Package ‘plyranges’

May 16, 2025

Type Package

Title A fluent interface for manipulating GenomicRanges

Version 1.29.0

Maintainer Michael Love <michaelisaiahlove@gmail.com>

Description A dplyr-like interface for interacting with the common Bioconductor classes Ranges and GenomicRanges. By providing a grammatical and consistent way of manipulating these classes their accessiblity for new Bioconductor users is hopefully increased.

Depends R (>= 3.5), BiocGenerics, IRanges (>= 2.12.0), GenomicRanges (>= 1.28.4)

Imports methods, dplyr, rlang (>= 0.2.0), magrittr, tidyselect (>= 1.0.0), rtracklayer, GenomicAlignments, GenomeInfoDb, Rsamtools, S4Vectors (>= 0.23.10), utils

biocViews Infrastructure, DataRepresentation, WorkflowStep, Coverage

BugReports https://github.com/tidyomics/plyranges

License Artistic-2.0

Encoding UTF-8

ByteCompile true

Suggests knitr, BiocStyle, rmarkdown, testthat (>= 2.1.0), HelloRanges, HelloRangesData, BSgenome.Hsapiens.UCSC.hg19, pasillaBamSubset, covr, ggplot2

VignetteBuilder knitr

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Collate 'class-AnchoredRanges.R' 'class-Operator.R' 'class-DeferredGenomicRanges.R' 'class-GroupedRanges.R' 'dplyr-arrange.R' 'dplyr-filter.R' 'dplyr-groups.R' 'dplyr-mutate.R' 'dplyr-select.R' 'dplyr-slice.R' 'dplyr-summarize.R' 'endo-coverage.R' 'endo-tile.R' 'io-bam.R' 'io-bed.R' 'io-bigwig.R' 'io-gff.R' 'io-wig.R' 'methods-DeferredGenomicRanges.R' 'methods-Operator.R' 'plyranges.R' 'ranges-add-distance.R' 'ranges-anchors.R' 'ranges-arithmetic-flank.R' 'ranges-arithmetic-setters.R' 'ranges-arithmetic-shift.R' 'ranges-arithmetic-stretch.R' 'ranges-bind.R' 'ranges-chop.R' 'ranges-colwise.R' 'ranges-construct.R' 'ranges-disjoin.R' 'ranges-eval-quo.R' 'ranges-eval.R' 'ranges-expand.R' 'ranges-genomeinfo.R' 'ranges-join-follow.R' 'ranges-join-nearest.R' 'ranges-join-precede.R' 'ranges-overlap-count.R' 'ranges-overlap-filter.R' 'ranges-overlap-find.R' 'ranges-overlap-groups.R' 'ranges-overlap-joins-intersect.R' 'ranges-overlap-joins-outer.R' 'ranges-overlap-self-joins.R' 'ranges-pairs.R' 'ranges-rangewise-setops.R' 'ranges-reduce.R' 'ranges-setops.R' 'reexports.R'

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

git_branch devel

git_last_commit e3aa4b8

git_last_commit_date 2025-04-15

Repository Bioconductor 3.22

Date/Publication 2025-05-15

Author Stuart Lee [aut] (ORCID: <https://orcid.org/0000-0003-1179-8436>),
Michael Lawrence [aut, ctb],
Dianne Cook [aut, ctb],
Spencer Nystrom [ctb] (ORCID: <https://orcid.org/0000-0003-1000-1579>),
Pierre-Paul Axisa [ctb],
Michael Love [ctb, cre]

Contents

`plyranges-package` *plyranges: a grammar of genomic data manipulation*

Description

plyranges is a dplyr like API to the Ranges/GenomicRanges infrastructure in Bioconductor.

Details

plryanges provides a consistent interface for importing and wrangling genomics data from a variety of sources. The package defines a grammar of genomic data manipulation through a set of verbs. These verbs can be used to construct human readable analysis pipelines based on Ranges objects.

- Modify genomic regions with the `set_width()` and `stretch()` functions.
- Modify genomic regions while fixing the start/end/center coordinates with the `anchors()` family of functions.
- Sort genomic ranges with `arrange()`.
- Modify, subset, and aggregate genomic data with the `mutate()`, `filter()`, and `summarise()`functions.
- Any of the above operations can be performed on partitions of the data with `group_by()`.
- Find nearest neighbour genomic regions with the `join_nearest()` family of functions.
- Find overlaps between ranges with the `join_overlap_inner()` family of functions.

- Merge all overlapping and adjacent genomic regions with `reduce_ranges()`.
- Merge the end points of all genomic regions with `disjoin_ranges()`.
- Import and write common genomic data formats with the `read_/write_` family of functions.

For more details on the features of plryanges, read the vignette: `browseVignettes(package = "plyranges")`

Author(s)

Maintainer: Stuart Lee `<stuart.andrew.lee@gmail.com>` (ORCID)

Authors:

- Michael Lawrence [contributor]
- Dianne Cook [contributor]

Other contributors:

- Spencer Nystrom (ORCID) [contributor]

See Also

Useful links:

- Report bugs at `https://github.com/sa-lee/plyranges`

`add_nearest_distance` *Add distance to nearest neighbours between two Ranges objects*

Description

Appends distance to nearest subject range to query ranges similar to setting `distance` in `join_nearest_`. Distance is set to `NA` for features with no nearest feature by the selected nearest metric.

Usage

```
add_nearest_distance(x, y = x, name = "distance")

add_nearest_distance_left(x, y = x, name = "distance")

add_nearest_distance_right(x, y = x, name = "distance")

add_nearest_distance_upstream(x, y = x, name = "distance")

add_nearest_distance_downstream(x, y = x, name = "distance")
```

Arguments

`x`	The query ranges
`y`	the subject ranges within which the nearest ranges are found. If missing, query ranges are used as the subject.
`name`	column name to create containing distance values

Details

By default add_nearest_distance will find arbitrary nearest neighbours in either direction and ignore any strand information. The add_nearest_distance_left and add_nearest_distance_right methods will find arbitrary nearest neighbour ranges on x that are left/right of those on y and ignore any strand information.

The add_nearest_distance_upstream method will find arbitrary nearest neighbour ranges on x that are upstream of those on y. This takes into account strandedness of the ranges. On the positive strand nearest upstream will be on the left and on the negative strand nearest upstream will be on the right.

The add_nearest_distance_downstream method will find arbitrary nearest neighbour ranges on x that are upstream of those on y. This takes into account strandedness of the ranges. On the positive strand nearest downstream will be on the right and on the negative strand nearest upstream will be on the left.

Value

ranges in x with additional column containing the distance to the nearest range in y.

See Also

join_nearest

Examples

```
query <- data.frame(start = c(5,10, 15,20),
                   width = 5,
                   gc = runif(4)) %>%
             as_iranges()
subject <- data.frame(start = c(2:6, 24),
                      width = 3:8,
                      label = letters[1:6]) %>%
             as_iranges()

add_nearest_distance(query, subject)
add_nearest_distance_left(query, subject)
add_nearest_distance_left(query)
```

---

anchor *Anchored Ranges objects*

Description

The GRangesAnchored class and the IRangesAnchored class allow components of a GRanges or IRanges (start, end, center) to be held fixed.

Usage

```
anchor(x)

unanchor(x)
```

```
anchor_start(x)

anchor_end(x)

anchor_center(x)

anchor_centre(x)

anchor_3p(x)

anchor_5p(x)
```

Arguments

`x`	a Ranges object

Details

Anchoring will fix a Ranges start, end, or center positions, so these positions will remain the same when performing arithimetic. For `GRanges` objects, the function (`anchor_3p()`) will fix the start for the negative strand, while `anchor_5p()` will fix the end for the positive strand. Anchoring modifies how arithmetic is performed, for example modifying the width of a range with `set_width()` or stretching a range with `stretch()`. To remove anchoring use `unanchor()`.

Value

a RangesAnchored object which has the same appearance as a regular Ranges object but with an additional slot displaying an anchor.

Constructors

Depending on how you want to fix the components of a Ranges, there are five ways to construct a RangesAnchored class. Here `x` is either an `IRanges` or `GRanges` object.

- `anchor_start(x)`Fix the start coordinates
- `anchor_end(x)`Fix the end coordinates
- `anchor_center(x)`Fix the center coordinates
- `anchor_3p(x)`On the negative strand fix the start coordinates, and for positive or unstranded ranges fix the end coordinates.
- `anchor_5p(x)`On the positive or unstranded ranges fix the start coordinates, coordinates and for negative stranded ranges fix the end coordinates.

Accessors

To see what has been anchored use the function `anchor`. This will return a character vector containing a valid anchor. It will be set to one of `c("start", "end", "center")` for an `IRanges` object or one of `c("start", "end", "center", "3p", "5p")` for a `GRanges` object.

See Also

mutate, stretch

Examples

```
df <- data.frame(start = 1:10, width = 5)
rng <- as_iranges(df)
rng_by_start <- anchor_start(rng)
rng_by_start
anchor(rng_by_start)
mutate(rng_by_start, width = 3L)
grng <- as_granges(df,
                   seqnames = "chr1",
                   strand = c(rep("-", 5), rep("+", 5)))
rng_by_5p <- anchor_5p(grng)
rng_by_5p
mutate(rng_by_5p, width = 3L)
```

`arrange.Ranges` *Sort a Ranges object*

Description

Sort a Ranges object

Usage

```
## S3 method for class 'Ranges'
arrange(.data, ...)
```

Arguments

`.data`	A Ranges object.
`...`	Comma seperated list of variable names.

