Package 'BSgenome'

October 8, 2015

Title Infrastructure for Biostrings-based genome data packages

Description Infrastructure shared by all the Biostrings-based genome data packages

Version 1.36.3

Author Herve Pages

Maintainer H. Pages <hpages@fredhutch.org>

biocViews Genetics, Infrastructure, DataRepresentation, SequenceMatching, Annotation, SNP

Depends R (>= 2.8.0), methods, BiocGenerics (>= 0.13.8), S4Vectors (>= 0.5.10), IRanges (>= 2.1.33), GenomeInfoDb (>= 1.3.19), GenomicRanges (>= 1.19.23), Biostrings (>= 2.35.3), rtracklayer (>= 1.25.8)

Imports methods, stats, BiocGenerics, S4Vectors, IRanges, XVector, GenomeInfoDb, GenomicRanges, Biostrings, Rsamtools, rtracklayer

Suggests BiocInstaller, Biobase, BSgenome.Celegans.UCSC.ce2, BSgenome.Hsapiens.UCSC.hg38, BSgenome.Hsapiens.UCSC.hg38.masked, BSgenome.Mmusculus.UCSC.mm10, BSgenome.Rnorvegicus.UCSC.rn5, TxDb.Hsapiens.UCSC.hg38.knownGene, TxDb.Mmusculus.UCSC.mm10.knownGene, SNPlocs.Hsapiens.dbSNP141.GRCh38, XtraSNPlocs.Hsapiens.dbSNP141.GRCh38, hgu95av2probe, RUnit

License Artistic-2.0

LazyLoad yes

Collate utils.R OnDiskLongTable-class.R OnDiskNamedSequences-class.R SNPlocs-class.R InjectSNPsHandler-class.R XtraSNPlocs-class.R BSgenome-class.R available.genomes.R injectSNPs.R getSeq-methods.R bsapply.R BSgenomeViews-class.R BSgenome-utils.R export-methods.R GenomeData-class.R GenomeDataList-class.R gdapply.R gdReduce.R BSgenomeForge.R

NeedsCompilation no

R topics documented:

`available.genomes` *Find available/installed genomes*

Description

`available.genomes` gets the list of BSgenome data packages that are available in the Bioconductor repositories for your version of R/Bioconductor.

`installed.genomes` gets the list of BSgenome data packages that are currently installed on your system.

`getBSgenome` searchs the installed BSgenome data packages for the specified genome and returns it as a BSgenome object.

Usage

```
available.genomes(splitNameParts=FALSE, type=getOption("pkgType"))

installed.genomes(splitNameParts=FALSE)

getBSgenome(genome, masked=FALSE)
```

Arguments

`splitNameParts`	Whether to split or not the package names in parts. In that case the result is returned in a data frame with 5 columns.
`type`	Character string indicating the type of package (`"source"`, `"mac.binary"` or `"win.binary"`) to look for.

genome	A BSgenome object, or the full name of an installed BSgenome data package, or a short string specifying a genome assembly (a.k.a. provider version) that refers unambiguously to an installed BSgenome data package.
masked	TRUE or FALSE. Whether to search for the *masked* BSgenome object (i.e. the object that contains the masked sequences) or not (the default).

Details

A BSgenome data package contains the full genome sequences for a given organism.

Its name typically has 4 parts (5 parts if it's a *masked* BSgenome data package i.e. if it contains masked sequences) separated by a dot e.g. `BSgenome.Mmusculus.UCSC.mm10` or `BSgenome.Mmusculus.UCSC.mm10.masked`

1. The 1st part is always `BSgenome`.
2. The 2nd part is the name of the organism in abbreviated form e.g. `Mmusculus`, `Hsapiens`, `Celegans`, `Scerevisiae`, `Ecoli`, etc...
3. The 3rd part is the name of the organisation who provided the genome sequences. We formally refer to it as the *provider* of the genome. E.g. `UCSC`, `NCBI`, `TAIR`, etc...
4. The 4th part is the release string or number used by this organisation for this particular genome assembly. We formally refer to it as the *provider version* of the genome. E.g. `hg38`, `GRCh38`, `hg19`, `mm10`, `susScr3`, etc...
5. If the package contains masked sequences, its name has the `.masked` suffix added to it, which is typically the 5th part.

A BSgenome data package contains a single top-level object (a BSgenome object) named like the package itself that can be used to access the genome sequences.

Value

For `available.genomes` and `installed.genomes`: by default (i.e. if `splitNameParts=FALSE`), a character vector containing the names of the BSgenome data packages that are available (for `available.genomes`) or currently installed (for `installed.genomes`). If `splitNameParts=TRUE`, the list of packages is returned in a data frame with one row per package and the following columns: `pkgname` (character), `organism` (factor), `provider` (factor), `provider_version` (character), and `masked` (logical).

For `getBSgenome`: the BSgenome object containing the sequences for the specified genome. Or an error if the object cannot be found in the BSgenome data packages currently installed.

Author(s)

H. Pages

See Also

- BSgenome objects.
- `available.packages`.

Examples

```
## ---------------------------------------------------------------------
## available.genomes() and installed.genomes()
## ---------------------------------------------------------------------

# What genomes are currently installed:
installed.genomes()

# What genomes are available:
available.genomes()

# Split the package names in parts:
av_gen <- available.genomes(splitNameParts=TRUE)
table(av_gen$organism)
table(av_gen$provider)

# Make your choice and install with:
library(BiocInstaller)
biocLite("BSgenome.Scerevisiae.UCSC.sacCer1")

# Have a coffee 8-)

# Load the package and display the index of sequences for this genome:
library(BSgenome.Scerevisiae.UCSC.sacCer1)
Scerevisiae  # same as BSgenome.Scerevisiae.UCSC.sacCer1

## ---------------------------------------------------------------------
## getBSgenome()
## ---------------------------------------------------------------------

## Specify the full name of an installed BSgenome data package:
genome <- getBSgenome("BSgenome.Celegans.UCSC.ce2")
genome

## Specify a genome assembly (a.k.a. provider version):
genome <- getBSgenome("hg38")
class(genome)  # BSgenome object
providerVersion(genome)
genome$chrM

genome <- getBSgenome("hg38", masked=TRUE)
class(genome)  # MaskedBSgenome object
providerVersion(genome)
genome$chr22
```

---

bsapply *bsapply*

Description

Apply a function to each chromosome in a genome.

Usage

```
bsapply(BSParams, ...)
```

Arguments

BSParams	a BSParams object that holds the various parameters needed to configure the bsapply function
...	optional arguments to 'FUN'.

Details

By default the exclude parameter is set to not exclude anything. A popular option will probably be to set this to "rand" so that random bits of unassigned contigs are filtered out.

Value

If BSParams sets simplify=FALSE, an ordinary list is returned containing the results generated using the remaining BSParams specifications. If BSParams sets simplify=TRUE, an sapply-like simplification is performed on the results.

Author(s)

Marc Carlson

See Also

BSParams-class, BSgenome-class, BSgenome-utils

Examples

```
## Load the Worm genome:
library("BSgenome.Celegans.UCSC.ce2")

## Count the alphabet frequencies for every chromosome but exclude
## mitochrondrial and scaffold ones:
params <- new("BSParams", X = Celegans, FUN = alphabetFrequency,
exclude = c("M", "_"))
bsapply(params)

## Or we can do this same function with simplify = TRUE:
params <- new("BSParams", X = Celegans, FUN = alphabetFrequency,
exclude = c("M", "_"), simplify = TRUE)
bsapply(params)


## Examples to show how we might look for a string (in this case an
## ebox motif) across the whole genome.
Ebox <- DNAStringSet("CACGTG")
pdict0 <- PDict(Ebox)

params <- new("BSParams", X = Celegans, FUN = countPDict, simplify = TRUE)
```

```
bsapply(params, pdict = pdict0)

params@FUN <- matchPDict
bsapply(params, pdict = pdict0)

## And since its really overkill to use matchPDict to find a single pattern:
params@FUN <- matchPattern
bsapply(params, pattern = "CACGTG")


## Examples on how to use the masks
library(BSgenome.Hsapiens.UCSC.hg38.masked)
genome <- BSgenome.Hsapiens.UCSC.hg38.masked
## I can make things verbose if I want to see the chromosomes getting processed.
options(verbose=TRUE)
## For the 1st example, lets use default masks
params <- new("BSParams", X = genome, FUN = alphabetFrequency,
exclude = c(1:8,"M","X","_"), simplify = TRUE)
bsapply(params)

if (interactive()) {
  ## Set up the motifList to filter out all double T's and all double C's
  params@motifList <-c("TT","CC")
  bsapply(params)

  ## Get rid of the motifList
  params@motifList=as.character()
}

##Enable all standard masks
params@maskList <- c(RM=TRUE,TRF=TRUE)
bsapply(params)

##Disable all standard masks
params@maskList <- c(AGAPS=FALSE,AMB=FALSE)
bsapply(params)
```

BSgenome-class *BSgenome objects*

Description

The BSgenome class is a container for storing the full genome sequences of a given organism. BSgenome objects are usually made in advance by a volunteer and made available to the Bioconductor community as "BSgenome data packages". See ?available.genomes for how to get the list of "BSgenome data packages" curently available.

Accessor methods

In the code snippets below, x is a BSgenome object. Note that, because the BSgenome class contains the GenomeDescription class, then all the accessor methods for GenomeDescription objects can also be used on x.

