IRanges

November 11, 2009

R topics documented:

`Alignment-class` *Alignments between sequences*

Description

The `Alignment` class will represent alignments between multiple sequences. Its design is not yet finalized.

`AnnotatedList-class`
Annotated Lists

Description

The class `AnnotatedList` extends `TypedList` to contain two optional metadata elements: a list with arbitrary annotations, and a data frame with local metadata on each element.

Details

Data analysis should not occur in a vacuum: every dataset floats in a sea of metadata, describing the observed features, as well as the experimental design and methodology. Some metadata is local to an experiment, such as its design, while other information, such as the layout of a microarray chip, transcends individual experiments.

The list structure is fundamental to computationally representing datasets. For example, the `data.frame` is a list of column vectors. The `AnnotatedList` is a list that additionally aims to store local metadata and reference global annotation resources.

This is implemented by adding two optional components on top of the underlying `TypedList`:

metadata A list, possibly empty, containing arbitrary annotations. For example, the name of the genome might be stored in a character vector as an element named "genome".

elementMetadata A data frame of class `XDataFrame` (or `NULL`) with a row for each element and a column for each metadata variable. This is for storing metadata local to the experiment, such as experimental design, or chromosome-wide statistics for datasets of genomic features.

Accessors

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

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

`elementMetadata(x)`, `elementMetadata(x) <- value`: Get or set the `XDataFrame` holding local metadata on each element. The rows are named according to the names of the elements. Optional, may be `NULL`.

Constructor

`AnnotatedList(elements = list(), metadata = NULL, elementMetadata = NULL)`: Constructs an instance of an `AnnotatedList` containing the elements in the ordinary list `elements`. The parameters `metadata` and `elementMetadata` correspond to the components described above.

Author(s)

Michael Lawrence

See Also

`RangesList` and `XDataFrame` both extend this class.

Examples

```
## demonstrated on RangesList objects, a subtype of AnnotatedList

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))
collection <- RangesList(range1, range2)

## access 'metadata' component
metadata(collection)$genome <- "hg18"
metadata(collection)

## set some element metadata
elementMetadata(collection) <- XDataFrame(chrom = c("chr1", "chr2"))
elementMetadata(collection)
```

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

Description

An extension of `TypedList` 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.

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.

Author(s)

P. Aboyoun

See Also

`TypedList` for the applicable methods.

Examples

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

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

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

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

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

## combining
col1 <- IntegerList(one = int1, int2)
col2 <- IntegerList(two = int2, one = int1)
```

```
col3 <- IntegerList(int2)
append(col1, col2)
append(col1, col2, 0)
c(col1, col2, col3)

## get the mean for each element
lapply(col1, mean)
```

`coverage` *Coverage across a set of ranges*

Description

Counts the number of times a position is represented in a set of ranges.

Usage

```
## Old interface (IRanges < 1.1.58):
#coverage(x, start=NA, end=NA, ...)

## Transitional interface (the current one):
coverage(x, start=NA, end=NA, shift=0L, width=NULL, weight=1L, ...)

## New interface (in the near future):
#coverage(x, shift=0L, width=NULL, weight=1L, ...)
```

Arguments

`x`	An IRanges, Views or MaskCollection object, or any object for which a `coverage` method is defined.
`start, end`	Single integers specifying the position in `x` where to start and end the extraction of the coverage. IMPORTANT NOTE: Please do not use these arguments (use the shift/width arguments below). They are temporarily kept for backward compatibility with existing code and will be dropped in the near future.
`shift`	An integer vector (recycled to the length of `x`) specifying how each element in `x` should be (horizontally) shifted before the coverage is computed.
`width`	The length of the returned coverage vector. If `width=NULL` (the default), then the specific `coverage` method that is actually selected will choose the length of the returned vector "in a way that makes sense". For example, when `width=NULL`, the method for IRanges objects returns a vector that has just enough elements to have its last element aligned with the rightmost end of all the ranges in `x` after those ranges have been shifted (see the `shift` argument above). This ensures that any longer coverage vector would be a "padded with zeros" version of the vector returned when `width=NULL`. When `width=NULL`, the method for Views objects returns a vector with `length(subject(x))` elements. When `width=NULL`, the method for MaskCollection objects returns a vector with `width(x)` elements.
`weight`	An integer vector specifying how much each element in `x` counts.
`...`	Further arguments to be passed to or from other methods.

Value

An Rle object representing the coverage of x. An integer value called the "coverage" can be associated to each position in x, indicating how many times this position is covered by the elements contained in x. For example, if x is a Views object, the coverage of a given position in subject(x) is the number of views it belongs to.

Note

The interface of the coverage generic is currently being migrated from "start/end" to "shift/width". In the near future, the start and end arguments will be dropped and the remaining arguments will be: coverage(x, shift=0L, width=NULL, weight=1L, ...) The "shift/width" interface is more intuitive, more convenient and offers slighty more control than the "start/end" interface. Also it makes sense to add the weight argument to the generic (vs having it supported only by some methods) since weighting the elements in x can be considered part of the concept of coverage in general.

Author(s)

H. Pages and P. Aboyoun

See Also

IRanges-class, Views-class, Rle-class, MaskCollection-class

Examples

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

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)
coverage(mymasks)
```

---

disjoin *Making Ranges disjoint*

Description

Functions for making Ranges disjoint, where no ranges overlap another.

Usage

```
disjoin(x, ...)
disjointBins(x, ...)
```

Arguments

x	The Ranges instance, possibly overlapping intervals.
...	Additional arguments for methods

Details

The disjoin method returns a disjoint Ranges, 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 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.

Author(s)

M. Lawrence

See Also

reduce for making normal ranges, a subset of disjoint ranges, where there must be a gap of length >= 1 between each range.

Examples

```
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))

disjointBins(IRanges(1, 5)) # 1L
disjointBins(IRanges(c(3, 1, 10), c(5, 12, 13))) # c(2L, 1L, 2L)
```

FilterRules-class *Collection of Filter Rules*

Description

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

Details

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

Rules may be either expressions or functions. Evaluating an expression or invoking a function should result in a logical vector. Expressions are often more convenient, but functions (i.e. closures) are generally safer and more powerful, because the user can specify the enclosing environment. If

a rule is an expression, it is evaluated inside the `envir` argument to the `eval` method (see below). If a function, it is invoked with `envir` as its only argument. See examples.

Accessor methods

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

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

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

Constructor

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

Subsetting and Replacement

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

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

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

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

Combining

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

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

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

Evaluating

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

Author(s)

Michael Lawrence

See Also

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

Examples

```
## constructing a FilterRules instance

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

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

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

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

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

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

## set the active state directly

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

## toggle the active state by name or index

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

IntervalTree-class *Interval Search Trees*

Description

An IntervalTree object is an external representation of ranges (i.e. it is derived from XRanges) that is optimized for overlap queries.

Details

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

The usual workflow is to create an IntervalTree using the constructor described below and then perform overlap queries using the overlap method. The results of the query are returned as a RangesMatching object.

Constructor

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

Finding Overlaps

This main purpose of the interval tree is to optimize the search for ranges overlapping those in a query set. The interface for this operation is the overlap function.

overlap(object, query = object, maxgap = 0, multiple = TRUE):
Find the intervals in query, a Ranges or integer vector to be converted to length-one ranges, that overlap with the intervals object, an IntervalTree or, for convenience, a Ranges coercible to a IntervalTree. If query is omitted, object is queried against itself. if query is unsorted, it is sorted first, so it is usually better to sort up-front, to avoid a sort with each overlap call. Intervals with a separation of maxgap or less are considered to be overlapping. maxgap should be a scalar, non-negative, non-NA number. When multiple (a scalar non-NA logical) is TRUE, the results are returned as a RangesMatching object.

If multiple is FALSE, at most one overlapping interval in object is returned for each interval in query. The matchings are returned as an integer vector of length length(query), with NA indicating intervals that did not overlap any intervals in object. This is analogous to the return value of the match function.

query may also be a RangesList, in which case object must also be a RangesList. If both lists have names, each element from the subject is paired with the element from the query with the matching name, if any. Otherwise, elements are paired by position. The overlap is then computed between the pairs as described above. If multiple is TRUE, a RangesMatchingList is returned, otherwise a list of integer vectors. Each element of the result corresponds to a space in query. For spaces that did not exist in object, the overlap is nil.

x %in% table: Shortcut for finding the ranges in x that overlap any of the ranges in table. Both x and table should be Ranges objects. The result is a logical vector of the same length as x.

Coercion

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

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

Accessors

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

Notes on Time Complexity

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

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

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

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

Author(s)

Michael Lawrence

References

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

See Also

`XRanges`, the parent of this class, `RangesMatching`, the result of an overlap query.

Examples

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

## at most one hit per query
overlap(tree, query, multiple = FALSE) # c(2, NA, 3)
```

```
## allow multiple hits
overlap(tree, query)

## overlap as long as distance <= 1
overlap(tree, query, maxgap = 1)

## shortcut
overlap(subject, query)

## query and subject are easily interchangeable
query <- IRanges(c(1, 4, 9), c(5, 7, 10))
subject <- IRanges(c(2, 2), c(5, 4))
tree <- IntervalTree(subject)
t(overlap(tree, query))
# the same as:
overlap(query, subject)

## one Ranges with itself
overlap(query)

## single points as query
subject <- IRanges(c(1, 6, 13), c(4, 9, 14))
overlap(subject, c(3L, 7L, 10L), multiple=FALSE)
```

`IRanges-class` *IRanges and NormalIRanges objects*

Description

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

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

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

Constructor

See `¿IRanges-constructor`.