Value

A sorted Ranges object

Examples

```
rng <- as_iranges(data.frame(start = 1:10, width = 10:1))
rng <- mutate(rng, score = runif(10))
arrange(rng, score)
# you can also use dplyr::desc to arrange by descending order
```

as_iranges *Construct a I/GRanges object from a tibble or data.frame*

Description

The as_i(g)ranges function looks for column names in .data called start, end, width, seqnames and strand in order to construct an IRanges or GRanges object. By default other columns in .data are placed into the mcols (metadata columns) slot of the returned object.

Usage

```
as_iranges(.data, ..., keep_mcols = TRUE)

as_granges(.data, ..., keep_mcols = TRUE)
```

Arguments

`.data`	a `data.frame()` or `tibble()` to construct a Ranges object from
`...`	optional named arguments specifying which the columns in .data containin the core components a Ranges object.
`keep_mcols`	place the remaining columns into the metadata columns slot (default=TRUE)

Value

a Ranges object.

See Also

`IRanges::IRanges()`, `GenomicRanges::GRanges()`

Examples

```
df <- data.frame(start=c(2:-1, 13:15), width=c(0:3, 2:0))
as_iranges(df)

df <- data.frame(start=c(2:-1, 13:15), width=c(0:3, 2:0), strand = "+")
# will return an IRanges object
as_iranges(df)

df <- data.frame(start=c(2:-1, 13:15), width=c(0:3, 2:0),
strand = "+", seqnames = "chr1")
as_granges(df)

# as_g/iranges understand alternate name specification
df <- data.frame(start=c(2:-1, 13:15), width=c(0:3, 2:0),
strand = "+", chr = "chr1")
as_granges(df, seqnames = chr)

# can also handle DFrame input
df <- methods::as(df, "DFrame")
df$y <- IRanges::IntegerList(c(1,2,3), NA, 5, 6, 8, 9, 10:12)
as_iranges(df)
as_granges(df, seqnames = chr)
```

as_ranges — *Coerce an Rle or RleList object to Ranges*

Description

Coerce an Rle or RleList object to Ranges

Usage

```
as_ranges(.data)
```

Arguments

`.data`	a `Rle()` or an `RleList()` object.

Details

This function is behind `compute_coverage()`.

Value

an `IRanges()` object if the input is an `Rle()` object or a `GRanges()` object for an `RleList()` object.

See Also

`S4Vectors::Rle()`, `IRanges::RleList()`

Examples

```
x <- S4Vectors::Rle(10:1, 1:10)
as_ranges(x)

# must have names set
y <- IRanges::RleList(chr1 = x)
as_ranges(y)
```

bind_ranges — *Combine Ranges by concatentating them together*

Description

Combine Ranges by concatentating them together

Usage

```
bind_ranges(..., .id = NULL)
```

Arguments

`...`	Ranges objects to combine. Each argument can be a Ranges object, or a list of Ranges objects.
`.id`	Ranges object identifier. When .id is supplied a new column is created that links each row to the original Range object. The contents of the column correspond to the named arguments or the names of the list supplied.

Value

a concatenated Ranges object

Note

Currently GRangesList or IRangesList objects are not supported.

Examples

```
gr <- as_granges(data.frame(start = 10:15,
                            width = 5,
                            seqnames = "seq1"))
gr2 <- as_granges(data.frame(start = 11:14,
                            width = 1:4,
                            seqnames = "seq2"))
bind_ranges(gr, gr2)

bind_ranges(a = gr, b = gr2, .id = "origin")

bind_ranges(gr, list(gr, gr2), gr2)

bind_ranges(list(a = gr, b = gr2), c = gr, .id = "origin")
```

`chop_by_introns` *Group a GRanges object by introns or gaps*

Description

Group a GRanges object by introns or gaps

Usage

```
chop_by_introns(x)

chop_by_gaps(x)
```

Arguments

`x`	a GenomicRanges object with a cigar string column

Details

Creates a grouped Ranges object from a cigar string column, for `chop_by_introns()` will check for the presence of "N" in the cigar string and create a new column called `intron` where TRUE indicates the alignment has a skipped region from the reference. For `chop_by_gaps()` will check for the presence of "N" or "D" in the cigar string and create a new column called "gaps" where TRUE indicates the alignment has a deletion from the reference or has an intron.

Value

a GRanges object

Examples

```
if (require(pasillaBamSubset)) {
   bamfile <- untreated1_chr4()
   # define a region of interest
   roi <- data.frame(seqnames = "chr4", start = 5e5, end = 7e5) %>%
            as_granges()
   # results in a grouped ranges object
   rng <- read_bam(bamfile) %>%
            filter_by_overlaps(roi) %>%
            chop_by_gaps()
   # to find ranges that have gaps use filter with `n()`
   rng %>% filter(n() >= 2)

}
```

`compute_coverage` *Compute coverage over a Ranges object*

Description

Compute coverage over a Ranges object

Usage

```
compute_coverage(x, shift, width, weight, ...)
```

Arguments

`x`	a Ranges object
`shift`	shift how much should each range in x be shifted by? (default = 0L)
`width`	width how long should the returned coverage score be? This must be either a positive integer or NULL (default = NULL)
`weight`	weight how much weight should be assigned to each range? Either an integer or numeric vector or a column in x. (default = 1L)
`...`	other optional parameters to pass to coverage

Value

An expanded Ranges object with a score column corresponding to the coverage value over that interval. Note that compute_coverage drops metadata associated with the orginal ranges.

See Also

`IRanges::coverage()`, `GenomicRanges::coverage()`

Examples

```
rng <- as_iranges(data.frame(start = 1:10, width = 5))
compute_coverage(rng)
compute_coverage(rng, shift = 14L)
compute_coverage(rng, width = 10L)
```

count_overlaps — *Count the number of overlaps between two Ranges objects*

Description

Count the number of overlaps between two Ranges objects

Usage

```
count_overlaps(x, y, maxgap, minoverlap)

## S3 method for class 'IntegerRanges'
count_overlaps(x, y, maxgap = -1L, minoverlap = 0L)

## S3 method for class 'GenomicRanges'
count_overlaps(x, y, maxgap = -1L, minoverlap = 0L)

count_overlaps_within(x, y, maxgap, minoverlap)

## S3 method for class 'IntegerRanges'
count_overlaps_within(x, y, maxgap = 0L, minoverlap = 1L)

## S3 method for class 'GenomicRanges'
count_overlaps_within(x, y, maxgap = 0L, minoverlap = 1L)

count_overlaps_directed(x, y, maxgap, minoverlap)

## S3 method for class 'GenomicRanges'
count_overlaps_directed(x, y, maxgap = -1L, minoverlap = 0L)

count_overlaps_within_directed(x, y, maxgap, minoverlap)

## S3 method for class 'GenomicRanges'
count_overlaps_within_directed(x, y, maxgap = -1L, minoverlap = 0L)
```

Arguments

`x, y`	Objects representing ranges
`maxgap, minoverlap`	The maximimum gap between intervals as an integer greater than or equal to zero. The minimum amount of overlap between intervals as an integer greater than zero, accounting for the maximum gap.

Value

An integer vector of same length as x.

Examples

```
query <- data.frame(start = c(5,10, 15,20), width = 5, gc = runif(4)) %>%
             as_iranges()
subject <- data.frame(start = 2:6, width = 3:7, label = letters[1:5]) %>%
             as_iranges()
query %>% mutate(n_olap = count_overlaps(., subject),
                 n_olap_within = count_overlaps_within(., subject))
```

`DeferredGenomicRanges-class`

DeferredGenomiRanges objects

Description

Enables deferred reading of files (currently only BAM files) by caching results after a plyranges verb is called.

Slots

`delegate` a GenomicRanges object to be cached

`ops` A FileOperator object

See Also

`read_bam()`

`disjoin_ranges` *Disjoin then aggregate a Ranges object*

Description

Disjoin then aggregate a Ranges object

Usage

```
disjoin_ranges(.data, ...)

disjoin_ranges_directed(.data, ...)
```

Arguments

`.data`	a Ranges object to disjoin
`...`	Name-value pairs of summary functions.

Value

a Ranges object that is now disjoint (no bases overlap).

Examples

```
df <- data.frame(start = 1:10, width = 5,  seqnames = "seq1",
strand = sample(c("+", "-", "*"), 10, replace = TRUE), gc = runif(10))
rng <- as_granges(df)
rng %>% disjoin_ranges()
rng %>% disjoin_ranges(gc = mean(gc))
rng %>% disjoin_ranges_directed(gc = mean(gc))
```

`expand_ranges` *Expand list-columns in a Ranges object*

Description

Expand list-columns in a Ranges object

Usage

```
expand_ranges(
  data,
  ...,
  .drop = FALSE,
  .id = NULL,
  .keep_empty = FALSE,
  .recursive = FALSE
)
```

Arguments

`data`	A Ranges object
`...`	list-column names to expand then unlist
`.drop`	Should additional list columns be dropped (default = FALSE)? By default `expand_ranges()` will keep other list columns even if they are nested.
`.id`	A character vector of length equal to number of list columns. If supplied will create new column(s) with name `.id` identifying the index of the list column (default = NULL).
`.keep_empty`	If a list-like column contains empty elements, should those elements be kept? (default = FALSE)
`.recursive`	If there are multiple list-columns, should the columns be treated as parallel? If FALSE each column will be unnested recursively, otherwise they are treated as parallel, that is each list column has identical lengths. (deafualt = FALSE)

Value

a GRanges object with expanded list columns

Examples

```
grng <- as_granges(data.frame(seqnames = "chr1", start = 20:23, width = 1000))
grng <- mutate(grng,
               exon_id = IntegerList(a = 1, b = c(4,5), c = 3, d = c(2,5))
               )
expand_ranges(grng)
expand_ranges(grng, .id = "name")

# empty list elements are not preserved by default
grng <- mutate(grng,
               exon_id = IntegerList(a = NULL, b = c(4,5), c= 3, d = c(2,5))
               )
expand_ranges(grng)
expand_ranges(grng, .keep_empty = TRUE)
expand_ranges(grng, .id = "name", .keep_empty = TRUE)
```

`FileOperator-class` *An abstract class to represent operations performed over a file*

Description

An abstract class to represent operations performed over a file

Details

This class is used internally by DeferredGenomicRanges objects. Currently, this class is only implemented for bam files (as a BamFileOperator) but will eventually be extended to the other avaialable readers.

