Package 'GenomeInfoDb'

February 20, 2025

Title Utilities for manipulating chromosome names, including modifying them to follow a particular naming style

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.

biocViews Genetics, DataRepresentation, Annotation, GenomeAnnotation

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

Video http://youtu.be/wdEjCYSXa7w

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

Version 1.43.4

License Artistic-2.0

Encoding UTF-8

Depends R (>= 4.0.0), methods, BiocGenerics (>= 0.53.2), S4Vectors (>= 0.45.2), IRanges (>= 2.41.1)

Imports stats, stats4, utils, UCSC.utils, GenomeInfoDbData

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

VignetteBuilder knitr

Collate utils.R fetch_table_dump_from_Ensembl_FTP.R list_ftp_dir.R rankSeqlevels.R NCBI-utils.R UCSC-utils.R Ensembl-utils.R getChromInfoFromNCBI.R getChromInfoFromUCSC.R getChromInfoFromEnsembl.R loadTaxonomyDb.R mapGenomeBuilds.R seqinfo.R Seqinfo-class.R seqlevelsStyle.R seqlevels-wrappers.R GenomeDescription-class.R zzz.R

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

git_branch devel

git_last_commit a7d190e

git_last_commit_date 2025-01-23

Repository Bioconductor 3.21

Date/Publication 2025-02-20

Author Sonali Arora [aut],
Martin Morgan [aut],
Marc Carlson [aut],
Hervé Pagès [aut, cre],
Prisca Chidimma Maduka [ctb],
Atuhurira Kirabo Kakopo [ctb],
Haleema Khan [ctb] (vignette translation from Sweave to Rmarkdown / HTML),
Emmanuel Chigozie Elendu [ctb]

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")
```

GenomeInfoDb internals

GenomeInfoDb internals

Description

Symbols defined in the **GenomeInfoDb** package that are not intended to be used directly.

getChromInfoFromEnsembl

Get chromosome information for an Ensembl species

Description

getChromInfoFromEnsembl returns chromosome information like sequence names, lengths and circularity flags for a given Ensembl species e.g. Human, Cow, Saccharomyces cerevisiae, etc...

Usage

```
getChromInfoFromEnsembl(species,
                        release=NA, division=NA, use.grch37=FALSE,
                        assembled.molecules.only=FALSE,
                        include.non_ref.sequences=FALSE,
                        include.contigs=FALSE,
                        include.clones=FALSE,
                        map.NCBI=FALSE,
                        recache=FALSE,
                        as.Seqinfo=FALSE)
```

Arguments

species	A single string specifying the name of an Ensembl species e.g. "human", "hsapiens", or "Homo sapiens". Case is ignored. Alternatively the name of an assembly (e.g. "GRCh38") or a taxonomy id (e.g. 9606) can be supplied.
release	The Ensembl release to query e.g. 89. If set to NA (the default), the current release is used.
division	NA (the default) or one of the EnsemblGenomes marts i.e. "bacteria", "fungi", "metazoa", "plants", or "protists".
use.grch37	NOT TESTED YET! TRUE or FALSE (the default).
assembled.molecules.only	NOT IMPLEMENTED YET!
include.non_ref.sequences	TODO: DOCUMENT THIS!
include.contigs	Whether or not sequences for which coord_system is set to "contig" should be included. They are not included by default. Note that the dataset for Human contains more than one hundred thousands *contigs*!
include.clones	Whether or not sequences for which coord_system is set to "clone" should be included. They are not included by default. Note that the dataset for Human contains more than one hundred thousands *clones*!

`map.NCBI` — `TRUE` or `FALSE` (the default).

If `TRUE` then NCBI chromosome information is bound to the result. This information is retrieved from NCBI by calling `getChromInfoFromNCBI` on the NCBI assembly that the Ensembl species is based on. Then the data frame returned by `getChromInfoFromNCBI` ("NCBI chrom info") is *mapped* and bound to the data frame returned by `getChromInfoFromEnsembl` ("Ensembl chrom info"). This "map and bind" operation is similar to a `JOIN` in SQL.

Note that not all rows in the "Ensembl chrom info" data frame are necessarily mapped to a row in the "NCBI chrom info" data frame. For the unmapped rows the NCBI columns in the final data frame are filled with NAs (`LEFT JOIN` in SQL).

The primary use case for using `map.NCBI=TRUE` is to map Ensembl sequence names to NCBI sequence names.

`recache` — `getChromInfoFromEnsembl` uses a cache mechanism so the chromosome information of a given dataset only gets downloaded once during the current R session (note that the caching is done in memory so cached information does NOT persist across sessions). Setting `recache` to `TRUE` forces a new download (and recaching) of the chromosome information for the specified dataset.

`as.Seqinfo` — `TRUE` or `FALSE` (the default). If `TRUE` then a Seqinfo object is returned instead of a data frame. Note that only the `name`, `length`, and `circular` columns of the data frame are used to make the Seqinfo object. All the other columns are ignored (and lost).

Details

COMING SOON...

Value

For `getChromInfoFromEnsembl`: By default, a 7-column data frame with columns:

1. `name`: character.
2. `length`: integer.
3. `coord_system`: factor.
4. `synonyms`: list.
5. `toplevel`: logical.
6. `non_ref`: logical.
7. `circular`: logical.

and with attribute `species_info` which contains details about the species that was used to obtaine the data.

If `map.NCBI` is `TRUE`, then 7 "NCBI columns" are added to the result:

- `NCBI.SequenceName`: character.
- `NCBI.SequenceRole`: factor.
- `NCBI.AssignedMolecule`: factor.

- NCBI.GenBankAccn: character.
- NCBI.Relationship: factor.
- NCBI.RefSeqAccn: character.
- NCBI.AssemblyUnit: factor.

Note that the names of the "NCBI columns" are those returned by getChromInfoFromNCBI but with the NCBI. prefix added to them.

Author(s)

H. Pagès

See Also

- getChromInfoFromNCBI and getChromInfoFromUCSC for getting chromosome information for an NCBI assembly or UCSC genome.
- Seqinfo objects.

Examples

