IRanges

April 19, 2009

`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.

Accesor 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)
```

`GenomicData-class` *Data on a Genome*

Description

`GenomicData` extends `RangedData` to more conveniently manipulate data on genomic ranges. The spaces are now called chromosomes (but could still refer to some other type of sequence). The annotation refers to the genome and there is formal treatment of an optional `strand` variable.

Accessors

The `GenomicData` class essentially adds a set of convenience accessors on top of `RangedData`. In the code snippets below, `x` is a `RangedData` object.

`chrom(x)`: Gets the chromosome names (a factor) over the ranges in `x`.

`genome(x)`: Gets the genome for the ranges in `x`; a simple wrapper around `annotation(x)`.

`strand(x)`: Gets the strand factor in `x`, or, if `x` is missing, return an empty factor with the standard levels: `-`, `+` and `*`, referring to the negative, positive and both strands, respectively. Any strand factor stored in `GenomicData` should have those levels. `NA`'s are allowed; the value is all `NA` if no strand has been specified.

Constructor

`GenomicData(ranges, ..., strand = NULL, chrom = NULL, genome = NULL)`: Constructs a `GenomicData` instance with the given `ranges` and variables in `...` (see the `RangedData` constructor). If non-`NULL`, `strand` specifies the strand of each range. It should be a character vector or factor of length equal to that of `ranges`. All values should be either `-`, `+`, `*` or `NA`. To get these levels, call `levels(strand())`. `chrom` is analogous to `splitter` in `RangedData`; if non-`NULL` it should be coercible to a factor indicating how the ranges, variables and strand should be split up across the chromosomes. The `genome` argument should be a scalar string and is treated as the `RangedData` annotation. See the examples.

Coercion

`as(from, "GenomicData")`: coerces `from` to a `GenomicData`, according to its class:

XRle The bounds of the runs become the ranges and the values become a column named `score`.

Note

This may move to another package in the future, as it is not quite general enough for IRanges.

Author(s)

Michael Lawrence

See Also

`RangedData`, on which this class is based.

Examples

```
range1 <- IRanges(start=c(1,2,3), end=c(5,2,8))

## just ranges
gr <- GenomicData(range1)

## with a genome (annotation)
gr <- GenomicData(range1, genome = "hg18")
genome(gr) ## "hg18"

## with some data
filter <- c(1L, 0L, 1L)
score <- c(10L, 2L, NA)
strand <- factor(c("+", NA, "-"), levels = levels(strand()))
gr <- GenomicData(range1, score, genome = "hg18")
gr[["score"]]
```

```
strand(gr) ## all NA
gr <- GenomicData(range1, score, filt = filter, strand = strand)
gr[["filt"]]
strand(gr) ## equal to 'strand'

range2 <- IRanges(start=c(15,45,20,1), end=c(15,100,80,5))
ranges <- c(range1, range2)
score <- c(score, c(0L, 3L, NA, 22L))
chrom <- paste("chr", rep(c(1,2), c(length(range1), length(range2))), sep="")

gr <- GenomicData(ranges, score, chrom = chrom, genome = "hg18")
chrom(gr) # equal to 'chrom'
gr[["score"]] # unlists over the chromosomes
gr[1][["score"]] # equal to score[1:3]
```

`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).

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

`IRanges(start=NULL, end=NULL, width=NULL)`: Return the IRanges object containing the ranges specified by `start`, `end` and `width`. Exactly two of the `start`, `end` and `width` arguments must be specified as integer vectors (with no `NA`s) and the other argument must be `NULL`. If `start` and `end` are specified, then they must be vectors of the same length. If `start` and `width` (or `end` and `width`) are specified, then the length of `width` must be <= to the length of `start` and, if it is <, then `width` is expanded cyclically to the length of `start`.

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-utils, IRanges-setops

Examples

```
## 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-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, ...)
```

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 instance representing the union, intersection and (asymmetric!) difference of the sets of integers represented by `x` and `y`.

`punion`, `pintersect` and `psetdiff` are generic functions that perform the element-wise (aka "parallel") union, intersection and (asymmetric!) difference of `x` and `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 of 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.

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 <- IRanges(c(14, 0, -5, 6, 18), c(20, 2, 2, 8, 20))

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

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

## 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`, a vector of integers, eventually with NAs. See the SEW (Start/End/Width) interface for the details (`?solveUserSEW`).
`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`, `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`. For `narrow`, 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.
`keep.all.ranges`	`TRUE` or `FALSE`. Should ranges that become "out of limits" after restriction be kept?
`with.inframe.attrib`	`TRUE` or `FALSE`. For internal use.
`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.

`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 obtained by removing the set of integers described by `x` from the interval specified by the `start` and `end` arguments.

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

Author(s)

H. Pages

See Also

Ranges-class, IRanges-class, 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)
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))

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'.
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)
```

IRangesList-class *List of IRanges*

Description

A RangesList that only stores IRanges instances.