`filter-ranges` *Subset a* `Ranges` *object*

Description

Subset a Ranges object

Usage

```
## S3 method for class 'Ranges'
filter(.data, ..., .preserve = FALSE)
```

Arguments

`.data`	A Ranges object
`...`	valid logical predictates to subset .data by. These are determined by variables in `.data`. If more than one condition is supplied, the conditions are combined with `&`. Only rows where the condition evaluates to `TRUE` are kept.
`.preserve`	when FALSE (the default) grouping structure is recalculated, TRUE is currently not implemented.

Details

For any Ranges objects `filter` can act on all core components of the class including start, end, width (for IRanges) or seqnames and strand (for GRanges) in addition to metadata columns. If the Ranges object is grouped, `filter` will act seperately on each parition of the data.

Value

a Ranges object

See Also

`dplyr::filter()`

Examples

```
set.seed(100)
df <- data.frame(start = 1:10,
                 width = 5,
                 seqnames = "seq1",
                 strand = sample(c("+", "-", "*"), 10, replace = TRUE),
                 gc = runif(10))

rng <- as_granges(df)

filter(rng, strand == "+")
filter(rng, gc > 0.5)

# multiple criteria
filter(rng, strand == "+" | start > 5)
filter(rng, strand == "+" & start > 5)
```

```
# multiple conditions are the same as and
filter(rng, strand == "+", start > 5)

# grouping acts on each subset of the data
rng %>%
  group_by(strand) %>%
  filter(gc > 0.5)
```

`filter_by_overlaps` *Filter by overlapping/non-overlapping ranges*

Description

Filter by overlapping/non-overlapping ranges

Usage

```
filter_by_overlaps(x, y, maxgap = -1L, minoverlap = 0L)

filter_by_non_overlaps(x, y, maxgap, minoverlap)

filter_by_overlaps_directed(x, y, maxgap = -1L, minoverlap = 0L)

filter_by_non_overlaps_directed(x, y, maxgap, minoverlap)
```

Arguments

`x, y`	Objects representing ranges
`maxgap`	The maximimum gap between intervals as a single integer greater than or equal to -1. If you modify this argument, `minoverlap` must be held fixed.
`minoverlap`	The minimum amount of overlap between intervals as a single integer greater than 0. If you modify this argument, `maxgap` must be held fixed.

Details

By default, `filter_by_overlaps` and `filter_by_non_overlaps` ignore strandedness for `GRanges()` objects. To perform stranded operations use `filter_by_overlaps_directed` and `filter_by_non_overlaps_directe` The argument `maxgap` is the maximum number of positions between two ranges for them to be considered overlapping. Here the default is set to be -1 as that is the the gap between two ranges that has its start or end strictly inside the other. The argugment `minoverlap` refers to the minimum number of positions overlapping between ranges, to consider there to be overlap.

Value

a Ranges object

See Also

`IRanges::subsetByOverlaps()`

Examples

```
df <- data.frame(seqnames = c("chr1", rep("chr2", 2),
                              rep("chr3", 3), rep("chr4", 4)),
                 start = 1:10,
                 width = 10:1,
                 strand = c("-", "+", "+", "*", "*", "+", "+", "+", "-", "-"),
                 name = letters[1:10])
query <- as_granges(df)

df2 <- data.frame(seqnames = c(rep("chr2", 2), rep("chr1", 3), "chr2"),
                  start = c(4,3,7,13,1,4),
                  width = c(6,6,3,3,3,9),
                  strand = c(rep("+", 3), rep("-", 3)))
subject <- as_granges(df2)

filter_by_overlaps(query, subject)

filter_by_overlaps_directed(query, subject)

filter_by_non_overlaps(query, subject)

filter_by_non_overlaps_directed(query, subject)
```

`find_overlaps` *Find overlap between two Ranges*

Description

Find overlap between two Ranges

Usage

```
find_overlaps(x, y, maxgap, minoverlap, suffix = c(".x", ".y"))

## S3 method for class 'IntegerRanges'
find_overlaps(x, y, maxgap = -1L, minoverlap = 0L, suffix = c(".x", ".y"))

## S3 method for class 'GenomicRanges'
find_overlaps(x, y, maxgap = -1L, minoverlap = 0L, suffix = c(".x", ".y"))

find_overlaps_within(x, y, maxgap, minoverlap, suffix = c(".x", ".y"))

## S3 method for class 'IntegerRanges'
find_overlaps_within(
  x,
  y,
  maxgap = -1L,
  minoverlap = 0L,
  suffix = c(".x", ".y")
)
```

```
## S3 method for class 'GenomicRanges'
find_overlaps_within(
  x,
  y,
  maxgap = -1L,
  minoverlap = 0L,
  suffix = c(".x", ".y")
)

find_overlaps_directed(x, y, maxgap, minoverlap, suffix = c(".x", ".y"))

## S3 method for class 'GenomicRanges'
find_overlaps_directed(
  x,
  y,
  maxgap = -1L,
  minoverlap = 0L,
  suffix = c(".x", ".y")
)

find_overlaps_within_directed(x, y, maxgap, minoverlap, suffix = c(".x", ".y"))

## S3 method for class 'GenomicRanges'
find_overlaps_within_directed(x, y, maxgap, minoverlap, suffix = c(".x", ".y"))

group_by_overlaps(x, y, maxgap, minoverlap)

## S3 method for class 'IntegerRanges'
group_by_overlaps(x, y, maxgap = -1L, minoverlap = 0L)

## S3 method for class 'GenomicRanges'
group_by_overlaps(x, y, maxgap = -1L, minoverlap = 0L)
```

Arguments

`x, y`	Objects representing ranges
`maxgap, minoverlap`	The maximimum gap between intervals as an integer greater than or equal to negative one. The minimum amount of overlap between intervals as an integer greater than zero, accounting for the maximum gap.
`suffix`	A character vector of length two used to identify metadata columns coming from x and y.

Details

`find_overlaps()` will search for any overlaps between ranges x and y and return a Ranges object of length equal to the number of times x overlaps y. This Ranges object will have additional metadata columns corresponding to the metadata columns in y. `find_overlaps_within()` is the same but will only search for overlaps within y. For GRanges objects strand is ignored, unless `find_overlaps_directed()` is used. If the Ranges objects have no metadata, one could use `group_by_overlaps()` to be able to identify the index of the input Range x that overlaps a Range in y. Alternatively, `pair_overlaps()` could be used to place the x ranges next to the range in y they overlap.

Value

A Ranges object with rows corresponding to the ranges in x that overlap y. In the case of `group_by_overlaps()`, returns a GroupedRanges object, grouped by the number of overlaps of ranges in x that overlap y (stored in a column called query).

See Also

`IRanges::findOverlaps()`, `GenomicRanges::findOverlaps()`

Examples

```
query <- data.frame(start = c(5,10, 15,20), width = 5, gc = runif(4)) %>%
             as_iranges()
subject <- data.frame(start = 2:6, width = 3:7, label = letters[1:5]) %>%
             as_iranges()

find_overlaps(query, subject)
find_overlaps(query, subject, minoverlap = 5)
find_overlaps_within(query, subject) # same result as minoverlap
find_overlaps(query, subject, maxgap = 1)

# -- GRanges objects, strand is ignored by default
query  <- data.frame(seqnames = "chr1",
               start = c(11,101),
               end = c(21, 200),
               name = c("a1", "a2"),
               strand = c("+", "-"),
               score = c(1,2)) %>%
           as_granges()
subject <- data.frame(seqnames = "chr1",
                      strand = c("+", "-", "+", "-"),
                      start = c(21,91,101,201),
                      end = c(30,101,110,210),
                      name = paste0("b", 1:4),
                      score = 1:4) %>%
                   as_granges()

# ignores strandedness
find_overlaps(query, subject, suffix = c(".query", ".subject"))
find_overlaps(query, subject, suffix = c(".query", ".subject"), minoverlap = 2)
# adding directed prefix includes strand
find_overlaps_directed(query, subject, suffix = c(".query", ".subject"))
```

`flank_left` *Generate flanking regions*

Description

Find flanking regions to the left or right or upstream or downstream of a Ranges object.

Usage

```
flank_left(x, width = 0L)

flank_right(x, width = 0L)

flank_upstream(x, width = 0L)

flank_downstream(x, width = 0L)
```

Arguments

`x`	a Ranges object.
`width`	the width of the flanking region relative to the ranges in `x`. Either an integer vector of length 1 or an integer vector the same length as x. The width can be negative in which case the flanking region is reversed.