```
## ---------------------------------------------------------------------
## A. BASIC EXAMPLES
## ---------------------------------------------------------------------

## Internet access required!

## === Worm ===
## https://uswest.ensembl.org/Caenorhabditis_elegans

celegans <- getChromInfoFromEnsembl("celegans")
attr(celegans, "species_info")

getChromInfoFromEnsembl("celegans", as.Seqinfo=TRUE)

celegans <- getChromInfoFromEnsembl("celegans", map.NCBI=TRUE)

## === Yeast ===
## https://uswest.ensembl.org/Saccharomyces_cerevisiae

scerevisiae <- getChromInfoFromEnsembl("scerevisiae")
attr(scerevisiae, "species_info")

getChromInfoFromEnsembl("scerevisiae", as.Seqinfo=TRUE)

scerevisiae <- getChromInfoFromEnsembl("scerevisiae", map.NCBI=TRUE)

## Arabidopsis thaliana:
athaliana <- getChromInfoFromEnsembl("athaliana", division="plants",
                                     map.NCBI=TRUE)
attr(athaliana, "species_info")
```

```
## ------------------------------------------------------------------------
## Temporary stuff that needs to go away...
## ------------------------------------------------------------------------

## TODO: Check all species for which an NCBI assembly is registered!
## Checked so far (with current Ensembl release i.e. 99):
## - celegans       OK
## - scerevisiae    OK
## - athaliana      OK
## - btaurus        OK
## - sscrofa        OK

## Not run:
## WORK IN PROGRESS!!!
library(GenomeInfoDb)

.do_join <- GenomeInfoDb:::.do_join
.map_Ensembl_seqlevels_to_NCBI_seqlevels <-
    GenomeInfoDb:::.map_Ensembl_seqlevels_to_NCBI_seqlevels

.map_Ensembl_seqlevels_to_NCBI_seqlevels(
    paste0("ENS_", 1:26),
    CharacterList(c(list(c(aa="INSDC1", bb="GNBK7"), c("INSDC2", "RefSeq3")),
                    rep(list(NULL), 23), list("NCBI_7"))),
    paste0("NCBI_", 1:10),
    paste0("GNBK", c(1:8, NA, 9)),
    c(paste0("REFSEQ", c(1:7, 1, 1)), NA),
    verbose=TRUE
)

map_to_NCBI <- function(Ensembl_chrom_info, NCBI_chrom_info,
                        special_mappings=NULL)
{
    .map_Ensembl_seqlevels_to_NCBI_seqlevels(
         Ensembl_chrom_info[ , "name"],
         Ensembl_chrom_info[ , "synonyms"],
         NCBI_chrom_info[ , "SequenceName"],
         NCBI_chrom_info[ , "GenBankAccn"],
         NCBI_chrom_info[ , "RefSeqAccn"],
         special_mappings=special_mappings,
         verbose=TRUE)
}

