Package 'GenomeInfoDb'

March 24, 2019

Title Utilities for manipulating chromosome and other 'seqname' identifiers

Description Contains data and functions that define and allow translation between different chromosome sequence naming conventions (e.g., ``chr1'' versus ``1''), including a function that attempts to place sequence names in their natural, rather than lexicographic, order.

Version 1.19.2

Encoding UTF-8

Author Sonali Arora, Martin Morgan, Marc Carlson, H. Pagès

Maintainer Bioconductor Package Maintainer <maintainer@bioconductor.org>

biocViews Genetics, DataRepresentation, Annotation, GenomeAnnotation

Depends R (>= 3.1), methods, BiocGenerics (>= 0.13.8), S4Vectors (>= 0.17.25), IRanges (>= 2.13.12)

Imports stats, stats4, utils, RCurl, GenomeInfoDbData

Suggests GenomicRanges, Rsamtools, GenomicAlignments, BSgenome, GenomicFeatures, BSgenome.Scerevisiae.UCSC.sacCer2, BSgenome.Celegans.UCSC.ce2, BSgenome.Hsapiens.NCBI.GRCh38, TxDb.Dmelanogaster.UCSC.dm3.ensGene, RUnit, BiocStyle, knitr

License Artistic-2.0

Collate utils.R rankSeqlevels.R assembly-utils.R loadTaxonomyDb.R fetchExtendedChromInfoFromUCSC.R fetchSequenceInfo.R seqinfo.R seqlevelsStyle.R seqlevels-wrappers.R Seqinfo-class.R GenomeDescription-class.R mapGenomeBuilds.R zzz.R

VignetteBuilder knitr

Video http://youtu.be/wdEjCYSXa7w

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

git_branch master

git_last_commit 18c1056

git_last_commit_date 2019-02-11

Date/Publication 2019-03-23

R topics documented:

`fetchExtendedChromInfoFromUCSC` *Fetching chromosomes info for some of the UCSC genomes*

Description

Fetch the chromosomes info for some UCSC genomes. Only supports hg38, hg19, hg18, panTro4, panTro3, panTro2, bosTau8, bosTau7, bosTau6, canFam3, canFam2, canFam1, musFur1, mm10, mm9, mm8, susScr3, susScr2, rn6, rheMac3, rheMac2, galGal4, galGal3, gasAcu1, danRer7, apiMel2, dm6, dm3, ce10, ce6, ce4, ce2, sacCer3, and sacCer2 at the moment.

Usage

```
fetchExtendedChromInfoFromUCSC(genome,
        goldenPath_url="http://hgdownload.cse.ucsc.edu/goldenPath",
        quiet=FALSE)
```

Arguments

`genome`	A single string specifying the UCSC genome e.g. `"sacCer3"`.
`goldenPath_url`	A single string specifying the URL to the UCSC goldenPath location. This URL is used internally to build the full URL to the 'chromInfo' MySQL dump containing chromosomes information for `genome`. See Details section below.
`quiet`	`TRUE` or `FALSE` (the default). If `TRUE` then some warnings are suppressed. See below for the details.

Details

Chromosomes information (e.g. names and lengths) for any UCSC genome is stored in the UCSC database in the 'chromInfo' table, and is normally available as a MySQL dump at:

```
goldenPath_url/<genome>/database/chromInfo.txt.gz
```

`fetchExtendedChromInfoFromUCSC` downloads and imports that table into a data frame, keeps only the `UCSC_seqlevel` and `UCSC_seqlength` columns (after renaming them), and adds the `circular` logical column.

Then, if this UCSC genome is based on an NCBI assembly (e.g. hg38 is based on GRCh38), the NCBI seqlevels and GenBank accession numbers are extracted from the NCBI assembly report and the UCSC seqlevels matched to them (using some guided heuristic). Finally the NCBI seqlevels and GenBank accession numbers are added to the returned data frame.