Details

The function `flank_left` will create the flanking region to the left of starting coordinates in `x`, while `flank_right` will create the flanking region to the right of the starting coordinates in `x`. The function `flank_upstream` will `flank_left` if the strand of rows in `x` is not negative and will `flank_right` if the strand of rows in `x` is negative. The function `flank_downstream` will `flank_right` if the strand of rows in `x` is not negative and will `flank_leftt` if the strand of rows in `x` is negative.

By default `flank_left` and `flank_right` will ignore strandedness of any ranges, while `flank_upstream` and `flank_downstream` will take into account the strand of `x`.

Value

A Ranges object of same length as `x`.

See Also

`IRanges::flank()`, `GenomicRanges::flank()`

Examples

```
gr <- as_granges(data.frame(start = 10:15,
                            width = 5,
                            seqnames = "seq1",
                            strand = c("+", "+", "-", "-", "+", "*")))
flank_left(gr, width = 5L)
flank_right(gr, width = 5L)
flank_upstream(gr, width = 5L)
flank_downstream(gr, width = 5L)
```

GroupedGenomicRanges-class
Group a Ranges by one or more variables

Description

The function `group_by` takes a Ranges object and defines groups by one or more variables. Operations are then performed on the Ranges by their "group". `ungroup()` removes grouping.

Usage

```
## S3 method for class 'GenomicRanges'
group_by(.data, ..., add = FALSE)

## S3 method for class 'GroupedGenomicRanges'
ungroup(x, ...)

## S3 method for class 'GroupedGenomicRanges'
groups(x)

## S3 method for class 'GroupedIntegerRanges'
groups(x)
```

Arguments

`.data`	a Ranges object.
`...`	Variable names to group by. These can be either metadata columns or the core variables of a Ranges.
`add`	if `.data` is already a GroupedRanges object, when add = FALSE the (default), `group_by()` will override existing groups. If add = TRUE, additional groups will be added.
`x`	a GroupedRanges object.

Details

`group_by()` creates a new object of class `GroupedGenomicRanges` if the input is a `GRanges` object or an object of class `GroupedIntegerRanges` if the input is a `IRanges` object. Both of these classes contain a slot called `groups` corresponding to the names of grouping variables. They also inherit from their parent classes, `Ranges` and `GenomicRanges` respectively. `ungroup()` removes the grouping and will return either a `GRanges` or `IRanges` object.

Value

The `group_by()` function will return a GroupedRanges object. These have the same appearance as a regular Ranges object but with an additional groups slot.

Accessors

To return grouping variables on a grouped Ranges use either

- `groups(x)`Returns a list of symbols
- `group_vars(x)`Returns a character vector

Examples

```
set.seed(100)
df <- data.frame(start = 1:10,
                 width = 5,
                 gc = runif(10),
                 cat = sample(letters[1:2], 10, replace = TRUE))
rng <- as_iranges(df)
rng_by_cat <- rng %>% group_by(cat)
# grouping does not change appearance or shape of Ranges
rng_by_cat
# a list of symbols
groups(rng_by_cat)
# ungroup removes any grouping
ungroup(rng_by_cat)
# group_by works best with other verbs
grng <- as_granges(df,
                   seqnames = "chr1",
                   strand = sample(c("+", "-"), size = 10, replace = TRUE))

grng_by_strand <- grng %>% group_by(strand)
grng_by_strand
# grouping with other verbs
grng_by_strand %>% summarise(gc = mean(gc))
grng_by_strand %>% filter(gc == min(gc))
grng_by_strand %>%
  ungroup() %>%
  summarise(gc = mean(gc))
```

intersect_ranges *Vector-wise Range set-operations*

Description

Vector-wise Range set-operations

Usage

```
intersect_ranges(x, y)

intersect_ranges_directed(x, y)

union_ranges(x, y)

union_ranges_directed(x, y)

setdiff_ranges(x, y)

setdiff_ranges_directed(x, y)

complement_ranges(x)

complement_ranges_directed(x)
```

Arguments

x, y	Two Ranges objects to compare.

Details

These are usual set-operations that act on the sets of the ranges represented in x and y. By default these operations will ignore any strand information. The directed versions of these functions will take into account strand for GRanges objects.

Value

A Ranges object

Examples

```
gr1 <- data.frame(seqnames = "chr1",
                  start = c(2,9),
                  end = c(7,9),
                  strand = c("+", "-")) %>%
               as_granges()
gr2 <- data.frame(seqnames = "chr1", start = 5, width = 5, strand = "-") %>%
         as_granges()

union_ranges(gr1, gr2)
union_ranges_directed(gr1, gr2)

intersect_ranges(gr1, gr2)
intersect_ranges_directed(gr1, gr2)

setdiff_ranges(gr1, gr2)
setdiff_ranges_directed(gr1, gr2)
# taking the complement of a ranges requires annotation information
gr1 <- set_genome_info(gr1, seqlengths = 100)
complement_ranges(gr1)
```

interweave *Interweave a pair of Ranges objects together*

Description

Interweave a pair of Ranges objects together

Usage

```
interweave(left, right, .id = NULL)
```

Arguments

left, right	Ranges objects.
.id	When supplied a new column that represents the origin column and is linked to each row of the resulting Ranges object.

Details

The output of `interweave()` takes pairs of Ranges objects and combines them into a single Ranges object. If an .id argument is supplied, an origin column with name .id is created indicated which side the resulting Range comes from (eit)

Value

a Ranges object

Examples

```
gr <- as_granges(data.frame(start = 10:15,
                            width = 5,
                            seqnames = "seq1",
                            strand = c("+", "+", "-", "-", "+", "*")))
interweave(flank_left(gr, width = 5L), flank_right(gr, width = 5L))
interweave(flank_left(gr, width = 5L), flank_right(gr, width = 5L), .id = "origin")
```

`join_follow` *Find following Ranges*

Description

Find following Ranges

Usage

```
join_follow(x, y, suffix = c(".x", ".y"))

join_follow_left(x, y, suffix = c(".x", ".y"))

join_follow_upstream(x, y, suffix = c(".x", ".y"))
```

Arguments

`x, y`	Ranges objects, which ranges in x follow those in y.
`suffix`	A character vector of length two used to identify metadata columns coming from x and y.

Details

By default `join_follow` will find abritrary ranges in y that are followed by ranges in x and ignore any strand information. On the other hand `join_follow_left` will find all ranges in y that are on the left-hand side of the ranges in x ignoring any strand information. Finally, `join_follow_upstream` will find all ranges in x that are that are upstream of the ranges in y. On the positive strand this will result in ranges in y that are left of those in x and on the negative strand it will result in ranges in y that are right of those in x.

Value

A Ranges object corresponding to the ranges in x`` `that are followed by the ranges in y, all metadata is copi`

Examples

```
query <- data.frame(start = c(5,10, 15,20), width = 5, gc = runif(4)) %>%
             as_iranges()
subject <- data.frame(start = 2:6, width = 3:7, label = letters[1:5]) %>%
             as_iranges()

join_follow(query, subject)

subject  <- data.frame(seqnames = "chr1",
               start = c(11,101),
               end = c(21, 200),
               name = c("a1", "a2"),
               strand = c("+", "-"),
               score = c(1,2)) %>%
           as_granges()
query <- data.frame(seqnames = "chr1",
                      strand = c("+", "-", "+", "-"),
                      start = c(21,91,101,201),
                      end = c(30,101,110,210),
                      name = paste0("b", 1:4),
                      score = 1:4) %>%
                   as_granges()

join_follow(query, subject)
join_follow_left(query, subject)
join_follow_upstream(query, subject)
```

---

join_nearest — *Find nearest neighbours between two Ranges objects*

Description

Find nearest neighbours between two Ranges objects

Usage

```
join_nearest(x, y, suffix = c(".x", ".y"), distance = FALSE)

join_nearest_left(x, y, suffix = c(".x", ".y"), distance = FALSE)

join_nearest_right(x, y, suffix = c(".x", ".y"), distance = FALSE)

join_nearest_upstream(x, y, suffix = c(".x", ".y"), distance = FALSE)

join_nearest_downstream(x, y, suffix = c(".x", ".y"), distance = FALSE)
```

Arguments

`x, y`	Ranges objects, add the nearest neighbours of ranges in x to those in y.
`suffix`	A character vector of length two used to identify metadata columns
`distance`	logical vector whether to add a column named "distance" containing the distance to the nearest region. If set to a character vector of length 1, will use that as distance column name.

Details

By default `join_nearest` will find arbitrary nearest neighbours in either direction and ignore any strand information. The `join_nearest_left` and `join_nearest_right` methods will find arbitrary nearest neighbour ranges on x that are left/right of those on y and ignore any strand information.

The `join_nearest_upstream` method will find arbitrary nearest neighbour ranges on x that are upstream of those on y. This takes into account strandedness of the ranges. On the positive strand nearest upstream will be on the left and on the negative strand nearest upstream will be on the right.

The `join_nearest_downstream` method will find arbitrary nearest neighbour ranges on x that are upstream of those on y. This takes into account strandedness of the ranges.On the positive strand nearest downstream will be on the right and on the negative strand nearest upstream will be on the left.

Value

A Ranges object corresponding to the nearest ranges, all metadata is copied over from the right-hand side ranges `y`.