`sourceUrl(x)` Returns the source URL i.e. the permanent URL to the place where the FASTA files used to produce the sequences contained in x can be found (and downloaded).

`seqnames(x)`, `seqnames(x) <- value` Gets or sets the names of the single sequences contained in x. Each single sequence is stored in a DNAString or MaskedDNAString object and typically comes from a source file (FASTA) with a single record. The names returned by `seqnames(x)` usually reflect the names of those source files but a common prefix or suffix was eventually removed in order to keep them as short as possible.

`seqlengths(x)` Returns the lengths of the single sequences contained in x.

See ?`length,XVector-method` and ?`length,MaskedXString-method` for the definition of the length of a DNAString or MaskedDNAString object. Note that the length of a masked sequence (MaskedXString object) is not affected by the current set of active masks but the nchar method for MaskedXString objects is.

`names(seqlengths(x))` is guaranteed to be identical to `seqnames(x)`.

`mseqnames(x)` Returns the index of the multiple sequences contained in x. Each multiple sequence is stored in a DNAStringSet object and typically comes from a source file (FASTA) with multiple records. The names returned by `mseqnames(x)` usually reflect the names of those source files but a common prefix or suffix was eventually removed in order to keep them as short as possible.

`names(x)` Returns the index of all sequences contained in x. This is the same as `c(seqnames(x), mseqnames(x))`.

`length(x)` Returns the length of x, i.e., the total number of sequences in it (single and multiple sequences). This is the same as `length(names(x))`.

`x[[name]]` Returns the sequence (single or multiple) in x named name (name must be a single string). No sequence is actually loaded into memory until this is explicitely requested with a call to `x[[name]]` or `x$name`. When loaded, a sequence is kept in a cache. It will be automatically removed from the cache at garbage collection if it's not in use anymore i.e. if there are no reference to it (other than the reference stored in the cache). With `options(verbose=TRUE)`, a message is printed each time a sequence is removed from the cache.

`x$name` Same as `x[[name]]` but name is not evaluated and therefore must be a literal character string or a name (possibly backtick quoted).

`masknames(x)` The names of the built-in masks that are defined for all the single sequences. There can be up to 4 built-in masks per sequence. These will always be (in this order): (1) the mask of assembly gaps, aka "the AGAPS mask";

(2) the mask of intra-contig ambiguities, aka "the AMB mask";

(3) the mask of repeat regions that were determined by the RepeatMasker software, aka "the RM mask";

(4) the mask of repeat regions that were determined by the Tandem Repeats Finder software (where only repeats with period less than or equal to 12 were kept), aka "the TRF mask".

All the single sequences in a given package are guaranteed to have the same collection of built-in masks (same number of masks and in the same order).

`masknames(x)` gives the names of the masks in this collection. Therefore the value returned by `masknames(x)` is a character vector made of the first N elements of `c("AGAPS", "AMB", "RM", "TRF")`, where N depends only on the BSgenome data package being looked at (0 <= N <= 4). The man page for most BSgenome data packages should provide the exact list and permanent URLs of the source data files that were used to extract the built-in masks. For example, if you've installed the BSgenome.Hsapiens.UCSC.hg38 package, load it and see the Note section in `?BSgenome.Hsapiens.UCSC.hg38`.

Author(s)

H. Pages

See Also

`available.genomes`, GenomeDescription-class, BSgenome-utils, DNAString-class, DNAStringSet-class, MaskedDNAString-class, `getSeq`, `BSgenome-method`, `injectSNPs`, subseq,XVector-method, `rm`, `gc`

Examples

```
## Loading a BSgenome data package doesn't load its sequences
## into memory:
library(BSgenome.Celegans.UCSC.ce2)

## Number of sequences in this genome:
length(Celegans)

## Display a summary of the sequences:
Celegans

## Index of single sequences:
seqnames(Celegans)

## Lengths (i.e. number of nucleotides) of the single sequences:
seqlengths(Celegans)

## Load chromosome I from disk to memory (hence takes some time)
## and keep a reference to it:
chrI <- Celegans[["chrI"]]  # equivalent to Celegans$chrI

chrI

class(chrI)   # a DNAString instance
length(chrI)  # with 15080483 nucleotides

## Single sequence can be renamed:
seqnames(Celegans) <- sub("^chr", "", seqnames(Celegans))
seqlengths(Celegans)
Celegans$I
seqnames(Celegans) <- paste0("chr", seqnames(Celegans))

## Multiple sequences:
```

```
library(BSgenome.Rnorvegicus.UCSC.rn5)
rn5 <- BSgenome.Rnorvegicus.UCSC.rn5
rn5
seqnames(rn5)
rn5_chr1 <- rn5$chr1
mseqnames(rn5)
rn5_random  <- Rnorvegicus$random
rn5_random
class(rn5_random)  # a DNAStringSet instance
## Character vector containing the description lines of the first
## 4 sequences in the original FASTA file:
names(rn5_random)[1:4]

## ---------------------------------------------------------------------
## PASS-BY-ADDRESS SEMANTIC, CACHING AND MEMORY USAGE
## ---------------------------------------------------------------------

## We want a message to be printed each time a sequence is removed
## from the cache:
options(verbose=TRUE)

gc()  # nothing seems to be removed from the cache
rm(rn5_chr1, rn5_random)
gc()  # rn5_chr1 and rn5_random are removed from the cache (they are
      # not in use anymore)

options(verbose=FALSE)

## Get the current amount of data in memory (in Mb):
mem0 <- gc()["Vcells", "(Mb)"]

system.time(rn5_chr2 <- rn5$chr2)  # read from disk

gc()["Vcells", "(Mb)"] - mem0  # 'rn5_chr2' occupies 20Mb in memory

system.time(tmp <- rn5$chr2)  # much faster! (sequence
                              # is in the cache)

gc()["Vcells", "(Mb)"] - mem0  # we're still using 20Mb (sequences
                               # have a pass-by-address semantic
                               # i.e. the sequence data are not
                               # duplicated)

## subseq() doesn't copy the sequence data either, hence it is very
## fast and memory efficient (but the returned object will hold a
## reference to 'rn5_chr2'):
y <- subseq(rn5_chr2, 10, 8000000)
gc()["Vcells", "(Mb)"] - mem0

## We must remove all references to 'rn5_chr2' before it can be
## removed from the cache (so the 20Mb of memory used by this
## sequence are freed):
options(verbose=TRUE)
```

```
rm(rn5_chr2, tmp)
gc()

## Remember that 'y' holds a reference to 'rn5_chr2' too:
rm(y)
gc()

options(verbose=FALSE)
gc()["Vcells", "(Mb)"] - mem0
```

BSgenome-utils *BSgenome utilities*

Description

Utilities for BSgenome objects.

Usage

```
## S4 method for signature 'BSgenome'
matchPWM(pwm, subject, min.score = "80%", exclude = "",
       maskList = logical(0))
## S4 method for signature 'BSgenome'
countPWM(pwm, subject, min.score = "80%", exclude = "",
       maskList = logical(0))
## S4 method for signature 'BSgenome'
vmatchPattern(pattern, subject, max.mismatch = 0, min.mismatch = 0,
            with.indels = FALSE, fixed = TRUE, algorithm = "auto",
            exclude = "", maskList = logical(0),  userMask =
               RangesList(), invertUserMask = FALSE)
## S4 method for signature 'BSgenome'
vcountPattern(pattern, subject, max.mismatch = 0, min.mismatch = 0,
            with.indels = FALSE, fixed = TRUE, algorithm = "auto",
            exclude = "", maskList = logical(0),  userMask =
               RangesList(), invertUserMask = FALSE)
## S4 method for signature 'BSgenome'
vmatchPDict(pdict, subject, max.mismatch = 0, min.mismatch = 0,
          fixed = TRUE, algorithm = "auto", verbose = FALSE,
          exclude = "", maskList = logical(0))
## S4 method for signature 'BSgenome'
vcountPDict(pdict, subject, max.mismatch = 0, min.mismatch = 0,
          fixed = TRUE, algorithm = "auto", collapse = FALSE,
          weight = 1L, verbose = FALSE, exclude = "", maskList = logical(0))
```

Arguments

pwm	A numeric matrix with row names A, C, G and T representing a Position Weight Matrix.

pattern	A DNAString object containing the pattern sequence.
pdict	A DNAStringSet object containing the pattern sequences.
subject	A BSgenome object containing the subject sequences.
min.score	The minimum score for counting a match. Can be given as a character string containing a percentage (e.g. "85%") of the highest possible score or as a single number.
max.mismatch, min.mismatch	The maximum and minimum number of mismatching letters allowed (see ?`lowlevel-matching` for the details). If non-zero, an inexact matching algorithm is used.
with.indels	If TRUE then indels are allowed. In that case, min.mismatch must be 0 and max.mismatch is interpreted as the maximum "edit distance" allowed between any pattern and any of its matches (see ?`matchPattern` for the details).
fixed	If FALSE then IUPAC extended letters are interpreted as ambiguities (see ?`lowlevel-matching` for the details).
algorithm	For vmatchPattern and vcountPattern one of the following: "auto", "naive-exact", "naive-inexact", "boyer-moore", "shift-or", or "indels". For vmatchPDict and vcountPDict one of the following: "auto", "naive-exact", "naive-inexact", "boyer-moore", or "shift-or".
collapse, weight	ignored arguments.
verbose	TRUE or FALSE.
exclude	A character vector with strings that will be used to filter out chromosomes whose names match these strings.
maskList	A named logical vector of maskStates preferred when used with a BSGenome object. When using the bsapply function, the masks will be set to the states in this vector.
userMask	A RangesList, containing a mask to be applied to each chromosome. See bsapply.
invertUserMask	Whether the userMask should be inverted.