Value

A data frame with 1 row per seqlevel in the UCSC genome, and at least 3 columns:

- `UCSC_seqlevel`: Character vector with no NAs. This is the `chrom` field of the UCSC 'chromInfo' table for the genome. See Details section above.
- `UCSC_seqlength`: Integer vector with no NAs. This is the `size` field of the UCSC 'chromInfo' table for the genome. See Details section above.
- `circular`: Logical vector with no NAs. This knowledge is stored in the **GenomeInfoDb** package itself for the supported genomes.

If the UCSC genome is *not* based on an NCBI assembly (e.g. gasAcu1, ce10, sacCer2), there are no additional columns and a warning is emitted (unless `quiet` is set to `TRUE`). In this case, the rows are sorted by UCSC seqlevel rank as determined by `rankSeqlevels()`.

If the UCSC genome is based on an NCBI assembly (e.g. sacCer3), the returned data frame has 3 additional columns:

- `NCBI_seqlevel`: Character vector. This information is obtained from the NCBI assembly report for the genome. Will contain NAs for UCSC seqlevels with no corresponding NCBI seqlevels (e.g. for chrM in hg18 or chrUextra in dm3), in which case `fetchExtendedChromInfoFromUCSC` emits a warning (unless `quiet` is set to `TRUE`).
- `SequenceRole`: Factor with levels `assembled-molecule`, `alt-scaffold`, `unlocalized-scaffold`, `unplaced-scaffold`, and `pseudo-scaffold`. For UCSC seqlevels with corresponding NCBI seqlevels this information is obtained from the NCBI assembly report. Otherwise it is obtained from a base of knowledge included in the **GenomeInfoDb** package. Can contain NAs but no warning is emitted in that case.
- `GenBankAccn`: Character vector. This information is obtained from the NCBI assembly report for the genome. Can contain NAs but no warning is emitted in that case.

In this case, the rows are sorted first by level in the `SequenceRole` column, that is, `assembled-molecules` first, then `alt-scaffolds`, etc, and NAs last. Then within each group they are sorted by UCSC seqlevel rank as determined by `rankSeqlevels()`.

Note

`fetchExtendedChromInfoFromUCSC` queries the UCSC Genome Browser as well as the FTP site at NCBI and thus requires internet access.

Only supports the hg38, hg19, hg18, panTro4, panTro3, panTro2, bosTau8, bosTau7, bosTau6, canFam3, canFam2, canFam1, musFur1, mm10, mm9, mm8, susScr3, susScr2, rn6, rheMac3, rheMac2, galGal4, galGal3, gasAcu1, danRer7, apiMel2, dm6, dm3, ce10, ce6, ce4, ce2, sacCer3, and sacCer2 genomes at the moment. More will come...

Author(s)

H. Pagès

See Also

- The `seqlevels` getter and setter.
- The `rankSeqlevels` function for ranking sequence names.
- The `seqlevelsStyle` getter and setter.
- The `getBSgenome` utility in the **BSgenome** package for searching the installed BSgenome data packages.

Examples

```
## All the examples below require internet access!

## ---------------------------------------------------------------------
## A. BASIC EXAMPLE
## ---------------------------------------------------------------------

## The sacCer3 UCSC genome is based on an NCBI assembly (RefSeq Assembly
## ID is GCF_000146045.2):
sacCer3_chrominfo <- fetchExtendedChromInfoFromUCSC("sacCer3")
sacCer3_chrominfo

## But the sacCer2 UCSC genome is not:
sacCer2_chrominfo <- fetchExtendedChromInfoFromUCSC("sacCer2")
sacCer2_chrominfo

## ---------------------------------------------------------------------
## B. USING fetchExtendedChromInfoFromUCSC() TO PUT UCSC SEQLEVELS ON
##    THE GRCh38 GENOME
## ---------------------------------------------------------------------

## Load the BSgenome.Hsapiens.NCBI.GRCh38 package:
library(BSgenome)
genome <- getBSgenome("GRCh38")  # this loads the
                                 # BSgenome.Hsapiens.NCBI.GRCh38 package

## A quick look at the GRCh38 seqlevels:
length(seqlevels(genome))
head(seqlevels(genome), n=30)

## Fetch the extended chromosomes info for the hg38 genome:
hg38_chrominfo <- fetchExtendedChromInfoFromUCSC("hg38")
dim(hg38_chrominfo)
head(hg38_chrominfo, n=30)

## 2 sanity checks:
##   1. Check the NCBI seqlevels:
stopifnot(setequal(hg38_chrominfo$NCBI_seqlevel, seqlevels(genome)))
##   2. Check that the sequence lengths in 'hg38_chrominfo' (which are
##      coming from the same 'chromInfo' table as the UCSC seqlevels)
##      are the same as in 'genome':
stopifnot(
  identical(hg38_chrominfo$UCSC_seqlength,
            unname(seqlengths(genome)[hg38_chrominfo$NCBI_seqlevel]))
)

## Extract the hg38 seqlevels and put the GRCh38 seqlevels on it as
## the names:
hg38_seqlevels <- setNames(hg38_chrominfo$UCSC_seqlevel,
                           hg38_chrominfo$NCBI_seqlevel)

## Set the hg38 seqlevels on 'genome':
seqlevels(genome) <- hg38_seqlevels[seqlevels(genome)]
head(seqlevels(genome), n=30)
```