Examples

```
query <- data.frame(start = c(5,10, 15,20),
                   width = 5,
                   gc = runif(4)) %>%
             as_iranges()
subject <- data.frame(start = c(2:6, 24),
                      width = 3:8,
                      label = letters[1:6]) %>%
             as_iranges()

join_nearest(query, subject)
join_nearest_left(query, subject)
join_nearest_right(query, subject)

subject  <- data.frame(seqnames = "chr1",
               start = c(11,101),
               end = c(21, 200),
               name = c("a1", "a2"),
               strand = c("+", "-"),
               score = c(1,2)) %>%
           as_granges()
query <- data.frame(seqnames = "chr1",
                      strand = c("+", "-", "+", "-"),
                      start = c(21,91,101,201),
                      end = c(30,101,110,210),
                      name = paste0("b", 1:4),
                      score = 1:4) %>%
                   as_granges()
join_nearest_upstream(query, subject)
join_nearest_downstream(query, subject)
```

`join_overlap_intersect` *Join by overlapping Ranges*

Description

Join by overlapping Ranges

Usage

```
join_overlap_intersect(x, y, maxgap, minoverlap, suffix = c(".x", ".y"))

join_overlap_intersect_within(x, y, maxgap, minoverlap, suffix = c(".x", ".y"))

join_overlap_intersect_directed(
  x,
  y,
  maxgap,
  minoverlap,
  suffix = c(".x", ".y")
)

join_overlap_intersect_within_directed(
  x,
  y,
  maxgap,
  minoverlap,
  suffix = c(".x", ".y")
)

join_overlap_inner(x, y, maxgap = -1L, minoverlap = 0L, suffix = c(".x", ".y"))

join_overlap_inner_within(
  x,
  y,
  maxgap = -1L,
  minoverlap = 0L,
  suffix = c(".x", ".y")
)

join_overlap_inner_directed(
  x,
  y,
  maxgap = -1L,
  minoverlap = 0L,
  suffix = c(".x", ".y")
)

join_overlap_inner_within_directed(
  x,
  y,
  maxgap = -1L,
  minoverlap = 0L,
  suffix = c(".x", ".y")
)

join_overlap_left(x, y, maxgap, minoverlap, suffix = c(".x", ".y"))

join_overlap_left_within(x, y, maxgap, minoverlap, suffix = c(".x", ".y"))

join_overlap_left_directed(x, y, maxgap, minoverlap, suffix = c(".x", ".y"))

join_overlap_left_within_directed(
  x,
  y,
  maxgap,
  minoverlap,
  suffix = c(".x", ".y")
)
```

Arguments

`x, y`	Objects representing ranges
`maxgap, minoverlap`	The maximimum gap between intervals as an integer greater than or equal to zero. The minimum amount of overlap between intervals as an integer greater than zero, accounting for the maximum gap.
`suffix`	Character to vectors to append to common columns in x and y (default = `c(".x", ".y")`).

Details

The function `join_overlap_intersect()` finds the genomic intervals that are the overlapping ranges between x and y and returns a new ranges object with metadata columns from x and y.

The function `join_overlap_inner()` is equivalent to `find_overlaps()`.

The function `join_overlap_left()` performs a left outer join between x and y. It returns all ranges in x that overlap or do not overlap ranges in y plus metadata columns common to both. If there is no overlapping range the metadata column will contain a missing value.

The function `join_overlap_self()` find all overlaps between a ranges object x and itself.

All of these functions have two suffixes that modify their behavior. The `within` suffix, returns only ranges in x that are completely overlapped within in y. The `directed` suffix accounts for the strandedness of the ranges when performing overlaps.

Value

a GRanges object

See Also

`join_overlap_self()`, `join_overlap_left()`, `find_overlaps()`

Examples

```
x <- as_iranges(data.frame(start = c(11, 101), end = c(21, 201)))
y <- as_iranges(data.frame(start = c(10, 20, 50, 100, 1),
                           end = c(19, 21, 105, 202, 5)))

# self
join_overlap_self(y)

# intersect takes common interval
join_overlap_intersect(x,y)

# within
join_overlap_intersect_within(x,y)

# left, and inner join, it's often useful having an id column here
y <- y %>% mutate(id = 1:n())
x <- x %>% mutate(id = 1:n())
join_overlap_inner(x,y)
join_overlap_left(y,x, suffix = c(".left", ".right"))
```

`join_overlap_self` *Find overlaps within a Ranges object*

Description

Find overlaps within a Ranges object

Usage

```
join_overlap_self(x, maxgap, minoverlap)

join_overlap_self_within(x, maxgap, minoverlap)

join_overlap_self_directed(x, maxgap, minoverlap)

join_overlap_self_within_directed(x, maxgap, minoverlap)
```

Arguments

`x`	A Ranges object
`maxgap, minoverlap`	The maximimum gap between intervals as an integer greater than or equal to zero. The minimum amount of overlap between intervals as an integer greater than zero, accounting for the maximum gap.

Details

Self overlaps find any overlaps (or overlaps within or overlaps directed) between a ranges object and itself.

Value

a Ranges object

See Also

`find_overlaps()`, `join_overlap_inner()`

Examples

```
query <- data.frame(start = c(5,10, 15,20), width = 5, gc = runif(4)) %>%
             as_iranges()

join_overlap_self(query)

# -- GRanges objects, strand is ignored by default
query  <- data.frame(seqnames = "chr1",
               start = c(11,101),
               end = c(21, 200),
               name = c("a1", "a2"),
               strand = c("+", "-"),
               score = c(1,2)) %>%
           as_granges()

# ignores strandedness
join_overlap_self(query)
join_overlap_self_within(query)
# adding directed prefix includes strand
join_overlap_self_directed(query)
```

`join_precede` *Find preceding Ranges*

Description

Find preceding Ranges

Usage

```
join_precede(x, y, suffix = c(".x", ".y"))

join_precede_right(x, y, suffix = c(".x", ".y"))

join_precede_downstream(x, y, suffix = c(".x", ".y"))
```

Arguments

`x, y`	Ranges objects, which ranges in x precede those in y.
`suffix`	A character vector of length two used to identify metadata columns coming from x and y.

Details

By default `join_precede` will return the ranges in x that come before the ranges in y and ignore any strand information. The function `join_precede_right` will find all ranges in y that are on the right-hand side of the ranges in x ignoring any strand information. Finally, `join_precede_downstream` will find all ranges in y that are that are downstream of the ranges in x. On the positive strand this will result in ranges in y that are right of those in x and on the negative strand it will result in ranges in y that are left of those in x.

Value

A Ranges object corresponding to the ranges in `y` that are preceded by the ranges in `x`, all metadata is copied over from the right-hand side ranges `y`.

Examples

```
subject <- data.frame(start = c(5,10, 15,20), width = 5, gc = runif(4)) %>%
             as_iranges()
query <- data.frame(start = 2:6, width = 3:7, label = letters[1:5]) %>%
             as_iranges()

join_precede(query, subject)

query  <- data.frame(seqnames = "chr1",
               start = c(11,101),
               end = c(21, 200),
               name = c("a1", "a2"),
               strand = c("+", "-"),
               score = c(1,2)) %>%
           as_granges()
subject <- data.frame(seqnames = "chr1",
                      strand = c("+", "-", "+", "-"),
                      start = c(21,91,101,201),
                      end = c(30,101,110,210),
                      name = paste0("b", 1:4),
                      score = 1:4) %>%
                   as_granges()

join_precede(query, subject)
join_precede_right(query, subject)
join_precede_downstream(query, subject)
```

`mutate.Ranges` *Modify a Ranges object*

Description

Modify a Ranges object

Usage

```
## S3 method for class 'Ranges'
mutate(.data, ...)
```

Arguments

`.data`	a `Ranges` object
`...`	Pairs of name-value expressions. The name-value pairs can either create new metadata columns or modify existing ones.

Value

a Ranges object

Examples

```
df <- data.frame(start = 1:10,
                 width = 5,
                 seqnames = "seq1",
                 strand = sample(c("+", "-", "*"), 10, replace = TRUE),
                 gc = runif(10))
rng <- as_granges(df)

# mutate adds new columns
rng %>%
    mutate(avg_gc = mean(gc), row_id = 1:n())
# can also compute on newly created columns
rng %>%
    mutate(score = gc * width, score2 = score + 1)
# group by partitions the data and computes within each group
rng %>%
    group_by(strand) %>%
    mutate(avg_gc = mean(gc), row_id = 1:n())

# mutate can be used in conjuction with anchoring to resize ranges
rng %>%
    mutate(width = 10)
# by default width modfication fixes by start
rng %>%
    anchor_start() %>%
    mutate(width = 10)
# fix by end or midpoint
rng %>%
    anchor_end() %>%
    mutate(width = width + 1)
rng %>%
    anchor_center() %>%
    mutate(width = width + 1)
# anchoring by strand
rng %>%
    anchor_3p() %>%
    mutate(width = width * 2)
rng %>%
    anchor_5p() %>%
    mutate(width = width * 2)
```

n — *Compute the number of ranges in each group.*

Description

This function should only be used within `summarise()`, `mutate()` and `filter()`.

Usage

```
n()
```

Value

`n()` will only be evaluated inside a function call, where it returns an integer.