## ------------------------------------------------------------------------
## Human
## https://uswest.ensembl.org/Homo_sapiens/
## Based on GRCh38.p13 (GCA_000001405.28)

## Return 944 rows
human_chrom_info <- getChromInfoFromEnsembl("hsapiens")
#                 1 id: 131550  <- ref chromosome
# CHR_HSCHR1_1_CTG3 id: 131561  <- non-ref chromosome
#     HSCHR1_1_CTG3 id: 131562  <- scaffold (no scaffold is non_ref)
```

```
## Map to NCBI
## Summary:
## - 639/640 NCBI sequences are reverse-mapped.
## - Restricted mapping is one-to-one.
GRCh38.p13 <- getChromInfoFromNCBI("GRCh38.p13")
L2R <- map_to_NCBI(human_chrom_info, GRCh38.p13)
## The only sequence in GRCh38.p13 that cannot be mapped to Ensembl is
## HG2139_PATCH (was introduced in GRCh38.p2)! Why? What's special about
## this patch?
GRCh38.p13$mapped <- tabulate(L2R, nbins=nrow(GRCh38.p13)) != 0L
table(GRCh38.p13$SequenceRole, GRCh38.p13$mapped)
#                        FALSE TRUE
#   assembled-molecule       0   25
#   alt-scaffold             0  261
#   unlocalized-scaffold     0   42
#   unplaced-scaffold        0  127
#   pseudo-scaffold          0    0
#   fix-patch                1  112
#   novel-patch              0   72
human_chrom_info <- .do_join(human_chrom_info, GRCh38.p13, L2R)
table(human_chrom_info$SequenceRole, human_chrom_info$toplevel)
#                       FALSE TRUE
#  assembled-molecule       0   25
#  alt-scaffold           261    0
#  unlocalized-scaffold     0   42
#  unplaced-scaffold        0  127
#  pseudo-scaffold          0    0
#  fix-patch              112    0
#  novel-patch             72    0

#hsa_seqlevels <- readRDS("hsapiens_gene_ensembl_txdb_seqlevels.rds")

## ---------------------------------------------------------------------
## Mouse
## https://uswest.ensembl.org/Mus_musculus/
## Based on GRCm38.p6 (GCA_000001635.8)

## Return 258 rows
mouse_chrom_info <- getChromInfoFromEnsembl("mmusculus")

## Map to NCBI
## Summary:
## - 139/239 NCBI sequences are reverse-mapped.
## - Restricted mapping is NOT one-to-one: 2 Ensembl sequences (NC_005089.1
##   and MT) are both mapped to NCBI MT.
GRCm38.p6 <- getChromInfoFromNCBI("GRCm38.p6")
L2R <- map_to_NCBI(mouse_chrom_info, GRCm38.p6)
## 100 sequences in GRCm38.p6 are not mapped:
GRCm38.p6$mapped <- tabulate(L2R, nbins=nrow(GRCm38.p6)) != 0L
table(GRCm38.p6$SequenceRole, GRCm38.p6$mapped)
#                        FALSE TRUE
#   assembled-molecule       0   22
```

```
#   alt-scaffold            99    0
#   unlocalized-scaffold     0   22
#   unplaced-scaffold        0   22
#   pseudo-scaffold          0    0
#   fix-patch                1   64
#   novel-patch              0    9
## OK so Ensembl doesn't include the alt-scaffolds for Mouse. BUT WHAT
## HAPPENED TO THIS ONE fix-patch SEQUENCE (MG4237_PATCH) THAT IS NOT
## MAPPED? Found it in seq_region_synonym table! It's seq_region_id=100405.
## Hey but that seq_region_id is **NOT** in the seq_region table!!! THIS
## VIOLATES FOREIGN KEY CONSTRAINT!!!!
mouse_chrom_info <- .do_join(mouse_chrom_info, GRCm38.p6, L2R)
## Ensembl does NOT comsider NC_005089.1 (duplicate entry for MT) toplevel:
mouse_chrom_info[mouse_chrom_info$SequenceName
#            name length coord_system                      synonyms toplevel
# 184 NC_005089.1  16299     scaffold                                 FALSE
# 201          MT  16299   chromosome NC_005089.1, chrM, AY172335.1     TRUE
#     SequenceName GenBankAccn  RefSeqAccn
# 184           MT  AY172335.1 NC_005089.1
# 201           MT  AY172335.1 NC_005089.1

