Package 'Seqinfo'

April 7, 2026

Title A simple S4 class for storing basic information about a collection of genomic sequences

Description The Seqinfo class stores the names, lengths, circularity flags, and genomes for a particular collection of sequences. These sequences are typically the chromosomes and/or scaffolds of a specific genome assembly of a given organism. Seqinfo objects are rarely used as standalone objects. Instead, they are used as part of higher-level objects to represent their seqinfo() component. Examples of such higher-level objects are GRanges, RangedSummarizedExperiment, VCF, GAlignments, etc... defined in other Bioconductor infrastructure packages.

biocViews Infrastructure, DataRepresentation, GenomeAssembly, Annotation, GenomeAnnotation

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

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

Version 1.1.0

License Artistic-2.0

Encoding UTF-8

Depends methods, BiocGenerics

Imports stats, S4Vectors (>= 0.47.6), IRanges

Suggests GenomeInfoDb, GenomicRanges, BSgenome, GenomicFeatures, TxDb.Hsapiens.UCSC.hg38.knownGene, TxDb.Dmelanogaster.UCSC.dm3.ensGene, BSgenome.Hsapiens.UCSC.hg38, BSgenome.Celegans.UCSC.ce2, RUnit, knitr, rmarkdown, BiocStyle

VignetteBuilder knitr

Collate utils.R rankSeqlevels.R seqinfo.R sortSeqlevels.R Seqinfo-class.R seqlevelsInUse.R GenomeDescription-class.R zzz.R

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

git_branch devel

git_last_commit 17fc74a

git_last_commit_date 2025-10-29

Repository Bioconductor 3.23

Date/Publication 2026-04-06

Author Hervé Pagès [aut, cre] (ORCID: <https://orcid.org/0009-0002-8272-4522>)

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

Contents

`GenomeDescription-class`
GenomeDescription objects

Description

A GenomeDescription object holds the meta information describing a given genome.

Constructor

Even though a constructor function is provided (`GenomeDescription()`), it is rarely needed GenomeDescription objects are typically obtained by coercing a BSgenome object to GenomeDescription. This has the effect of stripping the sequences from the object and retaining only the meta information that describes the genome. See the Examples section below for an example.

Accessor methods

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

`organism(object)`: Return the scientific name of the organism of the genome e.g. `"Homo sapiens"`, `"Mus musculus"`, `"Caenorhabditis elegans"`, etc...

`commonName(object)`: Return the common name of the organism of the genome e.g. `"Human"`, `"Mouse"`, `"Worm"`, etc...

`providerVersion(x)`: Return the *name* of the genome. This is typically the name of an NCBI assembly (e.g. `GRCh38.p13`, `WBcel235`, `TAIR10.1`, `ARS-UCD1.2`, etc...) or UCSC genome (e.g. `hg38`, `bosTau9`, `galGal6`, `ce11`, etc...).

`provider(x)`: Return the provider of this genome e.g. `"UCSC"`, `"BDGP"`, `"FlyBase"`, etc...

`releaseDate(x)`: Return the release date of this genome e.g. `"Mar. 2006"`.

`bsgenomeName(x)`: Uses the meta information stored in GenomeDescription object `x` to construct the name of the corresponding BSgenome data package (see the `available.genomes` function in the **BSgenome** package for details about the naming scheme used for those packages). Note that there is no guarantee that a package with that name actually exists.

`seqinfo(x)` Gets information about the genome sequences. This information is returned in a Seqinfo object. Each part of the information can be retrieved separately with `seqnames(x)`, `seqlengths(x)`, and `isCircular(x)`, respectively, as described below.

`seqnames(x)` Gets the names of the genome sequences. `seqnames(x)` is equivalent to `seqnames(seqinfo(x))`.

`seqlengths(x)` Gets the lengths of the genome sequences. `seqlengths(x)` is equivalent to `seqlengths(seqinfo(x))`.

`isCircular(x)` Returns the circularity flags of the genome sequences. `isCircular(x)` is equivalent to `isCircular(seqinfo(x))`.

Author(s)

H. Pagès

See Also

- The `available.genomes` function and the BSgenome class in the **BSgenome** package.
- The Seqinfo class.

