IRanges

April 19, 2010

`Alignment-class` *Alignments between sequences*

Description

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

`Annotated-class` *Annotated class*

Description

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

Details

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

Accessors

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

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

Author(s)

P. Aboyoun

See Also

`Sequence` for example implementations

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

Description

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

Details

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

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

Constructors

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

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

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

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

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

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

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

Coercion

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

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

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

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

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

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

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

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

`as.data.frame(x, row.names = NULL, optional = FALSE)`, `as(from, "data.frame")`: Creates a `data.frame` object based on the values contained in `x`. Essentially the same as calling `data.frame(space=rep(names(x), elementLengths(x)), as.vector(x))`.

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

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

Group Generics

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

```
Arith "+", "-", "*", "^", "%%", "%/%", "/"
Compare "==", ">", "<", "!=", "<=", ">="
Logic "&", "|"
Ops "Arith", "Compare", "Logic"
Math "abs", "sign", "sqrt", "ceiling", "floor", "trunc", "cummax", "cummin",
    "cumprod", "cumsum", "log", "log10", "log2", "log1p", "acos", "acosh",
    "asin", "asinh", "atan", "atanh", "exp", "expm1", "cos", "cosh", "sin",
    "sinh", "tan", "tanh", "gamma", "lgamma", "digamma", "trigamma"
Math2 "round", "signif"
Summary "max", "min", "range", "prod", "sum", "any", "all"
Complex "Arg", "Conj", "Im", "Mod", "Re"
```

See S4groupGeneric for more details.

Fixed Width Running Window Summaries

RleList objects have support for the following fixed width running window summary methods:

`runmean(x, k, endrule = c("drop", "constant"))`: Calculates the means for fixed width running windows across `x`.

`x` An RleList object containing integer or numeric Rle elements.

`k` An integer indicating the fixed width of the running window. Must be odd when `endrule == "constant"`.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

`"drop"` do not extend the running statistics to be the same length as the underlying vectors;

`"constant"` copies running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

`runmed(x, k, endrule = c("median", "keep", "constant"))`: Calculates the medians for fixed width running windows across `x`.

`x` An RleList object containing integer or numeric Rle elements.

`k` An integer indicating the fixed width of the running window. Must be odd when `endrule != "drop"`.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

`"drop"` do not extend the running statistics to be the same length as the underlying vectors;

`"keep"` keeps the first and last k_2 values at both ends, where k_2 is the half-bandwidth `k2 = k %/% 2`, i.e., `y[[i]][j] = x[[i]][j]` for $j \in \{1, \ldots, k_2; n - k_2 + 1, \ldots, n\}$ j = 1,..,k2 and (n-k2+1),..,n;

`"constant"` copies the running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

`"median"` the default, smooths the ends by using symmetrical medians of subsequently smaller bandwidth, but for the very first and last value where Tukey's robust end-point rule is applied, see `smoothEnds`.

`runsum(x, k, endrule = c("drop", "constant"))`: Calculates the sums for fixed width running windows across `x`.

`x` An RleList object containing integer or numeric Rle elements.

`k` An integer indicating the fixed width of the running window. Must be odd when `endrule == "constant"`.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

`"drop"` do not extend the running statistics to be the same length as the underlying vectors;

`"constant"` copies running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

`runwtsum(x, k, wt, endrule = c("drop", "constant"))`: Calculates the sums for fixed width running windows across `x`.

`x` An RleList object containing integer or numeric Rle elements.

`k` An integer indicating the fixed width of the running window. Must be odd when `endrule == "constant"`.

`wt` A numeric vector of length `k` that provides the weights to use.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

`"drop"` do not extend the running statistics to be the same length as the underlying vectors;

`"constant"` copies running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

`runq(x, k, i, endrule = c("drop", "constant"))`: Calculates the order statistic for fixed width running windows across `x`.

`x` An RleList object containing integer or numeric Rle elements.

`k` An integer indicating the fixed width of the running window. Must be odd when `endrule == "constant"`.

`i` An integer indicating which order statistic to calculate.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

"drop" do not extend the running statistics to be the same length as the underlying vectors;

"constant" copies running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

Author(s)

P. Aboyoun

See Also

Sequence for the applicable methods.

Examples

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

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

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

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

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

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

## group generics
```

```
2 * col1
col1 + col1
col1 > 2
log(col1)
sum(col1)

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

`coverage` *Coverage across a set of ranges*

Description

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

Usage

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

## Transitional interface (the current one):
coverage(x, start=NA, end=NA, shift=0L, width=NULL, weight=1L, ...)
## S4 method for signature 'RangesList':
coverage(x,
       start = structure(rep(list(NA), length(x)), names = names(x)),
       end = structure(rep(list(NA), length(x)), names = names(x)),
       shift = structure(rep(list(0L), length(x)), names = names(x)),
       width = structure(rep(list(NULL), length(x)), names = names(x)),
       weight = structure(rep(list(1L), length(x)), names = names(x)))

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

Arguments

`x`	An IRanges, Views, MaskCollection, RangesList, RangedData object, or any object for which a `coverage` method is defined.
`start, end`	For most methods, single integers specifying the position in `x` where to start and end the extraction of the coverage. For RangesList and RangedData objects, a list or vector of the same length as `x` to be used for the corresponding element in `x`. In addition for RangedData objects, can also be a character vector of length 1 denoting the column to use in `values(x)`. IMPORTANT NOTE: Please do not use these arguments (use the shift/width arguments below). They are temporarily kept for backward compatibility with existing code and will be dropped in the near future.
`shift`	For most methods, an integer vector (recycled to the length of `x`) specifying how each element in `x` should be (horizontally) shifted before the coverage is computed. Only shifted indices in the range [1, `width`] will be included in the coverage calculation. For RangesList and RangedData objects, a list or vector of the same length as `x` to be used for the corresponding element in `x`. In addition

for RangedData objects, can also be a character vector of length 1 denoting the column to use in `values(x)`.

`width` For most methods, the length of the returned coverage vector. For RangesList and RangedData objects, a list or vector of the same length as `x` to be used for the corresponding element in `x`. In addition for RangedData objects, can also be a character vector of length 1 denoting the column to use in `values(x)`.

If `width=NULL` (the default), then the specific `coverage` method that is actually selected will choose the length of the returned vector "in a way that makes sense".

For example, when `width=NULL`, the method for IRanges objects returns a vector that has just enough elements to have its last element aligned with the rightmost end of all the ranges in `x` after those ranges have been shifted (see the `shift` argument above). This ensures that any longer coverage vector would be a "padded with zeros" version of the vector returned when `width=NULL`.

When `width=NULL`, the method for Views objects returns a vector with `length(subject(x))` elements.

When `width=NULL`, the method for MaskCollection objects returns a vector with `width(x)` elements.

`weight` For most methods, an integer vector specifying how much each element in `x` counts. For RangesList and RangedData objects, a list or vector of the same length as `x` to be used for the corresponding element in `x`.

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

Value

For most methods, an Rle object representing the coverage of `x`. For RangesList and RangedData objects, a SimpleRleList object representing a list of coverage vectors.

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

Note

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

Author(s)

H. Pages and P. Aboyoun

See Also

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

Examples

```
x <- IRanges(start=c(-2L, 6L, 9L, -4L, 1L, 0L, -6L, 10L),
             width=c( 5L, 0L, 6L,  1L, 4L, 3L,  2L,  3L))
coverage(x)
coverage(x, shift=7)
coverage(x, shift=7, width=27)
coverage(restrict(x, 1, 10))
coverage(reduce(x), shift=7)
coverage(gaps(shift(x, 7), start=1, end=27))

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

DataFrame-class *External Data Frame*

Description

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

Details

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

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

Accessors

In the following code snippets, x is a DataFrame.

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

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

Subsetting

In the following code snippets, x is a DataFrame.

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

x[i,j] <- value: Behaves very similarly to the [<-.data.frame method.

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

`x[[i]] <- value`: Behaves very similarly to the `[[<-.data.frame` method, except argument `j` is not supported.

Constructor

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

Splitting and Combining

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

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

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

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

Coercion

`as(from, "DataFrame")`: By default, constructs a new `DataFrame` with `from` as its only column. If `from` is a `matrix` or `data.frame`, all of its columns become columns in the new `DataFrame`. If `from` is a list, each element becomes a column, recycling as necessary. Note that for the `DataFrame` to behave correctly, each column object must support element-wise subsetting via the `[` method and return the number of elements with `length`. It is recommended to use the `DataFrame` constructor, rather than this interface.

`as.list(x)`: Coerces `x`, a `DataFrame`, to a `list`.

`as.data.frame(x, row.names=NULL, optional=FALSE)`: Coerces `x`, a `DataFrame`, to a `data.frame`. Each column is coerced to a `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 `DataFrame` to a `data.frame` by calling `as.data.frame(from)`.

Author(s)

Michael Lawrence

See Also

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

Examples

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

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

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

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

# subsetting

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

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

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

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

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

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

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

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

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

```
subsw["C",] # partially matches

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

## column replacement

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

## split

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

## rbind

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

## cbind

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

DataFrameList-class

List of DataFrames

Description

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

Accessors

In the following code snippets, x is a DataFrameList.