Constructor

`IRangesList(...)`: 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`.

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
```

`IntervalTree-class` *Interval Search Trees*

Description

An `IntervalTree` instance 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` instance.

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` instance, 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. 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.

`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` instances. 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` instance 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, which is an assertion of the `overlap` method.

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)
```

`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 accesor 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]]
```

`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)
```

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 SplitXDataFrame, holding the split data. There is also an annotation 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.

Accesor 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)`: Gets the names of the ranges in `x`.
- `colnames(x)`: Gets the names of the variables in `x`.
- `dimnames(x)`: A list with two elements, essentially `list(rownames(x), colnames(x))`.

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)`: The names of the spaces (e.g. `"chr1"`). `NULL` or a character vector of the same length as `x`.
- `names(x) <- value`: Set the names of the spaces, where `value` is either `NULL` or a character vector of the same length as `x`.

Other accessors:

- `annotation(object)`: Here, `object` is a `RangedData` object. Get the scalar string identifying the source of the data in some way (e.g. genome, experimental platform, etc).
- `ranges(x)`: Gets the ranges in `x` as a `RangesList`.
- `values(x)`: Gets the data values in `x` as a `SplitXDataFrame`.

Constructor

`RangedData(ranges = IRanges(), ..., splitter = NULL, annotation = 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 annotation may be specified as a scalar string by the `annotation` 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")`: coerces `from` to a `RangedData`, according to its class:

  XRle The bounds of the runs become the ranges and the values become 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[[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.

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.

`c(x, ..., recursive = FALSE)`: Combines `x` with arguments specified in `...`, which must all be `RangedData` instances. 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` instances on a space-by-space basis and then needing to combine them.

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[["filter"]]
rd <- RangedData(ranges, score + score)
rd[["score...score"]] # names made valid
## use an annotation
rd <- RangedData(ranges, annotation = "hg18")
annotation(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, splitter = chrom, annotation = "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

## variable replacement

count <- c(1L, 0L, 2L)
rd <- RangedData(ranges, count, splitter = 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

## combining/splitting

rd <- RangedData(ranges, score, splitter = 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'
```

`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())
```

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`.

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.

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)
evalq(score > 3, rd)
```

`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(...)`: Contatenates the `RangedData` instances 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)
```

`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.

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).

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`

`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.

`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.

`duplicated(x)`: Determine 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` for more information).

`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, ...)`: Concatenate `x` and the Ranges objects in `...` together. The result is returned as an IRanges object.

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

See Also

IRanges-class, IRanges-utils, IRanges-setops, XRanges-class, RangedData-class, IntervalTree-class, `update`, `as.matrix`, `as.data.frame`, `duplicated`, `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
duplicated(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
```

`RangesList-class` *List of Ranges*

Description

An extension of `TypedList` that holds only `Ranges` instances. Useful for storing ranges over a set of spaces (e.g. chromosomes), each of which requires a separate `Ranges` instance.

Accessors

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

`isEmpty(x)`: Gets a logical vector indicating which elements are empty (length zero).

Constructor

`RangesList(...)`: 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`.

Coercion

In the code snippets below, `x` is 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, "IRangesList")`: Coerces `from`, a `RangesList`, 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).

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

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

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

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)
```

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 instance.
end	The end of the range over which to calculate the gaps. If NA, use the maximum end position in the Ranges instance.
with.inframe.attrib	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 instance and returns the result as a length-one RangesList.

Value

A RangesList instance. For gaps, 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))
```

RangesMatching-class

Matchings between Ranges

Description

The `RangesMatching` class stores a set of matchings between the ranges in one `Ranges` instance 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 matchings between the ranges are stored as a `Matrix-class` instance. While that structure is accessible, it is usually more convenient to coerce the `RangesMatching` instance to a more amenable representation.

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

To transpose a `RangesMatrix` `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`.

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)`: Get the `Matrix-class`, which may be a dense logical matrix (`lgeMatrix-class`) or sparse non-zero pattern matrix (`ngCMatrix-class`), that encodes the matchings, with columns corresponding to query ranges and rows corresponding to subject ranges. It is not recommended to work with this matrix directly, unless the coercion methods above are inadequate.

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)

if (interactive()) {
  ## This code seems to not work anymore (because of a change in
  ## the Matrix package?)
  as.table(matchings) # hits per query
  as.table(t(matchings)) # hits per subject
}
```

`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 two types: XSequence and XRle.

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 virtual class is a general container for storing an "external sequence" that is stored in a run-time encoding format. The following classes derive directly from the XRle class.