Examples

```
ir <- as_iranges(
                 data.frame(start = 1:10,
                            width = 5,
                            name = c(rep("a", 5), rep("b", 3), rep("c", 2))
                            )
                )
by_names <- group_by(ir, name)
summarise(by_names, n = n())
mutate(by_names, n = n())
filter(by_names, n() >= 3)
```

n_distinct — *Compute the number of distinct unique values in a vector or List*

Description

This is a wrapper to `length(unique(x))` or `lengths(unique(x))` if `x` is a List object

Usage

```
n_distinct(var)
```

Arguments

`var`	a vector of values

Value

an integer vector

Examples

```
x <- CharacterList(c("a", "b", "c", "a"),  "d")
n_distinct(x)
n_distinct(unlist(x))
```

`overscope_ranges` *Create an overscoped environment from a Ranges object*

Description

Create an overscoped environment from a Ranges object

Usage

```
overscope_ranges(x, envir = parent.frame())
```

Arguments

`x`	a Ranges object
`envir`	the environment to place the Ranges in (default = `parent.frame()`)

Details

This is the backend for non-standard evaluation in `plyranges`.

Value

an environment

See Also

`rlang::new_data_mask()`, `rlang::eval_tidy()`

`pair_overlaps` *Pair together two ranges objects*

Description

Pair together two ranges objects

Usage

```
pair_overlaps(x, y, maxgap, minoverlap, suffix)

pair_nearest(x, y, suffix)

pair_precede(x, y, suffix)

pair_follow(x, y, suffix)
```

Arguments

`x, y`	Ranges objects to pair together.
`maxgap, minoverlap`	The maximimum gap between intervals as an integer greater than or equal to negative one. The minimum amount of overlap between intervals as an integer greater than zero, accounting for the maximum gap.
`suffix`	A character vector of length two used to identify metadata columns coming from x and y.

Details

These functions return a DataFrame object, and is one way of representing paired alignments with plyranges.

Value

a DataFrame with two ranges columns and the corresponding metadata columns.

See Also

[join_nearest()][join_overlap_inner()][join_precede()][join_follow()]

Examples

```
query <- data.frame(start = c(5,10, 15,20), width = 5, gc = runif(4)) %>%
             as_iranges()
subject <- data.frame(start = 2:6, width = 3:7, label = letters[1:5]) %>%
             as_iranges()

pair_overlaps(query, subject)
pair_overlaps(query, subject, minoverlap = 5)
pair_nearest(query, subject)


query  <- data.frame(seqnames = "chr1",
               start = c(11,101),
               end = c(21, 200),
               name = c("a1", "a2"),
               strand = c("+", "-"),
               score = c(1,2)) %>%
           as_granges()
subject <- data.frame(seqnames = "chr1",
                      strand = c("+", "-", "+", "-"),
                      start = c(21,91,101,201),
                      end = c(30,101,110,210),
                      name = paste0("b", 1:4),
                      score = 1:4) %>%
                   as_granges()

# ignores strandedness
pair_overlaps(query, subject, suffix = c(".query", ".subject"))
pair_follow(query, subject, suffix = c(".query", ".subject"))
pair_precede(query, subject, suffix = c(".query", ".subject"))
pair_precede(query, subject, suffix = c(".query", ".subject"))
```

ranges-info *Construct annotation information*

Description

To construct annotations by supplying annotation information use `genome_info`. To add annotations to an existing Ranges object use `set_genome_info`. To retrieve an annotation as a Ranges object use `get_genome_info`.

Usage

```
genome_info(
  genome = NULL,
  seqnames = NULL,
  seqlengths = NULL,
  is_circular = NULL
)

set_genome_info(
  .data,
  genome = NULL,
  seqnames = NULL,
  seqlengths = NULL,
  is_circular = NULL
)

get_genome_info(.data)
```

Arguments

`genome`	A character vector of length one indicating the genome build.
`seqnames`	A character vector containing the name of sequences.
`seqlengths`	An optional integer vector containg the lengths of sequences.
`is_circular`	An optional logical vector indicating whether a sequence is ciruclar.
`.data`	A Ranges object to annotate or retrieve an annotation for.

Value

a GRanges object containing annotations. To retrieve the annotations as a Ranges object use `get_genome_info`.

See Also

`GenomeInfoDb::Seqinfo()`

Examples

```
x <- genome_info(genome = "toy",
                 seqnames = letters[1:4],
                 seqlengths = c(100, 300, 15, 600),
                 is_circular = c(NA, FALSE, FALSE, TRUE))
```

```
x

rng <- as_granges(data.frame(seqnames = "a", start = 30:50, width = 10))
rng
rng <- set_genome_info(rng,
                       genome = "toy",
                       seqnames = letters[1:4],
                       seqlengths = c(100, 300, 15, 600),
                       is_circular = c(NA, FALSE, FALSE, TRUE))
get_genome_info(rng)

## Not run:
if (interactive()) {
 # requires internet connection
 genome_info(genome = "hg38")
}

## End(Not run)
```

`read_bam` *Read a BAM file*

Description

Read a BAM file

Usage

```
read_bam(file, index = file, paired = FALSE)
```

Arguments

`file`	A connection or path to a BAM file
`index`	The path to the BAM index file
`paired`	Whether to treat alignments as paired end (TRUE) or single end (FALSE). Default is FALSE.

Details

Reading a BAM file is deferred until an action such as using `summarise()` or `mutate()` occurs. If paired is set to TRUE, when alignments are loaded, the GRanges has two additional columns called read_pair_id and read_pair_group corresponding to paired reads and is grouped by the read_pair_group.

Certain verbs have different behaviour, after using `read_bam()`.

For `select()` valid columns are the fields available in the BAM file. Valid entries are qname (QNAME), flag (FLAG), rname (RNAME), strand, pos (POS), qwidth (width of query), mapq (MAPQ), cigar (CIGAR), mrnm (RNEXT), mpos (PNEXT), isize (TLEN), seq (SEQ), and qual (QUAL). Any two character tags in the BAM file are also valid.

For `filter()` the following fields are valid, to select the FALSE option place `!` in front of the field:

- `is_paired` Select either unpaired (FALSE) or paired (TRUE) reads.

- `is_proper_pair` Select either improperly paired (FALSE) or properly paired (TRUE) reads. This is dependent on the alignment software used.
- ‘is_unmapped_query“ Select unmapped (TRUE) or mapped (FALSE) reads.
- `has_unmapped_mate` Select reads with mapped (FALSE) or unmapped (TRUE) mates.
- `is_minus_strand` Select reads aligned to plus (FALSE) or minus (TRUE) strand.
- `is_mate_minus_strand` Select reads where mate is aligned to plus (FALSE) or minus (TRUE) strand.
- `is_first_mate_read` Select reads if they are the first mate (TRUE) or not (FALSE).
- `is_second_mate_read` Select reads if they are the second mate (TRUE) or not (FALSE).
- `is_secondary_alignment` Select reads if their alignment status is secondary (TRUE) or not (FALSE). This might be relevant if there are multimapping reads.
- `is_not_passing_quality_controls` Select reads that either pass quality controls (FALSE) or that do not (TRUE).
- `is_duplicate` Select reads that are unduplicated (FALSE) or duplicated (TRUE). This may represent reads that are PCR or optical duplicates.

Value

A DeferredGenomicRanges object

See Also

`Rsamtools::BamFile()`,`GenomicAlignments::readGAlignments()`

Examples

```
if (require(pasillaBamSubset)) {
   bamfile <- untreated1_chr4()
   # nothing is read until an action has been performed
   print(read_bam(bamfile))
   # define a region of interest
   roi <- data.frame(seqnames = "chr4", start = 5e5, end = 7e5) %>%
            as_granges()
   rng <- read_bam(bamfile) %>%
            select(mapq) %>%
            filter_by_overlaps(roi)
}
```

`read_bed` *Read a BED or BEDGraph file*

Description

This is a lightweight wrapper to the import family of functions defined in **rtracklayer**.

Read common interval based formats as GRanges.

Usage

```
read_bed(file, col_names = NULL, genome_info = NULL, overlap_ranges = NULL)

read_bed_graph(
  file,
  col_names = NULL,
  genome_info = NULL,
  overlap_ranges = NULL
)

read_narrowpeaks(
  file,
  col_names = NULL,
  genome_info = NULL,
  overlap_ranges = NULL
)
```

Arguments

`file`	A path to a file or a connection.
`col_names`	An optional character vector for including additional columns in `file` that are not part of the BED/narrowPeaks specification.
`genome_info`	An optional character string or a Ranges object that contains information about the genome build. For example the USSC identifier "hg19" will add build information to the returned GRanges.
`overlap_ranges`	An optional Ranges object. Only the intervals in the file that overlap the Ranges will be returned.

Details

This is a lightweight wrapper to the import family of functions defined in **rtracklayer**. The `read_narrowpeaks` function parses the ENCODE narrowPeak BED format (see `https://genome.ucsc.edu/FAQ/FAQformat.html#format12` for details.). As such the parser expects four additional columns called (corresponding to the narrowPeaks spec):

- signalValue
- pValue
- qValue
- peak

Value

A GRanges object

See Also

`rtracklayer::BEDFile()`

Examples

```
test_path <- system.file("tests", package = "rtracklayer")
bed_file <- file.path(test_path, "test.bed")
gr <- read_bed(bed_file)
gr
gr <- read_bed(bed_file, genome_info = "hg19")
gr
olap <-  as_granges(data.frame(seqnames = "chr7", start = 1, end = 127473000))
gr <- read_bed(bed_file,
              overlap_ranges = olap)
# bedGraph
bg_file <- file.path(test_path, "test.bedGraph")
gr <- read_bed_graph(bg_file)
gr
# narrowpeaks
np_file <- system.file("extdata", "demo.narrowPeak.gz",  package="rtracklayer")
gr <- read_narrowpeaks(np_file, genome_info = "hg19")
gr
```

`read_bigwig` *Read a BigWig file*

Description

Read a BigWig file

Usage

```
read_bigwig(file, genome_info = NULL, overlap_ranges = NULL)
```

Arguments

`file`	A path to a file or URL.
`genome_info`	An optional character string or a Ranges object that contains information about the genome build. For example the identifier "hg19" will add build information to the returned GRanges.
`overlap_ranges`	An optional Ranges object. Only the intervals in the file that overlap the Ranges will be loaded.