Methods for NormalIRanges objects

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

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

Author(s)

H. Pages

See Also

Ranges-class, IRanges-constructor, IRanges-utils, IRanges-setops.

Some direct subclasses of the IRanges class (other than NormalIRanges): Views-class.

Examples

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

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

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

IRanges-constructor

The IRanges constructor and supporting functions

Description

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

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

Usage

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

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

Arguments

`start, end, width`
: For `IRanges` and `solveUserSEW0`: `NULL`, or vector of integers (eventually with NAs).
For `solveUserSEW`: vector of integers (eventually with NAs).

`names`
: A character vector or `NULL`.

`refwidths`
: Vector of non-negative integers containing the reference widths.

`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 expanded cyclically to the length of the longest (NULL arguments are filled with NAs). After this expansion, each row in the 3-column matrix obtained by binding those 3 vectors together is "solved" i.e. NAs are treated as unknown in the equation `end = start + width - 1`. Finally, the solved matrix is returned as an IRanges instance.

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

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

Supporting functions

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

`solveUserSEW(refwidths, start=NA, end=NA, width=NA, translate.negative.coord=TR allow.nonnarrowing=FALSE)`: `start`, `end` and `width` must have the same number of elements as, or less elements than, `refwidths`. In the latter case, they are expanded cyclically to the length of `refwidths` (provided none are of zero length). After this expansion, 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 (with the same number of elements as `refwidths`).

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

Author(s)

H. Pages

See Also

IRanges-class, `narrow`

Examples

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

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

## ---------------------------------------------------------------------
## B. USING solveUserSEW()
## ---------------------------------------------------------------------
refwidths <- c(5:3, 6:7)
refwidths

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

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

IRanges internals *IRanges internals*

Description

IRanges objects, classes and methods that are not intended to be used directly.

IRangesList-class *List of IRanges*

Description

A RangesList that only stores IRanges objects.

Constructor

`IRangesList(..., compress = TRUE)`: Each `IRanges` in `...` becomes an element in the new `IRangesList`, in the same order. This is analogous to the `list` constructor, except every argument in `...` must be derived from `IRanges`. If `compress`, the internal storage of the data is compressed.

Coercion

`as(from, "NormalIRanges")`: Merges each of the elements into a single `NormalIRanges` through `reduce`.

`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.

Author(s)

Michael Lawrence

See Also

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

Examples

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

IRanges-setops *Set operations on IRanges objects*

Description

Performs set operations on IRanges objects.

Usage

```
## Vector-wise operations:
## S4 method for signature 'IRanges, IRanges':
union(x, y)
## S4 method for signature 'IRanges, IRanges':
intersect(x, y)
## S4 method for signature 'IRanges, IRanges':
setdiff(x, y)

## Element-wise (aka "parallel") operations:
punion(x, y, ...)
pintersect(x, y, ...)
psetdiff(x, y, ...)
pgap(x, y, ...)
```

Arguments

x, y	IRanges objects.
...	Further arguments to be passed to or from other methods. For example, the fill.gap argument can be passed to the punion method for IRanges objects (see below).

Details

The union, intersect and setdiff methods for IRanges objects return a "normal" IRanges object (of the same class as x) representing the union, intersection and (asymmetric!) difference of the sets of integers represented by x and y.

punion, pintersect, psetdiff and pgap are generic functions that compute the element-wise (aka "parallel") union, intersection, (asymmetric!) difference and gap between each element in x and its corresponding element in y. Methods for IRanges objects are defined. For these methods, x and y must have the same length (i.e. same number of ranges) and they return an IRanges instance of the same length as x and y where each range represents the union/intersection/difference/gap of/between the corresponding ranges in x and y.

Note that the union or difference of 2 ranges cannot always be represented by a single range so punion and psetdiff cannot always return their result in an IRanges instance of the same length as the input. This happens to punion when there is a gap between the 2 ranges to combine. In that case, the user can use the fill.gap argument to enforce the union by filling the gap. This happens to psetdiff when a range in y has its end points strictly inside the corresponding range in x. In that case, psetdiff will simply fail.

If two ranges overlap, then the gap between them is empty.

Author(s)

H. Pages and M. Lawrence

See Also

Ranges-class, IRanges-class, IRanges-utils

Examples

```
x <- IRanges(c(1, 5, -2, 0, 14), c(10, 9, 3, 11, 17))
y <- Views(as(4:-17, "XInteger"), start=c(14, 0, -5, 6, 18), end=c(20, 2, 2, 8, 20))

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

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

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

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

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)

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

## Modify an IRanges object (endomorphisms):
shift(x, shift, use.names=TRUE)
restrict(x, start=NA, end=NA, keep.all.ranges=FALSE, use.names=TRUE)
narrow(x, start=NA, end=NA, width=NA, use.names=TRUE)
```

```
threebands(x, start=NA, end=NA, width=NA)
reduce(x, with.inframe.attrib=FALSE)
gaps(x, start=NA, end=NA)

## Other utilities
## S4 method for signature 'Ranges':
reflect(x, bounds)
## S4 method for signature 'Ranges':
flank(x, width, start = TRUE, both = FALSE)
## S4 method for signature 'Ranges':
range(x, ..., na.rm = FALSE)

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

Arguments

width	For successiveIRanges, must be a vector of non-negative integers (with no NAs) specifying the widths of the ranges to create. For narrow and threebands, a vector of integers, eventually with NAs. See the SEW (Start/End/Width) interface for the details (?solveUserSEW). For flank, the width of the 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.
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.
x	A logical vector for whichAsIRanges. An IRanges object for shift, restrict, narrow, threebands, reduce, gaps and asNormalIRanges.
shift	A single integer.
use.names	TRUE or FALSE. Should names be preserved?
start, end	A single integer or NA for all functions except narrow and threebands. For narrow and threebands, the supplied start and end arguments must be vectors of integers, eventually with NAs, that contain coordinates relative to the current ranges. 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.
keep.all.ranges	TRUE or FALSE. Should ranges that don't overlap with the interval specified by start and end 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.
with.inframe.attrib	TRUE or FALSE. For internal use.
bounds	An IRanges object to serve as the reference bounds for the reflection, see below.
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.

...	Additional Ranges to consider.
na.rm	Ignored
force	TRUE or FALSE. Should x be turned into a NormalIRanges object even if isNormal(x) is FALSE?

Details

successiveIRanges returns an IRanges object containing the ranges on subject 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.

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

shift shifts all the ranges in x.

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

narrow narrows the ranges in x i.e. each range in the returned IRanges 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.

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).

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

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

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..]
```

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
```

`range` returns an `IRanges` instance with a single range, from the minimum start to the maximum end in the combined ranges of `x` and the arguments in `...`.

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

Author(s)

H. Pages and M. Lawrence

See Also

Ranges-class, IRanges-class, IRanges-setops, `solveUserSEW`, `successiveViews`

Examples

```
vec <- as.integer(c(19, 5, 0, 8, 5))
whichAsIRanges(vec >= 5)
x <- successiveIRanges(vec)
x
shift(x, -3)
restrict(x, start=12, end=34)
restrict(x, start=20)
restrict(x, start=21)
restrict(x, start=21, keep.all.ranges=TRUE)

y <- x[width(x) != 0]
narrow(y, start=4, end=-2)
narrow(y, start=-4, end=-2)
narrow(y, end=5, width=3)
narrow(y, start = c(3, 4, 2, 3), end = c(12, 5, 7, 4))

z <- threebands(y, start=4, end=-2)
y0 <- punion(z$left, z$right, fill.gap=TRUE)
identical(y, y0)  # TRUE
threebands(y, start=-5)
```

```
x <- IRanges(start=c(-2L, 6L, 9L, -4L, 1L, 0L, -6L, 10L),
             width=c( 5L, 0L, 6L,  1L, 4L, 3L,  2L,  3L))
reduce(x)
gaps(x)
gaps(x, start=-6, end=20)  # Regions of the -6:20 range that are not masked by 'x'.