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

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

Subsetting

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

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

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

Constructor

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

`SplitDataFrameList(..., compress = TRUE)`: The `...` arguments can either be a set of `DataFrame` objects with the same number and names of columns or a set of objects (e.g. `AtomicList`) that represent split columns that will be column bound to form `DataFrame` objects. If `compress = TRUE`, returns a `CompressedSplitDataFrameList`; else returns a `SimpleSplitDataFrameList`.

Combining

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

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

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

Coercion

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

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

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

Author(s)

Michael Lawrence

See Also

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

DataTable-class *DataTable objects*

Description

The DataTable virtual class provides an interface for the storing rectangular data sets, like a basic data.frame object. It extends `Sequence`.

Accessors

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

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

`dimnames(x)`: Get the length two list of character vectors indicating in the first and second element the names of the rows and columns, respectively. This implies the existence of `rownames(x)` and `colnames(x)`.

Combining

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

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

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

Subsetting

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

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

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

`length(x)`: Return the number of columns in the DataTable object.

`seqselect(x, start=NULL, end=NULL, width=NULL)`: Similar to `window`, except that multiple subsequences can be requested. The requested subsequences are concatenated.

`seqselect(x, start=NULL, end=NULL, width=NULL) <- value`: Similar to `window<-`, except that multiple consecutive subsequences can be replaced by a `value` that spans those windows.

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

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

select expression indicating columns to keep.

drop passed on to `[` indexing operator.

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

`window(x, start = NA, end = NA, width = NA, frequency = NULL, delta = NULL, ...)`: Extract the subsequence window from the DataTable object using:

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

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

In general, this is more efficient than using "`[`" operator.

`window(x, start = NA, end = NA, width = NA, keepLength = TRUE) <- value`: Replace the subsequence window specified on the left (i.e. the subsequence in `x` specified by `start`, `end` and `width`) by `value`. `value` must either be of class `class(x)`, belong to a subclass of `class(x)`, be coercible to `class(x)`, or be `NULL`. If `keepLength` is `TRUE`, the number of rows of `value` are repeated to create a DataTable with the same number of rows as the width of the subsequence window it is replacing. If `keepLength` is `FALSE`, this replacement method can modify the number of rows of `x`, depending on how the number of rows of the left subsequence window compares to the number of rows of `value`.

Looping

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

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

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

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

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

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

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

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

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

Coercion

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

See Also

DataTable-stats for statistical functionality, like fitting regression models, DataFrame-class for an implementation that mimics `data.frame`, Sequence-class

Examples

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

DataTable-stats *Statistical modeling with DataTable*

Description

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

Tabulation

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

See Also

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

Examples

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

disjoin *Making Ranges disjoint*

Description

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

Usage

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

Arguments

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

Details

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

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

Author(s)

M. Lawrence

See Also

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

Examples

```
ir <- IRanges(c(1, 1, 4, 10), c(6, 3, 8, 10))
disjoin(ir) # IRanges(c(1, 4, 7, 10), c(3, 6, 8, 10))

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

---

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

Description

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

Usage

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

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

Arguments

X	a list, data.frame or Sequence object.
FUN	the function to be applied to each element of X (for endoapply) or for the elements in ... (for mendoapply).
...	For endoapply, optional arguments to FUN. For mendoapply, a set of list, data.frame or Sequence objects to compute over.
MoreArgs	a list of other arguments to FUN.

Value

endoapply returns an object of the same class as X, each element of which is the result of applying FUN to the corresponding element of X.

mendoapply returns an object of the same class as the first object specified in ..., each element of which is the result of applying FUN to the corresponding elements of ....

See Also

lapply, mapply

Examples

```
a <- data.frame(x = 1:10, y = rnorm(10))
b <- data.frame(x = 1:10, y = rnorm(10))

endoapply(a, function(x) (x - mean(x))/sd(x))
mendoapply(function(e1, e2) (e1 - mean(e1)) * (e2 - mean(e2)), a, b)
```

FilterRules-class *Collection of Filter Rules*

Description

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

Details

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

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

Accessor methods

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

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

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

Constructor

`FilterRules(exprs = list(), ..., active = TRUE)`: Constructs a `FilterRules` with the rules given in the list `exprs` or in `...`. The initial active state of the rules is given by `active`, which is recycled as necessary. Elements in `exprs` may be either character (parsed into an expression), a language object (coerced to an expression), an expression, or a function that takes at least one argument. **IMPORTANTLY**, all arguments in `...` are `quote()`'d and then coerced to an expression. So, for example, character data is only parsed if it is a

literal. The names of the filters are taken from the names of `exprs` and `...`, if given. Otherwise, the character vectors take themselves as their name and the others are deparsed (before any coercion). Thus, it is recommended to always specify meaningful names. In any case, the names are made valid and unique.

Subsetting and Replacement

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

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

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

`x[[i]] <- value`: The same interface as for `Sequence`. 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)
```

`Grouping-class` *Grouping objects*

Description

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

The Grouping core API

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

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

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

Every object must belong to one group and only one.

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

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

If `x` is a Grouping object:

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

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

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

Going from groups to objects:

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

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

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

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

Going from objects to groups:

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

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

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

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

The H2LGrouping and Dups subclasses

DOCUMENT ME

The Partitioning subclass

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

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

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

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

Binning subclass

A Binning container represents a grouping where each observation is assigned to a group or bin. It is similar in nature to taking a the integer codes of a factor object and splitting it up by its levels (i.e. myFactor <- factor(...); split(as.integer(myFactor), myFactor)).

Constructors

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

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

`PartitioningByEnd(end=integer(), names=NULL)`: Return the PartitioningByEnd object made of the partitions ending at the values specified by `end`. `end` must contain sorted non-negative integer values. If the `names` argument is non NULL, it is used to name the partitions.

`PartitioningByWidth(width=integer(), names=NULL)`: Return the PartitioningByWidth object made of the partitions with the widths specified by `width`. `width` must contain non-negative integer values. If the `names` argument is non NULL, it is used to name the partitions.

`Binning(group=integer(), names=NULL)`: Return the Binning object made from the `group` argument, which takes a factor or positive valued integer vector. If the `names` argument is non NULL, it is used to name the bins. When `group` is a factor, the `names` are set to `levels(group)` unless specified otherwise.