Value

a GRanges object

See Also

`rtracklayer::BigWigFile()`

Examples

```
if (.Platform$OS.type != "windows") {
  test_path <- system.file("tests", package = "rtracklayer")
  bw_file <- file.path(test_path, "test.bw")
  gr <- read_bigwig(bw_file)
  gr
}
```

---

read_gff *Read a GFF/GTF/GVT file*

Description

This is a lightweight wrapper to the import family of functions defined in **rtracklayer**.

Usage

```
read_gff(file, col_names = NULL, genome_info = NULL, overlap_ranges = NULL)

read_gff1(file, col_names = NULL, genome_info = NULL, overlap_ranges = NULL)

read_gff2(file, col_names = NULL, genome_info = NULL, overlap_ranges = NULL)

read_gff3(file, col_names = NULL, genome_info = NULL, overlap_ranges = NULL)
```

Arguments

`file`	A path to a file or a connection.
`col_names`	An optional character vector for parsing specific columns in `file` that are part of the GFF specification. These should name either fixed fields, like source or type, or, for GFF2 and GFF3, any attribute.
`genome_info`	An optional character string or a Ranges object that contains information about the genome build. For example the UCSC identifier "hg19" will add build information to the returned GRanges.
`overlap_ranges`	An optional Ranges object. Only the intervals in the file that overlap the Ranges will be returned.

Value

A GRanges object

a GRanges object

See Also

`rtracklayer::GFFFile()`

Examples

```
test_path <- system.file("tests", package = "rtracklayer")
# gff3
test_gff3 <- file.path(test_path, "genes.gff3")
gr <- read_gff3(test_gff3)
gr
# alternatively with read_gff
gr <- read_gff(test_gff3, genome_info = "hg19")
gr
```

---

read_wig *Read a WIG file*

Description

This is a lightweight wrapper to the import family of functions defined in **rtracklayer**.

Usage

```
read_wig(file, genome_info = NULL, overlap_ranges = NULL)
```

Arguments

file	A path to a file or a connection.
genome_info	An optional character string or a Ranges object that contains information about the genome build. For example the USSC identifier "hg19" will add build information to the returned GRanges.
overlap_ranges	An optional Ranges object. Only the intervals in the file that overlap the Ranges will be returned.

Value

A GRanges object

A GRanges object

See Also

`rtracklayer::WIGFile()`

Examples

```
test_path <- system.file("tests", package = "rtracklayer")
test_wig <- file.path(test_path, "step.wig")
gr <- read_wig(test_wig)
gr
gr <- read_wig(test_wig, genome_info = "hg19")
```

`reduce_ranges` *Reduce then aggregate a Ranges object*

Description

Reduce then aggregate a Ranges object

Usage

```
reduce_ranges(.data, min.gapwidth = 1L, ...)

reduce_ranges_directed(.data, min.gapwidth = 1L, ...)
```

Arguments

`.data`	a Ranges object to reduce
`min.gapwidth`	Ranges separated by a gap of at least min.gapwidth positions are not merged.
`...`	Name-value pairs of summary functions.

Value

a Ranges object with the

Examples

```
set.seed(10)
df <- data.frame(start = sample(1:10),
                 width = 5,
                 seqnames = "seq1",
                 strand = sample(c("+", "-", "*"), 10, replace = TRUE),
                 gc = runif(10))

rng <- as_granges(df)
rng %>% reduce_ranges()
rng %>% reduce_ranges(gc = mean(gc))
rng %>% reduce_ranges_directed(gc = mean(gc))
rng %>% reduce_ranges_directed(gc = mean(gc), min.gapwidth = 10)

x <- data.frame(start = c(11:13, 2, 7:6),
               width=3,
               id=sample(letters[1:3], 6, replace = TRUE),
               score= sample(1:6))
x <- as_iranges(x)
x %>% reduce_ranges()
x %>% reduce_ranges(score = sum(score))
x %>% group_by(id) %>% reduce_ranges(score = sum(score))
```

reexports *Objects exported from other packages.*

Description

These objects are imported from other packages. Follow the links below to see their documentation.

dplyr `arrange`, `filter`, `group_by`, `group_vars`, `groups`, `mutate`, `select`, `slice`, `summarise`, `summarize`, `ungroup`

magrittr `%>%`

rlang `!!`, `!!!`

remove_names *Tools for working with named Ranges*

Description

Tools for working with named Ranges

Usage

```
remove_names(.data)

names_to_column(.data, var = "name")

id_to_column(.data, var = "id")
```

Arguments

`.data`	a Ranges object
`var`	Name of column to use for names

Details

The function `names_to_column()` and `id_to_column()` always places `var` as the first column in `mcols(.data)`, shifting all other columns to the left. The `id_to_column()` creates a column with sequential row identifiers starting at 1, it will also remove any existing names.

Value

Returns a Ranges object with empty names

Examples

```
ir <- IRanges::IRanges(start = 1:3, width = 4, names = c("a", "b", "c"))
remove_names(ir)
ir_noname <- names_to_column(ir)
ir_noname
ir_with_id <- id_to_column(ir)
ir_with_id
```

select.Ranges — *Select metadata columns of the Ranges object by name or position*

Description

Select metadata columns of the Ranges object by name or position

Usage

```
## S3 method for class 'Ranges'
select(.data, ..., .drop_ranges = FALSE)
```

Arguments

`.data`	a Ranges object
`...`	One or more metadata column names.
`.drop_ranges`	If TRUE select will always return a tibble. In this case, you may select columns that form the core part of the Ranges object.

Details

Note that by default select only acts on the metadata columns (and will therefore return a Ranges object) if a core component of a Ranges is dropped or selected without the other required components (this includes the seqnames, strand, start, end, width names), then select will throw an error unless .drop_ranges is set to TRUE.

Value

a Ranges object or a tibble

See Also

`dplyr::select()`

Examples

```
df <- data.frame(start = 1:10, width = 5,  seqnames = "seq1",
strand = sample(c("+", "-", "*"), 10, replace = TRUE), gc = runif(10), counts = rpois(10, 2))
rng <- as_granges(df)
select(rng, -gc)
select(rng, gc)
select(rng, counts, gc)
select(rng, 2:1)
select(rng, seqnames, strand, .drop_ranges = TRUE)
```

`set_width` *Functional setters for Ranges objects*

Description

Functional setters for Ranges objects

Usage

```
set_width(x, width)

set_start(x, start = 0L)

set_end(x, end = 0L)

set_seqnames(x, seqnames)

set_strand(x, strand)
```

Arguments

`x`	a Ranges object
`width`	integer amount to modify width by
`start`	integer amount to modify start by
`end`	integer amount to modify end by
`seqnames`	update seqnames column
`strand`	update strand column

Details

These methods are used internally in `mutate()` to modify core columns in Ranges objects.

Value

a Ranges object

`shift_left` *Shift all coordinates in a genomic interval left or right, upstream or downstream*

Description

Shift all coordinates in a genomic interval left or right, upstream or downstream

Usage

```
shift_left(x, shift = 0L)

shift_right(x, shift = 0L)

shift_upstream(x, shift = 0L)

shift_downstream(x, shift = 0L)
```

Arguments

`x`	a Ranges object .
`shift`	the amount to move the genomic interval in the Ranges object by. Either a non-negative integer vector of length 1 or an integer vector the same length as x.

Details

Shifting left or right will ignore any strand information in the Ranges object, while shifting upstream/downstream will shift coordinates on the positive strand left/right and the negative strand right/left. By default, unstranded features are treated as positive. When using `shift_upstream()` or `shift_downstream()` when the `shift` argument is indexed by the strandedness of the input ranges.

Value

a Ranges object with start and end coordinates shifted.

See Also

`IRanges::shift()`, `GenomicRanges::shift()`

Examples

```
ir <- as_iranges(data.frame(start = 10:15, width = 5))
shift_left(ir, 5L)
shift_right(ir, 5L)
gr <- as_granges(data.frame(start = 10:15,
                            width = 5,
                            seqnames = "seq1",
                            strand = c("+", "+", "-", "-", "+", "*")))
shift_upstream(gr, 5L)
shift_downstream(gr, 5L)
```

`slice.Ranges` *Choose rows by their position*

Description

Choose rows by their position

Usage

```
## S3 method for class 'Ranges'
slice(.data, ..., .preserve = FALSE)

## S3 method for class 'GroupedGenomicRanges'
slice(.data, ..., .preserve = FALSE)

## S3 method for class 'GroupedIntegerRanges'
slice(.data, ..., .preserve = FALSE)
```

Arguments

`.data`	a Ranges object
`...`	Integer row values indicating rows to keep. If `.data` has been grouped via `group_by()`, then the positions are selected within each group.
`.preserve`	when FALSE (the default) the grouping structure is recomputed, otherwise it is kept as is. Currently ignored.

