Package 'slider'

July 1, 2021

Title Sliding Window Functions

Version 0.2.2

Description Provides type-stable rolling window functions over any R data type. Cumulative and expanding windows are also supported. For more advanced usage, an index can be used as a secondary vector that defines how sliding windows are to be created.

License MIT + file LICENSE

URL https://github.com/DavisVaughan/slider

BugReports https://github.com/DavisVaughan/slider/issues

Depends R (>= 3.3.0)

Imports ellipsis (>= 0.3.1), glue, rlang (>= 0.4.5), vctrs (>= 0.3.6), warp

Suggests covr, dplyr (>= 1.0.0), knitr, lubridate, rmarkdown, testthat (>= 3.0.0)

LinkingTo vctrs (>= 0.3.6)

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.1.1

SystemRequirements C++11

Collate 'block.R' 'conditions.R' 'hop-common.R' 'hop-index-common.R' 'hop-index.R' 'hop-index2.R' 'hop.R' 'hop2.R' 'names.R' 'phop-index.R' 'phop.R' 'slide-index2.R' 'pslide-index.R' 'slide-period2.R' 'pslide-period.R' 'slide2.R' 'pslide.R' 'segment-tree.R' 'slide-common.R' 'slide-index-common.R' 'slide-index.R' 'slide-period-common.R' 'slide-period.R' 'slide.R' 'slider-package.R' 'summary-index.R' 'summary-slide.R' 'utils.R' 'zzz.R'

NeedsCompilation yes

Author Davis Vaughan [aut, cre], RStudio [cph]

Maintainer Davis Vaughan <davis@rstudio.com>

Repository CRAN

Date/Publication 2021-07-01 19:50:01 UTC

R topics documented:

`block` *Break a vector into blocks*

Description

`block()` breaks up the `i`-ndex by `period`, and then uses that to define the indices to chop `x` with.

For example, it can split `x` into monthly or yearly blocks. Combined with `purrr::map()`, it is a way to iterate over a vector in "time blocks".

Usage

```
block(x, i, period, every = 1L, origin = NULL)
```

Arguments

`x`	[vector] The vector to block.
`i`	[Date / POSIXct / POSIXlt] The datetime index to block by. There are 3 restrictions on the index: • The size of the index must match the size of `x`, they will not be recycled to their common size. • The index must be an *increasing* vector, but duplicate values are allowed.

- The index cannot have missing values.

`period` [character(1)]

A string defining the period to group by. Valid inputs can be roughly broken into:

- `"year"`, `"quarter"`, `"month"`, `"week"`, `"day"`
- `"hour"`, `"minute"`, `"second"`, `"millisecond"`
- `"yweek"`, `"mweek"`
- `"yday"`, `"mday"`

`every` [positive integer(1)]

The number of periods to group together.

For example, if the period was set to `"year"` with an every value of 2, then the years 1970 and 1971 would be placed in the same group.

`origin` [Date(1) / POSIXct(1) / POSIXlt(1) / NULL]

The reference date time value. The default when left as `NULL` is the epoch time of 1970-01-01 00:00:00, *in the time zone of the index*.

This is generally used to define the anchor time to count from, which is relevant when the every value is > 1.

Details

`block()` determines the indices to block by with `warp::warp_boundary()`, and splits `x` by those indices using `vctrs::vec_chop()`.

Like `slide()`, `block()` splits data frame `x` values row wise.

Value

A vector fulfilling the following invariants:

- `vec_size(block(x)) == vec_size(unique(warp::warp_boundary(i)))`
- `vec_ptype(block(x)) == list()`
- `vec_ptype(block(x)[[1]]) == vec_ptype(x)`

See Also

`slide_period()`, `slide()`, `slide_index()`

Examples

```
x <- 1:6
i <- as.Date("2019-01-01") + c(-2:2, 31)

block(i, i, period = "year")

# Data frames are split row wise
df <- data.frame(x = x, i = i)
block(df, i, period = "month")

# Iterate over these blocks to apply a function over
```

```
# non-overlapping period blocks. For example, to compute a
# mean over yearly or monthly blocks.
vapply(block(x, i, "year"), mean, numeric(1))
vapply(block(x, i, "month"), mean, numeric(1))

# block by every 2 months, ensuring that we start counting
# the 1st of the 2 months from `2019-01-01`
block(i, i, period = "month", every = 2, origin = as.Date("2019-01-01"))

# Use the `origin` to instead start counting from `2018-12-01`, meaning
# that [2018-12, 2019-01] gets bucketed together.
block(i, i, period = "month", every = 2, origin = as.Date("2018-12-01"))
```

---

hop *Hop*

Description

`hop()` is the lower level engine that powers `slide()` (at least in theory). It has slightly different invariants than `slide()`, and is useful when you either need to hand craft boundary locations, or want to compute a result with a size that is different from `.x`.

