dups2)  # frequency for each element
table(x)

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

## The core Grouping API:
```

```
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)

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

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

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

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

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

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

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

`Hits-class-leftovers` *Examples of basic manipulation of Hits objects*

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.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 ManyToManyGrouping, 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))
```

`IntegerRanges-class` *IntegerRanges objects*

Description

The IntegerRanges *virtual* class is a general container for storing ranges on the space of integers.

Details

TODO

`IntegerRangesList-class`
IntegerRangesList objects

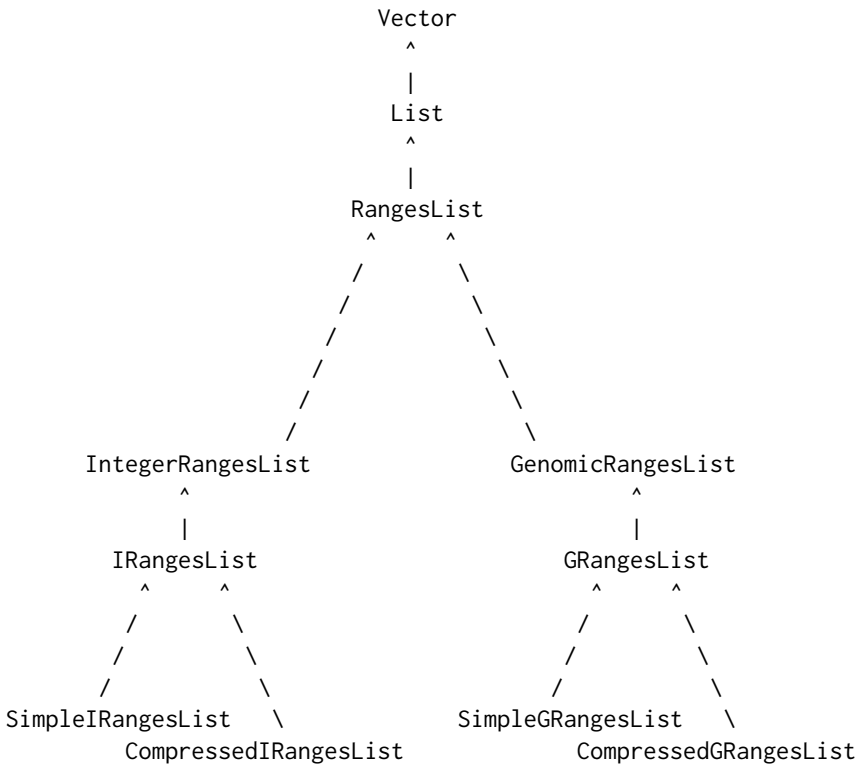
Description

The IntegerRangesList *virtual* class is a general container for storing a list of IntegerRanges objects.

Most users are probably more interested in the IRangesList container, an IntegerRangesList derivative for storing a list of IRanges objects.

Details

The place of IntegerRangesList in the *Vector class hierarchy*:



Note that the *Vector class hierarchy* has many more classes. In particular Vector, List, RangesList, and IntegerRangesList have other subclasses not shown here.

Accessors

In the code snippets below, x is a IntegerRangesList 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.

Coercion

In the code snippet below, x is an IntegerRangesList 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).

In the following code snippet, from is something other than a IntegerRangesList:

as(from, "IntegerRangesList"): When from is a IntegerRanges, analogous to as.list on a vector.

Arithmetic Operations

Any arithmetic operation, such as x + y, x * y, etc, where x is a IntegerRangesList, is performed identically on each element. Currently, IntegerRanges supports only the * operator, which zooms the ranges by a numeric factor.

Author(s)

M. Lawrence & H. Pagès

See Also

- IRangesList objects.
- IntegerRanges and IRanges objects.

Examples

```
## ---------------------------------------------------------------------
## 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 <- IRangesList(one = range1, two = range2)
length(named) # 2
start(named) # same as start(c(range1, range2))
names(named) # "one" and "two"
named[[1]] # range1
unnamed <- IRangesList(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(IRangesList(range1, range2))

# coerce to data.frame
as.data.frame(named)

IRangesList(range1, range2)

## zoom in 2X
collection <- IRangesList(one = range1, range2)
collection * 2
```

`inter-range-methods` *Inter range transformations of an IntegerRanges, Views, IntegerRangesList, or MaskCollection object*

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 'IntegerRanges'
range(x, ..., with.revmap=FALSE, na.rm=FALSE)

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

## reduce()
```

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

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

isDisjoint(x, ...)
disjointBins(x, ...)
```

Arguments

x	A IntegerRanges or IntegerRangesList object for range, disjoin, isDisjoint, and disjointBins. A IntegerRanges, Views, or IntegerRangesList object for reduce and gaps.
...	For range, additional IntegerRanges or IntegerRangesList 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 IntegerRanges 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 IntegerRangesList object: Integer vectors containing the coordinate bounds for each IntegerRangesList top-level element.

Details

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

reduce:

`reduce` first orders the ranges in `x` from left to right, then merges the overlapping or adjacent ones.

range:

`range` first concatenates `x` and the objects in `...` together. If the IRanges object resulting from this concatenation contains at least 1 range, then `range` returns an IRanges instance with a single range, from the minimum start to the maximum end of the concatenated object. Otherwise (i.e. if the concatenated object contains no range), `IRanges()` is returned (i.e. an IRanges instance of length 0).

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

gaps:

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

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

isDisjoint:

An IntegerRanges object `x` is considered to be "disjoint" if its ranges are non-overlapping. `isDisjoint` tests whether the object is "disjoint" or not.

Note that a "normal" IntegerRanges object is always "disjoint" but the opposite is not true. See `?isNormal` for more information about normal IntegerRanges objects.

About empty ranges. `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

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

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

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

and also equivalent to:

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

Finally, it is also equivalent to:

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

See ?`IPosRanges-comparison` for the meaning of the codes returned by the `pcompare` function.

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 an IntegerRanges 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 IntegerRangesList object:

- `range`, `reduce`, `gaps`, and `disjoin` return a IntegerRangesList 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 IntegerRanges, Views, IntegerRangesList, 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

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

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

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

```
}

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

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

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

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

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

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

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

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

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

## On an IntegerRanges 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 an IntegerRangesList object:
disjoin(collection)
disjoin(collection, with.revmap=TRUE)

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

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

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

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

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

## On an IntegerRangesList object:
isDisjoint(collection)

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

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

## On an IntegerRanges object:
disjointBins(IRanges(1, 5))  # 1L
disjointBins(IRanges(c(3, 1, 10), c(5, 12, 13)))  # c(2L, 1L, 2L)