## ------------------------------------------------------------------------
## Rat
## https://uswest.ensembl.org/Rattus_norvegicus/
## Based on Rnor_6.0 (GCA_000001895.4)

# Return 1418 rows
rat_chrom_info <- getChromInfoFromEnsembl("rnorvegicus")

## Map to NCBI
## Summary:
## - 955/955 NCBI sequences are reverse-mapped.
## - Reverse mapping is one-to-many: 2 Ensembl sequences (NC_001665.2 and MT)
##   are mapped to NCBI MT.
Rnor_6.0 <- getChromInfoFromNCBI("Rnor_6.0")
L2R <- map_to_NCBI(rat_chrom_info, Rnor_6.0)
rat_chrom_info <- .do_join(rat_chrom_info, Rnor_6.0, L2R)

## Ensembl does NOT comsider NC_001665.2 (duplicate entry for MT) toplevel:
rat_chrom_info[rat_chrom_info$SequenceName
#             name length coord_system                      synonyms toplevel
# 1417 NC_001665.2  16313     scaffold                                 FALSE
# 1418          MT  16313   chromosome NC_001665.2, AY172581.1, chrM     TRUE
#      SequenceName GenBankAccn  RefSeqAccn
# 1417           MT  AY172581.1 NC_001665.2
# 1418           MT  AY172581.1 NC_001665.2

table(rat_chrom_info$SequenceRole, rat_chrom_info$toplevel)
#                       FALSE TRUE
#   assembled-molecule      1   23
#   alt-scaffold            0    0
#   unlocalized-scaffold    0  354
#   unplaced-scaffold       0  578
```

```
#   pseudo-scaffold          0    0
#   fix-patch                0    0
#   novel-patch              0    0

## End(Not run)
```

---

getChromInfoFromNCBI *Get chromosome information for an NCBI assembly*

Description

getChromInfoFromNCBI returns chromosome information like sequence names, lengths and circularity flags for a given NCBI assembly e.g. for GRCh38, ARS-UCD1.2, R64, etc...

Note that getChromInfoFromNCBI behaves slightly differently depending on whether the assembly is *registered* in the **GenomeInfoDb** package or not. See below for the details.

Use registered_NCBI_assemblies to list all the NCBI assemblies currently registered in the **GenomeInfoDb** package.

Usage

```
getChromInfoFromNCBI(assembly,
                     assembled.molecules.only=FALSE,
                     assembly.units=NULL,
                     recache=FALSE,
                     as.Seqinfo=FALSE)