Examples

```
library(BSgenome.Celegans.UCSC.ce2)
BSgenome.Celegans.UCSC.ce2
as(BSgenome.Celegans.UCSC.ce2, "GenomeDescription")
```

---

rankSeqlevels *Assign sequence IDs to sequence names*

Description

`rankSeqlevels` assigns a unique ID to each unique sequence name in the input vector. The returned IDs span 1:N where N is the number of unique sequence names in the input vector.

`orderSeqlevels` is similar to `rankSeqlevels` except that the returned vector contains the order instead of the rank.

Usage

```
rankSeqlevels(seqnames, X.is.sexchrom=NA)
orderSeqlevels(seqnames, X.is.sexchrom=NA)
```

Arguments

seqnames	A character vector or factor containing sequence names.
X.is.sexchrom	A logical indicating whether X refers to the sexual chromosome or to chromosome with Roman Numeral X. If NA, rankSeqlevels does its best to "guess".

Value

An integer vector of the same length as seqnames that tries to reflect the "natural" order of seqnames, e.g.,chr1, chr2, chr3, ...

The values in the returned vector span 1:N where N is the number of unique sequence names in the input vector.

Author(s)

H. Pagès for rankSeqlevels, orderSeqlevels added by Sonali Arora

See Also

- sortSeqlevels for sorting the sequence levels of an object in "natural" order.

Examples

```
si <- Seqinfo(genome="sacCer2")
rankSeqlevels(seqnames(si))
rankSeqlevels(seqnames(si)[c(1:5,5:1)])

newchr <- paste0("chr",c(1:3,6:15,4:5,16:22))
newchr
orderSeqlevels(newchr)
rankSeqlevels(newchr)
```

---

seqinfo *Accessing/modifying sequence information*

Description

A set of generic functions for getting/setting/modifying the sequence information stored in an object.

Usage

```
seqinfo(x)
seqinfo(x,
        new2old=NULL,
        pruning.mode=c("error", "coarse", "fine", "tidy")) <- value

seqnames(x)
```

```
seqnames(x) <- value

seqlevels(x)
seqlevels(x,
          pruning.mode=c("error", "coarse", "fine", "tidy")) <- value
seqlevels0(x)
restoreSeqlevels(x)

seqlengths(x)
seqlengths(x) <- value

isCircular(x)
isCircular(x) <- value

genome(x)
genome(x) <- value
```

Arguments

x	Any object containing sequence information i.e. with a `seqinfo()` component.

new2old	The `new2old` argument allows the user to rename, drop, add and/or reorder the "sequence levels" in `x`.
	`new2old` can be `NULL` or an integer vector with one element per entry in Seqinfo object `value` (i.e. `new2old` and `value` must have the same length) describing how the "new" sequence levels should be mapped to the "old" sequence levels, that is, how the entries in `value` should be mapped to the entries in `seqinfo(x)`. The values in `new2old` must be >= 1 and <= `length(seqinfo(x))`. NAs are allowed and indicate sequence levels that are being added. Old sequence levels that are not represented in `new2old` will be dropped, but this will fail if those levels are in use (e.g. if `x` is a GRanges object with ranges defined on those sequence levels) unless a pruning mode is specified via the `pruning.mode` argument (see below).
	If `new2old=NULL`, then sequence levels can only be added to the existing ones, that is, `value` must have at least as many entries as `seqinfo(x)` (i.e. `length(values) >= length(seqinfo(x))`) and also `seqlevels(values)[seq_len(length(seqlevels(x)))]` must be identical to `seqlevels(x)`.
	Note that most of the times it's easier to proceed in 2 steps:
	1. First align the seqlevels on the left (`seqlevels(x)`) with the seqlevels on the right.
	2. Then call `seqinfo(x) <- value`. Because `seqlevels(x)` and `seqlevels(value)` now are identical, there's no need to specify `new2old`.
	This 2-step approach will typically look like this:
	`seqlevels(x) <- seqlevels(value)  # align seqlevels`
	`seqinfo(x) <- seqinfo(value)  # guaranteed to work`
	Or, if `x` has seqlevels not in `value`, it will look like this:

```
seqlevels(x, pruning.mode="coarse") <- seqlevels(value)
seqinfo(x) <- seqinfo(value)  # guaranteed to work
```

The `pruning.mode` argument will control what happens to `x` when some of its seqlevels get droppped. See below for more information.