## On an IntegerRangesList object:
disjointBins(collection)
```

---

intra-range-methods *Intra range transformations of an IRanges, IPos, Views, RangesList, or MaskCollection object*

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

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

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

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

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

promoters(x, upstream=2000, downstream=200, use.names=TRUE, ...)
terminators(x, upstream=2000, downstream=200, use.names=TRUE, ...)

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

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

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

Arguments

x	An IRanges, IPos, 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. Can also be a list-like object *parallel* to x if x is a RangesList object.
use.names	TRUE or FALSE. Should names be preserved?
start, end	If x is an IRanges, IPos 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. Can also be list-like objects *parallel* to x if x is a RangesList object.
width	If x is an IRanges, IPos 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. Can also be a list-like object *parallel* to x if x is a RangesList object.
fix	If x is an IRanges, IPos 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. Can also be a list-like object *parallel* to x if x is a RangesList object.
...	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.
upstream, downstream	Vectors of non-NA non-negative integers. Recycled as necessary so that each element corresponds to a range in x. Can also be list-like objects *parallel* to x if x is a RangesList object. 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 the upstream and downstream arguments of promoters() 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. Note that the same default values are used in terminators() at the moment. However this could be revisited if the case is made to use values that reflect more closely the biology of terminator regions in mammalian genomes.
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 list element in x.

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 IntegerRanges 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 and terminators:

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.

terminators is similar to promoters except that the generated ranges are relative to the transcription end sites (TES) returned by end(x).

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 IRanges, IPos, 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

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

## On an IRanges object:
ir1 <- successiveIRanges(c(19, 5, 0, 8, 5))
ir1
shift(ir1, shift=-3)

## On an IRangesList 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 an IRanges 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 an IRangesList 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 an IRanges object:
resize(ir2, 200)
resize(ir2, 2, fix="end")

## On an IRangesList object:
resize(collection, width=200)

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

## On an IRanges object:
ir3 <- IRanges(start=c(2,5,1), end=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 an IRangesList object:
flank(collection, width=10)
```

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

## On an IRanges object:
ir4 <- IRanges(start=10001:10004, end=12000)
promoters(ir4, upstream=800, downstream=0)
promoters(ir4, upstream=0, downstream=50)
promoters(ir4, upstream=800, downstream=50)

promoters(ir4, upstream=0, downstream=1)  # TSS

## On an IRangesList object:
promoters(collection, upstream=5, downstream=2)

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

## On an IRanges object:
bounds <- IRanges(start=c(0, 5, 3), end=c(10, 6, 9))
reflect(ir3, bounds)

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

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

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

## On an IRangesList object:
restrict(collection, start=2, end=8)
restrict(collection, start=2, end=8, keep.all.ranges=TRUE)

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

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

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

IPos-class *IPos objects*

Description

The IPos class is a container for storing a set of *integer positions*. It exists in 2 flavors: UnstitchedIPos and StitchedIPos. Each flavor uses a particular internal representation:

- In an UnstitchedIPos instance the positions are stored as an integer vector.
- In a StitchedIPos instance the positions are stored as an IRanges object where each range represents a run of *consecutive positions* (i.e. a run of positions that are adjacent and in *ascending order*). This storage is particularly memory-efficient when the vector of positions contains long runs of consecutive positions.

Because integer positions can be seen as integer ranges of width 1, the IPos class extends the IntegerRanges virtual class.

Usage

```
IPos(pos=integer(0), names=NULL, ..., stitch=NA)  # constructor function
```

Arguments

pos	An integer or numeric vector, or an IRanges object (or other IntegerRanges derivative). If pos is anything else, IPos() will first try to coerce it to an IRanges object with as(pos, "IRanges"). When pos is an IRanges object (or other IntegerRanges derivative), each range in it is interpreted as a run of consecutive positions.
names	A character vector or NULL.
...	Metadata columns to set on the IPos object. All the metadata columns must be vector-like objects of the same length as the object to construct.
stitch	TRUE, FALSE, or NA (the default). Controls which internal representation should be used: StitchedIPos (when stitch is TRUE) or UnstitchedIPos (when stitch is FALSE). When stitch is NA (the default), which internal representation will be used depends on the type of pos: UnstitchedIPos if pos is an integer or numeric vector, and StitchedIPos otherwise.

Details

Even though an IRanges object can be used for storing integer positions, using an IPos object is more efficient. In particular the memory footprint of an UnstitchedIPos object is half that of an IRanges object.

OTOH the memory footprint of a StitchedIPos object can vary a lot but will never be worse than that of an IRanges object. However it will reduce dramatically if the vector of positions contains long runs of consecutive positions. In the worst case scenario (i.e. when the object contains no consecutive positions) its memory footprint will be the same as that of an IRanges object.

Like for any Vector derivative, the length of an IPos object cannot exceed .Machine$integer.max (i.e. 2^31 on most platforms). IPos() will return an error if pos contains too many positions.

Value

An UnstitchedIPos or StitchedIPos object. If the input object pos is itself an IPos derivative, its metadata columns are propagated.

Accessors

Getters: IPos objects support the same set of getters as other IntegerRanges derivatives (i.e. `length()`, `start()`, `end()`, `names()`, `mcols()`, etc...), plus the `pos()` getter which is equivalent to `start()` and `end()`. See `?IntegerRanges` for the list of getters supported by IntegerRanges derivatives.

Setters: IPos derivatives support the `names()`, `mcols()` and `metadata()` setters only.

In particular there is no `pos()` setter for IPos derivatives at the moment (although one might be added in the future).

Coercion

From UnstitchedIPos to StitchedIPos and vice-versa: coercion back and forth between UnstitchedIPos and StitchedIPos is supported via `as(x, "StitchedIPos")` and `as(x, "UnstitchedIPos")`. This is the most efficient and recommended way to switch between the 2 internal representations. Note that this switch can have dramatic consequences on memory usage so is for advanced users only. End users should almost never need to do this switch when following a typical workflow.

From IntegerRanges to UnstitchedIPos, StitchedIPos, or IPos: An IntegerRanges derivative `x` in which all the ranges have a width of 1 can be coerced to an UnstitchedIPos or StitchedIPos object with `as(x, "UnstitchedIPos")` or `as(x, "StitchedIPos")`, respectively. For convenience `as(x, "IPos")` is supported and is equivalent to `as(x, "UnstitchedIPos")`.

From IPos to IRanges: An IPos derivative `x` can be coerced to an IRanges object with `as(x, "IRanges")`. However be aware that if `x` is a StitchedIPos instance, the memory footprint of the resulting object can be thousands times (or more) than that of `x`! See "MEMORY USAGE" in the Examples section below.

From IPos to ordinary R objects: Like with any other IntegerRanges derivative, `as.character()`, `as.factor()`, and `as.data.frame()` work on an IPos derivative `x`. Note however that `as.data.frame(x)` returns a data frame with a `pos` column (containing `pos(x)`) instead of the `start`, `end`, and `width` columns that one gets with other IntegerRanges derivatives.

Subsetting

An IPos derivative can be subsetted exactly like an IRanges object.

Concatenation

IPos derivatives can be concatenated with `c()` or `append()`. See `?c` in the **S4Vectors** package for more information about concatenating Vector derivatives.

Splitting and Relisting

Like with an IRanges object, `split()` and `relist()` work on an IPos derivative.

Author(s)

Hervé Pagès; based on ideas borrowed from Georg Stricker <georg.stricker@in.tum.de> and Julien Gagneur <gagneur@in.tum.de>

See Also

- The GPos class in the **GenomicRanges** package for representing a set of *genomic positions* (i.e. genomic ranges of width 1, a.k.a. *genomic loci*).
- The IRanges class for storing a set of *integer ranges* of arbitrary width.
- IPosRanges-comparison for comparing and ordering integer ranges and/or positions.
- findOverlaps-methods for finding overlapping integer ranges and/or positions.
- intra-range-methods and inter-range-methods for *intra range* and *inter range* transformations.
- coverage-methods for computing the coverage of a set of ranges and/or positions.
- nearest-methods for finding the nearest integer range/position neighbor.