GenomeDescription-class *GenomeDescription objects*

Description

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

Details

In general the user will not need to manipulate directly a GenomeDescription instance but will manipulate instead a higher-level object that belongs to a class that extends the GenomeDescription class. For example, the top-level object defined in any BSgenome data package is a BSgenome object and the BSgenome class contains the GenomeDescription class. Thus a BSgenome object is also a GenomeDescription object and can therefore be treated as such. In other words all the methods described below will work on it.

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...

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

`providerVersion(x)`: Return the provider-side version of this genome. For example UCSC uses versions `"hg18"`, `"hg17"`, etc... for the different Builds of the Human genome.

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

`releaseName(x)`: Return the release name of this genome, which is generally made of the name of the organization who assembled it plus its Build version. For example, UCSC uses `"hg18"` for the version of the Human genome corresponding to the Build 36.1 from NCBI hence the release name for this genome is `"NCBI Build 36.1"`.

`bsgenomeName(x)`: Uses the meta information stored in `x` to make 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). Of course 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)
class(Celegans)
is(Celegans, "GenomeDescription")
provider(Celegans)
seqinfo(Celegans)
gendesc <- as(Celegans, "GenomeDescription")
class(gendesc)
gendesc
provider(gendesc)
seqinfo(gendesc)
bsgenomeName(gendesc)
```

---

loadTaxonomyDb *Return a data.frame that lists the known taxonomy IDs and their corresponding organisms.*

Description

NCBI maintains a collection of unique taxonomy IDs and pairs these with associated genus and species designations. This function returns the set of pre-processed values that we use to check that something is a valid Taxonomy ID (or organism).

Usage

```
loadTaxonomyDb()
```

Value

A data frame with 1 row per genus/species designation and three columns. The 1st column is the taxonomy ID. The second columns is the genus and the third is the species name.

Author(s)

Marc Carlson

Examples

```
## get the data
taxdb <- loadTaxonomyDb()
tail(taxdb)
## which can then be searched etc.
taxdb[grepl('yoelii', taxdb$species), ]
```

mapGenomeBuilds *Mapping between UCSC and Ensembl Genome Builds*

Description

genomeBuilds lists the available genomes for a given species while mapGenomeBuilds maps between UCSC and Ensemble genome builds.

Usage

```
genomeBuilds(organism, style = c("UCSC", "Ensembl"))

mapGenomeBuilds(genome, style = c("UCSC", "Ensembl"))