Value

A GRanges object for matchPWM with two elementMetadata columns: "score" (numeric), and "string" (DNAStringSet).

A GRanges object for vmatchPattern.

A GRanges object for vmatchPDict with one elementMetadata column: "index", which represents a mapping to a position in the original pattern dictionary.

A data.frame object for countPWM and vcountPattern with three columns: "seqname" (factor), "strand" (factor), and "count" (integer).

A DataFrame object for vcountPDict with four columns: "seqname" ('factor' Rle), "strand" ('factor' Rle), "index" (integer) and "count" ('integer' Rle). As with vmatchPDict the index column represents a mapping to a position in the original pattern dictionary.

Author(s)

P. Aboyoun

See Also

matchPWM, matchPattern, matchPDict, bsapply

Examples

```
library(BSgenome.Celegans.UCSC.ce2)
data(HNF4alpha)

pwm <- PWM(HNF4alpha)
matchPWM(pwm, Celegans)
countPWM(pwm, Celegans)

pattern <- consensusString(HNF4alpha)
vmatchPattern(pattern, Celegans, fixed = "subject")
vcountPattern(pattern, Celegans, fixed = "subject")

vmatchPDict(HNF4alpha[1:10], Celegans)
vcountPDict(HNF4alpha[1:10], Celegans)
```

BSgenomeForge *The BSgenomeForge functions*

Description

A set of functions for making a BSgenome data package.

Usage

```
## Top-level BSgenomeForge function:

forgeBSgenomeDataPkg(x, seqs_srcdir=".", destdir=".", verbose=TRUE)

## Low-level BSgenomeForge functions:

forgeSeqlengthsFile(seqnames, prefix="", suffix=".fa",
                    seqs_srcdir=".", seqs_destdir=".", verbose=TRUE)

forgeSeqFiles(seqnames, mseqnames=NULL,
              seqfile_name=NA, prefix="", suffix=".fa",
              seqs_srcdir=".", seqs_destdir=".",
              ondisk_seq_format=c("2bit", "rda", "fa.rz", "fa"),
              verbose=TRUE)

forgeMasksFiles(seqnames, nmask_per_seq,
                seqs_destdir=".",
                ondisk_seq_format=c("2bit", "rda", "fa.rz", "fa"),
                masks_srcdir=".", masks_destdir=".",
                AGAPSfiles_type="gap", AGAPSfiles_name=NA,
```

```
                AGAPSfiles_prefix="", AGAPSfiles_suffix="_gap.txt",
                RMfiles_name=NA, RMfiles_prefix="", RMfiles_suffix=".fa.out",
                TRFfiles_name=NA, TRFfiles_prefix="", TRFfiles_suffix=".bed",
                verbose=TRUE)
```

Arguments

x	A BSgenomeDataPkgSeed object or the name of a BSgenome data package seed file. See the BSgenomeForge vignette in this package for more information.
seqs_srcdir, masks_srcdir	Single strings indicating the path to the source directories i.e. to the directories containing the source data files. Only read access to these directories is needed. See the BSgenomeForge vignette in this package for more information.
destdir	A single string indicating the path to the directory where the source tree of the target package should be created. This directory must already exist. See the BSgenomeForge vignette in this package for more information.
ondisk_seq_format	Specifies how the single sequences should be stored in the forged package. Can be "2bit", "rda", "fa.rz", or "fa". If "2bit" (the default), then all the single sequences are stored in a single twoBit file. If "rda", then each single sequence is stored in a separated serialized XString object (one per single sequence). If "fa.rz" or "fa", then all the single sequences are stored in a single FASTA file (compressed in the RAZip format if "fa.rz").
verbose	TRUE or FALSE.
seqnames, mseqnames	A character vector containing the names of the single (for seqnames) and multiple (for mseqnames) sequences to forge. See the BSgenomeForge vignette in this package for more information.
seqfile_name, prefix, suffix	See the BSgenomeForge vignette in this package for more information, in particular the description of the seqfile_name, seqfiles_prefix and seqfiles_suffix fields of a BSgenome data package seed file.
seqs_destdir, masks_destdir	During the forging process the source data files are converted into serialized Biostrings objects. seqs_destdir and masks_destdir must be single strings indicating the path to the directories where these serialized objects should be saved. These directories must already exist. forgeSeqlengthsFile will produce a single .rda file. Both forgeSeqFiles and forgeMasksFiles will produce one .rda file per sequence.
nmask_per_seq	A single integer indicating the desired number of masks per sequence. See the BSgenomeForge vignette in this package for more information.
AGAPSfiles_type, AGAPSfiles_name, AGAPSfiles_prefix, AGAPSfiles_suffix, RMfiles_name, RMfiles_prefi	These arguments are named accordingly to the corresponding fields of a BSgenome data package seed file. See the BSgenomeForge vignette in this package for more information.

Details

These functions are intended for Bioconductor users who want to make a new BSgenome data package, not for regular users of these packages. See the BSgenomeForge vignette in this package (`vignette("BSgenomeForge")`) for an extensive coverage of this topic.

Author(s)

H. Pages

Examples

```
seqs_srcdir <- system.file("extdata", package="BSgenome")
seqnames <- c("chrX", "chrM")

## Forge .rda sequence files:
forgeSeqFiles(seqnames, prefix="ce2", suffix=".fa.gz",
              seqs_srcdir=seqs_srcdir,
              seqs_destdir=tempdir(), ondisk_seq_format="rda")

## Forge .2bit sequence files:
forgeSeqFiles(seqnames, prefix="ce2", suffix=".fa.gz",
              seqs_srcdir=seqs_srcdir,
              seqs_destdir=tempdir(), ondisk_seq_format="2bit")

## Sanity checks:
library(BSgenome.Celegans.UCSC.ce2)
genome <- BSgenome.Celegans.UCSC.ce2

load(file.path(tempdir(), "chrX.rda"))
stopifnot(genome$chrX == chrX)
load(file.path(tempdir(), "chrM.rda"))
stopifnot(genome$chrM == chrM)

ce2_sequences <- import(file.path(tempdir(), "single_sequences.2bit"))
ce2_sequences0 <- DNAStringSet(list(chrX=genome$chrX, chrM=genome$chrM))
stopifnot(identical(names(ce2_sequences0), names(ce2_sequences)) &&
          all(ce2_sequences0 == ce2_sequences))
```

BSgenomeViews-class *BSgenomeViews objects*

Description

The BSgenomeViews class is a container for storing a set of genomic positions on a BSgenome object, called the "subject" in this context.

Usage

```
## Constructor
## ------------

BSgenomeViews(subject, granges)

## Accessors
## ---------

## S4 method for signature 'BSgenomeViews'
subject(x)
## S4 method for signature 'BSgenomeViews'
granges(x, use.mcols=FALSE)

## S4 method for signature 'BSgenomeViews'
length(x)
## S4 method for signature 'BSgenomeViews'
names(x)
## S4 method for signature 'BSgenomeViews'
seqnames(x)
## S4 method for signature 'BSgenomeViews'
start(x)
## S4 method for signature 'BSgenomeViews'
end(x)
## S4 method for signature 'BSgenomeViews'
width(x)
## S4 method for signature 'BSgenomeViews'
strand(x)
## S4 method for signature 'BSgenomeViews'
ranges(x, use.mcols=FALSE)
## S4 method for signature 'BSgenomeViews'
elementLengths(x)
## S4 method for signature 'BSgenomeViews'
seqinfo(x)

## DNAStringSet methods
## --------------------

## S4 method for signature 'BSgenomeViews'
seqtype(x)

## S4 method for signature 'BSgenomeViews'
nchar(x, type="chars", allowNA=FALSE)

## S4 method for signature 'BSgenomeViews'
unlist(x, recursive=TRUE, use.names=TRUE)

## S4 method for signature 'BSgenomeViews'
```

```
alphabetFrequency(x, as.prob=FALSE, collapse=FALSE, baseOnly=FALSE)

## S4 method for signature 'BSgenomeViews'
hasOnlyBaseLetters(x)

## S4 method for signature 'BSgenomeViews'
uniqueLetters(x)

## S4 method for signature 'BSgenomeViews'
letterFrequency(x, letters, OR="|", as.prob=FALSE, collapse=FALSE)

## S4 method for signature 'BSgenomeViews'
oligonucleotideFrequency(x, width, step=1,
                         as.prob=FALSE, as.array=FALSE,
                      fast.moving.side="right", with.labels=TRUE, simplify.as="matrix")

## S4 method for signature 'BSgenomeViews'
nucleotideFrequencyAt(x, at, as.prob=FALSE, as.array=TRUE,
                        fast.moving.side="right", with.labels=TRUE)

## S4 method for signature 'BSgenomeViews'
consensusMatrix(x, as.prob=FALSE, shift=0L, width=NULL, baseOnly=FALSE)

## S4 method for signature 'BSgenomeViews'
consensusString(x, ambiguityMap=IUPAC_CODE_MAP, threshold=0.25,
                shift=0L, width=NULL)
```

Arguments

`subject`	A BSgenome object or the name of a reference genome specified in a way that is accepted by the `getBSgenome` function. In that case the corresponding BSgenome data package needs to be already installed (see ?`getBSgenome` for the details).
`granges`	A GRanges object containing ranges relative to the genomic sequences stored in `subject`.
`x`	A BSgenomeViews object.
`use.mcols`	`TRUE` or `FALSE` (the default). Whether the metadata columns on `x` (accessible with `mcols(x)`) should be propagated to the returned object or not.
`type, allowNA, recursive, use.names`	Ignored.
`as.prob, letters, OR, width`	See ?`alphabetFrequency` and ?`oligonucleotideFrequency` in the **Biostrings** package.
`collapse, baseOnly`	See ?`alphabetFrequency` in the **Biostrings** package.
`step, as.array, fast.moving.side, with.labels, simplify.as, at`	See ?`oligonucleotideFrequency` in the **Biostrings** package.