Examples

```
showClass("IPos")  # shows the known subclasses

## ---------------------------------------------------------------------
## BASIC EXAMPLES
## ---------------------------------------------------------------------

## Example 1:
ipos1a <- IPos(c(44:53, 5:10, 2:5))
ipos1a  # unstitched

length(ipos1a)
pos(ipos1a)  # same as 'start(ipos1a)' and 'end(ipos1a)'
as.character(ipos1a)
as.data.frame(ipos1a)
as(ipos1a, "IRanges")
as.data.frame(as(ipos1a, "IRanges"))
ipos1a[9:17]

ipos1b <- IPos(c(44:53, 5:10, 2:5), stitch=TRUE)
ipos1b  # stitched

## 'ipos1a' and 'ipos1b' are semantically equivalent, only their
## internal representations differ:
all(ipos1a == ipos1b)

ipos1c <- IPos(c("44-53", "5-10", "2-5"))
ipos1c  # stitched

identical(ipos1b, ipos1c)

## Example 2:
my_pos <- IRanges(c(1, 6, 12, 17), c(5, 10, 16, 20))
ipos2 <- IPos(my_pos)
ipos2  # stitched

## Example 3:
ipos3A <- ipos3B <- IPos(c("1-15000", "15400-88700"))
npos <- length(ipos3A)

mcols(ipos3A)$sample <- Rle("sA")
sA_counts <- sample(10, npos, replace=TRUE)
mcols(ipos3A)$counts <- sA_counts
```

```
mcols(ipos3B)$sample <- Rle("sB")
sB_counts <- sample(10, npos, replace=TRUE)
mcols(ipos3B)$counts <- sB_counts

ipos3 <- c(ipos3A, ipos3B)
ipos3

## ---------------------------------------------------------------------
## MEMORY USAGE
## ---------------------------------------------------------------------

## Coercion to IRanges works on a StitchedIPos object...
ipos4 <- IPos(c("1-125000", "135000-575000"))
ir4 <- as(ipos4, "IRanges")
ir4
## ... but is generally not a good idea:
object.size(ipos4)
object.size(ir4)  # 1652 times bigger than the StitchedIPos object!

## Shuffling the order of the positions impacts memory usage:
ipos4r <- rev(ipos4)
object.size(ipos4r)
ipos4s <- sample(ipos4)
object.size(ipos4s)

## If one anticipates a lot of shuffling of the positions,
## then an UnstitchedIPos object should be used instead:
ipos4b <- as(ipos4, "UnstitchedIPos")
object.size(ipos4b)  # initial size is bigger than stitched version
object.size(rev(ipos4b))  # size didn't change
object.size(sample(ipos4b))  # size didn't change

## AN IMPORTANT NOTE: In the worst situations, IPos still performs
## as good as an IRanges object.
object.size(as(ipos4r, "IRanges"))  # same size as 'ipos4r'
object.size(as(ipos4s, "IRanges"))  # same size as 'ipos4s'

## Best case scenario is when the object is strictly sorted (i.e.
## positions are in strict ascending order).
## This can be checked with:
is.unsorted(ipos4, strict=TRUE)  # 'ipos4' is strictly sorted

## ---------------------------------------------------------------------
## USING MEMORY-EFFICIENT METADATA COLUMNS
## ---------------------------------------------------------------------
## In order to keep memory usage as low as possible, it is recommended
## to use a memory-efficient representation of the metadata columns that
## we want to set on the object. Rle's are particularly well suited for
## this, especially if the metadata columns contain long runs of
## identical values. This is the case for example if we want to use an
## IPos object to represent the coverage of sequencing reads along a
## chromosome.

## Example 5:
library(pasillaBamSubset)
library(Rsamtools)  # for the BamFile() constructor function
```

```
bamfile1 <- BamFile(untreated1_chr4())
bamfile2 <- BamFile(untreated3_chr4())
ipos5 <- IPos(IRanges(1, seqlengths(bamfile1)[["chr4"]]))
library(GenomicAlignments)  # for "coverage" method for BamFile objects
cvg1 <- coverage(bamfile1)$chr4
cvg2 <- coverage(bamfile2)$chr4
mcols(ipos5) <- DataFrame(cvg1, cvg2)
ipos5

object.size(ipos5)  # lightweight

## Keep only the positions where coverage is at least 10 in one of the
## 2 samples:
ipos5[mcols(ipos5)$cvg1 >= 10 | mcols(ipos5)$cvg2 >= 10]
```

`IPosRanges-class` *IPosRanges objects*

Description

The IPosRanges *virtual* class is a general container for storing a vector of ranges of integer positions.

Details

An IPosRanges object is a vector-like object where each element describes a "range of integer positions".

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 an IPosRanges object is the number of ranges in it, not the number of integer values in its ranges.

An IPosRanges object is considered empty iff all its ranges are empty.

IPosRanges 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 IPosRanges class itself is a virtual class. The following classes derive directly from it: IRanges, IPos, NCList, and GroupingRanges.

Methods

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

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

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

start(x): The start values of the ranges. This is an integer vector of the same length as x.

width(x): The number of integer values in each range. This is a vector of non-negative integers of the same length as x.

end(x): start(x) + width(x) - 1L

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

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

tile(x, n, width, ...): Splits each range in x into subranges as specified by n (number of ranges) or width. Only one of n or width can be specified. The return value is a IRangesList the same length as x. IPosRanges 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 IRangesList the same length as x. IPosRanges 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, ...): Convert x into a 2-column integer matrix containing start(x) and width(x). Extra arguments (...) are ignored.

as.data.frame(x, row.names=NULL, optional=FALSE): Convert x into a standard R data frame object. row.names must be NULL or a character vector giving the row names for the data frame, and optional is ignored. See ?as.data.frame for more information about these arguments.

x[[i]]: Return integer vector start(x[i]):end(x[i]) denoted by i. Subscript i can be a single integer or a character string.

x[i]: Return a new IPosRanges 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 IPosRanges 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, ..., ignore.mcols=FALSE): Concatenate IPosRanges object x and the IPosRanges objects in ... together. See ?c in the **S4Vectors** package for more information about concatenating Vector derivatives.

`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 display of IRanges, IPos, Hits, GRanges, GPos, GAlignments, XStringSet objects, and more...

Normality

An IPosRanges 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 IPosRanges 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 an IPosRanges 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* IPosRanges 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 NormalIRanges object is just an IRanges object that is guaranteed to be *normal*.

Here are some methods related to the notion of *normal* IPosRanges:

`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

- The IRanges class, a concrete IPosRanges direct subclass for storing a set of *integer ranges*.
- The IPos class, an IPosRanges direct subclass for representing a set of *integer positions* (i.e. *integer ranges* of width 1).
- IPosRanges-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 IntegerRanges derivatives.
- 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

```
## ---------------------------------------------------------------------
## 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))
ir*1 # no change
ir*c(1,2) # zoom second range by 2X
ir*-2 # zoom out 2X
```

IPosRanges-comparison *Comparing and ordering ranges*

Description

Methods for comparing and/or ordering the ranges in IPosRanges derivatives (e.g. IRanges, IPos, or NCList objects).