listOrganisms()
```

Arguments

organism	A character vector of common names or organism
genome	A character vector of genomes equivalent to UCSC version or Ensembl Assemblies
style	A single value equivalent to "UCSC" or "Ensembl" specifying the output genome

Details

genomeBuilds lists the currently available genomes for a given list of organisms. The genomes can be shown as "UCSC" or "Ensembl" IDs determined by style. organism must be specified as a character vector and match common names (i.e "Dog", "Mouse") or organism name (i.e "Homo sapiens", "Mus musculus") . A list of available organisms can be shown using listOrganisms().

mapGenomeBuilds provides a mapping between "UCSC" builds and "Ensembl" builds. genome must be specified as a character vector and match either a "UCSC" ID or an "Ensembl" Id. genomeBuilds can be used to get a list of available build Ids for a given organism. NA's may be present in the output. This would occur when the current genome build removed a previously defined genome for an organism.

In both functions, if style is not specified, "UCSC" is used as default.

Value

A data.frame of builds for a given organism or genome in the specified style. If style == "UCSC", ucscID, ucscDate and ensemblID are given. If style == "Ensembl", ensemblID, ensemblVersion, ensemblDate, and ucscID are given. The opposing ID is given so that it is possible to distinguish between many-to-one mappings.

Author(s)

Valerie Obenchain <Valerie.Obenchain@roswellpark.org> and Lori Shepherd <Lori.Shepherd@roswellpark.org>

References

UCSC genome builds https://genome.ucsc.edu/FAQ/FAQreleases.html Ensembl genome builds http://useast.ensembl.org/info/website/archives/assembly.html

Examples

```
listOrganisms()

genomeBuilds("mouse")
genomeBuilds(c("Mouse", "dog", "human"), style="Ensembl")

mapGenomeBuilds(c("canFam3", "GRCm38", "mm9"))
mapGenomeBuilds(c("canFam3", "GRCm38", "mm9"), style="Ensembl")
```

`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

```
library(BSgenome.Scerevisiae.UCSC.sacCer2)
rankSeqlevels(seqnames(Scerevisiae))
rankSeqlevels(seqnames(Scerevisiae)[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
sortSeqlevels(x, X.is.sexchrom=NA)
seqlevelsInUse(x)
seqlevels0(x)

seqlengths(x)
seqlengths(x) <- value

isCircular(x)
isCircular(x) <- value

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

Arguments

`x`	The object from/on which to get/set the sequence information.
`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 row 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 rows in `value` should be mapped to the rows 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 rows 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)`.

`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.

`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".

Details

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

- All these functions (except `seqinfo`, `seqlevelsInUse`, and `seqlevels0`) work on a Seqinfo object.
- For classes that implement it, the `seqinfo` getter should return a Seqinfo object.
- 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; etc...

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`.

Note

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

Author(s)

H. Pagès

See Also

- The seqlevelsStyle generic getter and setter.
- 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.
- seqlevels-wrappers for convenience wrappers to the `seqlevels` getter and setter.
- `rankSeqlevels`, on which `sortSeqlevels` is based.

Examples

```
## Finding overlaps, comparing, and matching operations between objects
## containing genomic ranges require the objects to have the same
## seqlevels, or they fail. So before one can perform these operations,
## it is often necessary to modify the seqlevels in some of the objects
## involved in the operation so that all the objects have the same
## seqlevels. This is typically done with the seqlevels() setter. It can
## rename, drop, add and reorder seqlevels of an object. Examples below
## show how to mofify the seqlevels of GRanges, GRangesList, and TxDb
## objects but the approach is the same for any object that has seqlevels.

## ---------------------------------------------------------------------
## A. MODIFY THE SEQLEVELS OF A GRanges OBJECT
## ---------------------------------------------------------------------
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)

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

## Drop all unused seqlevels:
seqlevels(gr) <- seqlevelsInUse(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:
seqlevels(txdb) <- seqlevels0(txdb)
seqlevels(txdb)

## ---------------------------------------------------------------------
## D. SORT SEQLEVELS IN "NATURAL" ORDER
## ---------------------------------------------------------------------

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)

gr <- GRanges()
seqlevels(gr) <- seqlevels
sortSeqlevels(gr)

## ---------------------------------------------------------------------
## E. 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)

## ---------------------------------------------------------------------
## F. 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 a table-like object that contains basic information about a set of genomic sequences. The table has 1 row per sequence and 1 column per sequence attribute. Currently the only attributes are the length, circularity flag, and genome provenance (e.g. hg19) of the sequence, but more attributes might be added in the future as the need arises.

Details

Typically Seqinfo objects are not used directly but are part of higher level objects. Those higher level objects will generally provide a `seqinfo` accessor for getting/setting their Seqinfo component.

Constructor

`Seqinfo(seqnames,seqlengths=NA,isCircular=NA,genome=NA)`: Creates a Seqinfo object.

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 rows (aka the sequence levels) of 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

There are no `c` or `rbind` method for Seqinfo objects. Both would be expected to just append the rows in `y` to the rows in `x` resulting in an object of length `length(x) + length(y)`. But that would tend to break the constraint that the seqnames of a Seqinfo object must be unique keys.

So instead, a `merge` method is provided.

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

`merge(x,y)`: Merge `x` and `y` into a single Seqinfo object where the keys (aka the seqnames) are `union(seqnames(x),seqnames(y))`. If a row in `y` has the same key as a row in `x`, and if the 2 rows contain compatible information (NA values are compatible with anything), then they are merged into a single row 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 based on different reference genomes.

`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 the objects not being subsets of each other during overlap operations.

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

- `seqinfo`
- The `fetchExtendedChromInfoFromUCSC` utility function that is used behind the scene to make a Seqinfo object for a supported genome (see examples below).

Examples