Usage

```
hop(.x, .starts, .stops, .f, ...)

hop_vec(.x, .starts, .stops, .f, ..., .ptype = NULL)
```

Arguments

`.x`	[vector] The vector to iterate over and apply `.f` to.
`.starts, .stops`	[integer] Vectors of boundary locations that make up the windows to bucket `.x` with. Both `.starts` and `.stops` will be recycled to their common size, and that common size will be the size of the result. Both vectors should be integer locations along `.x`, but out-of-bounds values are allowed.
`.f`	[function / formula] If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions.

`...`	Additional arguments passed on to the mapped function.
`.ptype`	[vector(0) / NULL] A prototype corresponding to the type of the output. If `NULL`, the default, the output type is determined by computing the common type across the results of the calls to `.f`. If supplied, the result of each call to `.f` will be cast to that type, and the final output will have that type. If `getOption("vctrs.no_guessing")` is `TRUE`, the `.ptype` must be supplied. This is a way to make production code demand fixed types.

Details

`hop()` is very close to being a faster version of:

```
map2(
  .starts,
  .stops,
  function(start, stop) {
    x_slice <- vec_slice(.x, start:stop)
    .f(x_slice, ...)
  }
)
```

Because of this, `hop_index()` is often the more useful function. `hop()` mainly exists for API completeness.

The main difference is that the start and stop values make up ranges of *possible* locations along `.x`, and it is not enforced that these locations actually exist along `.x`. As an example, with `hop()` you can do the following, which would be an error with `vec_slice()` because `0L` is out of bounds.

```
hop(c("a", "b"), .starts = 0L, .stops = 1L, ~.x)
#> [[1]]
#> [1] "a"
```

`hop()` allows these out of bounds values to be fully compatible with `slide()`. It is always possible to construct a `hop()` call from a `slide()` call. For example, the following are equivalent:

```
slide(1:2, ~.x, .before = 1)

hop(1:2, .starts = c(0, 1), .stops = c(1, 2), ~.x)

#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 1 2
```

Value

A vector fulfilling the following invariants:

`hop()`:

- `vec_size(hop(.x,.starts,.stops)) == vec_size_common(.starts,.stops)`
- `vec_ptype(hop(.x,.starts,.stops)) == list()`

`hop_vec()`:

- `vec_size(hop_vec(.x,.starts,.stops)) == vec_size_common(.starts,.stops)`
- `vec_size(hop_vec(.x,.starts,.stops)[[1]]) == 1L`
- `vec_ptype(hop_vec(.x,.starts,.stops,.ptype = ptype)) == ptype`

See Also

`hop2()`, `hop_index()`, `slide()`

Examples

```
# `hop()` let's you manually specify locations to apply `.f` at.
hop(1:3, .starts = c(1, 3), .stops = 3, ~.x)

# `hop()`'s start/stop locations are allowed to be out of bounds relative
# to the size of `.x`.
hop(
  mtcars,
  .starts = c(-1, 3),
  .stops  = c(2, 6),
  ~.x
)
```

---

hop2 *Hop along multiple inputs simultaneously*

Description

`hop2()` and `phop()` represent the combination of `slide2()` and `pslide()` with `hop()`, allowing you to iterate over multiple vectors at once, hopping along them using boundaries defined by `.starts` and `.stops`.

Usage

```
hop2(.x, .y, .starts, .stops, .f, ...)

hop2_vec(.x, .y, .starts, .stops, .f, ..., .ptype = NULL)

phop(.l, .starts, .stops, .f, ...)

phop_vec(.l, .starts, .stops, .f, ..., .ptype = NULL)
```

Arguments

`.x, .y` [vector]

Vectors to iterate over. Vectors of size 1 will be recycled.

`.starts, .stops`

[integer]

Vectors of boundary locations that make up the windows to bucket `.x` with. Both `.starts` and `.stops` will be recycled to their common size, and that common size will be the size of the result. Both vectors should be integer locations along `.x`, but out-of-bounds values are allowed.

`.f` [function / formula]

If a **function**, it is used as is.