registered_NCBI_assemblies(organism=NA)
```

Arguments

assembly	A single string specifying the name of an NCBI assembly (e.g. "GRCh38"). Alternatively, an assembly accession (GenBank or RefSeq) can be supplied (e.g. "GCF_000001405.12").
assembled.molecules.only	If FALSE (the default) then chromosome information is returned for all the sequences in the assembly (unless assembly.units is specified, see below), that is, for all the chromosomes, plasmids, and scaffolds. If TRUE then chromosome information is returned only for the *assembled molecules*. These are the chromosomes (including the mitochondrial chromosome) and plasmids only. No scaffolds.
assembly.units	If NULL (the default) then chromosome information is returned for all the sequences in the assembly (unless assembled.molecules.only is set to TRUE, see above), that is, for all the chromosomes, plasmids, and scaffolds. assembly.units can be set to a character vector containing the names of *Assembly Units* (e.g. "non-nuclear") in which case chromosome information is returned only for the sequences that belong to these Assembly Units.

`recache` — `getChromInfoFromNCBI` uses a cache mechanism so the chromosome information of a given assembly only gets downloaded once during the current R session (note that the caching is done in memory so cached information does NOT persist across sessions). Setting `recache` to `TRUE` forces a new download (and recaching) of the chromosome information for the specified assembly.

`as.Seqinfo` — `TRUE` or `FALSE` (the default). If `TRUE` then a Seqinfo object is returned instead of a data frame. Note that only the `SequenceName`, `SequenceLength`, and `circular` columns of the data frame are used to make the Seqinfo object. All the other columns are ignored (and lost).

`organism` — When `organism` is specified, `registered_NCBI_assemblies()` will only return the subset of assemblies that are registered for that organism. `organism` must be specified as a single string and will be used to perform a search (with `grep()`) on the `"organism"` column of the data frame returned by `registered_NCBI_assemblies()`. The search is case-insensitive.

Details

registered vs *unregistered* NCBI assemblies:

- All NCBI assemblies can be looked up by assembly accession (GenBank or RefSeq) but only *registered* assemblies can also be looked up by assembly name.
- For *registered* assemblies, the returned circularity flags are guaranteed to be accurate. For *unregistered* assemblies, a heuristic is used to determine the circular sequences.

Please contact the maintainer of the **GenomeInfoDb** package to request registration of additional assemblies.

Value

For `getChromInfoFromNCBI`: By default, a 10-column data frame with columns:

1. `SequenceName`: character.
2. `SequenceRole`: factor.
3. `AssignedMolecule`: factor.
4. `GenBankAccn`: character.
5. `Relationship`: factor.
6. `RefSeqAccn`: character.
7. `AssemblyUnit`: factor.
8. `SequenceLength`: integer. Note that this column **can** contain NAs! For example this is the case in assembly Amel_HAv3.1 where the length of sequence MT is missing or in assembly Release 5 where the length of sequence Un is missing.
9. `UCSCStyleName`: character.
10. `circular`: logical.

For `registered_NCBI_assemblies`: A data frame summarizing all the NCBI assemblies currently *registered* in the **GenomeInfoDb** package.

Author(s)

H. Pagès

See Also

- getChromInfoFromUCSC for getting chromosome information for a UCSC genome.
- getChromInfoFromEnsembl for getting chromosome information for an Ensembl species.
- Seqinfo objects.

Examples

```
## All registered NCBI assemblies for Triticum aestivum (bread wheat):
registered_NCBI_assemblies("tri")[1:4]

## All registered NCBI assemblies for Homo sapiens:
registered_NCBI_assemblies("homo")[1:4]

## Internet access required!
getChromInfoFromNCBI("GRCh37")
getChromInfoFromNCBI("GRCh37", as.Seqinfo=TRUE)
getChromInfoFromNCBI("GRCh37", assembled.molecules.only=TRUE)

## The GRCh38.p14 assembly only adds "patch sequences" to the GRCh38
## assembly:
GRCh38 <- getChromInfoFromNCBI("GRCh38")
table(GRCh38$SequenceRole)
GRCh38.p14 <- getChromInfoFromNCBI("GRCh38.p14")
table(GRCh38.p14$SequenceRole)  # 254 patch sequences (164 fix + 90 novel)

## All registered NCBI assemblies for Arabidopsis thaliana:
registered_NCBI_assemblies("arabi")[1:4]
getChromInfoFromNCBI("TAIR10.1")
getChromInfoFromNCBI("TAIR10.1", assembly.units="non-nuclear")

## Sanity checks:
idx <- match(GRCh38$SequenceName, GRCh38.p14$SequenceName)
stopifnot(!anyNA(idx))
tmp1 <- GRCh38.p14[idx, ]
rownames(tmp1) <- NULL
tmp2 <- GRCh38.p14[-idx, ]
stopifnot(
  identical(tmp1[ , -(5:7)], GRCh38[ , -(5:7)]),
  identical(tmp2, GRCh38.p14[GRCh38.p14$AssemblyUnit == "PATCHES", ])
)
```

---

getChromInfoFromUCSC *Get chromosome information for a UCSC genome*

Description

getChromInfoFromUCSC returns chromosome information like sequence names, lengths and circularity flags for a given UCSC genome e.g. for hg19, panTro6, sacCer3, etc...

Note that getChromInfoFromUCSC behaves slightly differently depending on whether a genome is *registered* in the **GenomeInfoDb** package or not. See below for the details.

Use registered_UCSC_genomes to list all the UCSC genomes currently registered in the **GenomeInfoDb** package.

Usage