Usage

```
## match() & selfmatch()
## ---------------------

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

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

## order() and related methods
## ---------------------------

## S4 method for signature 'IPosRanges'
is.unsorted(x, na.rm=FALSE, strictly=FALSE)

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

## Generalized parallel comparison of 2 IPosRanges derivatives
## -----------------------------------------------------------

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

rangeComparisonCodeToLetter(code)
```

Arguments

x, table, y	IPosRanges derivatives e.g. IRanges, IPos, or NCList 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.
na.rm	Ignored.
strictly	Logical indicating if the check should be for *strictly* increasing values.
...	One or more IPosRanges derivatives. The 2nd and following objects are used to break ties.

na.last	Ignored.
decreasing	TRUE or FALSE.
code	A vector of codes as returned by pcompare.

Details

Two ranges of an IPosRanges derivative are considered equal iff they share the same start and width. duplicated() and unique() on an IPosRanges derivative are conforming to this.

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.

The "natural order" for the elements of an IPosRanges derivative is to order them (a) first by start and (b) then by width. This way, the space of integer ranges is totally ordered.

pcompare(), ==, !=, <=, >=, < and > on IPosRanges derivatives behave accordingly to this "natural order".

is.unsorted(), order(), sort(), rank() on IPosRanges derivatives also behave accordingly to this "natural order".

Finally, note that some *inter range transformations* like reduce or disjoin also use this "natural order" implicitly when operating on IPosRanges derivatives.

pcompare(x, y): Performs element-wise (aka "parallel") comparison of 2 IPosRanges 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]: ...ooooooo...       3 j: x[i]: .....oooo...
      y[i]: .....oooo...             y[i]: ...ooooooo...

-2 e: x[i]: ..oooooooo..        2 i: x[i]: ....oooo....
      y[i]: ....oooo....             y[i]: ..oooooooo..

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

                0 g: x[i]: ...ooooooo...
                     y[i]: ...ooooooo...
```

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 IPosRanges derivatives 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 an IPosRanges derivative. 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 an IPosRanges derivative. 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 (an IPosRanges derivative) into ascending order, breaking ties by further arguments (also IPosRanges derivatives).

`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 IPosRanges 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 range* and *inter range* transformations.
- setops-methods for set operations on IRanges objects.

Examples

```
## ---------------------------------------------------------------------
## A. ELEMENT-WISE (AKA "PARALLEL") COMPARISON OF 2 IPosRanges
##    DERIVATIVES
## ---------------------------------------------------------------------
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

which(width(x2) == 0)  # 3 empty ranges
x2[2] == x2[2]  # TRUE
x2[2] == x2[5]  # FALSE
x2 == x2[4]
x2 >= x2[3]

## ---------------------------------------------------------------------
## B. match(), selfmatch(), %in%, duplicated(), unique()
## ---------------------------------------------------------------------
table <- x2[c(2:4, 7:8)]
match(x2, table)

x2 %in% table

duplicated(x2)
unique(x2)

## ---------------------------------------------------------------------
## C. findMatches(), countMatches()
## ---------------------------------------------------------------------
```

```
findMatches(x2, table)
countMatches(x2, table)

x2_levels <- unique(x2)
countMatches(x2_levels, x2)

## --------------------------------------------------------------------
## D. order() AND RELATED METHODS
## --------------------------------------------------------------------
is.unsorted(x2)
order(x2)
sort(x2)
rank(x2, ties.method="first")
```

`IRanges internals` *IRanges internals*

Description

Objects, classes and methods defined in the **IRanges** package that are not intended to be used directly.

`IRanges-class` *IRanges and NormalIRanges objects*

Description

The IRanges class is a simple implementation of the IntegerRanges container where 2 integer vectors of the same length are used to store the start and width values. See the IntegerRanges virtual class for a formal definition of IntegerRanges 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 IntegerRanges objects for the definition and properties of "normal" IntegerRanges objects.

Constructor

See `?`IRanges-constructor``.

Coercion

`ranges(x, use.names=FALSE, use.mcols=FALSE)`**:** Squeeze the ranges out of IntegerRanges object `x` and return them in an IRanges object *parallel* to `x` (i.e. same length as `x`).

`as(from, "IRanges")`**:** Creates an IRanges instance from an IntegerRanges derivative, or from a logical 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.

Coercing a data.frame or DataFrame into an IRanges object is also supported. See `makeIRangesFromDataFrame` for the details.

`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.

Concatenation

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

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

- The GRanges class in the **GenomicRanges** package for storing a set of *genomic ranges*.
- The IPos class for representing a set of *integer positions* (i.e. *integer ranges* of width 1).
- IPosRanges-comparison for comparing and ordering integer ranges and/or positions.
- IRanges-utils for some utility functions for creating or modifying IRanges objects.
- findOverlaps-methods for finding overlapping integer ranges and/or positions.
- intra-range-methods and inter-range-methods for *intra range* and *inter range* transformations.
- coverage-methods for computing the coverage of a set of ranges and/or positions.
- setops-methods for set operations on IRanges objects.
- nearest-methods for finding the nearest integer range/position neighbor.

Examples

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

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

## Concatenating IRanges objects
ir1 <- IRanges(c(1, 10, 20), width=5)
mcols(ir1) <- DataFrame(score=runif(3))
ir2 <- IRanges(c(101, 110, 120), width=10)
mcols(ir2) <- DataFrame(score=runif(3))
```

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

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

IRanges-constructor *The IRanges constructor and supporting functions*

Description

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

solveUserSEW is a low-level utility function for solving a set of user-supplied start/end/width triplets.

Usage

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

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

Arguments

start, end, width

For IRanges: NULL or vector of integers.

For solveUserSEW: vector of integers (eventually with NAs).

`names` A character vector or `NULL`.

`...` Metadata columns to set on the IRanges object. All the metadata columns must be vector-like objects of the same length as the object to construct.

`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

`solveUserSEW(refwidths, start=NA, end=NA, width=NA, rep.refwidths=FALSE, translate.negative.coord=`

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

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

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

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

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

Author(s)

Hervé Pagès

See Also

- IRanges-class for the IRanges class.
- `narrow`

Examples

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

## With ranges specified as strings:
IRanges(c("11-20", "15-14", "-4--2"))

## 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, nchunk, chunksize)

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

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

Arguments

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

Details

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

breakInChunks returns a PartitioningByEnd object describing the "chunks" that result from breaking a vector-like object of length totalsize in the chunks described by nchunk or chunksize.

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

- IRanges objects.
- Partitioning objects.
- equisplit for splitting a list-like object into a specified number of partitions.
- intra-range-methods and inter-range-methods for intra range and inter range transformations.
- setops-methods for performing set operations on IRanges objects.
- solveUserSEW
- successiveViews

Examples

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

successiveIRanges(vec)

breakInChunks(600999, chunksize=50000)  # chunks of size 50000 (last
                                        # chunk is smaller)

whichAsIRanges(vec >= 5)

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

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

IRangesList-class *List of IRanges and NormalIRanges*

Description

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

Constructor

`IRangesList(..., 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

In the code snippets below, `from` is a *list-like* object.