`pruning.mode` When some of the seqlevels to drop from `x` are in use (i.e. have ranges on them), the ranges on these sequences need to be removed before the seqlevels can be dropped. We call this *pruning*. The `pruning.mode` argument controls how to *prune* `x`. Four pruning modes are currently defined: `"error"`, `"coarse"`, `"fine"`, and `"tidy"`. `"error"` is the default. In this mode, no pruning is done and an error is raised. The other pruning modes do the following:

- `"coarse"`: Remove the elements in `x` where the seqlevels to drop are in use. Typically reduces the length of `x`. Note that if `x` is a list-like object (e.g. GRangesList, GAlignmentPairs, or GAlignmentsList), then any list element in `x` where at least one of the sequence levels to drop is in use is *fully* removed. In other words, when `pruning.mode="coarse"`, the `seqlevels` setter will keep or remove *full list elements* and not try to change their content. This guarantees that the exact ranges (and their order) inside the individual list elements are preserved. This can be a desirable property when the list elements represent compound features like exons grouped by transcript (stored in a GRangesList object as returned by `exonsBy( , by="tx")`), or paired-end or fusion reads, etc...
- `"fine"`: Supported on list-like objects only. Removes the ranges that are on the sequences to drop. This removal is done within each list element of the original object `x` and doesn't affect its length or the order of its list elements. In other words, the pruned object is guaranteed to be *parallel* to the original object.
- `"tidy"`: Like the `"fine"` pruning above but also removes the list elements that become empty as the result of the pruning. Note that this pruning mode is particularly well suited on a GRangesList object that contains transcripts grouped by gene, as returned by `transcriptsBy( , by="gene")`. Finally note that, as a convenience, this pruning mode is supported on non list-like objects (e.g. GRanges or GAlignments objects) and, in this case, is equivalent to the `"coarse"` mode.

See the "B. DROP SEQLEVELS FROM A LIST-LIKE OBJECT" section in the examples below for an extensive illustration of these pruning modes.

`value` Typically a Seqinfo object for the `seqinfo` setter.

Either a named or unnamed character vector for the `seqlevels` setter.

A vector containing the sequence information to store for the other setters.

Details

"It all revolves around Seqinfo objects"

The Seqinfo class plays a central role for the functions described in this man page because:

1. All these functions (except `seqinfo`, `seqlevels0`, and `restoreSeqlevels`) work on a Seqinfo object.

2. For classes that implement it, the `seqinfo` getter should return a Seqinfo object.
3. Default `seqlevels`, `seqlengths`, `isCircular`, and `genome` getters and setters are provided. By default, `seqlevels(x)` does `seqlevels(seqinfo(x))`, `seqlengths(x)` does `seqlengths(seqinfo(x))`, `isCircular(x)` does `isCircular(seqinfo(x))`, and `genome(x)` does `genome(seqinfo(x))`. So any class with a `seqinfo` getter will have all the above getters work out-of-the-box. If, in addition, the class defines a `seqinfo` setter, then all the corresponding setters will also work out-of-the-box.

   Examples of containers that have a `seqinfo` getter and setter:

   - the GRanges and GRangesList classes in the **GenomicRanges** package;
   - the SummarizedExperiment class in the **SummarizedExperiment** package;
   - the GAlignments, GAlignmentPairs, and GAlignmentsList classes in the **GenomicAlignments** package;
   - the TxDb class in the **GenomicFeatures** package;
   - the BSgenome class in the **BSgenome** package;
   - and more...

Value

The `seqinfo()` getter returns a Seqinfo object.

The `seqnames()`, `seqlevels()`, and `seqlevels0()` getters return a character vector with no `NA`s.

`restoreSeqlevels()` returns input object `x` with its original seqlevels restored i.e. reset to `seqlevels0(x)`.

The `seqlengths()` getter returns a named integer vector, possibly with `NA`s.

The `isCircular()` getter returns a named logical vector, possibly with `NA`s.

The `genome()` getter returns a named character vector, possibly with `NA`s.

Note

The full list of methods defined for a given generic function can be seen with e.g. `showMethods("seqinfo")` or `showMethods("seqnames")` (for the getters), and `showMethods("seqinfo<-")` or `showMethods("seqnames<-")` (for the setters a.k.a. *replacement methods*). Please be aware that this shows only methods defined in packages that are currently attached.