```
getChromInfoFromUCSC(genome,
                     assembled.molecules.only=FALSE,
                     map.NCBI=FALSE,
                     add.ensembl.col=FALSE,
                     goldenPath.url=getOption("UCSC.goldenPath.url"),
                     recache=FALSE,
                     as.Seqinfo=FALSE)

registered_UCSC_genomes(organism=NA)
```

Arguments

genome	A single string specifying the name of a UCSC genome e.g. "panTro6", "mm39", "sacCer3", etc...
assembled.molecules.only	If FALSE (the default) then chromosome information is returned for all the sequences in the genome, that is, for all the chromosomes, plasmids, and scaffolds. If TRUE then chromosome information is returned only for the *assembled molecules*. These are the chromosomes (including the mitochondrial chromosome) and plasmids only. No scaffolds. Note that assembled.molecules.only=TRUE is supported only for *registered* genomes. When used on an *unregistered* genome, assembled.molecules.only is ignored with a warning.
map.NCBI	TRUE or FALSE (the default). If TRUE then NCBI chromosome information is bound to the result. This information is retrieved from NCBI by calling getChromInfoFromNCBI on the NCBI assembly that the UCSC genome is based on. Then the data frame returned by getChromInfoFromNCBI ("NCBI chrom info") is *mapped* and bound to the data frame returned by getChromInfoFromUCSC ("UCSC chrom info"). This "map and bind" operation is similar to a JOIN in SQL. Note that not all rows in the "UCSC chrom info" data frame are necessarily mapped to a row in the "NCBI chrom info" data frame. For example chrM in hg19 has no corresponding sequence in the GRCh37 assembly (the mitochondrial chromosome was omitted from GRCh37). For the unmapped rows the NCBI columns in the final data frame are filled with NAs (LEFT JOIN in SQL).

The primary use case for using `map.NCBI=TRUE` is to map UCSC sequence names to NCBI sequence names. This is only supported for *registered* UCSC genomes based on an NCBI assembly!

`add.ensembl.col`

`TRUE` or `FALSE` (the default). Whether or not the Ensembl sequence names should be added to the result (in column `ensembl`).

`goldenPath.url` A single string specifying the URL to the UCSC goldenPath location where the chromosome sizes are expected to be found.

`recache` `getChromInfoFromUCSC` uses a cache mechanism so the chromosome sizes of a given genome only get downloaded once during the current R session (note that the caching is done in memory so cached information does NOT persist across sessions). Setting `recache` to `TRUE` forces a new download (and recaching) of the chromosome sizes for the specified genome.

`as.Seqinfo` `TRUE` or `FALSE` (the default). If `TRUE` then a Seqinfo object is returned instead of a data frame. Note that only the `chrom`, `size`, and `circular` columns of the data frame are used to make the Seqinfo object. All the other columns are ignored (and lost).

`organism` When `organism` is specified, `registered_UCSC_genomes()` will only return the subset of genomes that are registered for that organism. `organism` must be specified as a single string and will be used to perform a search (with `grep()`) on the `"organism"` column of the data frame returned by `registered_UCSC_genomes()`. The search is case-insensitive.

Details

*** *Registered* vs *unregistered* UCSC genomes ***

- For *registered* genomes, the returned data frame contains information about which sequences are assembled molecules and which are not, and the `assembled.molecules.only` argument is supported. For *unregistered* genomes, this information is missing, and the `assembled.molecules.only` argument is ignored with a warning.
- For *registered* genomes, the returned circularity flags are guaranteed to be accurate. For *unregistered* genomes, a heuristic is used to determine the circular sequences.
- For *registered* genomes, special care is taken to make sure that the sequences are returned in a sensible order. For *unregistered* genomes, a heuristic is used to return the sequences in a sensible order.

Please contact the maintainer of the **GenomeInfoDb** package to request registration of additional genomes.

*** Offline mode ***

`getChromInfoFromUCSC()` supports an "offline mode" when called with `assembled.molecules.only=TRUE`, but only for a selection of *registered* genomes. The "offline mode" works thanks to a collection of tab-delimited files stored in the package, that contain the "assembled molecules info" for the supported genomes. This makes calls like:

```
getChromInfoFromUCSC("hg38", assembled.molecules.only=TRUE)
```

fast and reliable i.e. the call will always work, even when offline!

See README.TXT in GenomeInfoDb/inst/extdata/assembled_molecules_db/UCSC/ for more information.

Note that calling `getChromInfoFromUCSC()` with `assembled.molecules.only=FALSE` (the default), or with `recache=TRUE`, will trigger retrieval of the chromosome info from UCSC, and will issue a warning if this info no longer matches the "assembled molecules info" stored in the package.

Please contact the maintainer of the **GenomeInfoDb** package to request genome additions to the "offline mode".

Value

For `getChromInfoFromUCSC`: By default, a 4-column data frame with columns:

1. `chrom`: character.
2. `size`: integer.
3. `assembled`: logical.
4. `circular`: logical.

If `map.NCBI` is `TRUE`, then 7 "NCBI columns" are added to the result:

- `NCBI.SequenceName`: character.
- `NCBI.SequenceRole`: factor.
- `NCBI.AssignedMolecule`: factor.
- `NCBI.GenBankAccn`: character.
- `NCBI.Relationship`: factor.
- `NCBI.RefSeqAccn`: character.
- `NCBI.AssemblyUnit`: factor.

Note that the names of the "NCBI columns" are those returned by `getChromInfoFromNCBI` but with the `NCBI.` prefix added to them.

If `add.ensembl.col` is `TRUE`, the column `ensembl` is added to the result.

For `registered_UCSC_genomes`: A data frame summarizing all the UCSC genomes currently *registered* in the **GenomeInfoDb** package.

Author(s)

H. Pagès

See Also

- `getChromInfoFromNCBI` for getting chromosome information for an NCBI assembly.
- `getChromInfoFromEnsembl` for getting chromosome information for an Ensembl species.
- Seqinfo objects.
- The `getBSgenome` convenience utility in the **BSgenome** package for getting a BSgenome object from an installed BSgenome data package.

Examples

```
## ---------------------------------------------------------------------
## A. BASIC EXAMPLES
## ---------------------------------------------------------------------

## --- Internet access required! ---

getChromInfoFromUCSC("hg19")

getChromInfoFromUCSC("hg19", as.Seqinfo=TRUE)

## Map the hg38 sequences to their corresponding sequences in
## the GRCh38.p13 assembly:
getChromInfoFromUCSC("hg38", map.NCBI=TRUE)[c(1, 5)]

## Note that some NCBI-based UCSC genomes contain sequences that
## are not mapped. For example this is the case for chrM in hg19:
hg19 <- getChromInfoFromUCSC("hg19", map.NCBI=TRUE)
hg19[is.na(hg19$NCBI.SequenceName), ]

## Map the hg19 sequences to the Ensembl sequence names:
getChromInfoFromUCSC("hg19", add.ensembl.col=TRUE)

## --- No internet access required! (offline mode) ---

getChromInfoFromUCSC("hg19", assembled.molecules.only=TRUE)

getChromInfoFromUCSC("panTro6", assembled.molecules.only=TRUE)

getChromInfoFromUCSC("bosTau9", assembled.molecules.only=TRUE)

## --- List of UCSC genomes currently registered in the package ---

registered_UCSC_genomes()

## All registered UCSC genomes for Felis catus (domestic cat):
registered_UCSC_genomes(organism = "Felis catus")

## All registered UCSC genomes for Homo sapiens:
registered_UCSC_genomes("homo")

## ---------------------------------------------------------------------
## B. USING getChromInfoFromUCSC() TO SET UCSC SEQUENCE NAMES ON THE
##    GRCh38 GENOME
## ---------------------------------------------------------------------

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

```
## Get the chromosome info for the hg38 genome:
hg38_chrom_info <- getChromInfoFromUCSC("hg38", map.NCBI=TRUE)
ncbi2ucsc <- setNames(hg38_chrom_info$chrom,
                      hg38_chrom_info$NCBI.SequenceName)

## Set the UCSC sequence names on 'genome':
seqlevels(genome) <- ncbi2ucsc[seqlevels(genome)]
genome

## Sanity check: check that the sequence lengths in 'genome' are the same
## as in 'hg38_chrom_info':
m <- match(seqlevels(genome), hg38_chrom_info$chrom)
stopifnot(identical(unname(seqlengths(genome)), hg38_chrom_info$size[m]))
```

`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")
```

NCBI-utils *Utility functions to access NCBI resources*

Description

Low-level utility functions to access NCBI resources. Not intended to be used directly by the end user.

Usage