ir1 <- IRanges(c(2,5,1), c(3,7,3))

bounds <- IRanges(c(0, 5, 3), c(10, 6, 9))
reflect(ir1, bounds)

flank(ir1, 2)
flank(ir1, 2, FALSE)
flank(ir1, 2, c(FALSE, TRUE, FALSE))
flank(ir1, c(2, -2, 2))
flank(ir1, 2, both = TRUE)
flank(ir1, 2, FALSE, TRUE)
flank(ir1, -2, FALSE, TRUE)

asNormalIRanges(x)   # 3 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)
```

`ListLike-class` *ListLike objects*

Description

The ListLike class is just an interface i.e. a virtual class with no slots. ListLike subclasses (i.e. classes that contain the ListLike class) must define the following minimal set of functions/operators: `length`, `"[["` and `names`. Then, instances of these subclasses can be considered to have the shape of a list, and the methods described below apply to them.

Some direct subclasses of the ListLike class are: Views, MaskCollection, XStringSet (defined in the Biostrings package), etc...

Methods

In the code snippets below, `x` and `X` are ListLike objects.

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

`lapply(X, FUN, ...)`: Like the standard `lapply` function defined in the base package, the `lapply` method for ListLike objects returns a list of the same length as `X`, each element of which is the result of applying `FUN` to the corresponding element of `X`.

`sapply(X, FUN, ..., simplify=TRUE, USE.NAMES=TRUE)`: Like the standard `sapply` function defined in the base package, the `sapply` method for ListLike objects is a user-friendly version of `lapply` by default returning a vector or matrix if appropriate.

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

`isEmpty(x)`: Here `x` can be an atomic, list or ListLike object, or any object for which an `isEmpty` method is defined. If `x` is atomic, returns `length(x) == 0L`. If `x` is a list or ListLike object, then it works elementwise and is defined recursively by `sapply(x, function(xx) all(isEmpty(xx)))`.

Author(s)

H. Pages

See Also

`lapply`, `sapply`, `as.list`.

Some direct subclasses of the ListLike class: TypedList-class, Views-class, MaskCollection-class, XStringSet-class.

Examples

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

MaskCollection-class

MaskCollection objects

Description

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

Details

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

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

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

Basic accessor methods

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

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

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

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

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

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

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

Constructor

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

Other methods

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

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

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

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

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

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

Subsetting and appending

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

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

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

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

Other methods

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

`reduce(x)`: Return a MaskCollection object of length 1 made of the union (or merging, or collapsing) of all the active masks in x.

`gaps(x)`: Invert the masks in x.

`subseq(x, start=NA, end=NA, width=NA)`: If y is a sequence that x has been put on top of, then `subseq` will return the set of submasks that go on top of the subsequence obtained by calling `subseq` on y (`subseq` must be called on x with the same arguments that have been used when called on y).

Author(s)

H. Pages

See Also

NormalIRanges-class, read.Mask, MaskedXString-class, `alphabetFrequency`, `reverse`, `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)
reduce(mymasks)
gaps(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
reduce(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]))
mymasks2 <- subseq(mymasks, start=8)
mymasks2
mymasks2[[2]]
```

nearest *Nearest neighbor finding*

Description

The nearest, precede and follow methods find nearest neighbors between Ranges instances.

Usage

```
nearest(x, subject, ...)
precede(x, subject = x, ...)
follow(x, subject = x, ...)
```

Arguments

x	The query Ranges instance.
subject	The subject Ranges instance, within which the nearest neighbors are found. Can be missing, in which case the query, x, is also the subject.
...	Additional arguments for methods

Details

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

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

1. Find the ranges in subject that overlap xi. If a single range si in subject overlaps xi, si is returned as the nearest neighbor of xi. If there are multiple overlaps, one of the overlapping ranges is chosen arbitrarily.
2. If no ranges in subject overlap with xi, then the range in subject with the shortest distance from its end to the start xi or its start to the end of xi is returned.

precede returns an integer vector of the index of range in subject that ends before and closest to the start of each range in x. Note that any overlapping ranges are excluded. NA is returned when there are no qualifying ranges in subject.

follow is the opposite of precede: it returns the index of the range in subject that starts after and closest to the end of each range in x.

Author(s)

M. Lawrence

See Also

overlap for finding just the overlapping ranges.

Examples

```
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)

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)
```

RangedData-class *Data on ranges*

Description

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

Details

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

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

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

Object Updating

updateRangedData(object): Updates instances of objects the inherit from an older RangedData class definition to match the current RangedData class definition.

Accessor methods

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

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

Array accessors:

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

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

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

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

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

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

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

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

`start(x)`: The start value of each range.

`width(x)`: The width of each range.

`end(x)`: The end value of each range.

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

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

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

Other accessors:

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

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

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

`values(x)`: Gets the data values in `x` as a `SplitXDataFrameList`.

Constructor

`RangedData(ranges = IRanges(), ..., splitter = NULL, universe = NULL)`: Creates a `RangedData` with the ranges in `ranges` and variables given by the arguments in `...`. See the constructor `XDataFrame` for how the `...` arguments are interpreted. If `splitter` is `NULL`, all of the ranges and values are placed into the same space, resulting in a single-space (length one) `RangedData`. Otherwise, the ranges and values are split into spaces according to `splitter`, which is treated as a factor, like the `f` argument in `split`. The universe may be specified as a scalar string by the `universe` argument.

Coercion

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

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

`as(from, "RangedData")`: Coerce an `Rle` or an `XRle` to a `RangedData` by converting each run to a range and storing the run values in a column named "score".

Subsetting and Replacement

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

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

`x[i, j]`: Subsets `x` by indexing into its rows and columns. The result is of the same class, with a different set of rows and columns. Note that this differs from the subset form above, because we are now treating `x` as one contiguous dataset.

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

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

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

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

Splitting and Combining

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

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

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

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

Applying

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

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

Author(s)

Michael Lawrence

See Also

RangedData-utils for utlities and the `rdapply` function for applying a function to each space separately.

Examples

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

## constructing RangedData instances

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

## split some data over chromosomes

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

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

## subsetting

## list style: [i]

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

```
rd[TRUE] # like rd[]
rd[c(TRUE, FALSE)]
rd[1] # numeric index
rd[c(1,2)]
rd[-2]

## matrix style: [i,j]

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

## dimnames

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

## space names

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

## variable replacement

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

## combining/splitting

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

## applying
```

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

RangedDataList-class *Lists of RangedData*

Description

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

Constructor

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

Author(s)

Michael Lawrence

See Also

RangedData, the element type of this TypedList.

Examples

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

RangedData-utils *RangedData utility functions*

Description

Utility functions for manipulating RangedData objects.

Usage

```
## S4 method for signature 'expressionORlanguage,
##   RangedData':
eval(expr, envir, enclos =
parent.frame())
## S4 method for signature 'RangedData':
range(x, ..., na.rm = FALSE)
```

Arguments

expr	The expression, call, or name to be evaluated.
envir	The RangedData object in which to evaluate expr.
enclos	The environment in which to look for symbols that do not exist in the environment formed from RangedData.
x	A RangedData object
...	Additional RangedData objects
na.rm	Ignored

Details

The eval method converts the RangedData object specified in envir to an environmnent, with enclos as its parent, and then evaluates expr within that environment. The RangedData environment contains the following objects:

ranges The result of unlist(ranges(envir)), i.e. all of the ranges in a single Ranges object.

colnames(envir) The data columns in envir are stored individually by their column names.

The objects are not actually copied into the environment. Rather, they are dynamically bound using makeActiveBinding. This prevents unnecessary copying of the data from the external vectors into R vectors. The values are cached, so that the data is not copied every time the symbol is accessed.

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

Value

The result of expression evaluation.

Author(s)

Michael Lawrence

See Also

FilterRules objects, which can be evaluated on a RangedData, and the base eval function.

Examples

```
ranges <- IRanges(c(1,2,3),c(4,5,6))
score <- c(10L, 2L, NA)
rd <- RangedData(ranges, score)
eval(quote(score > 3), rd)

range(rd)
rd2 <- RangedData(IRanges(c(5,2,0), c(6,3,1)))
range(rd, rd2)
```

Ranges-class *Ranges objects*

Description

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

Details

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

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

The starting point (or "start") of a range can be any integer (see `start` below) but its "width" must be a non-negative integer (see `width` below). The ending point (or "end") of a range is equal to its "start" plus its "width" minus one (see `end` below). An "empty" range is a range that contains no value i.e. a range that has a null width. Note that for an empty range, the end is smaller than the start.

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

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

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

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

Methods

In the code snippets below, `x`, `y` and `object` are Ranges objects. Not all the functions described below will necessarily work with all kinds of Ranges objects but they should work at least for IRanges objects. Also more operations on Ranges objects are described in the man page for IRanges-utils (`shift`, `restrict`, `narrow`, `reduce`, `gaps`), for IntervalTree objects (`overlap`) and for RangesList objects (`split` method for Ranges objects).

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

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

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

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

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

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

`update(object, ...)`: Convenience method for combining multiple modifications of `object` in one single call. For example `object <- update(object, start=start(object)-2L, end=end(object)+2L)` is equivalent to `start(object) <- start(object)-2L; end(object) <- end(object)+2L`.

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

`isDisjoint(x)`: Return a logical value indicating whether the ranges `x` are disjoint (i.e. non-overlapping).

`as.matrix(x, ...)`: Convert `x` into a 2-column integer matrix containing `start(x)` and `width(x)`. Extra arguments (`...`) are ignored.

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

`as.integer(x)`: Convert `x` into an integer vector, by converting each range into the integer sequence formed by `from:to` and concatenating them together.

`x[i]`: Return a new Ranges object (of the same type as `x`) made of the selected ranges. `i` can be a numeric vector, a logical vector, `NULL` or missing. If `x` is a NormalIRanges object and `i` a positive numeric subscript (i.e. a numeric vector of positive values), then `i` must be strictly increasing.