The **GenomicRanges** package defines `seqinfo` and `seqinfo<-` methods for these low-level data types: List and IntegerRangesList. Those objects do not have the means to formally store sequence information. Thus, the wrappers simply store the Seqinfo object within `metadata(x)`. Initially, the metadata is empty, so there is some effort to generate a reasonable default Seqinfo. The names of any List are taken as the `seqnames`, and the `universe` of IntegerRangesList is taken as the `genome`.

Author(s)

H. Pagès

See Also

- The `seqlevelsStyle` generic getter and setter in the **GenomeInfoDb** package for conveniently renaming the seqlevels of an object according to a particular naming convention (e.g. NCBI or UCSC).

- Seqinfo objects.
- GRanges and GRangesList objects in the **GenomicRanges** package.
- SummarizedExperiment objects in the **SummarizedExperiment** package.
- GAlignments, GAlignmentPairs, and GAlignmentsList objects in the **GenomicAlignments** package.
- TxDb objects in the **GenomicFeatures** package.
- BSgenome objects in the **BSgenome** package.
- `sortSeqlevels` for sorting the sequence levels of an object in "natural" order.
- `seqlevelsInUse` for getting the sequence levels that are used by an object.
- seqlevels-wrappers in the **GenomeInfoDb** package for convenience wrappers to the `seqlevels` getter and setter.

Examples

```
## ---------------------------------------------------------------------
## A. BASIC USAGE OF THE seqlevels() GETTER AND SETTER
## ---------------------------------------------------------------------
## Operations between 2 or more objects containing genomic ranges (e.g.
## finding overlaps, comparing, or matching) only make sense if the
## operands have the same seqlevels. So before performing such
## operations, it is often necessary to adjust the seqlevels in
## the operands so that they all have the same seqlevels. This is
## typically done with the seqlevels() setter. The setter can be used
## to rename, drop, add and/or reorder seqlevels of an object. The
## examples below show how to mofify the seqlevels of a GRanges object
## but the same would apply to any object containing sequence
## information (i.e. with a seqinfo() component).
library(GenomicRanges)

gr <- GRanges(rep(c("chr2", "chr3", "chrM"), 2), IRanges(1:6, 10))

## Add new seqlevels:
seqlevels(gr) <- c("chr1", seqlevels(gr), "chr4")
seqlevels(gr)

## Reorder existing seqlevels:
seqlevels(gr) <- rev(seqlevels(gr))
seqlevels(gr)

## Drop some seqlevels in use:
seqlevels(gr, pruning.mode="coarse") <- setdiff(seqlevels(gr), "chr3")
gr

## Rename, add, and reorder the seqlevels all at once:
seqlevels(gr) <- c("chr1", chr2="chr2", chrM="Mitochondrion")
seqlevels(gr)

## ---------------------------------------------------------------------
## B. DROP SEQLEVELS FROM A LIST-LIKE OBJECT
```

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

grl0 <- GRangesList(A=GRanges("chr2", IRanges(3:2, 5)),
                    B=GRanges(c("chr2", "chrMT"), IRanges(7:6, 15)),
                    C=GRanges(c("chrY", "chrMT"), IRanges(17:16, 25)),
                    D=GRanges())
grl0

grl1 <- grl0
seqlevels(grl1, pruning.mode="coarse") <- c("chr2", "chr5")
grl1  # grl0[[2]] was fully removed! (even if it had a range on chr2)

## If what is desired is to remove the 2nd range in grl0[[2]] only (i.e.
## the chrMT:6-15 range), or, more generally speaking, to remove the
## ranges within each list element that are located on the seqlevels to
## drop, then use pruning.mode="fine" or pruning.mode="tidy":
grl2 <- grl0
seqlevels(grl2, pruning.mode="fine") <- c("chr2", "chr5")
grl2  # grl0[[2]] not removed, but chrMT:6-15 range removed from it

## Like pruning.mode="fine" but also removes grl0[[3]].
grl3 <- grl0
seqlevels(grl3, pruning.mode="tidy") <- c("chr2", "chr5")
grl3

library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene
## Pruning mode "coarse" is particularly well suited on a GRangesList
## object that contains exons grouped by transcript:
ex_by_tx <- exonsBy(txdb, by="tx")
seqlevels(ex_by_tx)
seqlevels(ex_by_tx, pruning.mode="coarse") <- "chr2L"
seqlevels(ex_by_tx)
## Pruning mode "tidy" is particularly well suited on a GRangesList
## object that contains transcripts grouped by gene:
tx_by_gene <- transcriptsBy(txdb, by="gene")
seqlevels(tx_by_gene)
seqlevels(tx_by_gene, pruning.mode="tidy") <- "chr2L"
seqlevels(tx_by_gene)