The XRleInteger class is a container for storing an external run-length encoding of integers (stored as char values at the C level).

The purpose of these 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.

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).

`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

Views-class, `solveUserSEW`, DNAString-class

Examples

```
x1 <- XInteger(12, c(-1:10))
x1
length(x1)

## Subsetting
x2 <- XInteger(99999, sample(99, 99999, replace=TRUE) - 50)
x2
subseq(x2, start=10)
subseq(x2, start=-10)
subseq(x2, start=-20, end=-10)
subseq(x2, start=10, width=5)
subseq(x2, end=10, width=5)
subseq(x2, end=10, width=0)

x1[length(x1):1]
x1[length(x1):1, drop=FALSE]
```

`SplitXDataFrame-class` *Split XDataFrame*

Description

Represents an `XDataFrame` split along some factor. Internally a list of `XDataFrame` instances and extends `TypedList`. Asserts all elements have the same number and names of columns.

Accessors

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

`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

`SplitXDataFrame(...)`: Concatenates the `XDataFrame` instances in `...` into a new `SplitXDataFrame`. Note that all arguments should have the same number and names of columns.

Coercion

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

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

`unlist(x, recursive = TRUE, use.names = TRUE)`: Same as above, except specifying `use.names` to `FALSE` drops the row names. `recursive` is ignored.

`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 this class. It is not clear if it is of general use and might disappear.

Author(s)

Michael Lawrence

See Also

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

`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.

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[[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.

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.

`elements(x)`: Returns the internal `list` holding the elements. It is not recommended to access the elements this way, as for some subclasses of `TypedList` this may be an internal representation that is not consistent with what is extracted with, for example, the `[[` method.

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` instances 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.

Applying

lapply(X, FUN, ...): Applies the function FUN over the TypedList X, with arguments in ... passed on to FUN. Returns a list, with each element resulting from invoking FUN on the corresponding element of X. Same semantics as the default lapply.

Author(s)

Michael Lawrence

See Also

RangesList for an example implementation

Examples

```
## demonstrated on RangesList, as TypedList is virtual

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)

## 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
```

```
## combining
col1 <- RangesList(one = range1, range2)
col2 <- RangesList(two = range2, one = range1)
col3 <- RangesList(range2)
append(col1, col2, 1)
append(col1, col2, -5)
c(col1, col2, col3)

## get the starts of each Ranges
lapply(col1, start)
```

`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".

More specific classes like the XIntegerViews container derive directly from the Views class.

The primary purpose of the Views virtual class is to introduce concepts and provide some facilities shared by the more specific containers.

Usage

```
## Constructor:

  Views(subject, start=NA, end=NA, names=NULL)

## Accessor:

  subject(x)

## View extraction:

  ## S4 method for signature 'Views':
  x[[i, j, ...]]

## Utilities:

  ## S4 method for signature 'Views':
  restrict(x, start, end, keep.all.ranges=FALSE, use.names=TRUE)

  trim(x, use.names=TRUE)

  ## S4 method for signature 'Views':
  narrow(x, start=NA, end=NA, width=NA, use.names=TRUE)

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

  ## S4 method for signature 'Views':
  gaps(x, start=NA, end=NA)
```

```
successiveViews(subject, width, gapwidth=0, from=1)
```

Arguments

`subject`	The Sequence object on which to create the views.
`start, end`	For `Views`, they must be integer vectors (eventually with NAs) specifying the starting and ending positions of the views to create. For `restrict`, they must be single integers specifying the restriction window. For `narrow` and `subviews`, they can be single integers or NAs. For `gaps`, they 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 from the entire subject.
`width`	For `narrow` and `subviews`, can be `NA`, a single integer, or an integer vector of the same length as `x`. For `successiveViews`, must be a vector of positive integers (with no NAs) specifying the widths of the views to create.
`names`	If not `NULL`, the names to assign to the views.
`x`	A Views object.
`i, j, ...`	Only one subscript is allowed (`x[[i]]`).
`keep.all.ranges`	Not supported for Views objects (must be `FALSE`).
`use.names`	`TRUE` or `FALSE`. Should names be preserved?
`gapwidth`	A single integer or an integer vector with one less element than the `width` vector specifying the widths of the gaps separating one view from the next one.
`from`	A single integer specifying the starting position of the first view.

Details

`restrict` will drop the views that don't overlap with the window specified by `start` and `end` and drop the parts of the remaining views that are outside the window.

[TODO: give some details about `trim`]

[TODO: give some details about `subviews`]

`x[[i, exact=TRUE]]` extracts the view selected by `i`. Subscript `i` can be a single integer or a character string. It cannot be used for extracting a view that is "out of limits" (an error will be raised). The returned object belongs to the same class as `subject(x)`.