```
## ---------------------------------------------------------------------
## A. MAKING A Seqinfo OBJECT FOR A SUPPORTED GENOME
## ---------------------------------------------------------------------

if (interactive()) {
  ## This uses fetchExtendedChromInfoFromUCSC() behind the scene and
  ## thus requires internet access. See ?fetchExtendedChromInfoFromUCSC
  ## for the list of UCSC genomes that are currently supported.
  Seqinfo(genome="hg38")
  Seqinfo(genome="bosTau8")
  Seqinfo(genome="canFam3")
  Seqinfo(genome="musFur1")
  Seqinfo(genome="mm10")
  Seqinfo(genome="rn6")
  Seqinfo(genome="galGal4")
  Seqinfo(genome="dm6")
  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="toy")
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. MERGING 2 Seqinfo OBJECTS
## ---------------------------------------------------------------------

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

## This issues a warning:
merge(x, y)  # rows for chr3 and chrM are merged

## 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, i.e., 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 (i.e. the same
## rows, but typically not in the same order):
merge(y, x)

## This contradicts what 'x' says about circularity of chr3 and chrM:
isCircular(y)[c("chr3", "chrM")] <- c(TRUE, FALSE)
y
if (interactive()) {
  merge(x, y)  # raises an error
}

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

## ---------------------------------------------------------------------
## D. checkCompatibleSeqinfo()
## ---------------------------------------------------------------------

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

`seqlevels-wrappers` *Convenience wrappers to the seqlevels() getter and setter*

Description

Keep, drop or rename seqlevels in objects with a Seqinfo class.

Usage

```
keepSeqlevels(x, value, pruning.mode=c("error", "coarse", "fine", "tidy"))
dropSeqlevels(x, value, pruning.mode=c("error", "coarse", "fine", "tidy"))
renameSeqlevels(x, value)
restoreSeqlevels(x)
standardChromosomes(x, species=NULL)
keepStandardChromosomes(x, species=NULL,
                        pruning.mode=c("error", "coarse", "fine", "tidy"))
```

Arguments

x	Any object having a Seqinfo class in which the seqlevels will be kept, dropped or renamed.

value	A named or unnamed character vector. Names are ignored by keepSeqlevels and dropSeqlevels. Only the values in the character vector dictate which seqlevels to keep or drop. In the case of renameSeqlevels, the names are used to map new sequence levels to the old (names correspond to the old levels). When value is unnamed, the replacement vector must the same length and in the same order as the original seqlevels(x).
pruning.mode	See ?seqinfo for a description of the pruning modes.
species	The genus and species of the organism. Supported species can be seen with names(genomeStyles()).

Details

Matching and overlap operations on range objects often require that the seqlevels match before a comparison can be made (e.g., findOverlaps). keepSeqlevels, dropSeqlevels and renameSeqlevels are high-level convenience functions that wrap the low-level seqlevels setter.

- keepSeqlevels, dropSeqlevels: Subsetting operations that modify the size of x. keepSeqlevels keeps only the seqlevels in value and removes all others. dropSeqlevels drops the levels in value and retains all others. If value does not match any seqlevels in x an empty object is returned.