`shift, ambiguityMap, threshold`
: See ?`consensusMatrix` in the **Biostrings** package.

Constructors

`BSgenomeViews(subject, granges)`: Make a BSgenomeViews object by putting the views specified by `granges` on top of the genomic sequences stored in `subject`. See above for how argument `subject` and `granges` should be specified.

`Views(subject, granges)`: Equivalent to `BSgenomeViews(subject, granges)`. Provided for convenience.

Accessors

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

`subject(x)`: Return the BSgenome object containing the full genomic sequences on top of which the views in `x` are defined.

`granges(x, use.mcols=FALSE)`: Return the genomic ranges of the views as a GRanges object. These ranges are relative to the genomic sequences stored in `subject(x)`.

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

`names(x)`: The names of the views in `x`.

`seqnames(x)`, `start(x)`, `end(x)`, `width(x)`, `strand(x)`: Equivalent to `seqnames(granges(x))`, `start(granges(x))`, `end(granges(x))`, `width(granges(x))`, `strand(granges(x))`, respectively.

`ranges(x, use.mcols=FALSE)`: Equivalent to `ranges(granges(x, use.mcols), use.mcols)`.

`elementLengths(x)`: Equivalent to `width(x)`.

`seqinfo(x)`: Equivalent to `seqinfo(subject(x))` and to `seqinfo(granges(x))` (both are guaranteed to be the same). See ?`seqinfo` in the **GenomeInfoDb** package for more information.

Coercion

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

`as(x, "DNAStringSet")`: Turn `x` into a DNAStringSet object by extxracting the DNA sequence corresponding to each view. Alternatively `as(x, "XStringSet")` can be used for this, and is equivalent to `as(x, "DNAStringSet")`.

`as.character(x)`: Equivalent to `as.character(as(x, "DNAStringSet"))`.

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

Subsetting

`x[i]`: Select the views specified by `i`.

`x[[i]]`: Extract the one view specified by `i`.

DNAStringSet methods

For convenience, some methods defined for DNAStringSet objects in the **Biostrings** package can be used directly on a BSgenomeViews object. In that case, everything happens like if the BSgenomeViews object `x` was turned into a DNAStringSet object (with `as(x, "DNAStringSet")`) before it's passed to the method for DNAStringSet objects.

At the moment, the list of such methods is: `seqtype`, `nchar,XStringSet-method`, `unlist,XStringSet-method`, `alphabetFrequency`, `hasOnlyBaseLetters`, `uniqueLetters`, `letterFrequency`, `oligonucleotideFrequency`, `nucleotideFrequencyAt`, `consensusMatrix`, and `consensusString`.

See the corresponding man page in the **Biostrings** package for a description of these methods.

Author(s)

H. Pages

See Also

- The BSgenome class.
- The GRanges class in the **GenomicRanges** package.
- The DNAStringSet class in the **Biostrings** package.
- The seqinfo and related getters in the **GenomeInfoDb** package for getting the sequence information stored in an object.
- TxDb objects in the **GenomicFeatures** package.

Examples

```
library(BSgenome.Mmusculus.UCSC.mm10)
genome <- BSgenome.Mmusculus.UCSC.mm10
library(TxDb.Mmusculus.UCSC.mm10.knownGene)
txdb <- TxDb.Mmusculus.UCSC.mm10.knownGene
ex <- exons(txdb, columns=c("exon_id", "tx_name", "gene_id"))
v <- Views(genome, ex)
v

subject(v)
granges(v)
seqinfo(v)
as(v, "DNAStringSet")

v10 <- v[1:10]  # select the first 10 views
subject(v10)    # same as subject(v)
granges(v10)
seqinfo(v10)    # same as seqinfo(v)
as(v10, "DNAStringSet")
alphabetFrequency(v10)
alphabetFrequency(v10, collapse=TRUE)

v12 <- v[width(v) <= 12]  # select the views of 12 nucleotides or less
head(as.data.frame(v12))
trinucleotideFrequency(v12, simplify.as="collapsed")
```

```
## BSgenomeViews objects are list-like objects. That is, the
## BSgenomeViews class derives from List and typical list/List
## operations (e.g. [[, elementLengths(), unlist(), elementType(),
## etc...) work on these objects:
is(v12, "List")  # TRUE
v12[[2]]
head(elementLengths(v12))  # elementLengths(v) is the same as width(v)
unlist(v12)
elementType(v12)
```

BSParams-class *Class "BSParams"*

Description

A parameter class for representing all parameters needed for running the bsapply method.

Objects from the Class

Objects can be created by calls of the form new("BSParams", ...).

Slots

X: a BSgenome object that contains chromosomes that you wish to apply FUN on

FUN: the function to apply to each chromosome in the BSgenome object 'X'

exclude: this is a character vector with strings that will be used to filter out chromosomes whose names match these strings.

simplify: TRUE/FALSE value to indicate whether or not the function should try to simplify the output for you.

maskList: A named logical vector of maskStates preferred when used with a BSGenome object. When using the bsapply function, the masks will be set to the states in this vector.

motifList: A character vector which should contain motifs that the user wishes to mask from the sequence.

userMask: A RangesList object, where each element masks the corresponding chromosome in X. This allows the user to conveniently apply masks besides those included in X.

invertUserMask: A logical indicating whether to invert each mask in userMask.

Methods

bsapply(p) Performs the function FUN using the parameters contained within BSParams.

Author(s)

Marc Carlson

See Also

bsapply

export-methods *Export a BSgenome object as a FASTA or twoBit file*

Description

export methods for BSgenome objects.

NOTE: The export generic function and most of its methods are defined and documented in the **rtracklayer** package. This man page only documents the 2 export methods define in the **BSgenome** package.

Usage

```
## S4 method for signature 'BSgenome,FastaFile,ANY'
export(object, con, format, ...)
## S4 method for signature 'BSgenome,TwoBitFile,ANY'
export(object, con, format, ...)
```

Arguments

object	The BSgenome object to export.
con	A FastaFile or TwoBitFile object. Alternatively con can be a single string containing the path to a FASTA or twoBit file, in which case either the file extension or the format argument needs to be "fasta", "twoBit", or "2bit". Also note that in this case, the export method that is called is either the method with signature c("ANY", "character", "missing") or the method with signature c("ANY", "character", "character"), both defined in the **rtracklayer** package. If object is a BSgenome object and the file extension or the format argument is "fasta", "twoBit", or "2bit", then the flow eventually reaches one of 2 methods documented here.
format	If not missing, should be "fasta", "twoBit", or "2bit" (case insensitive for "twoBit" and "2bit").
...	Extra arguments passed down to other methods. The method for TwoBitFile objects forwards them to bsapply.

Author(s)

Michael Lawrence

See Also

- BSgenome objects.
- The export generic, and FastaFile and TwoBitFile objects in the **rtracklayer** package.

Examples

```
library(BSgenome.Celegans.UCSC.ce2)
genome <- BSgenome.Celegans.UCSC.ce2

## Export as FASTA file.
out1_file <- file.path(tempdir(), "Celegans.fasta")
export(genome, out1_file)

## Export as twoBit file.
out2_file <- file.path(tempdir(), "Celegans.2bit")
export(genome, out2_file)

## Sanity checks:
dna0 <- DNAStringSet(as.list(genome))

system.time(dna1 <- import(out1_file))
stopifnot(identical(names(dna0), names(dna1)) && all(dna0 == dna1))

system.time(dna2 <- import(out2_file))  # importing twoBit is 10-20x
                                       # faster than importing non
                                       # compressed FASTA
stopifnot(identical(names(dna0), names(dna2)) && all(dna0 == dna2))
```

---

gdapply *Applies a function to elements of a GenomeData*

Description

WARNING: Starting with BioC 3.1, GenomeData and GenomeDataList objects are defunct. Note that the GenomeData/GenomeDataList containers predate the GRanges/GRangesList containers and, most of the times, the latters can be used instead of the formers. Please let us know on the bioc-devel mailing list (http://bioconductor.org/help/mailing-list/) if you have a use case where you think there are significant benefits in using GenomeData/GenomeDataList over GRanges/GRangesList, or if you have questions or concerns about this.

Returns a list of values obtained by applying a function to elements of a GenomeData or GenomeDataList object.