## ---------------------------------------------------------------------
## C. RENAME THE SEQLEVELS OF A TxDb OBJECT
## ---------------------------------------------------------------------

library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene
seqlevels(txdb)

seqlevels(txdb) <- sub("chr", "", seqlevels(txdb))
seqlevels(txdb)

seqlevels(txdb) <- paste0("CH", seqlevels(txdb))
seqlevels(txdb)
```

```
seqlevels(txdb)[seqlevels(txdb) == "CHM"] <- "M"
seqlevels(txdb)

## Restore original seqlevels:
txdb <- restoreSeqlevels(txdb)  # same as
                                # seqlevels(txdb) <- seqlevels0(txdb)
seqlevels(txdb)

## Note that seqlevels0() and restoreSeqlevels() only work on TxDb
## objects at the moment.

## ---------------------------------------------------------------------
## D. SUBSET OBJECTS BY SEQLEVELS
## ---------------------------------------------------------------------

tx <- transcripts(txdb)
seqlevels(tx)

## Drop 'M', keep all others.
seqlevels(tx, pruning.mode="coarse") <- seqlevels(tx)[seqlevels(tx) != "M"]
seqlevels(tx)

## Drop all except 'ch3L' and 'ch3R'.
seqlevels(tx, pruning.mode="coarse") <- c("ch3L", "ch3R")
seqlevels(tx)

## ---------------------------------------------------------------------
## E. FINDING METHODS
## ---------------------------------------------------------------------

showMethods("seqinfo")
showMethods("seqinfo<-")

showMethods("seqnames")
showMethods("seqnames<-")

showMethods("seqlevels")
showMethods("seqlevels<-")