`as(from, "SimpleIRangesList")`: Coerces `from`, to a `SimpleIRangesList`, requiring that all `IntegerRanges` elements are coerced to internal `IRanges` elements. This is a convenient way to ensure that all `IntegerRanges` 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 `IntegerRanges` elements are coerced to internal `IRanges` elements. This is a convenient way to ensure that all `IntegerRanges` 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 `IntegerRanges` elements are coerced to internal `NormalIRanges` elements.

`as(from, "CompressedNormalIRangesList")`: Coerces `from`, to a `CompressedNormalIRangesList`, requiring that all `IntegerRanges` elements are coerced to internal `NormalIRanges` elements.

In the code snippet below, `x` is an `IRangesList` object.

`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 and Hervé Pagès

See Also

- `IntegerRangesList`, the parent of this class, for more functionality.
- intra-range-methods and inter-range-methods for *intra range* and *inter range* transformations.
- setops-methods for set operations on IRangesList objects.

Examples

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

```
unnamed <- IRangesList(range1, range2)
names(unnamed) # NULL

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

---

makeIRangesFromDataFrame *Make a IRanges object from a data.frame or DataFrame*

Description

makeIRangesFromDataFrame takes a data-frame-like object as input and try to automatically find the columns that describe integer ranges. If successful, it returns them in an object.

The function is also the workhorse behind the coercion method from data.frame (or DataFrame) to IRanges.

Usage

```
makeIRangesFromDataFrame(df, keep.extra.columns=FALSE,
        start.field="start",
        end.field=c("end", "stop"),
        starts.in.df.are.0based=FALSE,
        na.rm=FALSE)
```

Arguments

df	A data.frame or DataFrame object. If not, then the function first tries to turn df into a data frame with as.data.frame(df).
keep.extra.columns	TRUE or FALSE (the default). If TRUE, then the columns in df that are not used to form the integer ranges of the returned IRanges are returned as metadata columns on the object. Otherwise, they are ignored. Note that if df has a width column, then makeIRangesFromDataFrame will always ignore it.
start.field	A character vector of recognized names for the column in df that contains the start values of the integer ranges. Only the first name in start.field that is found in colnames(df) is used. If no one is found, then an error is raised.
end.field	A character vector of recognized names for the column in df that contains the end values of the integer ranges. Only the first name in start.field that is found in colnames(df) is used. If no one is found, then an error is raised.
starts.in.df.are.0based	TRUE or FALSE (the default). If TRUE, then the start values of the integer ranges in df are considered to be *0-based* and are converted to *1-based* in the returned IRanges object. This feature is intended to make it more convenient to handle input that contains data obtained from resources using the "0-based start" convention. A notorious example of such resource is the UCSC Table Browser (http://genome.ucsc.edu/cgi-bin/hgTables).
na.rm	TRUE or FALSE (the default). If TRUE, then rows in the df with missing start or end values (i.e. the value is NA) are ignored. Otherwise, they trigger an error.

Value

An IRanges object.

If na.rm is set to FALSE (the default), then the returned object is guaranteed to have one element per row in the input. However, if na.rm is set to TRUE, then the length of the returned object can be less than nrow(df).

If df has non-automatic row names (i.e. rownames(df) is not NULL and is not seq_len(nrow(df))), then they will be used to set names on the returned IRanges object.

Note

Coercing a data.frame or DataFrame df to IRanges (with as(df, "IRanges")), or calling IRanges(df), are both equivalent to calling makeIRangesFromDataFrame(df, keep.extra.columns=TRUE).

Author(s)

H. Pagès

See Also

- makeGRangesFromDataFrame in the **GenomicRanges** package to make a GRanges object from a data.frame or DataFrame.
- IRanges objects.
- DataFrame objects in the **S4Vectors** package.

Examples

```
df <- data.frame(ID=letters[1:5],
                 locus_stART=11:15,
                 locus_STop=12:16,
                 score=1:5)

## makeIRangesFromDataFrame() tries hard to figure out what columns
## to use for the start and end values of the ranges.
makeIRangesFromDataFrame(df)

makeIRangesFromDataFrame(df, keep.extra.columns=TRUE)

as(df, "IRanges")  # equivalent to the above
```

MaskCollection-class *MaskCollection objects*

Description

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

Details

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

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

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

Basic accessor methods

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

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

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

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

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

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

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

Constructor

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

Other methods

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

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

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

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

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

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

Subsetting and appending

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

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

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

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

Other methods

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

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

Author(s)

Hervé Pagès

See Also

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

Examples

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

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

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

## Other advanced operations:
mymasks[[2]]
length(mymasks[[2]])
```

```
mymasks[[2]][-3]
append(mymasks[-2], gaps(mymasks[2]))
```

`multisplit` *Split elements belonging to multiple groups*

Description

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

Usage

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

Arguments

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

Value

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

Author(s)

Michael Lawrence

Examples

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

`NCList-class` *Nested Containment List objects*

Description

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

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

To preprocess a IntegerRanges or IntegerRangesList object, simply call the `NCList` or `NCLists` constructor function on it.

Usage

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

Arguments

x	The IntegerRanges or IntegerRangesList object to preprocess.
circle.length	Use only if the space (or spaces if x is a IntegerRangesList 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 IntegerRanges object (replacement for IntervalTree), NCLists for a IntegerRangesList object (replacement for IntervalForest), and GNCList for a GenomicRanges object (replacement for GIntervalTree). However, for a one-time use, it is NOT advised to explicitely 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 IntegerRanges 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 alogrithm, 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 mitochnodrial chromosome). See GNCList in the **GenomicRanges** package for some examples.

- 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

Alexander V. Alekseyenko and Christopher J. Lee – Nested Containment List (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases. Bioinformatics (2007) 23 (11): 1386-1393. doi: 10.1093/bioinformatics/btl647

See Also

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

Examples

```
## 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/position neighbor*

Description

The nearest(), precede(), follow(), distance() and distanceToNearest() methods for IntegerRanges derivatives (e.g. IRanges objects).

Usage

```
## S4 method for signature 'IntegerRanges,IntegerRanges_OR_missing'
nearest(x, subject, select=c("arbitrary", "all"))