  When x is a GRangesList it is possible to have 'mixed' list elements that have ranges from different chromosomes. keepSeqlevels will not keep 'mixed' list elements
- renameSeqlevels: Rename the seqlevels in x to those in value. If value is a named character vector, the names are used to map the new seqlevels to the old. When value is unnamed, the replacement vector must be the same length and in the same order as the original seqlevels(x).
- restoreSeqlevels: Perform seqlevels(txdb) <-seqlevels0(txdb), that is, restore the seqlevels in x back to the original values. Applicable only when x is a TxDb object.
- standardChromosomes: Lists the 'standard' chromosomes defined as sequences in the assembly that are not scaffolds; also referred to as an 'assembly molecule' in NCBI. standardChromosomes attempts to detect the seqlevel style and if more than one style is matched, e.g., 'UCSC' and 'Ensembl', the first is chosen.

  x must have a Seqinfo object. species can be specified as a character string; supported species are listed with names(genomeStyles()).

  When x contains seqlevels from multiple organisms all those considered standard will be kept. For example, if seqlevels are "chr1" and "chr3R" from human and fly both will be kept. If species="Homo sapiens" is specified then only "chr1" is kept.
- keepStandardChromosomes: Subsetting operation that returns only the 'standard' chromosomes.

  x must have a Seqinfo object. species can be specified as a character string; supported species are listed with names(genomeStyles()).

  When x contains seqlevels from multiple organisms all those considered standard will be kept. For example, if seqlevels are "chr1" and "chr3R" from human and fly both will be kept. If species="Homo sapiens" is specified then only "chr1" is kept.

Value

The x object with seqlevels removed or renamed. If x has no seqlevels (empty object) or no replacement values match the current seqlevels in x the unchanged x is returned.

Author(s)