if (interactive()) {
  library(GenomicRanges)
  ?`GRanges-class`
}
```

Seqinfo-class *Seqinfo objects*

Description

A Seqinfo object is used to store basic information about a set of genomic sequences, typically chromosomes (but not necessarily).

Details

A Seqinfo object has one entry per sequence. Each entry contains the following information about the sequence:

- The sequence name (a.k.a. the *seqlevel*) e.g. `"chr1"`.
- The sequence length.
- The sequence *circularity flag*. This is a logical indicating whether the sequence is circular (`TRUE`) or linear (`FALSE`).
- Which genome the sequence belongs to e.g. `"hg19"`.

All entries must contain at least the sequence name. The other information is optional. In addition, the *seqnames* in a given Seqinfo object must be unique, that is, the object is not allowed to have two entries with the same sequence name. In other words, the sequence name is used as the *primary key* of a Seqinfo object.

Note that Seqinfo objects are usually not used as standalone objects but are instead typically found inside higher level objects like GRanges or TxDb objects. These higher level objects will generally provide a `seqinfo()` accessor for getting/setting their Seqinfo component.

Constructor

`Seqinfo(seqnames, seqlengths=NA, isCircular=NA, genome=NA)`**:** Create a Seqinfo object and populate it with the supplied data.

One special form of calling the `Seqinfo()` constructor is to specify only the `genome` argument and set it to the name of an NCBI assembly (e.g. `Seqinfo(genome="GRCh38.p13")`) or UCSC genome (e.g. `Seqinfo(genome="hg38")`), in which case the sequence information is fetched from NCBI or UCSC. See Examples section below for some examples.

Accessor methods

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

`length(x)`**:** Return the number of sequences in `x`.

`seqnames(x)`**,** `seqnames(x) <- value`**:** Get/set the names of the sequences in `x`. Those names must be non-NA, non-empty and unique. They are also called the *sequence levels* or the *keys* of the Seqinfo object.

Note that, in general, the end user should not try to alter the sequence levels with `seqnames(x) <- value`. The recommended way to do this is with `seqlevels(x) <- value` as described below.

`names(x)`**,** `names(x) <- value`**:** Same as `seqnames(x)` and `seqnames(x) <- value`.

`seqlevels(x)`**:** Same as `seqnames(x)`.

`seqlevels(x) <- value`: Can be used to rename, drop, add and/or reorder the sequence levels. `value` must be either a named or unnamed character vector. When `value` has names, the names only serve the purpose of mapping the new sequence levels to the old ones. Otherwise (i.e. when `value` is unnamed) this mapping is implicitly inferred from the following rules:

(1) If the number of new and old levels are the same, and if the positional mapping between the new and old levels shows that some or all of the levels are being renamed, and if the levels that are being renamed are renamed with levels that didn't exist before (i.e. are not present in the old levels), then `seqlevels(x) <- value` will just rename the sequence levels. Note that in that case the result is the same as with `seqnames(x) <- value` but it's still recommended to use `seqlevels(x) <- value` as it is safer.

(2) Otherwise (i.e. if the conditions for (1) are not satisfied) `seqlevels(x) <- value` will consider that the sequence levels are not being renamed and will just perform `x <- x[value]`.

See below for some examples.

`seqlengths(x)`, `seqlengths(x) <- value`: Get/set the length for each sequence in `x`.

`isCircular(x)`, `isCircular(x) <- value`: Get/set the circularity flag for each sequence in `x`.

`genome(x)`, `genome(x) <- value`: Get/set the genome identifier or assembly name for each sequence in `x`.

Subsetting

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

`x[i]`: A Seqinfo object can be subsetted only by name i.e. `i` must be a character vector. This is a convenient way to drop/add/reorder the entries in a Seqinfo object.

See below for some examples.

Coercion

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

`as.data.frame(x)`: Turns `x` into a data frame.

Combining Seqinfo objects

Note that we provide no `c()` or `rbind()` methods for Seqinfo objects. Here is why:

`c()` (like `rbind()`) is expected to follow an "appending semantic", that is, `c(x, y)` is expected to form a new object by *appending* the entries in `y` to the entries in `x`, thus resulting in an object with `length(x) + length(y)` entries. The problem with such operation is that it won't be very useful in general, because it will tend to break the constraint that the seqnames of a Seqinfo object must be unique (primary key).

So instead, a `merge()` method is provided, with a more useful semantic. `merge(x, y)` does the following:

- If an entry in Seqinfo object `x` has the same seqname as an entry in Seqinfo object `y`, then the 2 entries are fusioned together to produce a single entry in the result. This fusion only happens if the 2 entries contain compatible information.

- If 2 entries cannot be fusioned because they contain incompatible information (e.g. different seqlengths or different circularity flags), then `merge(x, y)` fails with an informative error of why `x` and `y` could not be merged.

We also implement an `update()` method for Seqinfo objects.

See below for the details.

In the code snippet below, `x`, `y`, `object`, and `value`, are Seqinfo objects.

`merge(x, y, ...)`**:** Merge `x` and `y` into a single Seqinfo object where the keys (i.e. the seqnames) are `union(seqnames(x), seqnames(y))`. If an entry in `y` has the same key as an entry in `x`, and if the two entries contain compatible information (NA values are treated as wildcards i.e. they're compatible with anything), then the two entries are merged into a single entry in the result. If they cannot be merged (because they contain different seqlengths, and/or circularity flags, and/or genome identifiers), then an error is raised. In addition to check for incompatible sequence information, `merge(x, y)` also compares `seqnames(x)` with `seqnames(y)` and issues a warning if each of them has names not in the other. The purpose of these checks is to try to detect situations where the user might be combining or comparing objects that use different underlying genomes.

Note that `merge()` can take more than two Seqinfo objects, in which case the objects are merged from left to right e.g.

```
merge(x1, x2, x3, x4)
```

is equivalent to

```
merge(merge(merge(x1, x2), x3), x4)
```

`intersect(x, y)`**:** Finds the intersection between two `Seqinfo` objects by merging them and subsetting for the intersection of their sequence names. This makes it easy to avoid warnings about each objects not being a subset of the other one during overlap operations.

`update(object, value)`**:** Update the entries in Seqinfo object `object` with the corresponding entries in Seqinfo object `value`. Note that the seqnames in `value` must be a subset of the seqnames in `object`.

A convenience wrapper, `checkCompatibleSeqinfo()`, is provided for checking whether 2 objects have compatible Seqinfo components or not. `checkCompatibleSeqinfo(x, y)` is equivalent to `merge(seqinfo(x), seqinfo(y))` so will work on any objects `x` and `y` that support `seqinfo()`.

Author(s)

H. Pagès

See Also

- The `seqinfo` getter and setter.
- The `getChromInfoFromNCBI` and `getChromInfoFromUCSC` utility functions in the **GenomeInfoDb** package that are used behind the scene to generate a Seqinfo object for a given assembly/genome (see examples below).

Examples