## S4 method for signature 'IntegerRanges,IntegerRanges_OR_missing'
precede(x, subject, select=c("first", "all"))

## S4 method for signature 'IntegerRanges,IntegerRanges_OR_missing'
follow(x, subject, select=c("last", "all"))

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

## S4 method for signature 'IntegerRanges,IntegerRanges_OR_missing'
distanceToNearest(x, subject, select=c("arbitrary", "all"))
```

Arguments

x	The query IntegerRanges derivative, or (for distance()) a Pairs object containing both the query (first) and subject (second).
subject	The subject IntegerRanges object, 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 IntegerRanges derivative. Cannot be missing. If x and y are not the same length, the shortest will be recycled to match the length of the longest.
...	Additional arguments for methods

Details

nearest(x, subject, select=c("arbitrary", "all")): The conventional nearest neighbor finder. Returns an 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 how it proceeds, for a range xi in x:

Find the ranges in subject that minimize the distance to xi. If a single range si in subject achieves the shortest distance to xi, si is returned as the nearest neighbor of xi. If multiple ranges in subject achieve the shortest distance to xi, one of them is chosen arbitrarily.

See distance below for how the distance between two ranges is defined.

`precede(x, subject, select=c("first", "all"))`: 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(x, subject, select=c("last", "all"))`: 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`.

`distance(x, y)`: Returns the distance for each range in `x` to the range in `y`.

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

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

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

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

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

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

Instead we say

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

or

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

`distanceToNearest(x, subject, select=c("arbitrary", "all"))`: Returns the distance for each range in `x` to its nearest neighbor in `subject`.

`selectNearest(hits, x, subject)`: 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 object if `select="all"`.

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

For `distanceToNearest()`, a Hits object with a metadata column reporting the `distance` between the pair. Access the `distance` metadata column with the `mcols()` accessor.

For `selectNearest()`, a Hits object, sorted by query.

Author(s)

M. Lawrence

See Also

- Hits objects implemented in the **S4Vectors** package.
- `findOverlaps` for finding just the overlapping ranges.
- The IntegerRanges class.
- nearest-methods in the **GenomicRanges** package for the `nearest()`, `precede()`, `follow()`, `distance()`, and `distanceToNearest()` methods for GenomicRanges objects.

Examples

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

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

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

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

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

## ----------------------------------------
## distance()
## ----------------------------------------
## adjacent
distance(IRanges(1,5), IRanges(6,10)) # 0L
## overlap
distance(IRanges(1,5), IRanges(3,7))  # 0L
## zero-width
sapply(-3:3, function(i) distance(shift(IRanges(4,3), i), IRanges(4,3)))
```

`range-squeezers` *Squeeze the ranges out of a range-based object*

Description

S4 generic functions for squeezing the ranges out of a range-based object.

These are analog to range squeezers `granges` and `grglist` defined in the **GenomicRanges** package, except that `ranges` returns the ranges in an IRanges object (instead of a GRanges object for `granges`), and `rglist` returns them in an IRangesList object (instead of a GRangesList object for `grglist`).

Usage

```
ranges(x, use.names=TRUE, use.mcols=FALSE, ...)
rglist(x, use.names=TRUE, use.mcols=FALSE, ...)
```

Arguments

x	An object containing ranges e.g. a IntegerRanges, GenomicRanges, RangedSummarizedExperiment, GAlignments, GAlignmentPairs, or GAlignmentsList object, or a Pairs object containing ranges.
use.names	TRUE (the default) or FALSE. Whether or not the names on x (accessible with names(x)) should be propagated to the returned object.
use.mcols	TRUE or FALSE (the default). Whether or not the metadata columns on x (accessible with mcols(x)) should be propagated to the returned object.
...	Additional arguments, for use in specific methods.

Details

Various packages (e.g. **IRanges**, **GenomicRanges**, **SummarizedExperiment**, **GenomicAlignments**, etc...) define and document various range squeezing methods for various types of objects.

Note that these functions can be seen as *object getters* or as functions performing coercion.

For some objects (e.g. GAlignments and GAlignmentPairs objects defined in the **GenomicAlignments** package), `as(x, "IRanges")` and `as(x, "IRangesList")`, are equivalent to `ranges(x, use.names=TRUE, use.mcols=TRUE)` and `rglist(x, use.names=TRUE, use.mcols=TRUE)`, respectively.

Value

An IRanges object for `ranges`.

An IRangesList object for `rglist`.

If x is a vector-like object (e.g. GAlignments), the returned object is expected to be *parallel* to x, that is, the i-th element in the output corresponds to the i-th element in the input.

If use.names is TRUE, then the names on x (if any) are propagated to the returned object. If use.mcols is TRUE, then the metadata columns on x (if any) are propagated to the returned object.

Author(s)

H. Pagès

See Also

- IRanges and IRangesList objects.
- RangedSummarizedExperiment objects in the **SummarizedExperiment** packages.
- GAlignments, GAlignmentPairs, and GAlignmentsList objects in the **GenomicAlignments** package.

Examples

```
## See ?GAlignments in the GenomicAlignments package for examples of
## "ranges" and "rglist" methods.
```

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

Coercion

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

Accessors

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

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

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

Author(s)

Michael Lawrence

Examples

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

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

RangedSelection(rl, "score")
```

read.Mask *Read a mask from a file*

Description

read.agpMask and read.gapMask extract the AGAPS mask from an NCBI "agp" file or a UCSC "gap" file, respectively.

read.liftMask extracts the AGAPS mask from a UCSC "lift" file (i.e. a file containing offsets of contigs within sequences).

read.rmMask extracts the RM mask from a RepeatMasker .out file.

read.trfMask extracts the TRF mask from a Tandem Repeats Finder .bed file.

Usage

```
read.agpMask(file, seqname="?", mask.width=NA, gap.types=NULL, use.gap.types=FALSE)
read.gapMask(file, seqname="?", mask.width=NA, gap.types=NULL, use.gap.types=FALSE)
read.liftMask(file, seqname="?", mask.width=NA)
read.rmMask(file, seqname="?", mask.width=NA, use.IDs=FALSE)
read.trfMask(file, seqname="?", mask.width=NA)
```

Arguments

file	Either a character string naming a file or a connection open for reading.
seqname	The name of the sequence for which the mask must be extracted. If no sequence is specified (i.e. seqname="?") then an error is raised and the sequence names found in the file are displayed. If the file doesn't contain any information for the specified sequence, then a warning is issued and an empty mask of width mask.width is returned.
mask.width	The width of the mask to return i.e. the length of the sequence this mask will be put on. See ?`MaskCollection-class` for more information about the width of a MaskCollection object.
gap.types	NULL or a character vector containing gap types. Use this argument to filter the assembly gaps that are to be extracted from the "agp" or "gap" file based on their type. Most common gap types are "contig", "clone", "centromere", "telomere", "heterochromatin", "short_arm" and "fragment". With gap.types=NULL, all the assembly gaps described in the file are extracted. With gap.types="?", an error is raised and the gap types found in the file for the specified sequence are displayed.
use.gap.types	Whether or not the gap types provided in the "agp" or "gap" file should be used to name the ranges constituing the returned mask. See ?`IRanges-class` for more information about the names of an IRanges object.
use.IDs	Whether or not the repeat IDs provided in the RepeatMasker .out file should be used to name the ranges constituing the returned mask. See ?`IRanges-class` for more information about the names of an IRanges object.

See Also

MaskCollection-class, IRanges-class

Examples

```
## ---------------------------------------------------------------------
## A. Extract a mask of assembly gaps ("AGAPS" mask) with read.agpMask()
## ---------------------------------------------------------------------
## Note: The hs_b36v3_chrY.agp file was obtained by downloading,
## extracting and renaming the hs_ref_chrY.agp.gz file from
##
##   ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/Assembled_chromosomes/
##     hs_ref_chrY.agp.gz      5 KB  24/03/08  04:33:00 PM
##
## on May 9, 2008.