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

Author(s)

H. Pages and P. Aboyoun

See Also

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

Examples

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

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

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

```
members(x, 5:2)  # all the members are put together and sorted
togroup(x)
togroup(x, 5:2)
togrouplength(x)  # same as 'grouplength(x, togroup(x))'
togrouplength(x, 5:2)

## The Sequence API:
as.list(x)
sapply(x, length)

## ---------------------------------------------------------------------
## B. Dups OBJECTS
## ---------------------------------------------------------------------
x_dups <- as(x, "Dups")
x_dups
duplicated(x_dups)  # same as 'duplicated(togroup(x_dups))'

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

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

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

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

## The Sequence API:
as.list(x)
sapply(x, length)

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

```
x

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

## Other examples:
PartitioningByEnd(end=c(0, 0, 19), names=LETTERS[1:3])
PartitioningByEnd()  # no partition
PartitioningByEnd(end=integer(9))  # all partitions are empty

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

## The 3 following calls produce the same ranges:
x1 <- successiveIRanges(mywidths)  # IRanges instance.
x2 <- PartitioningByEnd(end=cumsum(mywidths))  # PartitioningByEnd instance.
x3 <- PartitioningByWidth(width=mywidths)  # PartitioningByWidth instance.
stopifnot(identical(as(x1, "PartitioningByEnd"), x2))
stopifnot(identical(as(x1, "PartitioningByWidth"), x3))

## ---------------------------------------------------------------------
## E. Binning OBJECTS
## ---------------------------------------------------------------------
set.seed(0)
x <- Binning(factor(sample(letters, 36, replace=TRUE), levels=letters))
x

grouplength(x)
togroup(x)
x[[2]]
x[["u"]]
```

`IntervalTree-class` *Interval Search Trees*

Description

Efficiently perform overlap queries with an interval tree.

Details

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

The simplest approach is to call the `findOverlaps` function on a `Ranges` or other object with range information, as described in the following section.

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

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 `findOverlaps` function.

`findOverlaps(query, subject = query, maxgap = 0, multiple = TRUE, drop = FALSE)`:

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

If `multiple` is `FALSE`, at most one overlapping interval in `subject` 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 `subject`. This is analogous to the default behavior of the `match` function.

If `query` is a `RangesList` or `RangedData`, `subject` must be a `RangesList` or `RangedData`. If both lists have names, each element from the subject is paired with the element from the query with the matching name, if any. Otherwise, elements are paired by position. The overlap is then computed between the pairs as described above. If `multiple` is `TRUE`, a `RangesMatchingList` is returned, otherwise a list of integer vectors or, if `drop` is `TRUE`, an integer vector with indices offset to align with the unlisted `query`. When `drop` is `FALSE`, an `IntegerList` is returned, where each element of the result corresponds to a space in `query`. For spaces that did not exist in `subject`, the overlap is nil.

`x %in% table`: Shortcut for finding the ranges in `x` that overlap any of the ranges in `table`. Both `x` and `table` should be `Ranges`, `RangesList` or `RangedData` objects. For `Ranges` objects, the result is a `logical` vector of length equal to the number of ranges in `x`. For `RangesList` and `RangedData` objects, the result is a `LogicalList` object, where each element of the result corresponds to a space in `x`.

`match(x, table, nomatch = NA_integer_, incomparables = NULL)`: Returns an integer vector of length `length(x)`, containing the index of the first overlapping range in `table` for each range in `x`. If a range in `x` does not overlap any ranges in `table`, its value is `nomatch`. The `x` and `table` arguments should either be both `Ranges` objects or both `RangesList` objects, in which case the indices are into the unlisted `table`. The `incomparables` argument is currently ignored.

`countOverlaps(query, subject)`: Return the count of the number of ranges in `query` that overlap a range in `subject`.

Constructor

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

Coercion

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

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

Accessors

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

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

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

Notes on Time Complexity

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

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

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

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

Author(s)

Michael Lawrence

References

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

See Also

`Ranges`, 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
findOverlaps(query, tree, multiple = FALSE) # c(2, NA, 3)

## allow multiple hits
findOverlaps(query, tree)

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

## shortcut
findOverlaps(query, subject)

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

## one Ranges with itself
findOverlaps(query)

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

`IRanges-class` *IRanges and NormalIRanges objects*

Description

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

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

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

Constructor

See ¿`IRanges-constructor`‘.

Coercion

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

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

Methods for NormalIRanges objects

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

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

Author(s)

H. Pages

See Also

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

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

Examples

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

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

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

IRanges-constructor

The IRanges constructor and supporting functions

Description

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

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

Usage

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

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

Arguments

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

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

names
: A character vector or NULL.

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

translate.negative.coord, allow.nonnarrowing
: TRUE or FALSE.

IRanges constructor

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

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

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

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

Supporting functions

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

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

each row in the 3-column matrix obtained by binding those 3 vectors together must contain at least one NA (otherwise an error is returned).

Then each row is "solved" i.e. the 2 following transformations are performed (`i` is the indice of the row): (1) if `translate.negative.coord` is TRUE then a negative value of `start[i]` or `end[i]` is considered to be a `-refwidths[i]`-based coordinate so `refwidths[i]+1` is added to it to make it 1-based; (2) the NAs in the row are treated as unknowns which values are deduced from the known values in the row and from `refwidths[i]`.

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

Finally, the set of solved rows is returned as an IRanges object (with the same number of elements as `refwidths`).

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

Author(s)

H. Pages

See Also

IRanges-class, `narrow`

Examples

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

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

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

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

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

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

IRangesList-class *List of IRanges*

Description

A RangesList that only stores IRanges objects.

Constructor

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

Coercion

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

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

IRanges-setops *Set operations on IRanges objects*

Description

Performs set operations on IRanges objects.

Usage

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

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

Arguments

x, y	IRanges objects.
start, end	A single integer or NA. Use these arguments to specify the interval of reference i.e. which interval the returned gaps should be relative to.
...	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

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

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

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

Note that the union or difference of 2 ranges cannot always be represented by a single range so punion and psetdiff cannot always return their result in an IRanges instance of the same length as the input. This happens to punion when there is a gap between the 2 ranges to combine.

In that case, the user can use the `fill.gap` argument to enforce the union by filling the gap. This happens to `psetdiff` when a range in `y` has its end points strictly inside the corresponding range in `x`. In that case, `psetdiff` will simply fail.

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

Author(s)

H. Pages and M. Lawrence

See Also

`pintersect` is similar to `narrow`, except the end points are absolute, not relative. `pintersect` is also similar to `restrict`, except ranges outside of the restriction become empty and are not discarded.

union, intersect, setdiff, Ranges-class, IRanges-class, IRanges-utils

Examples

```
x0 <- IRanges(start=c(-2, 6, 9, -4, 1, 0, -6, 10),
              width=c( 5, 0, 6,  1, 4, 3,  2,  3))
gaps(x0)
gaps(x0, start=-6, end=20)  # Regions of the -6:20 range that are not masked by 'x0'.

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

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

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

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

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

IRanges-utils *IRanges utility functions*

Description

Utility functions for creating or modifying IRanges objects.

Usage

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

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

  ## Modify an IRanges object (endomorphisms):
  shift(x, shift, use.names=TRUE)
  restrict(x, start=NA, end=NA, keep.all.ranges=FALSE, use.names=TRUE)
  narrow(x, start=NA, end=NA, width=NA, use.names=TRUE)
  resize(x, width, start=TRUE, use.names=TRUE)
  ## S4 method for signature 'Ranges':
flank(x, width, start=TRUE, both=FALSE, use.names=TRUE)
  ## S4 method for signature 'Ranges':
reflect(x, bounds, use.names=TRUE)
  threebands(x, start=NA, end=NA, width=NA)
  ## S4 method for signature 'Ranges':
reduce(x, with.inframe.attrib=FALSE)

  ## Other utilities
  ## S4 method for signature 'Ranges':
range(x, ..., na.rm = FALSE)

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

Arguments

`width`	For `successiveIRanges`, must be a vector of non-negative integers (with no NAs) specifying the widths of the ranges to create. For `narrow` and `threebands`, a vector of integers, eventually with NAs. See the SEW (Start/End/Width) interface for the details (`?solveUserSEW`). For `resize` and `flank`, the width of the resized or flanking regions. Note that if `both` is `TRUE`, this is effectively doubled. Recycled as necessary so that each element corresponds to a range in `x`.
`gapwidth`	A single integer or an integer vector with one less element than the `width` vector specifying the widths of the gaps separating one range from the next one.
`from`	A single integer specifying the starting position of the first range.
`x`	A logical vector for `whichAsIRanges`. An IRanges object for `shift`, `restrict`, `narrow`, `threebands`, `reduce` 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`, `threebands`, `resize`, and `flank`. For `narrow` and `threebands`, the supplied `start` and `end` arguments must be vectors of integers, eventually with NAs, that contain coordinates relative to the current ranges. See the Details section below.

For resize and flank, start is a logical indicating whether x should be flanked at the start (TRUE) or the end (FALSE). Recycled as necessary so that each element corresponds to a range in x.

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

with.inframe.attrib
: TRUE or FALSE. For internal use.