`rep(x, times)`: Return a new Ranges object made of the repeated elements.

`c(x, ...)`: Combine `x` and the Ranges objects in `...` together. Any object in `...` must belong to the same class as `x`, or to one of its subclasses, or must be `NULL`. The result is an object of the same class as `x`. NOTE: Only works for IRanges (and derived) objects for now.

`x * y`: The arithmetic operation `x * y` is for centered zooming. It symmetrically scales the width of `x` by `1/y`, where `y` is a numeric vector that is recycled as necessary. For example, `x * 2` results in ranges with half their previous width but with approximately the same midpoint. The ranges have been "zoomed in". If `y` is negative, it is equivalent to `x * (1/abs(y))`. Thus, `x * -2` would double the widths in `x`. In other words, `x` has been "zoomed out".

Normality

A Ranges object `x` is implicitly representing an arbitrary finite set of integers (that are not necessarily consecutive). This set is the set obtained by taking the union of all the values in all the ranges in `x`. This representation is clearly not unique: many different Ranges objects can be used to represent the same set of integers. However one and only one of them is guaranteed to be "normal".

By definition a Ranges object is said to be "normal" when its ranges are: (a) not empty (i.e. they have a non-null width); (b) not overlapping; (c) ordered from left to right; (d) not even adjacent (i.e. there must be a non empty gap between 2 consecutive ranges).

Here is a simple algorithm to determine whether `x` is "normal": (1) if `length(x) == 0`, then `x` is normal; (2) if `length(x) == 1`, then `x` is normal iff `width(x) >= 1`; (3) if `length(x) >= 2`, then `x` is normal iff:

```
start(x)[i] <= end(x)[i] < start(x)[i+1] <= end(x)[i+1]
```

for every 1 <= `i` < `length(x)`.

The obvious advantage of using a "normal" Ranges object to represent a given finite set of integers is that it is the smallest in terms of of number of ranges and therefore in terms of storage space. Also the fact that we impose its ranges to be ordered from left to right makes it unique for this representation.

A special container (NormalIRanges) is provided for holding a "normal" IRanges object: a NormalIRanges object is just an IRanges object that is guaranteed to be "normal".

Here are some methods related to the notion of "normal" Ranges:

`isNormal(x)`: Return a logical value indicating whether `x` is "normal" or not.

`whichFirstNotNormal(x)`: Return `NA` if `x` is normal, or the smallest valid indice `i` in `x` for which `x[1:i]` is not "normal".

Author(s)

H. Pages and M. Lawrence

See Also

Ranges-comparison, IRanges-class, IRanges-utils, IRanges-setops, XRanges-class, RangedData-class, IntervalTree-class, `update`, `as.matrix`, `as.data.frame`, `rep`

Examples

```
x <- IRanges(start=c(2:-1, 13:15), width=c(0:3, 2:0))
x
length(x)
start(x)
width(x)
end(x)
isEmpty(x)
as.matrix(x)
as.data.frame(x)

## Subsetting:
x[4:2]                  # 3 ranges
x[-1]                   # 6 ranges
x[FALSE]                # 0 range
x0 <- x[width(x) == 0]  # 2 ranges
isEmpty(x0)

## Use the replacement methods to resize the ranges:
width(x) <- width(x) * 2 + 1
x
end(x) <- start(x)            # equivalent to width(x) <- 0
x
width(x) <- c(2, 0, 4)
x
start(x)[3] <- end(x)[3] - 2  # resize the 3rd range
x

## Name the elements:
names(x)
names(x) <- c("range1", "range2")
x
x[is.na(names(x))]  # 5 ranges
x[!is.na(names(x))]  # 2 ranges

## Check for disjointedness
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

ir <- IRanges(c(1,5), c(3,10))
ir*1 # no change
ir*c(1,2) # zoom second range by 2X
ir*-2 # zoom out 2X
```

`Ranges-comparison` *Ranges comparison*

Description

Equality and ordering of ranges, and related methods.

Usage

```
## ==== Equality and related methods ====
## --------------------------------------

x == y
x != y

## S4 method for signature 'Ranges':
duplicated(x, incomparables=FALSE, fromLast=FALSE, ...)

## S4 method for signature 'Ranges':
unique(x, incomparables=FALSE, fromLast=FALSE, ...)

## ==== Ordering and related methods ====
## --------------------------------------

x <= y
x >= y
x < y
x > y

## S4 method for signature 'Ranges':
order(..., na.last=TRUE, decreasing=FALSE)

## S4 method for signature 'Ranges':
sort(x, decreasing=FALSE, ...)

## S4 method for signature 'Ranges':
rank(x, na.last=TRUE, ties.method=c("average", "first", "random", "max", "min"
```

Arguments

`x,y`	A Ranges object.
`incomparables`	Must be `FALSE`.
`fromLast`	`TRUE` or `FALSE`.

`...`	Ranges objects for `order`.
`na.last`	Ignored.
`decreasing`	`TRUE` or `FALSE`.
`ties.method`	A character string specifying how ties are treated. Only `"first"` is supported for now.

Details

Two ranges are considered equal iff they share the same start and width. Note that with this definition, 2 empty ranges are generally not equal (they need to share the same start to be considered equal).

Ranges are ordered by starting position first, and then by width. This way, the space of ranges is totally ordered. The `order`, `sort` and `rank` methods for Ranges objects are consistent with this order.

`duplicated(x)`: Determines which elements of `x` are equal to elements with smaller subscripts, and returns a logical vector indicating which elements are duplicates. It is semantically equivalent to `duplicated(as.data.frame(x))`. See `duplicated` in the base package for more details.

`unique(x)`: Removes duplicate ranges from `x`. See `unique` in the base package for more details.

`order(...)`: Returns a permutation which rearranges its first argument (a Ranges object) into ascending order, breaking ties by further arguments (also Ranges objects). See `order` in the base package for more details.

`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.

See Also

Ranges-class, IRanges-class, `duplicated`, `unique`, `order`, `sort`, `rank`

Examples

```
x <- IRanges(start=c(20L, 8L, 20L, 22L, 25L, 20L, 22L, 22L),
             width=c( 4L, 0L, 11L,  5L,  0L,  9L,  5L,  0L))
x
which(width(x) == 0)  # 3 empty ranges
x[2] == x[5]  # FALSE
x == x[4]
duplicated(x)
unique(x)
x >= x[3]
order(x)
sort(x)
rank(x, ties.method="first")
```

RangesList-class *List of Ranges*

Description

An extension of AnnotatedList that holds only Ranges objects. Useful for storing ranges over a set of spaces (e.g. chromosomes), each of which requires a separate Ranges object. As an AnnotatedList, RangesList may be annotated with its universe identifier (e.g. a genome) in which all of its spaces exist.

Accessors

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

All of these accessors collapse over the spaces:

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

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

width(x): Get the widths of the ranges.

space(x): Gets the spaces of the ranges as a character vector. This is equivalent to names(x), except each name is repeated according to the length of its element.

These accessors are for the universe identifier:

universe(x): gets the name of the universe as a single string, if one has been specified, NULL otherwise.

universe(x) <- value: sets the name of the universe to value, a single string or NULL.

Constructor

RangesList(..., universe = NULL): Each Ranges in ... becomes an element in the new RangesList, in the same order. This is analogous to the list constructor, except every argument in ... must be derived from Ranges. The universe is specified by the universe parameter, which should be a single string or NULL, to leave unspecified.

Subsetting

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

x[i]: Subset x by index i, with the same semantics as a basic TypedList, except i may itself be a RangesList, in which case only the ranges in x that overlap with those in i are kept. See the overlap method for more details.

Coercion

In the code snippets below, x and from are a RangesList object.

as.data.frame(x, row.names = NULL, optional = FALSE): Coerces x to a data.frame. Essentially the same as calling as.data.frame(unlist(x)).

as(from, "RangedData"): Coerces from to a RangedData with zero columns and the same ranges as in from.

`as(from, "IRangesList")`: Coerces `from`, to an `IRangesList`, requiring that all `Ranges` elements are coerced to internal `IRanges` elements. This is a convenient way to ensure that all `Ranges` have been imported into R (and that there is no unwanted overhead when accessing them).