Valerie Obenchain, Sonali Arora

See Also

- seqinfo ## Accessing sequence information
- Seqinfo ## The Seqinfo class

Examples

```
## ---------------------------------------------------------------------
## keepSeqlevels / dropSeqlevels
## ---------------------------------------------------------------------

##
## GRanges / GAlignments:
##

library(GenomicRanges)
gr <- GRanges(c("chr1", "chr1", "chr2", "chr3"), IRanges(1:4, width=3))
seqlevels(gr)
## Keep only 'chr1'
gr1 <- keepSeqlevels(gr, "chr1", pruning.mode="coarse")
## Drop 'chr1'. Both 'chr2' and 'chr3' are kept.
gr2 <- dropSeqlevels(gr, "chr1", pruning.mode="coarse")

library(Rsamtools)  # for the ex1.bam file
library(GenomicAlignments)  # for readGAlignments()

fl <- system.file("extdata", "ex1.bam", package="Rsamtools")
gal <- readGAlignments(fl)
## If 'value' is named, the names are ignored.
seq2 <- keepSeqlevels(gal, c(foo="seq2"), pruning.mode="coarse")
seqlevels(seq2)

##
## List-like objects:
##

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())
## See ?seqinfo for a description of the pruning modes.
keepSeqlevels(grl0, "chr2", pruning.mode="coarse")
keepSeqlevels(grl0, "chr2", pruning.mode="fine")
keepSeqlevels(grl0, "chr2", pruning.mode="tidy")

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)
ex_by_tx2 <- keepSeqlevels(ex_by_tx, "chr2L", pruning.mode="coarse")
seqlevels(ex_by_tx2)
```

```
## 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)
tx_by_gene2 <- keepSeqlevels(tx_by_gene, "chr2L", pruning.mode="tidy")
seqlevels(tx_by_gene2)

## ---------------------------------------------------------------------
## renameSeqlevels
## ---------------------------------------------------------------------

##
## GAlignments:
##

seqlevels(gal)
## Rename 'seq2' to 'chr2' with a named vector.
gal2a <- renameSeqlevels(gal, c(seq2="chr2"))
## Rename 'seq2' to 'chr2' with an unnamed vector that includes all
## seqlevels as they appear in the object.
gal2b <- renameSeqlevels(gal, c("seq1", "chr2"))
## Names that do not match existing seqlevels are ignored.
## This attempt at renaming does nothing.
gal3 <- renameSeqlevels(gal, c(foo="chr2"))
stopifnot(identical(gal, gal3))

##
## TxDb:
##

seqlevels(txdb)
## When the seqlevels of a TxDb are renamed, all future
## extractions reflect the modified seqlevels.
renameSeqlevels(txdb, sub("chr", "CH", seqlevels(txdb)))
renameSeqlevels(txdb, c(CHM="M"))
seqlevels(txdb)

transcripts <- transcripts(txdb)
identical(seqlevels(txdb), seqlevels(transcripts))

## ---------------------------------------------------------------------
## restoreSeqlevels
## ---------------------------------------------------------------------

## Restore seqlevels in a TxDb to original values.
## Not run:
txdb <- restoreSeqlevels(txdb)
seqlevels(txdb)

## End(Not run)

## ---------------------------------------------------------------------
## keepStandardChromosomes
## ---------------------------------------------------------------------

##
## GRanges:
```

```
##
gr <- GRanges(c(paste0("chr",c(1:3)), "chr1_gl000191_random",
              "chr1_gl000192_random"), IRanges(1:5, width=3))
gr
keepStandardChromosomes(gr, pruning.mode="coarse")

##
## List-like objects:
##

grl <- GRangesList(GRanges("chr1", IRanges(1:2, 5)),
                   GRanges(c("chr1_GL383519v1_alt", "chr1"), IRanges(5:6, 5)))
## Use pruning.mode="coarse" to drop list elements with mixed seqlevels:
keepStandardChromosomes(grl, pruning.mode="coarse")
## Use pruning.mode="tidy" to keep all list elements with ranges on
## standard chromosomes:
keepStandardChromosomes(grl, pruning.mode="tidy")

##
## The set of standard chromosomes should not be affected by the
## particular seqlevel style currently in use:
##

## NCBI
worm <- GRanges(c("I", "II", "foo", "X", "MT"), IRanges(1:5, width=5))
keepStandardChromosomes(worm, pruning.mode="coarse")

## UCSC
seqlevelsStyle(worm) <- "UCSC"
keepStandardChromosomes(worm, pruning.mode="coarse")

## Ensembl
seqlevelsStyle(worm) <- "Ensembl"
keepStandardChromosomes(worm, pruning.mode="coarse")
```

`seqlevelsStyle` *Conveniently rename the seqlevels of an object according to a given style*

Description

The `seqlevelsStyle` getter and setter can be used to get the current seqlevels style of an object and to rename its seqlevels according to a given style.

Usage

```
seqlevelsStyle(x)
seqlevelsStyle(x) <- value

## Related low-level utilities:
genomeStyles(species)
extractSeqlevels(species, style)
extractSeqlevelsByGroup(species, style, group)
mapSeqlevels(seqnames, style, best.only=TRUE, drop=TRUE)
seqlevelsInGroup(seqnames, group, species, style)
```

Arguments

x	The object from/on which to get/set the seqlevels style. x must have a seqlevels method or be a character vector.
value	A single character string that sets the seqlevels style for x.
species	The genus and species of the organism in question separated by a single space. Don't forget to capitalize the genus.
style	a character vector with a single element to specify the style.
group	Group can be 'auto' for autosomes, 'sex' for sex chromosomes/allosomes, 'circular' for circular chromosomes. The default is 'all' which returns all the chromosomes.
best.only	if TRUE (the default), then only the "best" sequence renaming maps (i.e. the rows with less NAs) are returned.
drop	if TRUE (the default), then a vector is returned instead of a matrix when the matrix has only 1 row.
seqnames	a character vector containing the labels attached to the chromosomes in a given genome for a given style. For example : For *Homo sapiens*, NCBI style - they are "1","2","3",...,"X","Y","MT"

Details

seqlevelsStyle(x), seqlevelsStyle(x) <-value: Get the current seqlevels style of an object, or rename its seqlevels according to the supplied style.

genomeStyles: Different organizations have different naming conventions for how they name the biologically defined sequence elements (usually chromosomes) for each organism they support. The Seqnames package contains a database that defines these different conventions.