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

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

...
: Additional Ranges to consider.

na.rm
: Ignored

force
: TRUE or FALSE. Should x be turned into a NormalIRanges object even if isNormal(x) is FALSE?

Details

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

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

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

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

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

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

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

reflect "reflects" or reverses each range in x relative to the corresponding range in bounds, which is recycled as necessary. Reflection preserves the width of a range, but shifts it such the

distance from the left bound to the start of the range becomes the distance from the end of the range to the right bound. This is illustrated below, where x represents a range in x and [and] indicate the bounds:

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

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

```
---xxxxxxx
```

If start were FALSE:

```
xxxxxxx---
```

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

```
xxxx***
```

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

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

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

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

Author(s)

H. Pages and M. Lawrence

See Also

threebands could be described as a parallel variant of disjoin.

Ranges-class, IRanges-setops, solveUserSEW, successiveViews

Examples

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

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

resize(y, width = 200)
resize(y, width = 2, start=FALSE)

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

x <- IRanges(start=c(-2, 6, 9, -4, 1, 0, -6, 10),
             width=c( 5, 0, 6,  1, 4, 3,  2,  3))
reduce(x)

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

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

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

asNormalIRanges(x)   # 3 ranges ordered from left to right and separated by
                     # gaps of width >= 1.

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

MaskCollection-class *MaskCollection objects*

Description

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

Details

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

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

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

Basic accessor methods

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

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

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

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

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

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

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

Constructor

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

Other methods

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

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

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

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

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

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

Subsetting and appending

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

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

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

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

Other methods

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

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

`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)
collapse(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
collapse(mymasks)
active(mymasks) <- FALSE  # deactivate all masks
mymasks
active(mymasks)[-1] <- TRUE  # reactivate all masks except mask 1
active(mymasks) <- !active(mymasks)  # toggle all masks

## Other advanced operations:
mymasks[[2]]
length(mymasks[[2]])
mymasks[[2]][-3]
append(mymasks[-2], gaps(mymasks[2]))
mymasks2 <- subseq(mymasks, start=8)
mymasks2
mymasks2[[2]]
```

`nearest` *Nearest neighbor finding*

Description

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

Usage

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

Arguments

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

Details

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

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

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

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

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

Author(s)

M. Lawrence

See Also

`findOverlaps` for finding just the overlapping ranges.

Examples

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

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

query <- IRanges(c(1, 3, 9), c(3, 7, 10))
subject <- IRanges(c(3, 2, 10), c(3, 13, 12))

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

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

RangedData-class *Data on ranges*

Description

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

Details

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

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

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

Accessor methods

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

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

Array accessors:

- nrow(x): The number of ranges in x.
- ncol(x): The number of data variables in x.
- dim(x): An integer vector of length two, essentially c(nrow(x), ncol(x)).
- rownames(x), rownames(x) <- value: Gets or sets the names of the ranges in x.
- colnames(x), colnames(x) <- value: Gets the names of the variables in x.
- dimnames(x): A list with two elements, essentially list(rownames(x), colnames(x)).
- dimnames(x) <- value: Sets the row and column names, where value is a list as described above.

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

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

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

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

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

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

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

Other accessors:

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

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

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

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

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

Constructor

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

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

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

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

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

Coercion

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

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

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

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

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

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

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

Subsetting and Replacement

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

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

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

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

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

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

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

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

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

Splitting and Combining

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

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

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

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

Utilities

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

`reduce(x, by, with.inframe.attrib = FALSE)`: Merges the ranges in each of the spaces after grouping by the `by` values columns and returns the result as a `RangedData` containing the reduced ranges and the `by` value columns.

Applying

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

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

Author(s)

Michael Lawrence

See Also

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

Examples

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

## constructing RangedData instances

## no variables
rd <- RangedData()
rd <- RangedData(ranges)
ranges(rd)
## one variable
rd <- RangedData(ranges, score)
rd[["score"]]
## multiple variables
rd <- RangedData(ranges, filter, vals = score)
rd[["vals"]] # same as rd[["score"]] above
rd$vals
rd[["filter"]]
rd <- RangedData(ranges, score + score)
rd[["score...score"]] # names made valid
## use a universe
rd <- RangedData(ranges, universe = "hg18")
universe(rd)
rd <- RangedData(
    RangesList(
      chrA = IRanges(start = c(1, 4, 6), width=c(3, 2, 4)),
      chrB = IRanges(start = c(1, 3, 6), width=c(3, 3, 4))),
    score = c(2, 7, 3, 1, 1, 1))
rd
reduce(rd)
```

```
## split some data over chromosomes

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

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

## subsetting

## list style: [i]

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

## matrix style: [i,j]

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

## dimnames

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

## space names

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

## variable replacement

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

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

## combining/splitting

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

## applying

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

RangedDataList-class

Lists of RangedData

Description

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

Constructor

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

Author(s)

Michael Lawrence

See Also

RangedData, the element type of this Sequence.

Examples

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

RangedData-utils *RangedData utility functions*

Description

Utility functions for manipulating RangedData objects.

Usage

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

Arguments

x	A RangedData object
...	Additional RangedData objects
na.rm	Ignored

Value

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

Author(s)

Michael Lawrence

See Also

RangedData, DataTable-class, Sequence-class

Examples

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

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

Ranges-class *Ranges objects*

Description

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

Details

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

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

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

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

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

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

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

Methods

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

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

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

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

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

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

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

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

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

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

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

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

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

`unlist(x, recursive = TRUE, use.names = TRUE)`: Similar to `as.integer(x)` except can add names to elements.

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

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

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

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

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

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

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

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

`x + y`: Expands the ranges in `x` on either side by the corresponding value in the numeric vector `y`.

Normality

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

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

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

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

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

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

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

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

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

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

Author(s)

H. Pages and M. Lawrence

See Also

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

Examples

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

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

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

```
start(x)[3] <- end(x)[3] - 2  # resize the 3rd range
x

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

## Check for disjointedness
isDisjoint(IRanges(c(2,5,1), c(3,7,3))) ## FALSE
isDisjoint(IRanges(c(2,9,5), c(3,9,6))) ## TRUE
isDisjoint(IRanges(1, 5)) ## TRUE

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

`Ranges-comparison` *Ranges comparison*

Description

Equality and ordering of ranges, and related methods.

Usage

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

  x == y
  x != y

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

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

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

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

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

  ## S4 method for signature 'Ranges':
```

```
sort(x, decreasing=FALSE, ...)

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

Arguments

x,y	A Ranges object.
incomparables	Must be FALSE.
fromLast	TRUE or FALSE.
...	Ranges objects for order.
na.last	Ignored.
decreasing	TRUE or FALSE.
ties.method	A character string specifying how ties are treated. Only "first" is supported for now.

Details

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

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

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

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

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

sort(x): Sorts x. See sort in the base package for more details.

rank(x, na.last=TRUE, ties.method=c("average", "first", "random", "max", "min")): Returns the sample ranks of the ranges in x. See rank in the base package for more details.

See Also

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

Examples

```
x <- IRanges(start=c(20L, 8L, 20L, 22L, 25L, 20L, 22L, 22L),
             width=c( 4L, 0L, 11L,  5L,  0L,  9L,  5L,  0L))
x
which(width(x) == 0)  # 3 empty ranges
x[2] == x[5]  # FALSE
```

```
x == x[4]
duplicated(x)
unique(x)
x >= x[3]
order(x)
sort(x)
rank(x, ties.method="first")
```

RangesList-class *List of Ranges*

Description

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

Accessors

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

All of these accessors collapse over the spaces:

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

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

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

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

These accessors are for the universe identifier:

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

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

Constructor

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

Subsetting

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

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

Coercion

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

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

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

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

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

Arithmetic Operations

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

Author(s)

Michael Lawrence

Examples

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

# edit the width of the ranges in the list
edited <- named
width(edited) <- rep(c(3,2), elementLengths(named))
edited

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

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

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

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

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

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

RangesList-utils *RangesList utility functions*

Description

Utility functions for manipulating RangesList objects.