If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:

- For a single argument function, use `.`
- For a two argument function, use `.x` and `.y`
- For more arguments, use `..1`, `..2`, `..3` etc

This syntax allows you to create very compact anonymous functions.

`...` Additional arguments passed on to the mapped function.

`.ptype` [vector(0) / NULL]

A prototype corresponding to the type of the output.

If `NULL`, the default, the output type is determined by computing the common type across the results of the calls to `.f`.

If supplied, the result of each call to `.f` will be cast to that type, and the final output will have that type.

If `getOption("vctrs.no_guessing")` is `TRUE`, the `.ptype` must be supplied. This is a way to make production code demand fixed types.

`.l` [list]

A list of vectors. The length of `.l` determines the number of arguments that `.f` will be called with. If `.l` has names, they will be used as named arguments to `.f`. Elements of `.l` with size 1 will be recycled.

Value

A vector fulfilling the following invariants:

`hop2()`:

- `vec_size(hop2(.x,.y,.starts,.stops)) == vec_size_common(.starts,.stops)`
- `vec_ptype(hop2(.x,.y,.starts,.stops)) == list()`

`hop2_vec()`:

- `vec_size(hop2_vec(.x,.y,.starts,.stops)) == vec_size_common(.starts,.stops)`
- `vec_size(hop2_vec(.x,.y,.starts,.stops)[[1]]) == 1L`
- `vec_ptype(hop2_vec(.x,.y,.starts,.stops,.ptype = ptype)) == ptype`

`phop()`:

- `vec_size(phop(.l,.starts,.stops)) == vec_size_common(.starts,.stops)`
- `vec_ptype(phop(.l,.starts,.stops)) == list()`

`phop_vec()`:

- `vec_size(phop_vec(.l,.starts,.stops)) == vec_size_common(.starts,.stops)`
- `vec_size(phop_vec(.l,.starts,.stops)[[1]]) == 1L`
- `vec_ptype(phop_vec(.l,.starts,.stops,.ptype = ptype)) == ptype`

See Also

`hop()`, `hop_index()`, `slide2()`

Examples

```
hop2(1:2, 3:4, .starts = 1, .stops = c(2, 1), ~c(x = .x, y = .y))

phop(
 list(1, 2:4, 5:7),
 .starts = c(0, 1),
 .stops  = c(2, 4),
 ~c(x = ..1, y = ..2, z = ..3)
)
```

---

hop_index *Hop relative to an index*

Description

`hop_index()` is the lower level engine that powers `slide_index()`. It has slightly different invariants than `slide_index()`, and is useful when you either need to hand craft boundary values, or want to compute a result with a size that is different from `.x`.

Usage

```
hop_index(.x, .i, .starts, .stops, .f, ...)

hop_index_vec(.x, .i, .starts, .stops, .f, ..., .ptype = NULL)
```

Arguments

`.x`	[vector] The vector to iterate over and apply `.f` to.
`.i`	[vector] The index vector that determines the window sizes. It is fairly common to supply a date vector as the index, but not required. There are 3 restrictions on the index:

- The size of the index must match the size of `.x`, they will not be recycled to their common size.
- The index must be an *increasing* vector, but duplicate values are allowed.
- The index cannot have missing values.

`.starts, .stops`
: [vector]

  Vectors of boundary values that make up the windows to bucket `.i` with. Both `.starts` and `.stops` will be recycled to their common size, and that common size will be the size of the result. Both vectors should be the same type as `.i`. These boundaries are both *inclusive*, meaning that the slice of `.x` that will be used in each call to `.f` is where `.i >= start & .i <= stop` returns `TRUE`.

`.f`
: [function / formula]

  If a **function**, it is used as is.

  If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:

  - For a single argument function, use `.`
  - For a two argument function, use `.x` and `.y`
  - For more arguments, use `..1`, `..2`, `..3` etc

  This syntax allows you to create very compact anonymous functions.

`...`
: Additional arguments passed on to the mapped function.

`.ptype`
: [vector(0) / NULL]

  A prototype corresponding to the type of the output.

  If `NULL`, the default, the output type is determined by computing the common type across the results of the calls to `.f`.

  If supplied, the result of each call to `.f` will be cast to that type, and the final output will have that type.

  If `getOption("vctrs.no_guessing")` is `TRUE`, the `.ptype` must be supplied. This is a way to make production code demand fixed types.