```
find_NCBI_assembly_ftp_dir(assembly_accession, assembly_name=NA)

fetch_assembly_report(assembly_accession, assembly_name=NA,
                      AssemblyUnits=NULL)
```

Arguments

`assembly_accession`

A single string containing either a GenBank assembly accession (e.g. `"GCA_000001405.15"`) or a RefSeq assembly accession (e.g. `"GCF_000001405.26"`).

Alternatively, for `fetch_assembly_report()`, the `assembly_accession` argument can be set to the URL to the *assembly report* (a.k.a. "Full sequence report").

`assembly_name` A single string or `NA`.

`AssemblyUnits` By default, all the *assembly units* are included in the data frame returned by `fetch_assembly_report()`. To include only a subset of assembly units, pass a character vector containing the names of the assembly units to include to the `AssemblyUnits` argument.

Value

For `find_NCBI_assembly_ftp_dir()`: A length-2 character vector:

- The 1st element in the vector is the URL to the FTP dir, without the trailing slash.
- The 2nd element in the vector is the prefix used in the names of most of the files in the FTP dir.

For `fetch_assembly_report()`: A data frame with 1 row per sequence in the assembly and 10 columns:

1. SequenceName
2. SequenceRole
3. AssignedMolecule
4. AssignedMoleculeLocationOrType
5. GenBankAccn
6. Relationship
7. RefSeqAccn
8. AssemblyUnit
9. SequenceLength
10. UCSCStyleName

Note

`fetch_assembly_report` is the workhorse behind higher-level and more user-friendly `getChromInfoFromNCBI`.

Author(s)

H. Pagès

See Also

`getChromInfoFromNCBI` for a higher-level and more user-friendly version of `fetch_assembly_report`.

Examples

```
ftp_dir <- find_NCBI_assembly_ftp_dir("GCA_000001405.15")
ftp_dir

url <- ftp_dir[[1]]     # URL to the FTP dir
prefix <- ftp_dir[[2]]  # prefix used in names of most files

list_ftp_dir(url)

assembly_report_url <- paste0(url, "/", prefix, "_assembly_report.txt")

## To fetch the assembly report for assembly GCA_000001405.15, you can
## call fetch_assembly_report() on the assembly accession or directly
## on the URL to the assembly report:
```

```
assembly_report <- fetch_assembly_report("GCA_000001405.15")
dim(assembly_report)
head(assembly_report)

## Sanity check:
assembly_report2 <- fetch_assembly_report(assembly_report_url)
stopifnot(identical(assembly_report, assembly_report2))
```

`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`	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.

`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".

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`, `seqlevelsInUse`, and `seqlevels0`) 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...

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 for conveniently renaming the seqlevels of an object according to a given 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.
- seqlevels-wrappers for convenience wrappers to the `seqlevels` getter and setter.
- `rankSeqlevels`, on which `sortSeqlevels` is based.

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)
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 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 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 ('getChromInfoFromNCBI()' or '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
}
```

`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 from its seqlevels using a heuristic helped by a seqlevels style database stored in the **GenomeInfoDb** package. If the underlying genome is known (i.e. if unique(genome(x)) is not NA), the name of the genome or assembly (e.g. ce11 or WBcel235) is also used by the heuristic.

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
## ---------------------------------------------------------------------

## On a character vector:
x <- paste0("chr", 1:5)
```

```
seqlevelsStyle(x)
seqlevelsStyle(x) <- "NCBI"
x

## On a GRanges object:
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

## In general the seqlevelsStyle() setter doesn't know how to rename
## scaffolds. However, if the genome is specified, it's very likely
## that seqlevelsStyle() will be able to take advantage of that:
gr <- GRanges(rep(c("2", "Y", "Hs6_111610_36"), 2), IRanges(1:6, 10))
genome(gr) <- "NCBI36"
seqlevelsStyle(gr) <- "UCSC"
gr

## On a Seqinfo object:
si <- si0 <- Seqinfo(genome="apiMel2")
si
seqlevelsStyle(si) <- "NCBI"
si
seqlevelsStyle(si) <- "RefSeq"
si
seqlevelsStyle(si) <- "UCSC"
stopifnot(identical(si0, si))

si <- si0 <- Seqinfo(genome="WBcel235")
si
seqlevelsStyle(si) <- "UCSC"
si
seqlevelsStyle(si) <- "RefSeq"
si
seqlevelsStyle(si) <- "NCBI"
stopifnot(identical(si0, si))

si <- Seqinfo(genome="macFas5")
si
seqlevelsStyle(si) <- "NCBI"
si

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

```
## 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