Arithmetic Operations

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

Author(s)

Michael Lawrence

Examples

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

# subset by RangesList
range1 <- IRanges(start=c(1,2,3), end=c(5,2,8))
range2 <- IRanges(start=c(1,15,20,45), end=c(5,15,100,80))
collection <- RangesList(one = range1, range2)
collection[RangesList()] # empty elements
collection[RangesList(IRanges(4, 6), IRanges(50, 70))]
collection[RangesList(IRanges(50, 70), one=IRanges(4, 6))]

# same as list(range1, range2)
as.list(RangesList(range1, range2))

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

# set the universe
universe(named) <- "hg18"
universe(named)
RangesList(range1, range2, universe = "hg18")

## zoom in 2X
collection * 2
```

RangesList-utils *RangesList utility functions*

Description

Utility functions for manipulating `RangesList` objects.

Usage

```
## S4 method for signature 'RangesList':
gaps(x, start=NA, end=NA)
## S4 method for signature 'RangesList':
reduce(x, with.inframe.attrib = FALSE)
## S4 method for signature 'RangesList':
range(x, ..., na.rm = FALSE)
```

Arguments

x	A RangesList
start	The start of the range over which to calculate the gaps. If NA, use the minimum start position in the Ranges object.
end	The end of the range over which to calculate the gaps. If NA, use the maximum end position in the Ranges object.
with.inframe.attrib	Ignored.
...	Additional RangesList objects
na.rm	Ignored

Details

The gaps function takes the complement (the gaps) of each element in the list and returns the result as a RangesList.

The reduce method merges (via reduce) all of the elements into a single Ranges object and returns the result as a length-one RangesList.

range finds the range, i.e. a Ranges with one range, from the minimum start to the maximum end, on each element in x and returns the result as a RangesList. If there are additional RangesList objects in ..., they are merged into x by name, if all objects have names, otherwise, if they are all of the same length, by position. Else, an exception is thrown.

Value

A RangesList object. For gaps and range, length is the same as that of x. For reduce, length is one.

Author(s)

Michael Lawrence, H. Pages

See Also

RangesList, reduce, gaps

Examples

```
# 'gaps'
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))
collection <- RangesList(one = range1, range2)

# these two are the same
```

```
RangesList(gaps(range1), gaps(range2))
gaps(collection)

# 'reduce'
range2 <- IRanges(start=c(45,20,1), end=c(100,80,5))
collection <- RangesList(one = range1, range2)

# and these two are the same
reduce(collection)
RangesList(asNormalIRanges(IRanges(c(1,20), c(8, 100)), force=FALSE))

# 'range'
rl <- RangesList(a = IRanges(c(1,2),c(4,3)), b = IRanges(c(4,6),c(10,7)))
rl2 <- RangesList(c = IRanges(c(0,2),c(4,5)), a = IRanges(c(4,5),c(6,7)))
range(rl, rl2) # matched by names
names(rl2) <- NULL
range(rl, rl2) # now by position
```

RangesMatching-class

Matchings between Ranges

Description

The RangesMatching class stores a set of matchings between the ranges in one Ranges object and the ranges in another. Currently, RangesMatching are used to represent the result of an overlap query, though other matching operations are imaginable.

Details

The as.matrix method coerces a RangesMatching to a two column matrix with one row for each matching, where the value in the first column is the index of a range in the first (query) Ranges and the index of the matched subject range is in the second column. The matchMatrix function returns the same thing, but use of as.matrix is preferred.

The as.table method counts the number of matchings for each query range and outputs the counts as a table.

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

To get the actual regions of intersection between the overlapping ranges, use the ranges accessor.

Coercion

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

as.matrix(x): Coerces x to a two column integer matrix, with each row representing a matching between a query index (first column) and subject index (second column).

as.table(x): counts the number of matchings for each query range in x and outputs the counts as a table.

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

Accessors

`matchMatrix(x)`: A synonym for `as.matrix`, above.

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

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

`nrow(x)`: get the number of subjects in the matching

`ncol(x)`: get the number of queries in the matching

`dim(x)`: get a two-element integer vector, essentially `c(nrow(x), ncol(x))`.

Author(s)

Michael Lawrence

See Also

`overlap`, which generates an instance of this class.

Examples

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

as.matrix(matchings)

as.table(matchings) # hits per query
as.table(t(matchings)) # hits per subject
```

RangesMatchingList-class

List of Matchings between Ranges

Description

The `RangesMatchingList` class stores a set of matchings, represented as `RangesMatching` objects, between the ranges in one `RangesList` object and the ranges in another.

Details

Roughly the same set of utilities are provided for `RangesMatchingList` as for `RangesMatching`:

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

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

To transpose a RangesMatchingList x, so that the subject and query in each space are interchanged, call t(x). This allows, for example, counting the number of subjects that matched using as.table.

To get the actual regions of intersection between the overlapping ranges, use the ranges accessor.

Coercion

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

as.matrix(x): calls as.matrix on each RangesMatching, combines them row-wise and offsets the indices so that they are aligned with the result of calling unlist on the query and subject (as long as the names on each, if any, are in the same order).

as.table(x): counts the number of matchings for each query range in x and outputs the counts as a table, which is aligned with the result of calling unlist on the query.

t(x): Interchange the query and subject in each space of x, returns a transposed RangesMatchingList.

Accessors

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

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

Note

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

Author(s)

Michael Lawrence

See Also

overlap, which generates an instance of this class.

---

rdapply *Applying over spaces*

Description

The rdapply function applies a user function over the spaces of a RangedData. The parameters to rdapply are collected into an instance of RDApplyParams, which is passed as the sole parameter to rdapply.

Usage

```
rdapply(x, ...)
```

Arguments

x	The RDApplyParams instance, see below for how to make one.
...	Additional arguments for methods

Details

The rdapply function is an attempt to facilitate the common operation of performing the same operation over each space (e.g. chromosome) in a RangedData. To facilitate a wide array of such tasks, the function takes a large number of options. The RDApplyParams class is meant to help manage this complexity. In particular, it facilitates experimentation through its support for incremental changes to parameter settings.

There are two RangedData settings that are required: the user function object and the RangedData over which it is applied. The rest of the settings determine what is actually passed to the user function and how the return value is processed before relaying it to the user. The following is the description and rationale for each setting.

rangedData **REQUIRED**. The RangedData instance over which applyFun is applied.

applyFun **REQUIRED**. The user function to be applied to each space in the RangedData. The function must expect the RangedData as its first parameter and also accept the parameters specified in applyParams.

applyParams The list of additional parameters to pass to applyFun. Usually empty.

filterRules The instance of FilterRules that is used to filter each subset of the RangedData passed to the user function. This is an efficient and convenient means for performing the same operation over different subsets of the data on a space-by-space basis. In particular, this avoids the need to store subsets of the entire RangedData. A common workflow is to invoke rdapply with one set of active filters, enable different filters, reinvoke rdapply, and compare the results.

simplify A scalar logical (TRUE or FALSE) indicating whether the list to be returned from rdapply should be simplified as by sapply. Defaults to FALSE.

reducerFun The function that is used to convert the list that would otherwise be returned from rdapply to something more convenient. The function should take the list as its first parameter and also accept the parameters specified in reducerParams. This is an alternative to the primitive behavior of the simplify option (so simplify must be FALSE if this option is set). The aim is to orthogonalize the applyFun operation (i.e. the statistics) from the data structure of the result.

reducerParams A list of additional parameters to pass to reducerFun. Can only be set if reducerFun is set. Usually empty.

Value

By default a list holding the result of each invocation of the user function, but see details.

Constructing an RDApplyParams object

RDApplyParams(rangedData, applyFun, applyParams, filterRules, simplify, reducerFun, reducerParams): Constructs a RDApplyParams object with each setting specified by the argument of the same name. See the Details section for more information.

Accessors

In the following code snippets, x is an RDApplyParams object.

rangedData(x), rangedData(x) <- value: Get or set the RangedData instance over which applyFun is applied.

applyFun(x), applyFun(x) <- value: Get or set the user function to be applied to each space in the RangedData.

applyParams(x), applyParams(x) <- value: Get or set the list of additional parameters to pass to applyFun.

filterRules(x), filterRules(x) <- value: Get or set the instance of FilterRules that is used to filter each subset of the RangedData passed to the user function.

simplify(x), simplify(x) <- value: Get or set a a scalar logical (TRUE or FALSE) indicating whether the list to be returned from rdapply should be simplified as by sapply.

reducerFun(x), reducerFun(x) <- value: Get or set the function that is used to convert the list that would otherwise be returned from rdapply to something more convenient.

reducerParams(x), reducerParams(x) <- value: Get or set a list of additional parameters to pass to reducerFun.

Author(s)

Michael Lawrence

See Also

RangedData, FilterRules

Examples

```
ranges <- IRanges(c(1,2,3),c(4,5,6))
score <- c(2L, 0L, 1L)
rd <- RangedData(ranges, score, splitter = c("chr1","chr2","chr1"))

## a single function
countrows <- function(rd) nrow(rd)
params <- RDApplyParams(rd, countrows)
rdapply(params) # list(chr1 = 2L, chr2 = 1L)

## with a parameter
params <- RDApplyParams(rd, function(rd, x) nrow(rd)*x, list(x = 2))
rdapply(params) # list(chr1 = 4L, chr2 = 2L)

## add a filter
cutoff <- 0
rules <- FilterRules(filter = score > cutoff)
params <- RDApplyParams(rd, countrows, filterRules = rules)
rdapply(params) # list(chr1 = 2L, chr2 = 0L)
rules <- FilterRules(list(fun = function(rd) rd[["score"]] < 2),
                     filter = score > cutoff)
params <- RDApplyParams(rd, countrows, filterRules = rules)
rdapply(params) # list(chr1 = 1L, chr2 = 0L)
active(filterRules(params))["filter"] <- FALSE
rdapply(params) # list(chr1 = 1L, chr2 = 1L)
```

```
## simplify
params <- RDApplyParams(rd, countrows, simplify = TRUE)
rdapply(params) # c(chr1 = 2L, chr2 = 1L)

## reducing
params <- RDApplyParams(rd, countrows, reducerFun = unlist,
                        reducerParams = list(use.names = FALSE))
rdapply(params) ## c(2L, 1L)
```