Usage

```
gdapply(X, FUN, ...)
```

Arguments

X	An object of class GenomeData or GenomeDataList.
FUN	A function to be applied to each chromosome-level sub-element of X.
...	Further arguments; passed to FUN

Value

Typically an object of the same class as X.

Author(s)

Deepayan Sarkar

See Also

GenomeData-class, GenomeDataList-class

---

gdReduce *Reduces arguments to a single GenomeData instance*

Description

WARNING: Starting with BioC 3.1, GenomeData and GenomeDataList objects are defunct. Note that the GenomeData/GenomeDataList containers predate the GRanges/GRangesList containers and, most of the times, the latters can be used instead of the formers. Please let us know on the bioc-devel mailing list (http://bioconductor.org/help/mailing-list/) if you have a use case where you think there are significant benefits in using GenomeData/GenomeDataList over GRanges/GRangesList, or if you have questions or concerns about this.

This function accepts one or more objects that are reduced, with a user-specified function, to a single GenomeData instance.

Usage

```
gdReduce(f, ..., init, right = FALSE, accumulate = FALSE, gdArgs = list())
```

Arguments

f	An object of class "function", accepting two instances of classes appropriate for the ... arguments, and returning an object suitable for subsequent use in f and incorporation into GenomeData.
...	Objects to be reduced. All objects should be of the same class, as dictated by methods defined on gdReduce A function to be applied to each chromosome-level sub-element of X.
init	An R object of the same kind as the elements of ....
right	A logical indicating whether to proceed from left to right (default) or right to left.
accumulate	A logical indicating whether the successive reduce combinations should be accumulated. By default, only the final combination is used.
gdArgs	Additional arguments passed to the GenomeData constructor used to assemble the final object.

Details

The gdReduce method for GenomeData objects successively combines GenomeData elements of ... using f; all arguments assigned to ... must be of class GenomeData. f is a function accepting two objects returned by "[[" applied to the successive elements of ..., returning a single GenomeData object to be used in subsequent calls to f. init, right, and accumulate are as described for Reduce. gdArgs can be used to provide metadata information to the constructor used to create the final GenomeData object.

Currently the gdReduce method for GenomeDataList objects works when a single GenomeDataList object x is provided as ... and it does gdReduce(f, x[[1]], x[[2]] ... x[[N]], init, right, accumulate, gdArgs where N is the length of x i.e. the number of GenomeData objects in it.

Value

An object of class GenomeData, containing elements corresponding to the intersection of all named elements of ... (in the case of the method for GenomeData objects) or all elements in the single GenomeDataList object passed to it (in the case of the method for GenomeDataList objects).

Author(s)

Martin Morgan

See Also

Reduce, GenomeData-class, GenomeDataList-class

Examples

```
## Not run:
gdReduce
showMethods("gdReduce")

gd <- GenomeData(list(chr1 = IRanges(1, 10), chrX = IRanges(2, 5)),
                 organism = "Mmusculus", provider = "UCSC",
                 providerVersion = "mm9")

gdr <- gdReduce(function(x, y) {
    ## "[[" returns IRanges instances, construct a synthetic version
    IRanges(c(start(x), start(y)), c(end(x), end(y)))
}, GenomeDataList(list(gd, gd[2])))
gdr[["chr1"]]
gdr[["chrX"]]

## End(Not run)
```

GenomeData-class *Data on the genome*

Description

WARNING: Starting with BioC 3.1, GenomeData and GenomeDataList objects are defunct. Note that the GenomeData/GenomeDataList containers predate the GRanges/GRangesList containers and, most of the times, the latters can be used instead of the formers. Please let us know on the bioc-devel mailing list (http://bioconductor.org/help/mailing-list/) if you have a use case where you think there are significant benefits in using GenomeData/GenomeDataList over GRanges/GRangesList, or if you have questions or concerns about this.

GenomeData formally represents genomic data as a list, with one element per chromosome in the genome.

Details

This class facilitates storing data on the genome by formalizing a set of metadata fields for storing the organism (e.g. Mmusculus), genome build provider (e.g. UCSC), and genome build version (e.g. mm9).

The data is represented as a list, with one element per chromosome (or really any sequence, like a gene). There are no constraints as to the data type of the elements.

Note that as a SimpleList, it is possible to store chromosome-level data (e.g. the lengths) in the elementMetadata slot. The organism, provider and providerVersion are all stored in the SimpleList metadata, so they may be retrieved in list form by calling metadata(x).

Accessor methods

In the code snippets below, object and x are GenomeData objects.

organism(object): Get the single string indicating the organism, if specified, otherwise NULL.

provider(x): Get the single string indicating the genome build provider, if specified, otherwise NULL.

providerVersion(x): Get the single string indicating the genome build version, if specified, otherwise NULL.

Constructor

GenomeData(listData = list(), providerVersion = metadata[["providerVersion"]],
Creates a GenomeData with the elements from the listData parameter, a list. The other arguments correspond to the metadata fields, and, with the exception of elementMetadata, should all be either single strings or NULL (unspecified). Additional global metadata elements may be passed in metadata, in list-form, and via .... The elements in metadata are always overridden by the explicit arguments, like organism and those in .... elementMetadata should be an DataTable or NULL.

Coercion

`as(from, "data.frame")`: Coerces each subelement to a data frame, and binds them into a single data frame with an additional column indicating chromosome

`as(from, "RangesList")`: Coerces each subelement to a `Ranges` and combines them into a `RangesList` with the same names. The "universe" metadata property is set to the `providerVersion` of `from`.

`as(from, "RangedData")`: Coerces each subelement to a `RangedData` and combines them into a single `RangedData` with the same names. The "universe" metadata property is set to the `providerVersion` of `from`.

Author(s)

Michael Lawrence

See Also

The GRanges and GRangesList classes defined and documented in the **GenomicRanges** package.

GenomeDataList-class, a container for storing a list of GenomeData objects and useful e.g. for storing data on multiple samples.

SimpleList-class, the base of this class.

`gdapply` for applying a function to elements of a GenomeData object.

`gdReduce` for successively combining GenomeData objects into a single GenomeData objects.

Examples

```
## Not run:
gd <- GenomeData(list(chr1 = IRanges(1, 10), chrX = IRanges(2, 5)),
                 organism = "Mmusculus", provider = "UCSC",
                 providerVersion = "mm9")
organism(gd)
providerVersion(gd)
provider(gd)
gd[["chr1"]] # get data for chromsome 1

## End(Not run)
```

GenomeDataList-class *List of GenomeData objects*

Description

WARNING: Starting with BioC 3.1, GenomeData and GenomeDataList objects are defunct. Note that the GenomeData/GenomeDataList containers predate the GRanges/GRangesList containers and, most of the times, the latters can be used instead of the formers. Please let us know on the bioc-devel mailing list (http://bioconductor.org/help/mailing-list/) if you have a use

case where you think there are significant benefits in using GenomeData/GenomeDataList over GRanges/GRangesList, or if you have questions or concerns about this.

GenomeDataList is a list of GenomeData objects. It could be useful for storing data on multiple experiments or samples.

Details

This class inherits from SimpleList and requires that all of its elements to be instances of GenomeData.

One should try to take advantage of the metadata storage facilities provided by SimpleList. The elementMetadata field, for example, could be used to store the experimental design, while the metadata field could store the experimental platform.

Constructor

GenomeDataList(listData = list(), metadata = list(), elementMetadata = NULL): Creates a GenomeDataList with the elements from the listData parameter, a list of GenomeData instances. The other arguments correspond to the optional metadata stored in SimpleList.

Coercion

as(from, "data.frame"): Coerces each subelement to a data frame, and binds them into a single data frame with an additional column indicating chromosome

Author(s)

Michael Lawrence

See Also

The GRanges and GRangesList classes defined and documented in the **GenomicRanges** package.

GenomeData, the type of elements stored in this class.

SimpleList

Examples

```
## Not run:
gd <- GenomeData(list(chr1 = IRanges(1, 10), chrX = IRanges(2, 5)),
                 organism = "Mmusculus", provider = "UCSC",
                 providerVersion = "mm9")
gdl <- GenomeDataList(list(gd), elementMetadata = DataFrame(induced = TRUE))
gdl[[1]] # get first element

## End(Not run)
```

getSeq-methods *getSeq method for BSgenome objects*

Description

A getSeq method for extracting a set of sequences (or subsequences) from a BSgenome object.

Usage

```
## S4 method for signature 'BSgenome'
getSeq(x, names, start=NA, end=NA, width=NA,
                 strand="+", as.character=FALSE)
```

Arguments

x	A BSgenome object. See the available.genomes function for how to install a genome.
names	A character vector containing the names of the sequences in x where to get the subsequences from, or a GRanges object, or a GRangesList object, or a named RangesList object, or a named Ranges object. The RangesList or Ranges object must be named according to the sequences in x where to get the subsequences from. If names is missing, then seqnames(x) is used. See ?`BSgenome-class` for details on how to get the lists of single sequences and multiple sequences (respectively) contained in a BSgenome object.
start, end, width	Vector of integers (eventually with NAs) specifying the locations of the subsequences to extract. These are not needed (and it's an error to supply them) when names is a GRanges, GRangesList, RangesList, or Ranges object.
strand	A vector containing "+"s or/and "-"s. This is not needed (and it's an error to supply it) when names is a GRanges or GRangesList object.
as.character	TRUE or FALSE. Should the extracted sequences be returned in a standard character vector?
...	Additional arguments. (Currently ignored.)