Usage

```
## S4 method for signature 'RangesList':
shift(x, shift, use.names=TRUE)
## S4 method for signature 'RangesList':
restrict(x, start=NA, end=NA, keep.all.ranges=FALSE, use.names=TRUE)
## S4 method for signature 'RangesList':
narrow(x, start=NA, end=NA, width=NA, use.names=TRUE)
## S4 method for signature 'RangesList':
resize(x, width, start=TRUE, use.names=TRUE)
## S4 method for signature 'RangesList':
flank(x, width, start=TRUE, both=FALSE, use.names=TRUE)
## S4 method for signature 'RangesList':
gaps(x, start=NA, end=NA)
## S4 method for signature 'RangesList':
disjoin(x)
## S4 method for signature 'RangesList':
reduce(x, with.inframe.attrib=FALSE)
## S4 method for signature 'RangesList':
range(x, ..., na.rm = FALSE)

# Set operations
## S4 method for signature 'RangesList,RangesList':
union(x, y)
## S4 method for signature 'RangesList,RangesList':
```

```
intersect(x, y)
## S4 method for signature 'RangesList,RangesList':
setdiff(x, y)
```

Arguments

x, y	A RangesList
start, end	A single integer or NA for all functions except narrow, resize, and flank. For narrow, the supplied start and end arguments must be vectors of integers, eventually with NAs, that contain coordinates relative to the current ranges. For resize and flank, start is a logical indicating whether x should be flanked at the start (TRUE) or the end (FALSE). Recycled as necessary so that each element corresponds to a range in x.
width	For narrow, a vector of integers, eventually with NAs. See the SEW (Start/End/Width) interface for the details (?solveUserSEW). For resize and flank, the width of the resized or flanking regions. Note that if both is TRUE, this is effectively doubled. Recycled as necessary so that each element corresponds to a range in x.
shift	A single integer.
both	If TRUE, extends the flanking region width positions *into* the range. The resulting range thus straddles the end point, with width positions on either side.
use.names	TRUE or FALSE. Should names be preserved?
keep.all.ranges	TRUE or FALSE. Should ranges that don't overlap with the interval specified by start and end be kept? Note that "don't overlap" means that they end strictly before start - 1 or start strictly after end + 1. Ranges that end at start - 1 or start at end + 1 are always kept and their width is set to zero in the returned RangesList object.
with.inframe.attrib	TRUE or FALSE. For internal use.
...	Additional RangesList to consider.
na.rm	Ignored

Details

The shift method shifts all the ranges in x.

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

The narrow method narrows the ranges in x i.e. each range in the returned RangesList object is a subrange of the corresponding range in x.

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

The flank method generates flanking ranges for each range in x.

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

The disjoin method returns disjoint ranges by finding the within element union of the end points of x.

The reduce method merges (via reduce) each of the elements in the list and returns the result as a RangesList.

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

The union method performs elementwise union operations for two RangesList objects.

The intersect method performs elementwise intersect operations for two RangesList objects.

The setdiff method performs elementwise setdiff operations for two RangesList objects.

Value

A RangesList object. For shift, restrict, narrow, resize, flank, gaps and range, length is the same as that of x. For reduce, length is one.

Author(s)

Michael Lawrence, H. Pages, P. Aboyoun

See Also

RangesList, IRanges-utils

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)

shift(collection, shift=5)
restrict(collection, start=2, end=8)
resize(collection, width=200)
flank(collection, width=10)
disjoin(collection)

# these two are the same
RangesList(gaps(range1), gaps(range2))
gaps(collection)

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

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

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

```
range(rl, rl2) # now by position

# set operations
union(rl, rl2)
intersect(rl, rl2)
setdiff(rl, rl2)
```

RangesMatching-class
Matchings between Ranges

Description

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

Details

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

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

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

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

Coercion

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

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

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

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

Accessors

queryHits(x): Gets the indices of overlapping ranges in the query, equivalent to as.matrix(x)[,1].

subjectHits(x): Gets the indices of overlapping ranges in the subject, equivalent to as.matrix(x)[,2].

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

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

length(x): get the number of matches

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

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

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

Author(s)

Michael Lawrence

See Also

`findOverlaps`, 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 <- findOverlaps(query, tree)

as.matrix(matchings)

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

RangesMatchingList-class

List of Matchings between Ranges

Description

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

Details

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

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

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

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

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

Coercion

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

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

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

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

Accessors

queryHits(x): Gets the indices of overlapping ranges in the query, equivalent to as.matrix(x)[,1].

subjectHits(x): Gets the indices of overlapping ranges in the subject, equivalent to as.matrix(x)[,2].

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

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

Note

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

Author(s)

Michael Lawrence

See Also

findOverlaps, which generates an instance of this class.

---

rdapply *Applying over spaces*

Description

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

Usage

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

Arguments

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

Details

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

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

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

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

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

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

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

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

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

Value

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

Constructing an RDApplyParams object

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

Accessors

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

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

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

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

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

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

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

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

Author(s)

Michael Lawrence

See Also

`RangedData`, `FilterRules`

Examples

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

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

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

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

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

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

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

Description

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

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

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

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

Usage

```
read.agpMask(file, seqname="?", mask.width=NA, gap.types=NULL, use.gap.types=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 an IRanges object*

Description

This operation is not intended to be used directly.

Usage

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

Arguments

`x`	An IRanges object.
`...`	Additional arguments to be passed to or from methods.

Value

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

Examples

```
## WARNING: The examples here are not relevant anymore and will be
## updated soon.
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)

## When omitted, 'start' and 'end' are considered to be 'min(start(x))'
## and 'max(end(x))', respectively:
reverse(x)

reverse(shift(x, 2), start=-6, end=20)
reverse(restrict(x, 1, 10), start=-6, end=20)
reverse(reduce(x), start=-6, end=20)
reverse(gaps(x, start=-6, end=20), start=-6, end=20)

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

Rle-class *Rle objects*

Description

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

Constructors

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

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

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

Accessors

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

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

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

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

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

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

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

Replacers

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

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

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

Coercion

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

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

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

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

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

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

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

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

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

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

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

Group Generics

Rle objects have support for S4 group generic functionality:

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

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

`Logic` "&", "|"

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

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

`Math2` "round", "signif"

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

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

See S4groupGeneric for more details.

General Methods

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

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

`x[i] <- value`: Equivalent to `seqselect(x, i) <- value`.

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

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

- `by` An object with `start`, `end`, and `width` methods.
- `FUN` The function, found via `match.fun`, to be applied to each window of x.
- `start`, `end`, `width` the start, end, or width of the window. If `by` is missing, then must supply two of the three.
- `frequency`, `delta` Optional arguments that specify the sampling frequency and increment within the window.
- **...** Further arguments for `FUN`.
- `simplify` A logical value specifying whether or not the result should be simplified to a vector or matrix if possible.

`append(x, values, after = length(x))`: Insert one Rle into another Rle.

- `values` the Rle to insert.
- `after` the subscript in x after which the values are to be inserted.

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

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

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

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

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

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

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

- `times` Vector giving the number of times to repeat each element if of length `length(x)`, or to repeat the whole vector if of length 1.
- `length.out` Non-negative integer. The desired length of the output vector.
- `each` Non-negative integer. Each element of x is repeated `each` times.

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

`seqselect(x, start = NULL, end = NULL, width = NULL)`: Creates a new Rle object using consecutive subsequences from x specified by two of the three following values: `start`, `end`, and `width`. See seqselect for more details.

`seqselect(x, start=NULL, end=NULL, width=NULL) <- value`: Similar to `window<-`, except that multiple consecutive subsequences can be replaced by a constant `value`. As such two of the three `start`, `end`, and `width` arguments can be used to specify the consecutive subsequences. Alternatively, `start` can take a Ranges object or something that can be converted to a Ranges object like an integer vector, logical vector or logical Rle.

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

`SHIFT` A non-negative integer vector of shift values.

`X`, `Y` The Rle objects to shift.

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

... Further arguments for `FUN`.

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

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

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

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

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

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

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

`split(x, f, drop = FALSE)`: Splits `x` according to `f` to create a CompressedRleList object. Empty list elements are removed if `drop` is `TRUE`.

`subset(x, subset)`: Return a new Rle object made of the subset using logical vector `subset`.

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

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

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

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

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

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

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

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

window it is replacing. If keepLength is FALSE, this replacement method can modify the length of x, depending on how the length of the left subsequence window compares to the length of value.

Logical Data Methods

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

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

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

Numerical Data Methods

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

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

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

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

lag An integer indicating which lag to use.

differences An integer indicating the order of the difference.