`read.Mask` *Read a mask from a file*

Description

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

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

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

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

Usage

```
read.agpMask(file, seqname="?", mask.width=NA, gap.types=NULL, use.gap.types=F
read.gapMask(file, seqname="?", mask.width=NA, gap.types=NULL, use.gap.types=F
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 ranges*

Description

Reverses a set of ranges.

Usage

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

Arguments

x	An IRanges, NormalIRanges, or MaskCollection object.
...	Additional arguments to be passed to or from methods.

Details

Reverses the order of the ranges.

Value

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

See Also

IRanges-class, NormalIRanges-class, MaskCollection-class

Examples

```
x <- IRanges(start=c(-2L, 6L, 9L, -4L, 1L, 0L, -6L, 10L),
             width=c( 5L, 0L, 6L,  1L, 4L, 3L,  2L,  3L))
reverse(x, start=-6, end=20)  # 'start' and 'end' must be specified for
                              # an IRanges object.
reverse(shift(x, 2), start=-6, end=20)
reverse(restrict(x, 1, 10), start=-6, end=20)
reverse(reduce(x), start=-6, end=20)
reverse(gaps(x, start=-6, end=20), start=-6, end=20)

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

Rle-class *Rle objects*

Description

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

Constructors

`Rle(values)`: This constructor creates an Rle instances out of an atomic vector `values`.

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

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

Accessors

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

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

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

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

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

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

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

Replacers

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

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

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

Coercion

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

`as.vector(x)`, `as(from, "vector")`: Creates an atomic vector of the most appropriate type based on the values contained in `x`.

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

`as.integer(x)`, `as(from, "integer")`: Creates an integer vector based on the values contained in `x`.

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

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

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

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

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

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

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

Group Generics

Rle objects have support for S4 group generic functionality:

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

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

Logic "&", "|"

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

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

Math2 "round", "signif"

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

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

See S4groupGeneric for more details.

General Methods

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

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

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

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

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

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

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

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

... Further arguments for `FUN`.

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

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

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

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

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

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

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

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

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

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

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

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

shiftApply(SHIFT, X, Y, FUN, ..., OFFSET = 0L, simplify = TRUE, verbose = FALSE): Let i be the indices in SHIFT, X_i = subseq(X, 1 + OFFSET, length(X) - SHIFT[i]), and Y_i = subseq(Y, 1 + SHIFT[i], length(Y) - OFFSET). Calculates the set of FUN(X_i, Y_i, ...) values and return the results in a convenient form:

SHIFT A non-negative integer vector of shift values.

X, Y The Rle objects to shift.

FUN The function, found via match.fun, to be applied to each set of shifted vectors.

... Further arguments for FUN.

OFFSET A non-negative integer offset to maintain throughout the shift operations.

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

verbose A logical value specifying whether or not to print the i indices to track the iterations.

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

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

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

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

subseq(x, start = NA, end = NA, width = NA): Extract the subsequence from x specified by two of the three following values: start, end, and width. This is more efficient for extracting consecutive values than [.

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

table(...): Returns a table containing the counts of the unique values.

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

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

window(x, start = NULL, end = NULL, width = NULL, frequency = NULL, delta = NULL, ...): Extract the subsequence window from x specified by:

start, end, width The start, end, or width of the window. Two of the three are required.

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

Logical Data Methods

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

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

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

Numerical Data Methods

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

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

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

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

lag An integer indicating which lag to use.

differences An integer indicating the order of the difference.

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

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

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

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

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

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

probs A numeric vector of probabilities with values in [0,1].

na.rm If TRUE, removes NAs from x before the quantiles are computed.

names If TRUE, the result has names describing the quantiles.

type An integer between 1 and 9 selecting one of the nine quantile algorithms detailed in quantile.

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

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

center The center to calculate the deviation from.

constant The scale factor.

na.rm If TRUE, removes NAs from x before the mad is computed.

low If TRUE, compute the 'lo-median'.

high If TRUE, compute the 'hi-median'.

Character Data Methods

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

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

type One of `c("bytes", "chars", "width")`.

allowNA Should NA be returned for invalid multibyte strings rather than throwing an error?

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

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

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

`new` A character string specifying the translations.

`tolower(x)`: Returns a lower case version of `x`.

`toupper(x)`: Returns an upper case version of `x`.

`sub(pattern, replacement, x, ignore.case = FALSE, extended = TRUE, perl = FALSE, fixed = FALSE, useBytes = FALSE)`: Returns a character Rle with replacements based on matches determined by regular expression matching. See `sub` for a description of the arguments.

`gsub(pattern, replacement, x, ignore.case = FALSE, extended = TRUE, perl = FALSE, fixed = FALSE, useBytes = FALSE)`: Returns a character Rle with replacements based on matches determined by regular expression matching. See `gsub` for a description of the arguments.

Author(s)

P. Aboyoun

See Also

`rle`, Sequence-class, S4groupGeneric, IRanges-class

Examples

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

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

diff(x)
unique(x)
sort(x)
sqrt(x)
x^2 + 2 * x + 1
x[c(1,3,5,7,9)]
subseq(x, 4, 14)
range(x)
table(x)
sum(x)
mean(x)
x > 4
aggregate(x, x > 4, mean)
aggregate(x, FUN = mean, start = 1:(length(x) - 50), end = 51:length(x))

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

```
c(y, x > 5)

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

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, Views-utils

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)
```

`Sequence-class` *Sequence objects*

Description

The Sequence virtual class is a general container for storing a sequence i.e. an ordered set of elements. These containers come in three types: XSequence, XRle [DEPRECATED], and Rle.

The XSequence virtual class is a general container for storing an "external sequence". The following classes derive directly from the XSequence class.

The XRaw class is a container for storing an external sequence of bytes (stored as char values at the C level).

The XInteger class is a container for storing an external sequence of integer values (stored as int values at the C level).

The XNumeric class is a container for storing an external sequence of numeric values (stored as double values at the C level).

Also the XString class from the Biostrings package

The XRle [DEPRECATED – use Rle] virtual class is a general container for storing an "external sequence" that is stored in a run-length encoding format. The following classes derive directly from the XRle class.

The XRleInteger [DEPRECATED – use Rle] class is a container for storing an external run-length encoding of integers (stored as int values at the C level).

The purpose of the X* containers is to provide a "pass by address" semantic and also to avoid the overhead of copying the sequence data when a linear subsequence needs to be extracted.

For information on the Rle class, type `help(Rle)`.

Combining

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

`c(x, ...)`: Combine `x` and the Sequence objects in `...` together. Any object in `...` must belong to the same class as `x`, or to one of its subclasses, or must be `NULL`. The result is an object of the same class as `x`. NOTE: Only works for XRaw (and derived) objects for now.

Subsetting

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

`subseq(x, start=NA, end=NA, width=NA)`: Extract the subsequence from `x` specified by `start`, `end` and `width`. The supplied start/end/width values are solved by a call to `solveUserSEW(length(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).

A note about performance: `subseq` does NOT copy the sequence data of an XSequence object. Hence it's very efficient and is therefore the recommended way to extract a linear subsequence (i.e. a set of consecutive elements) from an XSequence object. For example, extracting a 100Mb subsequence from Human chromosome 1 (a 250Mb DNAString object) with `subseq` is (almost) instantaneous and has (almost) no memory footprint (the cost in time and memory does not depend on the length of the original sequence or on the length of the subsequence to extract).

`subseq(x, start=NA, end=NA, width=NA) <- value`: Replace the subsequence specified on the left (i.e. the subsequence in `x` specified by `start`, `end` and `width`) by `value`. `value` must belong to the same class as `x`, or to one of its subclasses, or must be `NULL`. This replacement method can modify the length of `x`, depending on how the length of the left subsequence compares to the length of `value`. It can be used for inserting elements in `x` (specify an empty left subsequence for this) or deleting elements from `x` (use a `NULL` rigth value for this). Unlike the extraction method above, this replacement method always copies the sequence data of `x` (even for XSequence objects). NOTE: Only works for XRaw (and derived) objects for now.

`x[i, drop=TRUE]`: Return a new Sequence object made of the selected elements (subscript `i` must be an NA-free numeric vector specifying the positions of the elements to select). The `drop` argument specifies whether or not to coerce the returned sequence to a standard vector.

`rep(x, times)`: Return a new Sequence object made of the repeated elements.