Details

L, the number of sequences to extract, is determined as follow:

- If names is a GRanges or Ranges object then L = length(names).
- If names is a GRangesList or RangesList object then L = length(unlist(names)).
- Otherwise, L is the length of the longest of names, start, end and width and all these arguments are recycled to this length. NAs and negative values in these 3 arguments are solved according to the rules of the SEW (Start/End/Width) interface (see ?solveUserSEW for the details).

If names is neither a GRanges or GRangesList object, then the strand argument is also recycled to length L.

Here is how the names passed to the names argument are matched to the names of the sequences in BSgenome object x. For each name in names:

- (1): If x contains a single sequence with that name then this sequence is used for extraction;
- (2): Otherwise the names of all the elements in all the multiple sequences are searched. If the names argument is a character vector then name is treated as a regular expression and grep is used for this search, otherwise (i.e. when the names are supplied via a higher level object like GRanges or GRangesList) then name must match exactly the name of the sequence. If exactly 1 sequence is found, then it is used for extraction, otherwise (i.e. if no sequence or more than 1 sequence is found) then an error is raised.

Value

Normally a DNAStringSet object (or character vector if as.character=TRUE).

With the 2 following exceptions:

1. A DNAStringSetList object (or CharacterList object if as.character=TRUE) of the same shape as names if names is a GRangesList object.
2. A DNAString object (or single character string if as.character=TRUE) if L = 1 and names is not a GRanges, GRangesList, RangesList, or Ranges object.

Note

Be aware that using as.character=TRUE can be very inefficient when extracting a "big" amount of DNA sequences (e.g. millions of short sequences or a small number of very long sequences).

Note that the masks in x, if any, are always ignored. In other words, masked regions in the genome are extracted in the same way as unmasked regions (this is achieved by dropping the masks before extraction). See ?`MaskedDNAString-class` for more information about masked DNA sequences.

Author(s)

H. Pages; improvements suggested by Matt Settles and others

See Also

getSeq, available.genomes, BSgenome-class, DNAString-class, DNAStringSet-class, MaskedDNAString-class, GRanges-class, GRangesList-class, RangesList-class, Ranges-class, grep

Examples

```
## ---------------------------------------------------------------------
## A. SIMPLE EXAMPLES
## ---------------------------------------------------------------------

## Load the Caenorhabditis elegans genome (UCSC Release ce2):
library(BSgenome.Celegans.UCSC.ce2)
```

```
## Look at the index of sequences:
Celegans

## Get chromosome V as a DNAString object:
getSeq(Celegans, "chrV")
## which is in fact the same as doing:
Celegans$chrV

## Not run:
  ## Never try this:
  getSeq(Celegans, "chrV", as.character=TRUE)
  ## or this (even worse):
  getSeq(Celegans, as.character=TRUE)

## End(Not run)

## Get the first 20 bases of each chromosome:
getSeq(Celegans, end=20)

## Get the last 20 bases of each chromosome:
getSeq(Celegans, start=-20)

## ---------------------------------------------------------------------
## B. EXTRACTING SMALL SEQUENCES FROM DIFFERENT CHROMOSOMES
## ---------------------------------------------------------------------

myseqs <- data.frame(
  chr=c("chrI", "chrX", "chrM", "chrM", "chrX", "chrI", "chrM", "chrI"),
  start=c(NA, -40, 8510, 301, 30001, 9220500, -2804, -30),
  end=c(50, NA, 8522, 324, 30011, 9220555, -2801, -11),
  strand=c("+", "-", "+", "+", "-", "-", "+", "-")
)
getSeq(Celegans, myseqs$chr,
       start=myseqs$start, end=myseqs$end)
getSeq(Celegans, myseqs$chr,
       start=myseqs$start, end=myseqs$end, strand=myseqs$strand)

## ---------------------------------------------------------------------
## C. USING A GRanges OBJECT
## ---------------------------------------------------------------------

gr1 <- GRanges(seqnames=c("chrI", "chrI", "chrM"),
               ranges=IRanges(start=101:103, width=9))
gr1  # all strand values are "*"
getSeq(Celegans, gr1)  # treats strand values as if they were "+"

strand(gr1)[] <- "-"
getSeq(Celegans, gr1)

strand(gr1)[1] <- "+"
getSeq(Celegans, gr1)

strand(gr1)[2] <- "*"
```

```
if (interactive())
  getSeq(Celegans, gr1)  # Error: cannot mix "*" with other strand values

gr2 <- GRanges(seqnames=c("chrM", "NM_058280_up_1000"),
               ranges=IRanges(start=103:102, width=9))
gr2
if (interactive()) {
  ## Because the sequence names are supplied via a GRanges object, they
  ## are not treated as regular expressions:
  getSeq(Celegans, gr2)  # Error: sequence NM_058280_up_1000 not found
}

## ---------------------------------------------------------------------
## D. USING A GRangesList OBJECT
## ---------------------------------------------------------------------

gr1 <- GRanges(seqnames=c("chrI", "chrII", "chrM", "chrII"),
               ranges=IRanges(start=101:104, width=12),
               strand="+")
gr2 <- shift(gr1, 5)
gr3 <- gr2
strand(gr3) <- "-"

grl <- GRangesList(gr1, gr2, gr3)
getSeq(Celegans, grl)

## ---------------------------------------------------------------------
## E. EXTRACTING A HIGH NUMBER OF RANDOM 40-MERS FROM A GENOME
## ---------------------------------------------------------------------

extractRandomReads <- function(x, density, readlength)
{
    if (!is.integer(readlength))
        readlength <- as.integer(readlength)
    start <- lapply(seqnames(x),
                    function(name)
                    {
                      seqlength <- seqlengths(x)[name]
                      sample(seqlength - readlength + 1L,
                             seqlength * density,
                             replace=TRUE)
                    })
    names <- rep.int(seqnames(x), elementLengths(start))
    ranges <- IRanges(start=unlist(start), width=readlength)
    strand <- strand(sample(c("+", "-"), length(names), replace=TRUE))
    gr <- GRanges(seqnames=names, ranges=ranges, strand=strand)
    getSeq(x, gr)
}

## With a density of 1 read every 100 genome bases, the total number of
## extracted 40-mers is about 1 million:
rndreads <- extractRandomReads(Celegans, 0.01, 40)
```

```
## Notes:
## - The short sequences in 'rndreads' can be seen as the result of a
##   simulated high-throughput sequencing experiment. A non-realistic
##   one though because:
##     (a) It assumes that the underlying technology is perfect (the
##         generated reads have no technology induced errors).
##     (b) It assumes that the sequenced genome is exactly the same as
##         the reference genome.
##     (c) The simulated reads can contain IUPAC ambiguity letters only
##         because the reference genome contains them. In a real
##         high-throughput sequencing experiment, the sequenced genome
##         of course doesn't contain those letters, but the sequencer
##         can introduce them in the generated reads to indicate
##         ambiguous base-calling.
## - Those reads are coming from the plus and minus strands of the
##   chromosomes.
## - With a density of 0.01 and the reads being only 40-base long, the
##   average coverage of the genome is only 0.4 which is low. The total
##   number of reads is about 1 million and it takes less than 10 sec.
##   to generate them.
## - A higher coverage can be achieved by using a higher density and/or
##   longer reads. For example, with a density of 0.1 and 100-base reads
##   the average coverage is 10. The total number of reads is about 10
##   millions and it takes less than 1 minute to generate them.
## - Those reads could easily be mapped back to the reference by using
##   an efficient matching tool like matchPDict() for performing exact
##   matching (see ?matchPDict for more information). Typically, a
##   small percentage of the reads (4 to 5% in our case) will hit the
##   reference at multiple locations. This is especially true for such
##   short reads, and, in a lower proportion, is still true for longer
##   reads, even for reads as long as 300 bases.

## ---------------------------------------------------------------------
## F. SEE THE BSgenome CACHE IN ACTION
## ---------------------------------------------------------------------

options(verbose=TRUE)
first20 <- getSeq(Celegans, end=20)
first20
gc()
stopifnot(length(ls(Celegans@.seqs_cache)) == 0L)
## One more gc() call is needed in order to see the amount of memory in
## used after all the chromosomes have been removed from the cache:
gc()
```

`injectSNPs` *SNP injection*

Description

Inject SNPs from a SNPlocs data package into a genome.

Usage

```
injectSNPs(x, snps)

SNPlocs_pkgname(x)

## S4 method for signature 'BSgenome'
snpcount(x)
## S4 method for signature 'BSgenome'
snplocs(x, seqname, ...)

## Related utilities
available.SNPs(type=getOption("pkgType"))
installed.SNPs()
```

Arguments

x	A BSgenome object.
snps	A SNPlocs object or the name of a SNPlocs data package. This object or package must contain SNP information for the single sequences contained in x. If a package, it must be already installed (injectSNPs won't try to install it).
seqname	The name of a single sequence in x.
type	Character string indicating the type of package ("source", "mac.binary" or "win.binary") to look for.
...	Further arguments to be passed to snplocs method for SNPlocs objects.

Value

injectSNPs returns a copy of the original genome x where some or all of the single sequences from x are altered by injecting the SNPs stored in snps. The SNPs in the altered genome are represented by an IUPAC ambiguity code at each SNP location.