chrY_length <- 57772954
file1 <- system.file("extdata", "hs_b36v3_chrY.agp", package="IRanges")
mask1 <- read.agpMask(file1, seqname="chrY", mask.width=chrY_length,
                      use.gap.types=TRUE)
mask1
mask1[[1]]

mask11 <- read.agpMask(file1, seqname="chrY", mask.width=chrY_length,
                       gap.types=c("centromere", "heterochromatin"))
mask11[[1]]

## ---------------------------------------------------------------------
## B. Extract a mask of assembly gaps ("AGAPS" mask) with read.liftMask()
## ---------------------------------------------------------------------
## Note: The hg18liftAll.lft file was obtained by downloading,
## extracting and renaming the liftAll.zip file from
##
##   http://hgdownload.cse.ucsc.edu/goldenPath/hg18/bigZips/
##     liftAll.zip             03-Feb-2006 11:35  5.5K
##
## on May 8, 2008.

file2 <- system.file("extdata", "hg18liftAll.lft", package="IRanges")
mask2 <- read.liftMask(file2, seqname="chr1")
mask2
if (interactive()) {
    ## contigs 7 and 8 for chrY are adjacent
    read.liftMask(file2, seqname="chrY")

    ## displays the sequence names found in the file
    read.liftMask(file2)

    ## specify an unknown sequence name
    read.liftMask(file2, seqname="chrZ", mask.width=300)
}

## ---------------------------------------------------------------------
## C. Extract a RepeatMasker ("RM") or Tandem Repeats Finder ("TRF")
##    mask with read.rmMask() or read.trfMask()
## ---------------------------------------------------------------------
## Note: The ce2chrM.fa.out and ce2chrM.bed files were obtained by
## downloading, extracting and renaming the chromOut.zip and
## chromTrf.zip files from
##
```

```
##   http://hgdownload.cse.ucsc.edu/goldenPath/ce2/bigZips/
##     chromOut.zip            21-Apr-2004 09:05  2.6M
##     chromTrf.zip            21-Apr-2004 09:07  182K
##
## on May 7, 2008.

## Before you can extract a mask with read.rmMask() or read.trfMask(), you
## need to know the length of the sequence that 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
}
```

`reverse` *reverse*

Description

A generic function for reversing vector-like or list-like objects. This man page describes methods for reversing a character vector, a Views object, or a MaskCollection object. Note that `reverse` is similar to but not the same as `rev`.

Usage

```
reverse(x, ...)
```

Arguments

x	A vector-like or list-like object.
...	Additional arguments to be passed to or from methods.

Details

On a character vector or a Views object, `reverse` reverses each element individually, without modifying the top-level order of the elements. More precisely, each individual string of a character vector is reversed.

Value

An object of the same class and length as the original object.

See Also

- `rev` in base R.
- `revElements` in the **S4Vectors** package to reverse all or some of the list elements of a list-like object.
- Views objects.
- MaskCollection objects.
- reverse-methods in the **XVector** package.

Examples

```
## On a character vector:
reverse(c("Hi!", "How are you?"))
rev(c("Hi!", "How are you?"))

## On a Views object:
v <- successiveViews(Rle(c(-0.5, 12.3, 4.88), 4:2), 1:4)
v
reverse(v)
rev(v)

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

Rle-class-leftovers *Rle objects (old man page)*

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:

`as(from, "IRanges")`**:** Creates an IRanges instance from a logical Rle. Note that this instance is guaranteed to be normal.

`as(from, "NormalIRanges")`**:** Creates a NormalIRanges instance from a logical Rle.

General Methods

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

`split(x, f, drop=FALSE)`**:** Splits `x` according to `f` to create a CompressedRleList object. If `f` is a list-like object then `drop` is ignored and `f` is treated as if it was `rep(seq_len(length(f)), sapply(f, length))`, so the returned object has the same shape as `f` (it also receives the names of `f`). Otherwise, if `f` is not a list-like object, empty list elements are removed from the returned object if `drop` is `TRUE`.

`findRange(x, vec)`**:** Returns an IRanges object representing the ranges in Rle `vec` that are referenced by the indices in the integer vector `x`.

`splitRanges(x)`**:** 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 <- 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)`**:** 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 IntegerRanges objects respectively that are meshed together for form the RleViewsList.

`Views(subject, start=NULL, end=NULL, width=NULL, names=NULL)`**:** Same as `RleViewsList(rleList = subject, rangesList = start)`.

Coercion

In the code snippet below, `from` is an RleViewsList object:

`as(from, "IRangesList")`**:** Creates an `IRangesList` 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 IntegerRangesList 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)
```

---

seqapply *2 methods that should be documented somewhere else*

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

---

setops-methods *Set operations on IntegerRanges and IntegerRangesList objects*

Description

Performs set operations on IntegerRanges and IntegerRangesList objects.

Usage

```
## Vector-wise set operations
## --------------------------

## S4 method for signature 'IntegerRanges,IntegerRanges'
union(x, y)
## S4 method for signature 'Pairs,missing'
union(x, y, ...)

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

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

## Element-wise (aka "parallel") set operations
## --------------------------------------------

## S4 method for signature 'IntegerRanges,IntegerRanges'
punion(x, y, fill.gap=FALSE)
## S4 method for signature 'Pairs,missing'
punion(x, y, ...)

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

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

## S4 method for signature 'IntegerRanges,IntegerRanges'
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)), end = max(end(x), end(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 IntegerRanges objects return a "normal" IntegerRanges 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 IntegerRanges objects are defined. For these methods, x and y must have the same length (i.e. same number of ranges). They return a IntegerRanges object *parallel* to x and y i.e. where the i-th range corresponds to the i-th range in x and iny) and represents the union/intersection/difference/gap of/between the corresponding x[i] 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.

Value

On IntegerRanges objects, union, intersect, and setdiff return an IRanges *instance* that is guaranteed to be *normal* (see isNormal) but is NOT promoted to NormalIRanges.

On IntegerRanges objects, punion, pintersect, psetdiff, and pgap return an object of the same class and length as their first 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.
- IntegerRanges and IntegerRangesList objects. In particular, *normality* of an IntegerRanges object is discussed in the man page for IntegerRanges objects.
- `mendoapply` in the **S4Vectors** package.

Examples

```
x <- IRanges(c(1, 5, -2, 0, 14), c(10, 9, 3, 11, 17))
subject <- Rle(1:-3, 6:2)
y <- Views(subject, start=c(14, 0, -5, 6, 18), end=c(20, 2, 2, 8, 20))

## Vector-wise operations:
union(x, ranges(y))
union(ranges(y), x)

intersect(x, ranges(y))
intersect(ranges(y), x)

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

## Element-wise (aka "parallel") operations:
try(punion(x, ranges(y)))
punion(x[3:5], ranges(y)[3:5])
punion(x, ranges(y), fill.gap=TRUE)
try(pintersect(x, ranges(y)))
pintersect(x[3:4], ranges(y)[3:4])
pintersect(x, ranges(y), resolve.empty="max.start")
psetdiff(ranges(y), x)
try(psetdiff(x, ranges(y)))
start(x)[4] <- -99
end(y)[4] <- 99
psetdiff(x, ranges(y))
pgap(x, ranges(y))

## On IntegerRangesList objects:
irl1 <- IRangesList(a=IRanges(c(1,2),c(4,3)), b=IRanges(c(4,6),c(10,7)))
irl2 <- IRangesList(c=IRanges(c(0,2),c(4,5)), a=IRanges(c(4,5),c(6,7)))
union(irl1, irl2)
intersect(irl1, irl2)
setdiff(irl1, irl2)
```

`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 ?`link{view-summarization-methods}`

Value

The method for Rle objects returns an RleViews object if rangesOnly=FALSE or an IRanges object if rangesOnly=TRUE.