`successiveViews` returns a Views object containing the views on `subject` that have the widths specified in the `width` vector and are separated by the gaps specified in `gapwidth`. The first view starts at position `from`.

See Also

IRanges-class, IRanges-utils, Sequence, XSequence, XIntegerViews-class, XStringViews-class

Examples

```
## 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]]

## 'start' and 'end' are recycled
Views(subject, 2:1, 4)
Views(subject, 5:7, )
Views(subject, , 5:7)

## Some views can be "out of limits"
v2 <- Views(subject, 4:-1, 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

```
slice(x, lower=-Inf, upper=Inf, ...)

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

Arguments

`x`	An XRleInteger, XInteger object or an integer vector for `slice`. An XRleIntegerViews, XIntegerViews object for `viewMins`, `viewMaxs` and `viewSums`.
`lower, upper`	The lower and upper bounds for the slice.
`na.rm`	Logical indicating whether or not to include missing values in the results.
`...`	Additional arguments to be passed to or from methods.

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.

Value

An XRleIntegerViews or XIntegerViews object for `slice`.

An integer vector of `length(x)` containing the numeric summaries for the views.

Author(s)

P. Aboyoun

See Also

XRleIntegerViews-class, XIntegerViews-class

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
viewMins(vecViews)
viewMaxs(vecViews)
viewSums(vecViews)

## Views derived from coverage
x <- IRanges(start=c(1L, 9L, 4L, 1L, 5L, 10L),
             width=c(5L, 6L, 3L, 4L, 3L,  3L))
coverage(x, start=1, end=15)
```

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).

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 subsetting by matrix indices is not supported. Due to temporary limitations in the subsetting XSequence objects, indices containing NA's are not yet 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 `SplitXDataFrame`, 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` instances 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` instances 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 are placed into the new `XDataFrame`. In either case, there is an attempt to coerce columns to `XSequence` before inserting them into the `XDataFrame`. 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.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))
```

`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)
evalq(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)`.

`as.integer(x, use.names, check.limits)`: Convert `x` to a list of integer vectors of the same length as `x`. Can't be used if `x` has "out of limits" views (raise an error).

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 <- XInteger(6, c(45, 67, 84, 67, 45, 78))
v4 <- Views(subject, 3:0, 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:
Views(XInteger(6, c(97, 97, 97, 45, 45, 98)), -3:4, -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 instances 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)`.

`as.integer(x, use.names, check.limits)`: Convert `x` to a list of integer vectors of the same length as `x`. Can't be used if `x` has "out of limits" views (raise an error).

Author(s)

P. Aboyoun

See Also

Views-class, XRleInteger-class, Views-utils

Examples

```
## 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))
```

`coverage` *Coverage across a set of ranges*

Description

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

Usage

```
coverage(x, start=NA, end=NA, ...)
## S4 method for signature 'IRanges':
coverage(x, start=NA, end=NA, weight=1L)
```

Arguments

`x`	An IRanges or MaskCollection object.
`start`	A single integer specifying the position in `x` where to start the extraction of the coverage.
`end`	A single integer specifying the position in `x` where to end the extraction of the coverage.
`weight`	An integer vector specifying how much each element in `x` counts.
`...`	Further arguments to be passed to or from other methods.

Value

An XRleInteger object representing the coverage of `x` in the interval specified by the `start` and `end` arguments. An integer value called the "coverage" can be associated to each position in `x`, indicating how many times this position is covered by the ranges stored in `x`. Note that the positions in the returned XInteger object are to be interpreted as relative to the interval specified by the `start` and `end` arguments.

See Also

XRleInteger-class, IRanges-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, start=-6, end=20)  # 'start' and 'end' must be specified for
                               # an IRanges object.
coverage(shift(x, 2), start=-6, end=20)
coverage(restrict(x, 1, 10), start=-6, end=20)
coverage(reduce(x), start=-6, end=20)
coverage(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)
coverage(mymasks)
```

`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)
```

`solveUserSEW` *The SEW (Start/End/Width) interface*

Description

`solveUserSEW` is a utility function that solves a set of user-supplied start/end/width values.

Usage

```
solveUserSEW(refwidths, start=NA, end=NA, width=NA,
             translate.nonpositive.coord=TRUE,
             allow.nonnarrowing=FALSE)
```

Arguments

`refwidths`	Vector of non-negative integers containing the reference widths.
`start, end, width`	Vectors of integers, eventually with NAs, containing the set of user-supplied start/end/width values.
`translate.nonpositive.coord, allow.nonnarrowing`	`TRUE` or `FALSE`.

Details

`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.nonpositive.coord` is TRUE then a non-positive 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.

Value

An IRanges object (with the same number of elements as `refwidths`) representing the set of solved start/end/width values, or an error 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

```
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))
```

Index