SNPlocs_pkgname, snpcount and snplocs return NULL if no SNPs were injected in x (i.e. if x is not a BSgenome object returned by a previous call to injectSNPs). Otherwise SNPlocs_pkgname returns the name of the package from which the SNPs were injected, snpcount the number of SNPs for each altered sequence in x, and snplocs their locations in the sequence whose name is specified by seqname.

available.SNPs returns a character vector containing the names of the SNPlocs and XtraSNPlocs data packages that are currently available on the Bioconductor repositories for your version of R/Bioconductor. A SNPlocs data package contains basic information (location and alleles) about the known molecular variations of class *snp* for a given organism. A XtraSNPlocs data package contains information about the known molecular variations of other classes (*in-del*, *heterozygous*, *microsatellite*, *named-locus*, *no-variation*, *mixed*, *multinucleotide-polymorphism*) for a given organism. Only SNPlocs data packages can be used for SNP injection for now.

installed.SNPs returns a character vector containing the names of the SNPlocs and XtraSNPlocs data packages that are already installed.

Note

injectSNPs, SNPlocs_pkgname, snpcount and snplocs have the side effect to try to load the SNPlocs data package that was specified thru the snps argument if it's not already loaded.

Author(s)

H. Pages

See Also

BSgenome-class, IUPAC_CODE_MAP, injectHardMask, letterFrequencyInSlidingView, .inplaceReplaceLetterAt

Examples

```
## What SNPlocs data packages are already installed:
installed.SNPs()

## What SNPlocs data packages are available:
available.SNPs()

if (interactive()) {
  ## Make your choice and install with:
  source("http://bioconductor.org/biocLite.R")
  biocLite("SNPlocs.Hsapiens.dbSNP141.GRCh38")
}

## Inject SNPs from dbSNP into the Human genome:
library(BSgenome.Hsapiens.UCSC.hg38.masked)
genome <- BSgenome.Hsapiens.UCSC.hg38.masked
SNPlocs_pkgname(genome)

genome2 <- injectSNPs(genome, "SNPlocs.Hsapiens.dbSNP141.GRCh38")
genome2  # note the extra "with SNPs injected from ..." line
SNPlocs_pkgname(genome2)
snpcount(genome2)
head(snplocs(genome2, "chr1"))

alphabetFrequency(genome$chr1)
alphabetFrequency(genome2$chr1)

## Find runs of SNPs of length at least 25 in chr1. Might require
## more memory than some platforms can handle (e.g. 32-bit Windows
## and maybe some Mac OS X machines with little memory):
is_32bit_windows <- .Platform$OS.type == "windows" &&
                    .Platform$r_arch == "i386"
is_macosx <- substr(R.version$os, start=1, stop=6) == "darwin"
if (!is_32bit_windows && !is_macosx) {
    chr1 <- injectHardMask(genome2$chr1)
    ambiguous_letters <- paste(DNA_ALPHABET[5:15], collapse="")
    lf <- letterFrequencyInSlidingView(chr1, 25, ambiguous_letters)
    sl <- slice(as.integer(lf), lower=25)
    v1 <- Views(chr1, start(sl), end(sl)+24)
```

```
    v1
    max(width(v1))  # length of longest SNP run
}
```

SNPlocs-class *SNPlocs objects*

Description

The SNPlocs class is a container for storing known SNP locations for a given organism. SNPlocs objects are usually made in advance by a volunteer and made available to the Bioconductor community as "SNPlocs data packages". See ?available.SNPs for how to get the list of "SNPlocs data packages" curently available.

This man page's main focus is on how to extract information from a SNPlocs object.

Usage

```
snpcount(x)

snpsBySeqname(x, seqnames, ...)
## S4 method for signature 'SNPlocs'
snpsBySeqname(x, seqnames, drop.rs.prefix=FALSE)

snpsByOverlaps(x, ranges, maxgap=0L, minoverlap=0L,
               type=c("any", "start", "end", "within", "equal"), ...)
## S4 method for signature 'SNPlocs'
snpsByOverlaps(x, ranges, maxgap=0L, minoverlap=0L,
               type=c("any", "start", "end", "within", "equal"),
               drop.rs.prefix=FALSE, ...)

snpsById(x, ids, ...)
## S4 method for signature 'SNPlocs'
snpsById(x, ids, ifnotfound=c("error", "warning", "drop"))

## Old API
## ------------------------------------

snplocs(x, seqname, ...)
## S4 method for signature 'SNPlocs'
snplocs(x, seqname, as.GRanges=FALSE, caching=TRUE)

snpid2loc(x, snpid, ...)
## S4 method for signature 'SNPlocs'
snpid2loc(x, snpid, caching=TRUE)

snpid2alleles(x, snpid, ...)
## S4 method for signature 'SNPlocs'
```

```
snpid2alleles(x, snpid, caching=TRUE)

snpid2grange(x, snpid, ...)
## S4 method for signature 'SNPlocs'
snpid2grange(x, snpid, caching=TRUE)
```

Arguments

x	A SNPlocs object.
seqnames	The names of the sequences for which to get SNPs. Must be a subset of seqlevels(x). NAs and duplicates are not allowed.
...	Additional arguments, for use in specific methods. Arguments passed to the snpsByOverlaps method for SNPlocs objects thru ... are passed to internal call to subsetByOverlaps().
drop.rs.prefix	Should the rs prefix be dropped from the returned RefSNP ids? (RefSNP ids are stored in the RefSNP_id metadata column of the returned object.)
ranges	One or more regions of interest specified as a GRanges object. A single region of interest can be specified as a character string of the form "ch14:5201-5300".
maxgap, minoverlap, type	These arguments are passed to subsetByOverlaps() which is used internally by snpsByOverlaps. See ?IRanges::subsetByOverlaps in the **IRanges** package and ?GenomicRanges::subsetByOverlaps in the **GenomicRanges** package for more information about the subsetByOverlaps() generic and its method for GenomicRanges objects.
ids, snpid	The RefSNP ids to look up (a.k.a. rs ids). Can be integer or character vector, with or without the "rs" prefix. NAs are not allowed.
ifnotfound	What to do if SNP ids are not found.
seqname	The name of the sequence for which to get the SNP locations and alleles. If as.GRanges is FALSE, only one sequence can be specified (i.e. seqname must be a single string). If as.GRanges is TRUE, an arbitrary number of sequences can be specified (i.e. seqname can be a character vector of arbitrary length).
as.GRanges	TRUE or FALSE. If TRUE, then the SNP locations and alleles are returned in a GRanges object. Otherwise (the default), they are returned in a data frame.
caching	Should the loaded SNPs be cached in memory for faster further retrieval but at the cost of increased memory usage?

Value

snpcount returns a named integer vector containing the number of SNPs for each sequence in the reference genome.

snpsBySeqname, snpsByOverlaps, and snpsById return a GRanges object with 1 element (genomic range) per SNP and the following metadata columns:

- RefSNP_id: RefSNP ID (aka "rs id"). Character vector with no NAs and no duplicates.

- `alleles_as_ambig`: A character vector with no NAs containing the alleles for each SNP represented by an IUPAC nucleotide ambiguity code. See ?`IUPAC_CODE_MAP` in the **Biostrings** package for more information.

Note that all the elements (genomic ranges) in this GRanges object have their strand set to `"+"`.

If `ifnotfound="error"`, the object returned by `snpsById` is guaranteed to be *parallel* to `ids`, that is, the i-th element in the GRanges object corresponds to the i-th element in `ids`.

Old API: Note that `snplocs` is superseded by `snpsBySeqname`, and `snpid2loc`, `snpid2alleles`, and `snpid2grange` are superseded by `snpsById`.

By default (i.e. when `as.GRanges=FALSE`), `snplocs` returns a data frame with 1 row per SNP and the following columns:

1. `RefSNP_id`: Same as above but with `"rs"` prefix always removed.
2. `alleles_as_ambig`: Same as above.
3. `loc`: The 1-based location of the SNP relative to the first base at the 5' end of the plus strand of the reference sequence.

Otherwise (i.e. when `as.GRanges=TRUE`), it returns a GRanges object with metadata columns `"RefSNP_id"` and `"alleles_as_ambig"`.

`snpid2loc` and `snpid2alleles` both return a named vector (integer vector for the former, character vector for the latter) where each (name, value) pair corresponds to a supplied SNP id. For both functions the name in (name, value) is the chromosome of the SNP id. The value in (name, value) is the position of the SNP id on the chromosome for `snpid2loc`, and a single IUPAC code representing the associated alleles for `snpid2alleles`.

`snpid2grange` returns a GRanges object similar to the one returned by `snplocs` (when used with `as.GRanges=TRUE`) and where each element corresponds to a supplied SNP id.

Author(s)

H. Pages

See Also

- `available.SNPs`
- `injectSNPs`
- `IUPAC_CODE_MAP` in the **Biostrings** package.