See Also

Rle-class, Views-class, `solveUserSEW`, DNAString-class

Examples

```
## ---------------------------------------------------------------------
## A. XRaw OBJECTS
## ---------------------------------------------------------------------

x1 <- XRaw(4)  # values are not initialized
x1
x2 <- as(c(255, 255, 199), "XRaw")
x2
y <- c(x1, x2, NULL, x1)  # NULLs are ignored
y
subseq(y, start=-4)
subseq(y, start=-4) <- x2
y

## ---------------------------------------------------------------------
## B. XInteger OBJECTS
## ---------------------------------------------------------------------

x3 <- XInteger(12, val=c(-1:10))
x3
length(x3)

## Subsetting
x4 <- XInteger(99999, val=sample(99, 99999, replace=TRUE) - 50)
x4
subseq(x4, start=10)
subseq(x4, start=-10)
subseq(x4, start=-20, end=-10)
subseq(x4, start=10, width=5)
subseq(x4, end=10, width=5)
subseq(x4, end=10, width=0)

x3[length(x3):1]
x3[length(x3):1, drop=FALSE]
```

```
x5 <- LETTERS
subseq(x5, 8, 11) <- x5[11:8]  # swap 8<->11 and 9<->10 elements
subseq(x5, start=1, width=0) <- c("xx", "yyy")  # insert 2 elements at the beginning
subseq(x5, end=-1, width=0) <- letters[1:3]  # insert 3 elements at the end
subseq(x5, end=-4, width=5) <- NULL  # remove 5 elements before the just added ones
```

`TypedList-class` *Typed Lists*

Description

The virtual class `TypedList` is an emulation of an ordinary `list`, except all of the elements must derive from a particular type. This is useful for validity checking and for implementing vectorized type-specific operations.

Details

In general, a `TypedList` may be treated as any ordinary `list`, except with regard to the element type restriction.

The required element type is indicated by the `elementClass` slot, a scalar string naming the class from which all elements must derive. This slot should never be set after initialization.

`TypedList` is a virtual class, so a subclass must be derived for a particular element type. This turns out to be useful in almost all cases, as the explicit class can be used as the type of a slot in a class that requires a homogeneous list of elements. Also, methods may be implemented for the subclass that, for example, perform a vectorized operation specific to the element type. Using this approach, the convention is for the prototype of the subclass to set the `elementClass` slot and to leave it unchanged.

Object Updating

updateTypedObject(object): Updates instances of objects the inherit from an older TypedList class definition to match the current TypedList class definition.

Subsetting

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

`x[i]`: Get a subset of `x` containing the elements indexed by `i`, which may be numeric, character, logical, `NULL` or missing. The behavior is very similar to an ordinary `list`, except operations that would insert `NULL` elements are only allowed if `NULL` is a valid element type.

`x[[i]]`: Get the element in `x` indexed by `i`, which may be a scalar number or string. The behavior is nearly identical to that of an ordinary `list`.

`x$name`: similar to above, where `name` is taken literally as an element name.

`x[[i]] <- value`: Replace the element at index `i` (a scalar number or string) with `value`. The behavior is very similar to that of an ordinary `list`, except `value` must be coercible (and is coerced) to the required element class.

`x$name <- value`: similar to above, where `name` is taken literally as an element name.

Accessors

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

- length(x): Get the number of elements in x
- names(x), names(x) <- value: Get or set the names of the elements in the list. This behaves exactly the same as an ordinary list.
- elementClass(x): Get the scalar string naming the class from which all elements must derive.
- elementLengths(x): Get the 'length' of each of the elements.
- isEmpty(x): Gets a logical vector indicating which elements are empty.

Splitting and Combining

The following are methods for combining TypedList elements. In the signatures, x is a TypedList object.

- append(x, values, after = length(x)): Insert the TypedList values onto x at the position given by after. values must have an elementClass that extends that of x.
- c(x, ..., recursive = FALSE): Appends the TypedList objects in ... onto the end of x. All arguments must have an element class that extends that of x.

Note that the default split method happens to work on TypedList objects.

Coercion

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

- as.list(x), as(from, "list"): Coerces a TypedList to an ordinary list. Note that this is preferred over the elements accessor for getting a list of the elements.
- unlist(x): Combines all of the elements in this list into a single element via the c function and returns the result. Will not work if the elements have no method for c. Returns NULL if there are no elements in x, which may not be what is expected in many cases. Subclasses should implement their own logic.

Looping

- lapply(X, FUN, ...): Like the standard lapply function defined in the base package, the lapply method for TypedLike objects returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.
- sapply(X, FUN, ..., simplify=TRUE, USE.NAMES=TRUE): Like the standard sapply function defined in the base package, the sapply method for TypedList objects is a user-friendly version of lapply by default returning a vector or matrix if appropriate.

Author(s)

Michael Lawrence

See Also

ListLike, RangesList for an example implementation

Examples

```
## demonstrated on IntegerList objects, as TypedList is virtual

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

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

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

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

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

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

## get the mean for each element
lapply(col1, mean)
```

`Views-class` *Views objects*

Description

The Views virtual class is a general container for storing a set of views on an arbitrary Sequence 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: XIntegerViews, RleViews, 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 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 IRanges object. This ensures that `Views(subject, IRanges(mystarts, myends, mywidths, mynames))` and `Views(subject, mystarts, myends, mywidths, mynames)` are equivalent (except when `mystarts` is itself an IRanges object).

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 and appending

The `"["` and `c` methods defined for `IRanges` objects work on Views objects and return a Views object. In addition, the `"[["` operator is defined for Views objects:

`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 `subseq(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))`).

Other methods

`restrict(x, start, end, keep.all.ranges=FALSE, use.names=TRUE)`: `start` and `end` must be single integers specifying the restriction window. `restrict` will drop the views that don't overlap with the restriction window and drop the parts of the remaining views that are outside the window.

`trim(x, use.names=TRUE)`: [TODO]

`narrow(x, start=NA, end=NA, width=NA, use.names=TRUE)`: [TODO]

`subviews(x, start=NA, end=NA, width=NA, use.names=TRUE)`: [TODO]

`gaps(x, start=NA, end=NA)`: `start` and `end` can be single integers or NAs. The gap extraction will be restricted to the window specified by `start` and `end`. `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.

`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)

H. Pages

See Also

IRanges-class, ListLike-class, IRanges-utils, Sequence, XSequence.

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

Examples

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

## Create a set of 4 views on an XInteger subject of length 10:
subject <- XInteger(10, 3:-6)
v1 <- Views(subject, start=4:1, end=4:7)

## Extract the 2nd view:
v1[[2]]

## Some views can be "out of limits"
v2 <- Views(subject, start=4:-1, end=6)
trim(v2)
subviews(v2, end=-2)

## gaps()
v3 <- Views(subject, start=c(8, 3), end=c(14, 4))
gaps(v3)

## Views on a big XInteger subject:
subject <- XInteger(99999, sample(99, 99999, replace=TRUE) - 50)
v4 <- Views(subject, start=1:99*1000, end=1:99*1001)
v4
v4[-1]
v4[[5]]

## 31 adjacent views:
successiveViews(subject, 40:10)
```

Views-utils *Utility functions and numeric summary of Views of numerics*

Description

The `slice` function creates a Views object that contains the indices where the data are within the specified bounds.

The `viewMins`, `viewMaxs`, `viewSums` functions calculate the minimums, maximums, and sums on views respectively.

Usage

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

slice(x, lower=-Inf, upper=Inf, ...)
## S4 method for signature 'Rle':
slice(x, lower=-Inf, upper=Inf,
                      includeLower=TRUE, includeUpper=TRUE)

viewMins(x, na.rm=FALSE)
viewMaxs(x, na.rm=FALSE)
viewSums(x, na.rm=FALSE)

viewWhichMins(x, na.rm=FALSE)
viewWhichMaxs(x, na.rm=FALSE)

viewRangeMins(x, ...)
viewRangeMaxs(x, ...)
```

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 Rle, XRleInteger, XInteger object or an integer vector for slice. An RleViews, XRleIntegerViews, XIntegerViews object for viewMins, viewMaxs, and viewSums. An Rle, XRleIntegerViews, XIntegerViews object for viewWhichMins and viewWhichMaxs.
lower, upper	The lower and upper bounds for the slice.
includeLower, includeUpper	Logical indicating whether or not the specified boundary is open or closed.
na.rm	Logical indicating whether or not to include missing values in the results.

Details

The slice function creates views on XRleInteger or XInteger objects where the data are within the specified bounds. This is useful for finding areas of absolute maxima (peaks), absolute minima (troughs), or fluxuations within a specified limits.

The viewMins, viewMaxs, and viewSums 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 location of the minimums and maximums.

Value

An XRleIntegerViews or XIntegerViews object for slice.

A vector of length(x) containing the numeric summaries for the views for viewMins, viewMaxs, viewSums, viewWhichMins, and viewWhichMaxs.

A IRanges object containing the location ranges for `viewRangeMins` and `viewRangeMaxs`.