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

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

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

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

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

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

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

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

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

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

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

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

center The center to calculate the deviation from.

constant The scale factor.

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

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

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

`IQR(x, na.rm = FALSE)`: Calculates the interquartile range of `x`.

`na.rm` If `TRUE`, removes `NA`s from `x` before the IQR is computed.

`smoothEnds(y, k = 3)`: Smooth end points of an Rle `y` using subsequently smaller medians and Tukey's end point rule at the very end.

`k` An integer indicating the width of largest median window; must be odd.

`runmean(x, k, endrule = c("drop", "constant"))`: Calculates the means for fixed width running windows across `x`.

`k` An integer indicating the fixed width of the running window. Must be odd when `endrule == "constant"`.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

`"drop"` do not extend the running statistics to be the same length as the underlying vectors;

`"constant"` copies running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

`runmed(x, k, endrule = c("median", "keep", "drop", "constant"))`: Calculates the medians for fixed width running windows across `x`.

`k` An integer indicating the fixed width of the running window. Must be odd when `endrule != "drop"`.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

`"keep"` keeps the first and last k_2 values at both ends, where k_2 is the half-bandwidth `k2 = k %/% 2`, i.e., `y[j] = x[j]` for $j \in \{1, \ldots, k_2; n - k_2 + 1, \ldots, n\}$ j = 1,..,k2 and (n-k2+1),..,n;

`"constant"` copies the running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

`"median"` the default, smooths the ends by using symmetrical medians of subsequently smaller bandwidth, but for the very first and last value where Tukey's robust end-point rule is applied, see `smoothEnds`.

`runsum(x, k, endrule = c("drop", "constant"))`: Calculates the sums for fixed width running windows across `x`.

`k` An integer indicating the fixed width of the running window. Must be odd when `endrule == "constant"`.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

`"drop"` do not extend the running statistics to be the same length as the underlying vectors;

`"constant"` copies running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

`runwtsum(x, k, wt, endrule = c("drop", "constant"))`: Calculates the sums for fixed width running windows across `x`.

`k` An integer indicating the fixed width of the running window. Must be odd when `endrule == "constant"`.

`wt` A numeric vector of length `k` that provides the weights to use.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

`"drop"` do not extend the running statistics to be the same length as the underlying vectors;

`"constant"` copies running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

`runq(x, k, i, endrule = c("drop", "constant"))`: Calculates the order statistic for fixed width running windows across `x`.

`k` An integer indicating the fixed width of the running window. Must be odd when `endrule == "constant"`.

`i` An integer indicating which order statistic to calculate.

endrule A character string indicating how the values at the beginning and the end (of the data) should be treated.

`"drop"` do not extend the running statistics to be the same length as the underlying vectors;

`"constant"` copies running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

Character Data Methods

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

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

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

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

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

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

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

`new` A character string specifying the translations.

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

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

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

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

Factor Data Methods

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

`levels(x)`, `levels(x) <- value`: Gets and sets to the levels attribute of a variable, respectively.

Author(s)

P. Aboyoun

See Also

rle, Sequence-class, S4groupGeneric, IRanges-class

Examples

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

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

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

y <- Rle(c(TRUE,TRUE,FALSE,FALSE,TRUE,FALSE,TRUE,TRUE,TRUE))
y
as.vector(y)
rep(y, 10)
c(y, x > 5)

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

RleViews-class *The RleViews class*

Description

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

Details

An RleViews object contains a set of views (start/end locations) on the same Rle object called "the subject vector" or simply "the subject". Each view is defined by its start and end locations: both are integers such that start <= end. An RleViews object is in fact a particular case of a Views object (the RleViews class contains the Views class) so it can be manipulated in a similar manner: see `?Views` for more information. Note that two views can overlap and that a view can be "out of limits" i.e. it can start before the first element of the subject or/and end after its last element.

Author(s)

P. Aboyoun

See Also

Views-class, Rle-class, Views-utils

Examples

```
subject <- Rle(rep(c(3L, 2L, 18L, 0L), c(3,2,1,5)))
myViews <- Views(subject, 3:0, 5:8)
myViews
subject(myViews)
length(myViews)
start(myViews)
end(myViews)
width(myViews)
myViews[[2]]

set.seed(0)
vec <- Rle(sample(0:2, 20, replace = TRUE))
vec
Views(vec, vec > 0)
```

`RleViewsList-class` *List of RleViews*

Description

An extension of `RangesList` that holds only `RleViews` objects. Useful for storing coverage vectors over a set of spaces (e.g. chromosomes), each of which requires a separate `RleViews` object. As a `Sequence`, `RleViewsList` may be annotated with its universe identifier (e.g. a genome) in which all of its spaces exist.

Details

For more information on methods available for RleViewsList objects consult the man pages for RangesList and Views-utils.

Constructor

`RleViewsList(..., rleList, rangesList, universe = NULL)`: Either `...` or the `rleList`/`rangesList` couplet provide the RleViews for the list. If `...` is provided, each of these arguments must be RleViews objects. Alternatively, `rleList` and `rangesList` accept Rle and Ranges objects respectively that are meshed together for form the RleViewsList. The universe is specified by the `universe` parameter, which should be a single string or NULL, to leave unspecified.

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

Author(s)

P. Aboyoun

See Also

RangesList-class, Views-utils

Examples

```
## Rle objects
subject1 <- Rle(c(3L,2L,18L,0L), c(3,2,1,5))
set.seed(0)
subject2 <- Rle(c(0L,5L,2L,0L,3L), c(8,5,2,7,4))

## Views
rleViews1 <- Views(subject1, 3:0, 5:8)
rleViews2 <- Views(subject2, subject2 > 0)

## RleList and RangesList objects
rleList <- RleList(subject1, subject2)
rangesList <- IRangesList(IRanges(3:0, 5:8), IRanges(subject2 > 0))

## methods for construction
method1 <- RleViewsList(rleViews1, rleViews2)
method2 <- RleViewsList(rleList = rleList, rangesList = rangesList)
identical(method1, method2)

## calculation over the views
viewSums(method1)
```

---

runstat *Fixed width running window summaries across vector-like objects*

Description

The `runsum`, `runmean`, `runwtsum`, `runq` functions calculate the sum, mean, weighted sum, and order statistics for fixed width running windows.

Usage

```
runsum(x, k, endrule = c("drop", "constant"))
runmean(x, k, endrule = c("drop", "constant"))
runwtsum(x, k, wt, endrule = c("drop", "constant"))
runq(x, k, i, endrule = c("drop", "constant"))
```

Arguments

x	The data object.
k	An integer indicating the fixed width of the running window. Must be odd when endrule == "constant".
wt	A numeric vector of length k that provides the weights to use.
i	An integer indicating which order statistic to calculate.
endrule	A character string indicating how the values at the beginning and the end (of the data) should be treated.
	"drop" do not extend the running statistics to be the same length as the underlying vectors;
	"constant" copies running statistic to the first values and analogously for the last ones making the smoothed ends *constant*;

Details

The runsum, runmean, runwtsum, and runq functions provide efficient methods for calculating the specified numeric summary by performing the looping in compiled code.

Value

An object of the same class as x.

Author(s)

P. Aboyoun

See Also

runmed, Rle-class, RleList-class

Examples

```
x <- Rle(1:10, 1:10)
runsum(x, k = 3)
runsum(x, k = 3, endrule = "constant")
runmean(x, k = 3)
runwtsum(x, k = 3, wt = c(0.25, 0.5, 0.25))
runq(x, k = 5, i = 3, endrule = "constant")
```

score *Score accessor and setter*

Description

Gets and sets the score of an object.

Usage

```
score(x, ...)
score(x, ...) <- value
```

Arguments

x	An object to get or set the score value of.
value	A new score value.
...	Additional arguments.

Sequence-class *Sequence objects*

Description

The Sequence virtual class serves as the heart of the IRanges package and has over 80 subclasses. It serves a similar role as vector in base R. The Sequence class includes three slots: elementType, metadata (via extension of the Annotated class), and elementMetadata. Their purpose is defined below.