Examples

```
library(SNPlocs.Hsapiens.dbSNP141.GRCh38)
snps <- SNPlocs.Hsapiens.dbSNP141.GRCh38
snpcount(snps)

## ---------------------------------------------------------------------
## snpsBySeqname()
## ---------------------------------------------------------------------
## Get all SNPs located on chromosome 22 and MT:
snpsBySeqname(snps, c("ch22", "chMT"))
```

```
## ---------------------------------------------------------------------
## snpsByOverlaps()
## ---------------------------------------------------------------------
## Get all SNPs overlapping some regions of interest:
snpsByOverlaps(snps, "ch22:33.63e6-33.64e6")

## With the regions of interest being all the known CDS for hg38
## located on chr22 or chrMT (except for the chromosome naming
## convention, hg38 is the same as GRCh38):
library(TxDb.Hsapiens.UCSC.hg38.knownGene)
txdb <- TxDb.Hsapiens.UCSC.hg38.knownGene
my_cds <- cds(txdb)
seqlevels(my_cds, force=TRUE) <- c("chr22", "chrMT")
seqlevelsStyle(my_cds)  # UCSC
seqlevelsStyle(snps)  # dbSNP
seqlevelsStyle(my_cds) <- seqlevelsStyle(snps)
genome(my_cds) <- genome(snps)
snpsByOverlaps(snps, my_cds)

## ---------------------------------------------------------------------
## snpsById()
## ---------------------------------------------------------------------
## Lookup some RefSNP ids:
my_rsids <- c("rs10458597", "rs12565286", "rs7553394")
## Not run:
  snpsById(snps, my_rsids)  # error, rs7553394 not found

## End(Not run)
snpsById(snps, my_rsids, ifnotfound="drop")
```

`XtraSNPlocs-class` *XtraSNPlocs objects*

Description

The XtraSNPlocs class is a container for storing extra SNP locations and alleles for a given organism. While a SNPlocs object can store only molecular variations of class *snp*, an XtraSNPlocs object contains molecular variations of other classes (*in-del*, *heterozygous*, *microsatellite*, *named-locus*, *no-variation*, *mixed*, *multinucleotide-polymorphism*).

XtraSNPlocs objects are usually made in advance by a volunteer and made available to the Bioconductor community as *XtraSNPlocs data packages*. See ?`available.SNPs` for how to get the list of *SNPlocs and XtraSNPlocs data packages* curently available.

This man page's main focus is on how to extract data from an XtraSNPlocs object.

Usage

```
## S4 method for signature 'XtraSNPlocs'
snpcount(x)
```

```
## S4 method for signature 'XtraSNPlocs'
snpsBySeqname(x, seqnames,
              columns=c("seqnames", "start", "end", "strand", "RefSNP_id"),
              drop.rs.prefix=FALSE,
              as.DataFrame=FALSE)

## S4 method for signature 'XtraSNPlocs'
snpsByOverlaps(x, ranges, maxgap=0L, minoverlap=0L,
               type=c("any", "start", "end", "within", "equal"),
               columns=c("seqnames", "start", "end", "strand", "RefSNP_id"),
               drop.rs.prefix=FALSE, as.DataFrame=FALSE, ...)

## S4 method for signature 'XtraSNPlocs'
snpsById(x, ids,
         columns=c("seqnames", "start", "end", "strand", "RefSNP_id"),
         ifnotfound=c("error", "warning", "drop"),
         as.DataFrame=FALSE)

## S4 method for signature 'XtraSNPlocs'
colnames(x, do.NULL=TRUE, prefix="col")
```

Arguments

x	An XtraSNPlocs object.
seqnames	The names of the sequences for which to get SNPs. NAs and duplicates are not allowed. The supplied seqnames must be a subset of seqlevels(x).
columns	The names of the columns to return. Valid column names are: seqnames, start, end, width, strand, RefSNP_id, alleles, snpClass, loctype. See Details section below for a description of these columns.
drop.rs.prefix	Should the rs prefix be dropped from the returned RefSNP ids? (RefSNP ids are stored in the RefSNP_id metadata column of the returned object.)
as.DataFrame	Should the result be returned in a DataFrame instead of a GRanges object?
ranges	One or more regions of interest specified as a GRanges object. A single region of interest can be specified as a character string of the form "ch14:5201-5300".
maxgap, minoverlap, type	These arguments are passed to subsetByOverlaps() which is used internally by snpsByOverlaps. See ?IRanges::subsetByOverlaps in the **IRanges** package and ?GenomicRanges::subsetByOverlaps in the **GenomicRanges** package for more information about the subsetByOverlaps() generic and its method for GenomicRanges objects.
ids	The RefSNP ids to look up (a.k.a. *rs ids*). Can be integer or character vector, with or without the "rs" prefix. NAs are not allowed.
ifnotfound	What to do if SNP ids are not found.
...	Additional arguments, for use in specific methods. Further arguments passed to the snpsByOverlaps method for XtraSNPlocs objects (thru ...) are passed to subsetByOverlaps().

`do.NULL, prefix`
: These arguments are ignored.

Value

`snpcount` returns a named integer vector containing the number of SNPs for each chromosome in the reference genome.

`snpsBySeqname` and `snpsById` both return a GRanges object with 1 element per SNP, unless `as.DataFrame` is set to `TRUE` in which case they return a DataFrame with 1 row per SNP. When a GRanges object is returned, the columns requested via the `columns` argument are stored as metada columns of the object, except for the following columns: `seqnames`, `start`, `end`, `width`, and `strand`. These "spatial columns" (in the sense that they describe the genomic locations of the SNPs) can be accessed by calling the corresponding getter on the GRanges object.

Summary of available columns (`my_snps` being the returned object):

- `seqnames`: The name of the chromosome where each SNP is located. Access with `seqnames(my_snps)` when `my_snps` is a GRanges object.
- `start` and `end`: The starting and ending coordinates of each SNP with respect to the chromosome indicated in `seqnames`. Coordinated are 1-based and with respect to the 5' end of the plus strand of the chromosome in the reference genome. Access with `start(my_snps)`, `end(my_snps)`, or `ranges(my_snps)` when `my_snps` is a GRanges object.
- `width`: The number of nucleotides spanned by each SNP *on the reference genome* (e.g. a width of 0 means the SNP is an insertion). Access with `width( my_snps)` when `my_snps` is a GRanges object.
- `strand`: The strand that the alleles of each SNP was reported to. Access with `strand(my_snps)` when `my_snps` is a GRanges object.
- `RefSNP_id`: The RefSNP id (a.k.a. *rs id*) of each SNP. Access with `mcols(my_snps)$RefSNP_id` when `my_snps` is a GRanges object.
- `alleles`: The alleles of each SNP in the format used by dbSNP. Access with `mcols(my_snps)$alleles` when `my_snps` is a GRanges object.
- `snpClass`: Class of each SNP. Possible values are `in-del`, `heterozygous`, `microsatellite`, `named-locus`, `no-variation`, `mixed`, and `multinucleotide-polymorphism`. Access with `mcols(my_snps)$snpClass` when `my_snps` is a GRanges object.
- `loctype`: See `ftp://ftp.ncbi.nih.gov/snp/00readme.txt` for the 6 loctype codes used by dbSNP, and their meanings. WARNING: The code assigned to each SNP doesn't seem to be reliable. For example, loctype codes 1 and 3 officially stand for insertion and deletion, respectively. However, when looking at the SNP ranges it actually seems to be the other way around. Access with `mcols(my_snps)$loctype` when `my_snps` is a GRanges object.

`colnames(x)` returns the names of the available columns.

Author(s)

H. Pages

See Also

- available.SNPs
- SNPlocs objects.

Examples

```
library(XtraSNPlocs.Hsapiens.dbSNP141.GRCh38)
snps <- XtraSNPlocs.Hsapiens.dbSNP141.GRCh38
snpcount(snps)
colnames(snps)

## ---------------------------------------------------------------------
## snpsBySeqname()
## ---------------------------------------------------------------------
## Get the location, RefSNP id, and alleles for all "extra SNPs"
## located on chromosome 22 and MT:
snpsBySeqname(snps, c("ch22", "chMT"), columns=c("RefSNP_id", "alleles"))

## ---------------------------------------------------------------------
## snpsByOverlaps()
## ---------------------------------------------------------------------
## Get the location, RefSNP id, and alleles for all "extra SNPs"
## overlapping some regions of interest:
snpsByOverlaps(snps, "ch22:33.63e6-33.64e6",
               columns=c("RefSNP_id", "alleles"))

## With the regions of interest being all the known CDS for hg38
## (except for the chromosome naming convention, hg38 is the same
## as GRCh38):
library(TxDb.Hsapiens.UCSC.hg38.knownGene)
txdb <- TxDb.Hsapiens.UCSC.hg38.knownGene
hg38_cds <- cds(txdb)
seqlevelsStyle(hg38_cds)  # UCSC
seqlevelsStyle(snps)  # dbSNP
seqlevelsStyle(hg38_cds) <- seqlevelsStyle(snps)
genome(hg38_cds) <- genome(snps)
snpsByOverlaps(snps, hg38_cds, columns=c("RefSNP_id", "alleles"))

## ---------------------------------------------------------------------
## snpsById()
## ---------------------------------------------------------------------
## Get the location and alleles for some RefSNP ids:
my_rsids <- c("rs367617508", "rs398104919", "rs3831697", "rs372470289",
              "rs141568169", "rs34628976", "rs67551854")
snpsById(snps, my_rsids, c("RefSNP_id", "alleles"))

## See ?XtraSNPlocs.Hsapiens.dbSNP141.GRCh38 for examples of using
## snpsBySeqname() and snpsById().
```

Index