Author(s)

P. Aboyoun

See Also

RleViews-class, XRleIntegerViews-class, XIntegerViews-class, `which.min`, `colSums`

Examples

```
## Views derived from vector
vec <- as.integer(c(19, 5, 0, 8, 5))
slice(vec, lower=5, upper=8)

set.seed(0)
vec <- sample(24)
vecViews <- slice(vec, lower=4, upper=16)
vecViews
viewApply(vecViews, function(x) diff(as.integer(x)))
viewMins(vecViews)
viewMaxs(vecViews)
viewSums(vecViews)
viewWhichMins(vecViews)
viewWhichMaxs(vecViews)

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

XDataFrame-class *External Data Frame*

Description

The `XDataFrame` emulates the interface of `data.frame`, but it supports the storage of any type of object as a column, as long as the `length` and `[` methods are implemented. The "X" in its name indicates that it attempts to coerce its columns to external `XSequence` objects in a way that is completely transparent to the user. This helps to avoid unncessary copying.

Details

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

As `XDataFrame` derives from `AnnotatedList`, it is possible to set an `annotaiton` string. Also, another `XDataFrame` can hold metadata on the columns.

Accessors

In the following code snippets, x is an XDataFrame.

dim(x): Get the length two integer vector indicating in the first and second element the number of rows and columns, respectively.

dimnames(x), dimnames(x) <- value: Get and set the two element list containing the row names (character vector of length nrow(x) or NULL) and the column names (character vector of length ncol(x)).

Subsetting

In the following code snippets, x is an XDataFrame.

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

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

x[[i]] <- value: Behaves very similarly to the [[<-.data.frame method, except the argument j is not supported. An attempt is made to coerce value to a XSequence object.

Constructor

XDataFrame(..., row.names = NULL): Constructs an XDataFrame in similar fashion to data.frame. Each argument in ... is coerced to an XDataFrame and combined column-wise. No special effort is expended to automatically determine the row names from the arguments. The row names should be given in row.names; otherwise, there are no row names. This is by design, as row names are normally undesirable when data is large.

Splitting and Combining

In the following code snippets, x is an XDataFrame.

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

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

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

Coercion

as(from, "XDataFrame"): By default, constructs a new XDataFrame with from as its only column. If from is a matrix or data.frame, all of its columns become columns in the new XDataFrame. In any case, there is an attempt to coerce columns to XSequence before inserting them into the XDataFrame. If from is a list, its elements become columns in the same way. Note that for the XDataFrame to behave correctly, each column object

must support element-wise subsetting via the `[` method and return the number of elements with `length`. It is recommended to use the `XDataFrame` constructor, rather than this interface.

`as.list(x)`: Coerces `x`, an `XDataFrame`, to a `list`, converting any `XSequence` objects to vectors along the way.

`as.data.frame(x, row.names=NULL, optional=FALSE)`: Coerces `x`, an `XDataFrame`, to a `data.frame`. Each column is coerced to a `vector` and stored as a column in the `data.frame`. If `row.names` is `NULL`, they are retrieved from `x`, if it has any. Otherwise, they are inferred by the `data.frame` constructor.

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

Note

In the future, the general data frame functionality will probably be moved to a `DataFrame` class. `XDataFrame` will derive from `DataFrame` and encapsulate the behavior of attempting to coerce or even requiring columns to be `XSequence`.

Author(s)

Michael Lawrence

See Also

`RangedData`, which makes heavy use of this class.

Examples

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

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

xdf <- XDataFrame(vals = score) # explicit naming
xdf[["vals"]]

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

# subsetting

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

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

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

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

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

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

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

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

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

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

## column replacement

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

## split

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

## rbind

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

## cbind

cbind(XDataFrame(score), XDataFrame(counts))
```

XDataFrameList-class
List of XDataFrames

Description

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

Accessors

In the following code snippets, x is a XDataFrameList.

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

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

Constructor

XDataFrameList(...): Concatenates the XDataFrame objects in ... into a new XDataFrameList.

SplitXDataFrameList(...): Concatenates the XDataFrame objects in ... into a new SplitXDataFrameList. Note that all arguments must have the same number and names of columns.

Coercion

In the following code snippets, x is a SplitXDataFrameList.

as(from, "XDataFrame"): Coerces a XDataFrameList to an XDataFrame by combining the rows of the elements. This essentially unsplits the XDataFrame.

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

Note

The RangedData drove the development of these classes. It is not clear if they are of general use and might disappear.

Author(s)

Michael Lawrence

See Also

XDataFrame, RangedData, which uses a XDataFrameList to split the data by the spaces.

XDataFrame-utils *XDataFrame utility functions*

Description

Utility functions for manipulating XDataFrame objects.

Usage

```
## S4 method for signature 'expressionORlanguage,
##   XDataFrame':
eval(expr, envir, enclos = parent.frame())
```

Arguments

expr	The expression, call, or name to be evaluated.
envir	The XDataFrame object in which to evaluate expr.
enclos	The environment in which to look for symbols that do not exist in the environment formed from XDataFrame.

Details

The eval method converts the XDataFrame object specified in envir to an environmnent, with enclos as its parent, and then evaluates expr within that environment. As when evaluating within an ordinary data.frame, the environment formed from an XDataFrame contains a symbol for each column name which refers to the object stored in that column.

The objects are not actually copied into the environment. Rather, they are dynamically bound using makeActiveBinding. This prevents unnecessary copying of the data from the external vectors into R vectors. The values are cached, so that the data is not copied every time the symbol is accessed.

Value

The result of expression evaluation.

Author(s)

Michael Lawrence

See Also

FilterRules objects, which can be evaluated on a XDataFrame, and the base eval function.

Examples

```
score <- c(10L, 2L, NA)
rd <- XDataFrame(score)
eval(quote(score > 3), rd)
```

`XIntegerViews-class` *The XIntegerViews class*

Description

The XIntegerViews class is the basic container for storing a set of views (start/end locations) on the same XInteger object.

Details

An XIntegerViews object contains a set of views (start/end locations) on the same XInteger object called "the subject integer vector" or simply "the subject". Each view is defined by its start and end locations: both are integers such that start <= end. An XIntegerViews object is in fact a particular case of a Views object (the XIntegerViews 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.

Other methods

In the code snippets below, `x`, `object`, `e1` and `e2` are XIntegerViews objects, and `i` can be a numeric or logical vector.

`x[[i]]`: Extract a view as an XInteger object. `i` must be a single numeric value (a numeric vector of length 1). Can't be used for extracting a view that is "out of limits" (raise an error). The returned object has the same XInteger subtype as `subject(x)`.

`e1 == e2`: A vector of logicals indicating the result of the view by view comparison. The views in the shorter of the two XIntegerViews object being compared are recycled as necessary.

`e1 != e2`: Equivalent to `!(e1 == e2)`.

Author(s)

P. Aboyoun

See Also

Views-class, XInteger-class, Views-utils

Examples

```
## One standard way to create an XIntegerViews object is to use
## the Views() constructor:
subject <- as(c(45, 67, 84, 67, 45, 78), "XInteger")
v4 <- Views(subject, start=3:0, end=5:8)
v4
subject(v4)
length(v4)
start(v4)
end(v4)
width(v4)
```

```
## Attach a comment to views #3 and #4:
names(v4)[3:4] <- "out of limits"
names(v4)

## A more programatical way to "tag" the "out of limits" views:
names(v4)[start(v4) < 1 | length(subject(v4)) < end(v4)] <- "out of limits"
## or just:
names(v4)[length(subject(v4)) < width(v4)] <- "out of limits"

## Extract a view as an XInteger object:
v4[[2]]

## It is an error to try to extract an "out of limits" view:
#v4[[3]] # Error!

## Here the first view doesn't even overlap with the subject:
subject <- as(c(97, 97, 97, 45, 45, 98), "XInteger")
Views(subject, start=-3:4, end=-3:4 + c(3:6, 6:3))
```

XRanges-class *External Ranges*

Description

The XRanges class is meant to be the virtual parent for all Ranges derivatives that exist externally from R, such as search trees, databases, etc. It is the external analog of the internal IRanges.

Details

The primary requirement for a XRanges implementation is that it is coercible to IRanges, so that the data may be imported into R. Several of the most important accessors (start, end, width) and utilities (reduce, gaps) have default implementations for XRanges objects that simply coerce the XRanges to an IRanges and delegate. Subclasses are responsible for optimized implementations of those methods and should generally attempt to implement as much of the Ranges API as is feasible.

Author(s)

Michael Lawrence

See Also

The internal IRanges; IntervalTree for an implementation.

XRleIntegerViews-class *The XRleIntegerViews class*

Description

The XRleIntegerViews class is the basic container for storing a set of views (start/end locations) on the same XRleInteger object.

Details

An XRleIntegerViews object contains a set of views (start/end locations) on the same XRleInteger object called "the subject integer vector" or simply "the subject". Each view is defined by its start and end locations: both are integers such that start <= end. An XRleIntegerViews object is in fact a particular case of a Views object (the XRleIntegerViews 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.

Other methods

In the code snippets below, `x`, `object`, `e1` and `e2` are XRleIntegerViews objects, and `i` can be a numeric or logical vector.

`x[[i]]`: Extract a view as an XRleInteger object. `i` must be a single numeric value (a numeric vector of length 1). Can't be used for extracting a view that is "out of limits" (raise an error). The returned object has the same XRleInteger subtype as `subject(x)`.

`e1 == e2`: A vector of logicals indicating the result of the view by view comparison. The views in the shorter of the two XRleIntegerViews object being compared are recycled as necessary.

`e1 != e2`: Equivalent to `!(e1 == e2)`.

Author(s)

P. Aboyoun

See Also

Views-class, XRleInteger-class, Views-utils

Examples

```
## Not run:
  ## One standard way to create an XIntegerViews object is to use
  ## the Views() constructor:
  subject <- XRleInteger(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]]

  ## Here the first view doesn't even overlap with the subject:
  Views(XRleInteger(as.integer(c(97, 97, 97, 45, 45, 98))), -3:4, -3:4 + c(3:6, 6:3))
## End(Not run)
```

Index