The elementType slot is the preferred location for Sequence subclasses to store the type of data represented in the sequence. It is designed to take a character of length 1 representing the class of the sequence elements. While the Sequence class performs no validity checking based on elementType, if a subclass expects elements to be of a given type, that subclass is expected to perform the necessary validity checking. For example, the subclass IntegerList has elementType = "integer" and its validity method checks if this condition is TRUE.

The Sequence class supports the storage of global and element-wise metadata with its metadata and elementMetadata slots. The metadata slot can store a list of metadata pertaining to the whole object and the elementMetadata slot can store a DataTable (or NULL) for element-wise metadata with a row for each element and a column for each metadata variable.

To be functional, a class that inherits from Sequence must define a length and names methods as well as one or both of the subscript methods "[[" and "[".

Accessors

In the following code snippets, x is a Sequence 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 Sequence.

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

`elementLengths(x)`: Get the 'length' of each of the elements.

`isEmpty(x)`: Returns a logical indicating either if the sequence has no elements or if all its elements are empty.

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

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

Combining

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

`append(x, values, after = length(x))`: Insert the `Sequence values` onto `x` at the position given by `after`. `values` must have an `elementType` that extends that of `x`.

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

Subsetting

In the code snippets below, `x` is a Sequence object or regular R vector object. The R vector object methods for `window` and `seqselect` are defined in this package and the remaining methods are defined in base R.

`x[i, drop=TRUE]`: If defined, returns a new Sequence object made of selected elements `i`, which can be missing; an NA-free logical, numeric, or character vector; or a logical Rle object. The `drop` argument specifies whether or not to coerce the returned sequence to a standard vector.

`x[i] <- value`: Equivalent to `seqselect(x, i) <- value`.

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

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

`Filter(f, x)`: Extracts the elements of `x` for which function `f` is TRUE.

f A unary argument function.

`Find(f, x, right = FALSE, nomatch = NULL)`: Extracts the first or last such element in `x`.

f A unary argument function.

right A logical indicating whether to proceed from left to right (default) or from right to left.

nomatch The value to be returned in the case when "no match" (no element satisfying the predicate) is found.

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

`Position(f, x, right = FALSE, nomatch = NA_integer_)`: Extracts the first or last such position in `x`.

f A unary argument function.

right A logical indicating whether to proceed from left to right (default) or from right to left.

nomatch The value to be returned in the case when "no match" (no element satisfying the predicate) is found.

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

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

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

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

`rev(x)`: Return a new Sequence object made of the original elements in the reverse order.

`seqselect(x, start=NULL, end=NULL, width=NULL)`: Similar to `window`, except that multiple consecutive subsequences can be requested for concatenation. As such two of the three `start`, `end`, and `width` arguments can be used to specify the consecutive subsequences. Alternatively, `start` can take a Ranges object or something that can be converted to a Ranges object like an integer vector, logical vector or logical Rle. If the concatenation of the consecutive subsequences is undesirable, consider using `Views`.

`seqselect(x, start=NULL, end=NULL, width=NULL) <- value`: Similar to `window<-`, except that multiple consecutive subsequences can be replaced by a `value` whose length is a divisor of the number of elements it is replacing. As such two of the three `start`, `end`, and `width` arguments can be used to specify the consecutive subsequences. Alternatively, `start` can take a Ranges object or something that can be converted to a Ranges object like an integer vector, logical vector or logical Rle.

`subset(x, subset)`: Return a new Sequence object made of the subset using logical vector `subset`, where missing values are taken as FALSE.

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

`window(x, start = NA, end = NA, width = NA, frequency = NULL, delta = NULL, ...)`: Extract the subsequence window from the Sequence object using:

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

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

In general, this is more efficient than using `"["` operator.

`window(x, start = NA, end = NA, width = NA, keepLength = TRUE) <- value`: Replace the subsequence window specified on the left (i.e. the subsequence in `x` specified by `start`, `end` and `width`) by `value`. `value` must either be of class `class(x)`, belong to a subclass of `class(x)`, be coercible to `class(x)`, or be `NULL`. If `keepLength` is `TRUE`, the elements of `value` are repeated to create a Sequence with the same number of elements as the width of the subsequence window it is replacing. If `keepLength` is `FALSE`, this replacement method can modify the length of `x`, depending on how the length of the left subsequence window compares to the length of `value`.

Evaluating

In the code snippets below, `envir` and `data` is a Sequence object.

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

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

Coersion

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

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

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

Looping

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

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

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

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

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

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

... Further arguments for `FUN`.

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

`endoapply(X, FUN, ...)`: Similar to `lapply`, but performs an endomorphism, i.e. returns an object of `class(X)`.

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

`Map(f, ...)`: Applies a function to the corresponding elements of given Sequence objects.

`mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)`: Like the standard `mapply` function defined in the base package, the `mapply` method for Sequence objects is a multivariate version of `sapply`.

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

`Reduce(f, x, init, right = FALSE, accumulate = FALSE)`: Uses a binary function to successively combine the elements of x and a possibly given initial value.

f A binary argument function.

init An R object of the same kind as the elements of x.

right A logical indicating whether to proceed from left to right (default) or from right to left.

nomatch The value to be returned in the case when "no match" (no element satisfying the predicate) is found.

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

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

`SHIFT` A non-negative integer vector of shift values.

`X`, `Y` The Sequence or R vector objects to shift.

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

... Further arguments for `FUN`.

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

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

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

Author(s)

P. Aboyoun

See Also

Annotated, DataTable, SimpleList, Ranges, Rle, XVector for example implementations

Examples

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

SimpleList-class *Simple and Compressed List Classes*

Description

The (non-virtual) SimpleList and (virtual) CompressedList classes extend the Sequence virtual class.

Details

The SimpleList and CompressedList classes provide an implementation that subclasses can easily extend. The underlying storage in a SimpleList subclass is a list object. The underlying storage in a CompressedList object is a virtually partitioned vector-like object. For more information on the available methods, see the Sequence man page.

Constructor

The SimpleList class constructor is used to create SimpleList objects:

`SimpleList(...)`: takes possibly named objects as elements for the new SimpleList object.

Coercion

In the following code snippets, x is a SimpleList or CompressedList object.

`as.list(x)`: Copies the elements of x into a new R list object.

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

Subsetting

In the following code snippets, x is a SimpleList or CompressedList object.

`x[i]`: In addition to normal usage, the i parameter can be a `RangesList`, logical `RleList`, `LogicalList`, or `IntegerList` object to perform subsetting within the list elements rather than across them.

`x[i] <- value`: In addition to normal usage, the i parameter can be a `RangesList`, logical `RleList`, `LogicalList`, or `IntegerList` object to perform subsetting within the list elements rather than across them.

`seqselect(x, start=NULL, end=NULL, width=NULL)`: In addition to normal usage, the `start` parameter can be a `RangesList`, logical `RleList`, `LogicalList`, or `IntegerList` object to perform sequence extraction within the list elements rather than across them.

`seqselect(x, start=NULL, end=NULL, width=NULL) <- value`: In addition to normal usage, the `start` parameter can be a `RangesList`, logical `RleList`, `LogicalList`, or `IntegerList` object to perform sequence replacement within the list elements rather than across them.

Looping

In the following code snippets, x is a SimpleList or CompressedList object.

`aggregate(x, by, FUN, start = NULL, end = NULL, width = NULL, frequency = NULL, delta = NULL, ..., simplify = TRUE))`: In addition to normal usage, the `by` parameter can be a `RangesList` to aggregate within the list elements rather than across them. When `by` is a `RangesList`, the output is either a `SimpleAtomicList` object, if possible, or a `SimpleList` object, if not.

Author(s)

P. Aboyoun

See Also

Sequence, AtomicList and RangesList for example implementations

Examples

```
SimpleList(a = letters, ranges = IRanges(1:10, 1:10))
```

strutils *Low-level string utilities*

Description

Some low-level string utilities that operate on ordinary character vectors. For more advanced string manipulations, see the Biostrings package.

Usage

```
strsplitAsListOfIntegerVectors(x, sep=",")
```

Arguments

x	A character vector where each element is a string containing comma-separated decimal integer values.
sep	The value separator character.

Value

A list of integer vectors. The list is of the same length as the input.