genomeStyles() returns the list of all supported seqname mappings, one per supported organism. Each mapping is represented as a data frame with 1 column per seqname style and 1 row per chromosome name (not all chromosomes of a given organism necessarily belong to the mapping).

genomeStyles(species) returns a data.frame only for the given organism with all its supported seqname mappings.

extractSeqlevels: Returns a character vector of the seqnames for a single style and species.

extractSeqlevelsByGroup: Returns a character vector of the seqnames for a single style and species by group. Group can be 'auto' for autosomes, 'sex' for sex chromosomes/ allosomes, 'circular' for circular chromosomes. The default is 'all' which returns all the chromosomes.

mapSeqlevels: Returns a matrix with 1 column per supplied sequence name and 1 row per sequence renaming map compatible with the specified style. If best.only is TRUE (the default), only the "best" renaming maps (i.e. the rows with less NAs) are returned.

seqlevelsInGroup: It takes a character vector along with a group and optional style and species.If group is not specified , it returns "all" or standard/top level seqnames. Returns a character vector of seqnames after subsetting for the group specified by the user. See examples for more details.

Value

For seqlevelsStyle: A single string containing the style of the seqlevels in x, or a character vector containing the styles of the seqlevels in x if the current style cannot be determined unambiguously. Note that this information is not stored in x but inferred by looking up a seqlevels style database stored inside the **GenomeInfoDb** package.

For `extractSeqlevels` , `extractSeqlevelsByGroup` and `seqlevelsInGroup`: A character vector of seqlevels for given supported species and group.

For `mapSeqlevels`: A matrix with 1 column per supplied sequence name and 1 row per sequence renaming map compatible with the specified style.

For `genomeStyle`: If species is specified returns a data.frame containg the seqlevels style and its mapping for a given organism. If species is not specified, a list is returned with one list per species containing the seqlevels style with the corresponding mappings.

Author(s)

Sonali Arora, Martin Morgan, Marc Carlson, H. Pagès

Examples

```
## ---------------------------------------------------------------------
## seqlevelsStyle() getter and setter
## ---------------------------------------------------------------------

## character vectors:
x <- paste0("chr", 1:5)
seqlevelsStyle(x)
seqlevelsStyle(x) <- "NCBI"
x

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

seqlevelsStyle(gr)
seqlevelsStyle(gr) <- "NCBI"
gr

seqlevelsStyle(gr)
seqlevelsStyle(gr) <- "dbSNP"
gr

seqlevelsStyle(gr)
seqlevelsStyle(gr) <- "UCSC"
gr

## ---------------------------------------------------------------------
## Related low-level utilities
## ---------------------------------------------------------------------

## Genome styles:
names(genomeStyles())
genomeStyles("Homo_sapiens")
"UCSC" %in% names(genomeStyles("Homo_sapiens"))

## Extract seqlevels based on species, style and group:
## The 'group' argument can be 'sex', 'auto', 'circular' or 'all'.

## All:
extractSeqlevels(species="Drosophila_melanogaster", style="Ensembl")

## Sex chromosomes:
```

```
extractSeqlevelsByGroup(species="Homo_sapiens", style="UCSC", group="sex")

## Autosomes:
extractSeqlevelsByGroup(species="Homo_sapiens", style="UCSC", group="auto")


## Identify which seqnames belong to a particular 'group':
newchr <- paste0("chr",c(1:22,"X","Y","M","1_gl000192_random","4_ctg9"))
seqlevelsInGroup(newchr, group="sex")

newchr <- as.character(c(1:22,"X","Y","MT"))
seqlevelsInGroup(newchr, group="all","Homo_sapiens","NCBI")

## Identify which seqnames belong to a species and style:
seqnames <- c("chr1","chr9", "chr2", "chr3", "chr10")
all(seqnames %in% extractSeqlevels("Homo_sapiens", "UCSC"))

## Find mapped seqlevelsStyles for exsiting seqnames:
mapSeqlevels(c("chrII", "chrIII", "chrM"), "NCBI")
mapSeqlevels(c("chrII", "chrIII", "chrM"), "Ensembl")
```

Index