```
## ---------------------------------------------------------------------
## A. MAKING A Seqinfo OBJECT FOR A GIVEN NCBI ASSEMBLY OR UCSC GENOME
## ---------------------------------------------------------------------

## One special form of calling the 'Seqinfo()' constructor is to specify
## only the 'genome' argument and set it to the name of an NCBI assembly
## or UCSC genome, in which case the sequence information is fetched
## from NCBI or UCSC ('GenomeInfoDb::getChromInfoFromNCBI()' or
## 'GenomeInfoDb::getChromInfoFromUCSC()' are used behind the scene
## for this so internet access is required).

if (interactive()) {
  ## NCBI assemblies (see '?registered_NCBI_assemblies' for the list of
  ## NCBI assemblies that are currently supported):
  Seqinfo(genome="GRCh38")
  Seqinfo(genome="GRCh38.p13")
  Seqinfo(genome="Amel_HAv3.1")
  Seqinfo(genome="WBcel235")
  Seqinfo(genome="TAIR10.1")

  ## UCSC genomes (see '?registered_UCSC_genomes' for the list of UCSC
  ## genomes that are currently supported):
  Seqinfo(genome="hg38")
  Seqinfo(genome="mm10")
  Seqinfo(genome="rn6")
  Seqinfo(genome="bosTau9")
  Seqinfo(genome="canFam3")
  Seqinfo(genome="musFur1")
  Seqinfo(genome="galGal6")
  Seqinfo(genome="dm6")
  Seqinfo(genome="ce11")
  Seqinfo(genome="sacCer3")
}

## ---------------------------------------------------------------------
## B. BASIC MANIPULATION OF A Seqinfo OBJECT
## ---------------------------------------------------------------------

## Note that all the arguments (except 'genome') must have the
## same length. 'genome' can be of length 1, whatever the lengths
## of the other arguments are.
x <- Seqinfo(seqnames=c("chr1", "chr2", "chr3", "chrM"),
             seqlengths=c(100, 200, NA, 15),
             isCircular=c(NA, FALSE, FALSE, TRUE),
             genome="sasquatch")
x

## Accessors:
length(x)
seqnames(x)
names(x)
```

```
seqlevels(x)
seqlengths(x)
isCircular(x)
genome(x)

## Get a compact summary:
summary(x)

## Subset by names:
x[c("chrY", "chr3", "chr1")]

## Rename, drop, add and/or reorder the sequence levels:
xx <- x
seqlevels(xx) <- sub("chr", "ch", seqlevels(xx))  # rename
xx
seqlevels(xx) <- rev(seqlevels(xx))  # reorder
xx
seqlevels(xx) <- c("ch1", "ch2", "chY")  # drop/add/reorder
xx
seqlevels(xx) <- c(chY="Y", ch1="1", "22")  # rename/reorder/drop/add
xx

## ---------------------------------------------------------------------
## C. COMBINING 2 Seqinfo OBJECTS
## ---------------------------------------------------------------------

y <- Seqinfo(seqnames=c("chr3", "chr4", "chrM"),
             seqlengths=c(300, NA, 15))
y

## ------ merge() ------

## This issues a warning:
merge(x, y)  # the entries for chr3 and chrM contain information merged
             # from the corresponding entries in 'x' and 'y'

## To get rid of the above warning, either use suppressWarnings() or
## set the genome on 'y':
suppressWarnings(merge(x, y))
genome(y) <- genome(x)
merge(x, y)

## Note that, strictly speaking, merging 2 Seqinfo objects is not
## a commutative operation:
merge(y, x)

## More precisely: In general, 'z1 <- merge(x, y)' is not identical
## to 'z2 <- merge(y, x)'. However 'z1' and 'z2' are guaranteed to
## contain the same information but with their entries possibly in
## different order.