Value

a GRanges object

Examples

```
df <- data.frame(start = 1:10,
                 width = 5,
                 seqnames = "seq1",
                 strand = sample(c("+", "-", "*"), 10, replace = TRUE),
                 gc = runif(10))
rng <- as_granges(df)
dplyr::slice(rng, 1:2)
dplyr::slice(rng, -n())
dplyr::slice(rng, -5:-n())

by_strand <- group_by(rng, strand)

# slice with group by finds positions within each group
dplyr::slice(by_strand, n())
dplyr::slice(by_strand, which.max(gc))

# if the index is beyond the number of groups slice are ignored
dplyr::slice(by_strand, 1:3)
```

`stretch` *Stretch a genomic interval*

Description

By default, `stretch(x)` will anchor by the center of a Ranges object. This means that half of the value of `extend` will be added to the end of the range and the remaining half subtracted from the start of the Range. The other anchors will leave the start/end fixed and stretch the end/start respectively.

Usage

```
stretch(x, extend)
```

Arguments

`x`	a Ranges object, to fix by either the start, end or center of an interval use `anchor_start(x)`, `anchor_end(x)`, `anchor_center(x)`. To fix by strand use `anchor_3p(x)` or `anchor_5p(x)`.
`extend`	the amount to alter the width of a Ranges object by. Either an integer vector of length 1 or an integer vector the same length as x.

Value

a Ranges object with modified start or end (or both) coordinates

See Also

`anchor()`, `mutate()`

Examples

```
rng <- as_iranges(data.frame(start=c(2:-1, 13:15), width=c(0:3, 2:0)))
rng2 <- stretch(anchor_center(rng), 10)
stretch(anchor_start(rng2), 10)
stretch(anchor_end(rng2), 10)
grng <- as_granges(data.frame(seqnames = "chr1",
                         strand = c("+", "-", "-", "+", "+", "-", "+"),
                         start=c(2:-1, 13:15),
                         width=c(0:3, 2:0)))
stretch(anchor_3p(grng), 10)
stretch(anchor_5p(grng), 10)
```

`summarise.Ranges` *Reduce multiple values in a Ranges down to a single value*

Description

Reduce multiple values in a Ranges down to a single value

Usage

```
## S3 method for class 'Ranges'
summarise(.data, ...)
```

Arguments

`.data`	a Ranges object
`...`	Name-value pairs of summary functions. The name will be the name of the variable in the result. The value should be an expression that will return a value that has length one or length equal to the number of groups.

Details

Creates one or more variables as a S4Vectors::DataFrame() from the input Ranges object. If the ranges object is grouped, there will be a row for each group. Because grouping may remove whether a Ranges object is valid, a DataFrame is always returned.

Value

A S4Vectors::DataFrame()

Examples

```
df <- data.frame(start = 1:10, width = 5,  seqnames = "seq1",
strand = sample(c("+", "-", "*"), 10, replace = TRUE), gc = runif(10))
rng <- as_granges(df)
rng %>% summarise(gc = mean(gc))
rng %>% group_by(strand) %>% summarise(gc = mean(gc))
```

`tile_ranges` *Slide or tile over a Ranges object*

Description

Slide or tile over a Ranges object

Usage

```
tile_ranges(x, width)

slide_ranges(x, width, step)
```

Arguments

x	a Ranges object
width	the maximum width of each window/tile (integer vector of length 1)
step	the distance between start position of each sliding window (integer vector of length 1)

Details

The `tile_ranges()` function paritions a Ranges object `x` by the given the `width` over all ranges in `x`, truncated by the sequence end. The `slide_ranges()` function makes sliding windows within each range of `x` of size `width` and sliding by `step`. Both `slide_ranges()` and `tile_ranges()` return a new Ranges object with a metadata column called "partition" which contains the index of the input range `x` that a parition belongs to.

Value

a Ranges object

See Also

GenomicRanges::tile()

Examples

```
gr <- data.frame(seqnames = c("chr1", rep("chr2", 3), rep("chr1", 2), rep("chr3", 4)),
                 start = 1:10,
                 end = 11,
                strand = c("-", rep("+", 2), rep("*", 2), rep("+", 3), rep("-", 2))) %>%
      as_granges() %>%
      set_genome_info(seqlengths = c(11,12,13))

# partition ranges into subranges of width 2, odd width ranges
# will have one subrange of width 1
tile_ranges(gr, width = 2)

# make sliding windows of width 3, moving window with step size of 2
slide_ranges(gr, width = 3, step = 2)
```

`write_bed` *Write a BED or BEDGraph file*

Description

This is a lightweight wrapper to the export family of functions defined in **rtracklayer**.

Usage

```
write_bed(x, file, index = FALSE)

write_bed_graph(x, file, index = FALSE)

write_narrowpeaks(x, file)
```

Arguments

`x`	A GRanges object
`file`	File name, URL or connection specifying a file to write x to. Compressed files with extensions such as '.gz' are handled automatically. If you want to index the file with tabix use the `index` argument.
`index`	Compress and index the output file with bgzf and tabix (default = FALSE). Note that tabix indexing will sort the data by chromosome and start.

Value

The write functions return a BED(Graph)File invisibly

See Also

`rtracklayer::BEDFile()`

Examples

```
## Not run:
  test_path <- system.file("tests", package = "rtracklayer")
  bed_file <- file.path(test_path, "test.bed")
  gr <- read_bed(bed_file)
  bed_file_out <- file.path(tempdir(), "new.bed")
  write_bed(gr, bed_file_out)
  read_bed(bed_file_out)
  #' bedgraph
  bg_file <- file.path(test_path, "test.bedGraph")
  gr <- read_bed_graph(bg_file)
  bg_file_out <- file.path(tempdir(), "new.bg")
  write_bed(gr, bg_file_out)
  read_bed(bg_file_out)
  # narrowpeaks
  np_file <- system.file("extdata", "demo.narrowPeak.gz",package="rtracklayer")
  gr <- read_narrowpeaks(np_file, genome_info = "hg19")
  np_file_out <- file.path(tempdir(), "new.bg")
  write_narrowpeaks(gr, np_file_out)
  read_narrowpeaks(np_file_out)

## End(Not run)
```

`write_bigwig` *Write a BigWig file*

Description

This is a lightweight wrapper to the export family of functions defined in **rtracklayer**.

Usage

```
write_bigwig(x, file)
```

Arguments

`x`	A GRanges object
`file`	File name, URL or connection specifying a file to write x to. Compressed files with extensions such as '.gz' are handled automatically.

Value

The write functions return a BigWigFile invisibly

See Also

`rtracklayer::BigWigFile()`

Examples

```
## Not run:
 if (.Platform$OS.type != "windows") {
  test_path <- system.file("tests", package = "rtracklayer")
  bw_file <- file.path(test_path, "test.bw")
  gr <- read_bigwig(bw_file)
  gr
  bw_out <- file.path(tempdir(), "test_out.bw")
  write_bigwig(gr ,bw_out)
  read_bigwig(bw_out)
 }

## End(Not run)
```

`write_gff` *Write a GFF(123) file*

Description

This is a lightweight wrapper to the export family of functions defined in **rtracklayer**.

Usage

```
write_gff(x, file, index = FALSE)

write_gff1(x, file, index = FALSE)

write_gff2(x, file, index = FALSE)

write_gff3(x, file, index = FALSE)
```

Arguments

`x`	A GRanges object
`file`	Path or connection to write to
`index`	If TRUE the output file will be compressed and indexed using bgzf and tabix.

Value

The write function returns a GFFFile object invisibly

See Also

`rtracklayer::GFFFile()`

Examples

```
## Not run:
 test_path <- system.file("tests", package = "rtracklayer")
 test_gff3 <- file.path(test_path, "genes.gff3")
 gr <- read_gff3(test_gff3)
 out_gff3 <- file.path(tempdir(), "test.gff3")
 write_gff3(gr, out_gff3)
 read_gff3(out_gff3)

## End(Not run)
```

`write_wig` *Write a WIG file*

Description

Write a WIG file

Usage

```
write_wig(x, file)
```

Arguments

`x`	A GRanges object
`file`	File name, URL or connection specifying a file to write x to. Compressed files with extensions such as '.gz' are handled automatically.

Value

The write function returns a WIGFile invisibly.

See Also

`rtracklayer::WIGFile()`

`%union%` *Row-wise set operations on Ranges objects*

Description

Row-wise set operations on Ranges objects

Usage

```
x %union% y

x %intersect% y

x %setdiff% y

between(x, y)

span(x, y)
```

Arguments

x, y	Ranges objects

Details

Each of these functions acts on the rows between pairs of Ranges object. The function `%union%()`. will return the entire range between two ranges objects assuming there are no gaps, if you would like to force gaps use `span()` instead. The function `%intersect%()` will create a new ranges object with a hit column indicating whether or not the two ranges intersect. The function `%setdiff%()`will return the ranges for each row in x that are not in the corresponding row of y. The function `between()` will return the gaps between two ranges.

Value

A Ranges object

See Also

[IRanges::punion()][IRanges::pintersect()][IRanges::pgap()][IRanges::psetdiff()]

Examples

```
x <- as_iranges(data.frame(start = 1:10, width = 5))
# stretch x by 3 on the right
y <- stretch(anchor_start(x), 3)
# take the rowwise union
x %union% y
# take the rowwise intersection
x %intersect% y
# asymetric difference
y %setdiff% x
x %setdiff% y
# if there are gaps between the rows of each range use span
y <- as_iranges(data.frame(start = c(20:15, 2:5),
width = c(10:15,1:4)))
# fill in the gaps and take the rowwise union
span(x,y)
# find the gaps
between(x,y)
```

Index