Value

A vector fulfilling the following invariants:

`hop_index()`:

- `vec_size(hop_index(.x,.starts,.stops)) == vec_size_common(.starts,.stops)`
- `vec_ptype(hop_index(.x,.starts,.stops)) == list()`

`hop_index_vec()`:

- `vec_size(hop_index_vec(.x,.starts,.stops)) == vec_size_common(.starts,.stops)`
- `vec_size(hop_index_vec(.x,.starts,.stops)[[1]]) == 1L`
- `vec_ptype(hop_index_vec(.x,.starts,.stops,.ptype = ptype)) == ptype`

See Also

`slide()`, `slide_index()`, `hop_index2()`

Examples

```
library(vctrs)
library(lubridate, warn.conflicts = FALSE)

# ------------------------------------------------------------------------
# Returning a size smaller than `.x`

i <- as.Date("2019-01-25") + c(0, 1, 2, 3, 10, 20, 35, 42, 45)

# slide_index() allows you to slide relative to `i`
slide_index(i, i, ~.x, .before = weeks(1))

# But you might be more interested in coarser summaries. This groups
# by year-month and computes 2 `.f` on 2 month windows.
i_yearmonth <- year(i) + (month(i) - 1) / 12
slide_index(i, i_yearmonth, ~.x, .before = 1)

# ^ This works nicely when working with dplyr if you are trying to create
# a new column in a data frame, but you'll notice that there are really only
# 3 months, so only 3 values are being calculated. If you only want to return
# a vector of those 3 values, you can use `hop_index()`. You'll have to
# hand craft the boundaries, but this is a general strategy
# I've found useful:
first_start <- floor_date(i[1], "months")
last_stop <- ceiling_date(i[length(i)], "months")
dates <- seq(first_start, last_stop, "1 month")
inner <- dates[2:(length(dates) - 1L)]
starts <- vec_c(first_start, inner)
stops <- vec_c(inner - 1, last_stop)

hop_index(i, i, starts, stops, ~.x)

# ------------------------------------------------------------------------
# Non-existant dates with `lubridate::months()`

# Imagine you want to compute a 1 month rolling average on this
# irregular daily data.
i <- vec_c(as.Date("2019-02-27") + 0:3, as.Date("2019-03-27") + 0:5)
x <- rnorm(vec_seq_along(i))

# You might try `slide_index()` like this, but you'd run into this error
library(rlang)

with_options(
  catch_cnd(
    slide_index(x, i, mean, .before = months(1))
  ),
  rlang_backtrace_on_error = current_env()
)

# This is because when you actually compute the `.i - .before` sequence,
# you hit non-existant dates. i.e. `"2019-03-29" - months(1)` doesn't exist.
```

```
i - months(1)

# To get around this, lubridate provides `add_with_rollback()`,
# and the shortcut operation `%m-%`, which subtracts the month, then rolls
# forward/backward if it hits an `NA`. You can manually generate boundaries,
# then provide them to `hop_index()`.
starts <- i %m-% months(1)
stops <- i

hop_index(x, i, starts, stops, mean)

hop_index(i, i, starts, stops, ~.x)
```

`hop_index2` *Hop along multiple inputs simultaneously relative to an index*

Description

`hop_index2()` and `phop_index()` represent the combination of `slide2()` and `pslide()` with `hop_index()`, allowing you to iterate over multiple vectors at once, relative to an `.i`-ndex with boundaries defined by `.starts` and `.stops`.

Usage

```
hop_index2(.x, .y, .i, .starts, .stops, .f, ...)

hop_index2_vec(.x, .y, .i, .starts, .stops, .f, ..., .ptype = NULL)

phop_index(.l, .i, .starts, .stops, .f, ...)

phop_index_vec(.l, .i, .starts, .stops, .f, ..., .ptype = NULL)
```

Arguments

`.x, .y`	[vector] Vectors to iterate over. Vectors of size 1 will be recycled.
`.i`	[vector] The index vector that determines the window sizes. It is fairly common to supply a date vector as the index, but not required. There are 3 restrictions on the index: • The size of the index must match the size of `.x`, they will not be recycled to their common size. • The index must be an *increasing* vector, but duplicate values are allowed. • The index cannot have missing values.

`.starts, .stops`
: [vector]

  Vectors of boundary values that make up the windows to bucket `.i` with. Both `.starts` and `.stops` will be recycled to their common size, and that common size will be the size of the result. Both vectors should be the same type as `.i`. These boundaries are both *inclusive*, meaning that the slice of `.x` that will be used in each call to `.f` is where `.i >= start & .i <= stop` returns `TRUE`.