The method for RleList objects returns an RleViewsList object if rangesOnly=FALSE or an IRangesList object if rangesOnly=TRUE.

Author(s)

P. Aboyoun

See Also

- view-summarization-methods for summarizing the views returned by slice.
- slice-methods in the **XVector** package for more slice methods.
- coverage for computing the coverage across a set of ranges.
- The Rle, RleList, RleViews, and RleViewsList classes.

Examples

```
## Views derived from coverage
x <- IRanges(start=c(1L, 9L, 4L, 1L, 5L, 10L),
             width=c(5L, 6L, 3L, 4L, 3L,  3L))
cvg <- coverage(x)
slice(cvg, lower=2)
slice(cvg, lower=2, rangesOnly=TRUE)
```

Vector-class-leftovers

Vector objects (old man page)

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.

`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.

`window(x, start=NA, end=NA, width=NA) <- value`**:** Replace the subsequence window specified on the left (i.e. the subsequence in `x` specified by `start`, `end` and `width`) by `value`. `value` must either be of class `class(x)`, belong to a subclass of `class(x)`, or be coercible to `class(x)` or a subclass of `class(x)`. The elements of `value` are repeated to create a Vector with the same number of elements as the width of the subsequence window it is replacing.

Looping

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

`tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)`**:** Like the standard `tapply` function defined in the base package, the `tapply` method for Vector objects applies a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

Coercion

`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.

`view-summarization-methods` *Summarize views on a vector-like object with numeric values*

Description

`viewApply` applies a function on each view of a Views or ViewsList object.

`viewMins`, `viewMaxs`, `viewSums`, `viewMeans` calculate respectively the minima, maxima, sums, and means of the views in a Views or ViewsList object.

Usage

```
viewApply(X, FUN, ..., simplify = TRUE)

viewMins(x, na.rm=FALSE)
## S4 method for signature 'Views'
min(x, ..., na.rm = FALSE)

viewMaxs(x, na.rm=FALSE)
## S4 method for signature 'Views'
max(x, ..., na.rm = FALSE)

viewSums(x, na.rm=FALSE)
## S4 method for signature 'Views'
sum(x, ..., na.rm = FALSE)

viewMeans(x, na.rm=FALSE)
## S4 method for signature 'Views'
mean(x, ...)

viewWhichMins(x, na.rm=FALSE)
## S4 method for signature 'Views'
which.min(x)

viewWhichMaxs(x, na.rm=FALSE)
## S4 method for signature 'Views'
which.max(x)

viewRangeMins(x, na.rm=FALSE)

viewRangeMaxs(x, na.rm=FALSE)
```

Arguments

`X`	A Views object.
`FUN`	The function to be applied to each view in `X`.
`...`	Additional arguments to be passed on.
`simplify`	A logical value specifying whether or not the result should be simplified to a vector or matrix if possible.

x	An RleViews or RleViewsList object.
na.rm	Logical indicating whether or not to include missing values in the results.

Details

The viewMins, viewMaxs, viewSums, and viewMeans functions provide efficient methods for calculating the specified numeric summary by performing the looping in compiled code.

The viewWhichMins, viewWhichMaxs, viewRangeMins, and viewRangeMaxs functions provide efficient methods for finding the locations of the minima and maxima.

Value

For all the functions in this man page (except viewRangeMins and viewRangeMaxs): A numeric vector of the length of x if x is an RleViews object, or a List object of the length of x if it's an RleViewsList object.

For viewRangeMins and viewRangeMaxs: An IRanges object if x is an RleViews object, or an IRangesList object if it's an RleViewsList object.

Note

For convenience, methods for min, max, sum, mean, which.min and which.max are provided as wrappers around the corresponding view* functions (which might be deprecated at some point).

Author(s)

P. Aboyoun and H. Pagès

See Also

- The slice function for slicing an Rle or RleList object.
- view-summarization-methods in the **XVector** package for more view summarization methods.
- The RleViews and RleViewsList classes.
- The which.min and colSums functions.

Examples

```
## Views derived from coverage
x <- IRanges(start=c(1L, 9L, 4L, 1L, 5L, 10L),
             width=c(5L, 6L, 3L, 4L, 3L,  3L))
cvg <- coverage(x)
cvg_views <- slice(cvg, lower=2)

viewApply(cvg_views, diff)

viewMins(cvg_views)
viewMaxs(cvg_views)

viewSums(cvg_views)
viewMeans(cvg_views)

viewWhichMins(cvg_views)
viewWhichMaxs(cvg_views)
```

```
viewRangeMins(cvg_views)
viewRangeMaxs(cvg_views)
```

`Views-class` *Views objects*

Description

The Views virtual class is a general container for storing a set of views on an arbitrary Vector object, called the "subject".

Its primary purpose is to introduce concepts and provide some facilities that can be shared by the concrete classes that derive from it.

Some direct subclasses of the Views class are: RleViews, XIntegerViews (defined in the XVector package), XStringViews (defined in the Biostrings package), etc...

Constructor

`Views(subject, start=NULL, end=NULL, width=NULL, names=NULL)`**:** This constructor is a generic function with dispatch on argument `subject`. Specific methods must be defined for the subclasses of the Views class. For example a method for XString subjects is defined in the Biostrings package that returns an XStringViews object. There is no default method.

The treatment of the `start`, `end` and `width` arguments is the same as with the `IRanges` constructor, except that, in addition, `Views` allows `start` to be an IntegerRanges 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 IntegerRanges 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))`).

Concatenation

`c(x, ..., ignore.mcols=FALSE)`**:** Concatenate `Views` objects. They must have the same subject.

Other methods

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

`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.

`findOverlaps`.

Examples

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

## Create a set of 4 views on an Rle subject of length 50:
subject <- Rle(LETTERS[1:10], 5)
v1 <- Views(subject, start=9:4, end=9:14)
v1

## Extract the 4th view:
v1[[4]]

## Some views can be "out of limits":
v2 <- Views(101:110, start=4:-1, end=6)
## Not run:
  v2[[5]]  # Error! view is out of limits

## End(Not run)

## Trimming the views so they're no longer out of limits:
trim(v2)

trim(v2)[[5]]

## Note that using subviews() with no additional arguments has
## the same effect as trimming:
subviews(v2)

## See ?`XIntegerViews-class` in the XVector package for more examples.
```

ViewsList-class *List of Views*

Description

An extension of List that holds only Views objects.

Details

ViewsList is a virtual class. Specialized subclasses like e.g. RleViewsList are useful for storing coverage vectors over a set of spaces (e.g. chromosomes), each of which requires a separate RleViews object.

As a 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