## This contradicts what 'x' says about circularity of chr3 and chrM:
yy <- y
```

```
isCircular(yy)[c("chr3", "chrM")] <- c(TRUE, FALSE)

## We say that 'x' and 'yy' are incompatible Seqinfo objects.
yy
if (interactive()) {
  merge(x, yy)  # raises an error
}

## Sanity checks:
stopifnot(identical(x, merge(x, Seqinfo())))
stopifnot(identical(x, merge(Seqinfo(), x)))
stopifnot(identical(x, merge(x, x)))

## ------ update() ------

z <- Seqinfo(seqnames=c("chrM", "chr2", "chr3"),
             seqlengths=c(25, NA, 300),
             genome="chupacabra")
z

update(x, z)

if (interactive()) {
  update(z, x)  # not allowed
  update(x, y)  # not allowed
}

## The seqnames in the 2nd argument can always be forced to be a subset
## of the seqnames in the 1st argument with:
update(x, y[intersect(seqnames(x), seqnames(y))])  # replace entries

## Note that the above is not the same as:
merge(x, y)[seqnames(x)]                           # fusion entries

## The former is guaranteed to work, whatever the Seqinfo objects 'x'
## and 'y'. The latter requires 'x' and 'y' to be compatible.

## Sanity checks:
stopifnot(identical(x, update(x, Seqinfo())))
stopifnot(identical(x, update(x, x)))
stopifnot(identical(z, update(x, z)[seqnames(z)]))

## ---------------------------------------------------------------------
## D. checkCompatibleSeqinfo()
## ---------------------------------------------------------------------
## A simple convenience wrapper to check that 2 objects have compatible
## Seqinfo components.

library(GenomicRanges)
gr1 <- GRanges("chr3:15-25", seqinfo=x)
gr2 <- GRanges("chr3:105-115", seqinfo=y)
if (interactive()) {
  checkCompatibleSeqinfo(gr1, gr2)  # raises an error
```

```
}
```

`seqlevelsInUse` *Get the sequence levels in use*

Description

Get the sequence levels that are "used" by an object, that is, the sequence levels on which the object defines genomic ranges or features.

Usage

```
seqlevelsInUse(x)
```

Arguments

`x`	Any object containing sequence information i.e. with a `seqinfo()` component.

Value

The sequence levels in use in a character vector.

Author(s)

H. Pagès

See Also

- The `seqlevels` getter and setter.
- Seqinfo objects.
- GRanges and GRangesList objects in the **GenomicRanges** package.

Examples

```
library(GenomicRanges)

gr <- GRanges(rep(c("chr2", "chr3", "chrM"), 2), IRanges(1:6, 10))

## Add new seqlevels:
seqlevels(gr) <- c("chr1", seqlevels(gr), "chr4")
seqlevels(gr)
seqlevelsInUse(gr)

## Drop all unused seqlevels:
seqlevels(gr) <- seqlevelsInUse(gr)
```

`sortSeqlevels` *Sort the sequence levels of an object*

Description

A generic function and methods to sort the sequence levels of an object in "natural" order.

Usage

```
sortSeqlevels(x, X.is.sexchrom=NA)
```

Arguments

`x`	Any object containing sequence information i.e. with a `seqinfo()` component.
`X.is.sexchrom`	A logical indicating whether X refers to the sexual chromosome or to chromosome with Roman Numeral X. If `NA`, `sortSeqlevels` does its best to "guess".

Value

The input object `x` with its seqlevels sorted.

Author(s)

H. Pagès

See Also

- The `seqlevels` getter and setter.
- `rankSeqlevels`, on which `sortSeqlevels` is based.
- Seqinfo objects.
- GRanges and GRangesList objects in the **GenomicRanges** package.

Examples

```
sortSeqlevels(c("11", "Y", "1", "10", "9", "M", "2"))

seqlevels <- c("chrXI", "chrY", "chrI", "chrX", "chrIX", "chrM", "chrII")
sortSeqlevels(seqlevels)
sortSeqlevels(seqlevels, X.is.sexchrom=TRUE)
sortSeqlevels(seqlevels, X.is.sexchrom=FALSE)

seqlevels <- c("chr2RHet", "chr4", "chrUextra", "chrYHet",
               "chrM", "chrXHet", "chr2LHet", "chrU",
               "chr3L", "chr3R", "chr2R", "chrX")
sortSeqlevels(seqlevels)

library(GenomicRanges)
gr <- GRanges()
```

```
seqlevels(gr) <- seqlevels
sortSeqlevels(gr)
```

Index