`.f`
: [function / formula]

  If a **function**, it is used as is.

  If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:

  - For a single argument function, use `.`
  - For a two argument function, use `.x` and `.y`
  - For more arguments, use `..1`, `..2`, `..3` etc

  This syntax allows you to create very compact anonymous functions.

`...`
: Additional arguments passed on to the mapped function.

`.ptype`
: [vector(0) / NULL]

  A prototype corresponding to the type of the output.

  If `NULL`, the default, the output type is determined by computing the common type across the results of the calls to `.f`.

  If supplied, the result of each call to `.f` will be cast to that type, and the final output will have that type.

  If `getOption("vctrs.no_guessing")` is `TRUE`, the `.ptype` must be supplied. This is a way to make production code demand fixed types.

`.l`
: [list]

  A list of vectors. The length of `.l` determines the number of arguments that `.f` will be called with. If `.l` has names, they will be used as named arguments to `.f`. Elements of `.l` with size 1 will be recycled.

Value

A vector fulfilling the following invariants:

`hop_index2()`:

- `vec_size(hop_index2(.x,.y,.starts,.stops)) == vec_size_common(.starts,.stops)`
- `vec_ptype(hop_index2(.x,.y,.starts,.stops)) == list()`

`hop_index2_vec()`:

- `vec_size(hop_index2_vec(.x,.y,.starts,.stops)) == vec_size_common(.starts,.stops)`
- `vec_size(hop_index2_vec(.x,.y,.starts,.stops)[[1]]) == 1L`
- `vec_ptype(hop_index2_vec(.x,.y,.starts,.stops,.ptype = ptype)) == ptype`

`phop_index()`:

- `vec_size(phop_index(.l,.starts,.stops)) == vec_size_common(.starts,.stops)`
- `vec_ptype(phop_index(.l,.starts,.stops)) == list()`

phop_index_vec():

- vec_size(phop_index_vec(.l,.starts,.stops)) == vec_size_common(.starts,.stops)
- vec_size(phop_index_vec(.l,.starts,.stops)[[1]]) == 1L
- vec_ptype(phop_index_vec(.l,.starts,.stops,.ptype = ptype)) == ptype

See Also

slide2(), slide_index2(), hop_index()

Examples

```
# Notice that `i` is an irregular index!
x <- 1:5
i <- as.Date("2019-08-15") + c(0:1, 4, 6, 7)

# Manually create starts/stops. They don't have to be equally spaced,
# and they don't have to be the same size as `.x` or `.i`.
starts <- as.Date(c("2019-08-15", "2019-08-18"))
stops <- as.Date(c("2019-08-16", "2019-08-23"))

# The output size is equal to the common size of `.starts` and `.stops`
hop_index2(x, i, i, starts, stops, ~data.frame(x = .x, y = .y))
```

---

slide *Slide*

Description

slide() iterates through .x using a sliding window, applying .f to each sub-window of .x.

Usage

```
slide(.x, .f, ..., .before = 0L, .after = 0L, .step = 1L, .complete = FALSE)

slide_vec(
  .x,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE,
  .ptype = NULL
)

slide_dbl(
  .x,
```

```
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

slide_int(
  .x,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

slide_lgl(
  .x,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

slide_chr(
  .x,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

slide_dfr(
  .x,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE,
  .names_to = rlang::zap(),
  .name_repair = c("unique", "universal", "check_unique")
```

```
)

slide_dfc(
  .x,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE,
  .size = NULL,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)
```

Arguments

`.x`	[vector] The vector to iterate over and apply `.f` to.
`.f`	[function / formula] If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions.
`...`	Additional arguments passed on to the mapped function.
`.before, .after`	[integer(1) / Inf] The number of values before or after the current element to include in the sliding window. Set to `Inf` to select all elements before or after the current element. Negative values are allowed, which allows you to "look forward" from the current element if used as the `.before` value, or "look backwards" if used as `.after`.
`.step`	[positive integer(1)] The number of elements to shift the window forward between function calls.
`.complete`	[logical(1)] Should the function be evaluated on complete windows only? If `FALSE`, the default, then partial computations will be allowed.
`.ptype`	[vector(0) / NULL] A prototype corresponding to the type of the output. If `NULL`, the default, the output type is determined by computing the common type across the results of the calls to `.f`. If supplied, the result of each call to `.f` will be cast to that type, and the final output will have that type. If `getOption("vctrs.no_guessing")` is `TRUE`, the `.ptype` must be supplied. This is a way to make production code demand fixed types.