Note

strsplitAsListOfIntegerVectors is similar to the strsplitAsListOfIntegerVectors2 function shown in the Examples section below, except that the former generally raises an error where the latter would have inserted an NA in the returned object. More precisely:

- The latter accepts NAs in the input, the former doesn't (raises an error).
- The latter introduces NAs by coercion (with a warning), the former doesn't (raises an error).
- The latter supports "inaccurate integer conversion in coercion" when the value to coerce is > INT_MAX (then it's coerced to INT_MAX), the former doesn't (raises an error).
- The latter coerces non-integer values (e.g. 10.3) to an int by truncating them, the former doesn't (raises an error).

When it fails, strsplitAsListOfIntegerVectors will print an informative error message. Finally, strsplitAsListOfIntegerVectors is faster and uses much less memory than strsplitAsListOfIntegerVectors2.

Author(s)

H. Pages

See Also

strsplit

Examples

```
x <- c("1116,0,-19",
       " +55291 , 2476,",
       "19184,4269,5659,6470,6721,7469,14601",
       "7778889, 426900, -4833,5659,6470,6721,7096",
       "19184 , -99999")
y <- strsplitAsListOfIntegerVectors(x)
y

## In normal situations (i.e. when the input is well-formed),
## strsplitAsListOfIntegerVectors() does actually the same as the
## function below but is faster and much more memory efficient:
strsplitAsListOfIntegerVectors2 <- function(x, sep=",")
{
  tmp <- strsplit(x, sep, fixed = TRUE)
  lapply(tmp, as.integer)
}
y2 <- strsplitAsListOfIntegerVectors2(x)
stopifnot(identical(y, y2))
```

`updateObject` *Update an IRanges object from BioC 2.4 to current version*

Description

Updates an old IRanges object to the current version. This updating process is useful when an object has been serialized (e.g., stored to disk) for some time (e.g., months), and the class definition has changed. This change in class definition makes the serialized instance is no longer valid.

Usage

```
updateObject(object, ..., verbose=FALSE)
```

Arguments

`object`	Object to be updated, or for slot information to be extracted from.
`verbose`	A logical, indicating whether information about the update should be reported. Use `message` to report this.
`...`	Additional arguments, for use in specific update methods.

Value

`updateObject` returns a valid instance of `object`.

Author(s)

Biocore team

Views-class *Views objects*

Description

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

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

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

Constructor

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

The treatment of the `start`, `end` and `width` arguments is the same as with the `IRanges` constructor, except that, in addition, `Views` allows `start` to be a Ranges object. With this feature, `Views(subject, IRanges(my_starts, my_ends, my_widths, my_names))` and `Views(subject, my_starts, my_ends, my_widths, my_names)` are equivalent (except when `my_starts` is itself a Ranges object).

Accessor-like methods

All the accessor-like methods defined for `IRanges` objects work on Views objects. In addition, the following accessors are defined for Views objects:

`subject(x)`: Return the subject of the views.

Subsetting and appending

`"["`, `c` and `"[["` work on a Views object. The first two operations are just inherited from the `IRanges` class but now they return a Views object. However, the `"[["` method for Views objects has a different semantic than the method for `IRanges` objects.

`x[[i]]`: Extracts the view selected by `i` as an object of the same class as `subject(x)`. Subscript `i` can be a single integer or a character string. The result is the subsequence of `subject(x)` defined by `window(subject(x), start=start(x)[i], end=end(x)[i])` or an error if the view is "out of limits" (i.e. `start(x)[i] < 1` or `end(x)[i] > length(subject(x))`).

Other methods

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

`subviews(x, start=NA, end=NA, width=NA, use.names=TRUE)`: `start`, `end`, and `width` arguments must be vectors of integers, eventually with NAs, that contain coordinates relative to the current ranges. Equivalent to `trim(narrow(x, start=start, end=end, width=width, use.names=use.names))`.

`gaps(x, start=NA, end=NA)`: `start` and `end` can be single integers or NAs. The gap extraction will be restricted to the window specified by `start` and `end`. `start=NA` and `end=NA` are interpreted as `start=1` and `end=length(subject(x))`, respectively, so, if `start` and `end` are not specified, then gaps are extracted with respect to the entire subject.

`successiveViews(subject, width, gapwidth=0, from=1)`: Equivalent to `Views(subject, successiveIRanges(width, gapwidth, from))`. See `?successiveIRanges` for a description of the `width`, `gapwidth` and `from` arguments.

Author(s)

H. Pages

See Also

IRanges-class, Sequence-class, IRanges-utils, XVector.

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

Examples

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

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

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

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

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

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

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

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

Description

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

The `viewMins`, `viewMaxs`, `viewSums`, `viewMeans` functions calculate the minima, maxima, sums, and means on views respectively.

Usage

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

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

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

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

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

Arguments

`X`	A Views object.
`FUN`	The function to be applied to each view in `X`.
`...`	Additional arguments to be passed on.
`simplify`	A logical value specifying whether or not the result should be simplified to a vector or matrix if possible.
`x`	An Rle, RleList, XInteger object or an integer vector for `slice`. An RleViews, RleViewsList, XIntegerViews object for `viewMins`, `viewMaxs`, `viewSums`, and `viewMeans`. An Rle, RleViewsList, XIntegerViews object for `viewWhichMins` and `viewWhichMaxs`.
`lower, upper`	The lower and upper bounds for the slice.
`includeLower, includeUpper`	Logical indicating whether or not the specified boundary is open or closed.
`rangesOnly`	For Rle and RleList objects, a logical indicating whether or not to drop the original data from the output.
`na.rm`	Logical indicating whether or not to include missing values in the results.

Details

The `slice` function is useful for finding areas of absolute maxima (peaks), absolute minima (troughs), or fluctuations within specified limits.

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

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

Value

An RleViews object for Rle or an RleViewsList for RleList containing the views when using `slice` with `rangesOnly = FALSE`. An IRanges object for Rle or a CompressedIRangesList for RleList containing the ranges when using `slice` with `rangesOnly = TRUE`. An XIntegerViews for an XInteger object when using `slice`.

A vector of `length(x)` for RleViews and XIntegerViews objects or a SimpleList object of `length(x)` for RleViewsList objects containing the numeric summaries for the views for `viewMins`, `viewMaxs`, `viewSums`, `viewMeans`, `viewWhichMins`, and `viewWhichMaxs`.

An IRanges object for RleViews objects or a SimpleIRangesList for RleViewsList objects containing the location ranges for `viewRangeMins` and `viewRangeMaxs`.

Author(s)

P. Aboyoun

See Also

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

Examples

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

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

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

`XIntegerViews-class` *The XIntegerViews class*

Description

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

Details

An XIntegerViews object contains a set of views (start/end locations) on the same XInteger object called "the subject integer vector" or simply "the subject". Each view is defined by its start and end locations: both are integers such that start <= end. An XIntegerViews object is in fact a particular case of a Views object (the XIntegerViews class contains the Views class) so it can be manipulated in a similar manner: see `?Views` for more information. Note that two views can overlap and that a view can be "out of limits" i.e. it can start before the first element of the subject or/and end after its last element.

Other methods

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

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

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

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

Author(s)

P. Aboyoun

See Also

Views-class, XInteger-class, Views-utils

Examples

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

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

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

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

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

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

`XVector-class` *XVector objects*

Description

The XVector virtual class is a general container for storing an "external vector". It inherits from the Sequence, which has a very rich interface.

The following classes derive directly from the XVector class:

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

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

The XDouble class is a container for storing an "external double vector" i.e. an external sequence of numeric values (stored as double values at the C level).

Also the XString class from the Biostrings package.

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

Additional Subsetting operations on XVector objects

In the code snippets below, `x` is an XVector 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 XVector 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 XVector object. For example, extracting a 100Mb subsequence from Human chromosome 1 (a 250Mb DNAString object) with `subseq` is (almost) instantaneous and has (almost) no memory footprint (the cost in time and memory does

not depend on the length of the original sequence or on the length of the subsequence to extract).

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

Author(s)

H. Pages

See Also

Sequence-class, Views-class, `solveUserSEW`, DNAString-class

Examples

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

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

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

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

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

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

`XVectorList-class` *XVectorList objects*

Description

THIS IS A WORK-IN-PROGRESS!

An XVectorList object is *conceptually* a list of XVector objects.

Author(s)

H. Pages

See Also

XVector-class

Index