`.names_to` This controls what to do with input names supplied in `...`.

- By default, input names are zapped.
- If a string, specifies a column where the input names will be copied. These names are often useful to identify rows with their original input. If a column name is supplied and `...` is not named, an integer column is used instead.
- If `NULL`, the input names are used as row names.

`.name_repair` One of `"unique"`, `"universal"`, or `"check_unique"`. See `vec_as_names()` for the meaning of these options.

With `vec_rbind()`, the repair function is applied to all inputs separately. This is because `vec_rbind()` needs to align their columns before binding the rows, and thus needs all inputs to have unique names. On the other hand, `vec_cbind()` applies the repair function after all inputs have been concatenated together in a final data frame. Hence `vec_cbind()` allows the more permissive minimal names repair.

`.size` If, `NULL`, the default, will determine the number of rows in `vec_cbind()` output by using the standard recycling rules.

Alternatively, specify the desired number of rows, and any inputs of length 1 will be recycled appropriately.

Details

Unlike `lapply()` or `purrr::map()`, which construct calls like

```
.f(.x[[i]], ...)
```

the equivalent with `slide()` looks like

```
.f(vctrs::vec_slice(.x, i), ...)
```

which is approximately

```
.f(.x[i], ...)
```

except in the case of data frames or arrays, which are iterated over row-wise.

If `.x` has names, then the output will preserve those names.

Using `vctrs::vec_cast()`, the output of `.f` will be automatically cast to the type required by the variant of slide_*() being used.

Value

A vector fulfilling the following invariants:

`slide()`**:**

- `vec_size(slide(.x)) == vec_size(.x)`
- `vec_ptype(slide(.x)) == list()`

`slide_vec()` **and** slide_*() **variants:**

- `vec_size(slide_vec(.x)) == vec_size(.x)`
- `vec_size(slide_vec(.x)[[1]]) == 1L`
- `vec_ptype(slide_vec(.x,.ptype = ptype)) == ptype`

See Also

slide2(), slide_index(), hop()

Examples

```
# The defaults work similarly to `map()`
slide(1:5, ~.x)

# Use `.before`, `.after`, and `.step` to control the window
slide(1:5, ~.x, .before = 1)

# This can be used for rolling means
slide_dbl(rnorm(5), mean, .before = 2)

# Or more flexible rolling operations
slide(rnorm(5), ~ .x - mean(.x), .before = 2)

# `.after` allows you to "align to the left" rather than the right
slide(1:5, ~.x, .after = 2)

# And a mixture of `.before` and `.after`
# allows you complete control over the exact alignment.
# Below, "center alignment" is used.
slide(1:5, ~.x, .before = 1, .after = 1)

# The `.step` controls how the window is shifted along `.x`,
# allowing you to "skip" iterations if you only need a less granular result
slide(1:10, ~.x, .before = 2, .step = 3)

# `.complete` controls whether or not partial results are computed.
# By default, they are, but setting `.complete = TRUE` restricts
# `slide()` to only evaluate the function where a complete window exists.
slide(1:5, ~.x, .before = 2, .after = 1)
slide(1:5, ~.x, .before = 2, .after = 1, .complete = TRUE)

# ------------------------------------------------------------------------------
# Data frames

# Data frames are iterated over rowwise
mtcars_rowwise <- slide(mtcars, ~.x)
mtcars_rowwise[1:3]

# This means that any column name is easily accessible
slide_dbl(mtcars, ~.x$mpg + .x$cyl)

# More advanced rowwise iteration is available as well by using the
# other arguments
mtcars_rowwise_window <- slide(mtcars, ~.x, .before = 1, .after = 1)
mtcars_rowwise_window[1:3]

# ------------------------------------------------------------------------------
# Cumulative sliding
```

```
# Using the special cased value, `Inf`, you can ask `slide()` to pin the
# start of the sliding window to the first element, effectively creating
# a cumulative window
slide(1:5, ~.x, .before = Inf)

# Same with `.after`, this creates a window where you start with all of the
# elements, but decrease the total number over each iteration
slide(1:5, ~.x, .after = Inf)

# ------------------------------------------------------------------------------
# Negative `.before` / `.after`

# `.before` is allowed to be negative, allowing you to "look forward" in
# your vector. Note that `abs(.before) <= .after` must hold if `.before` is
# negative. In this example, we look forward to elements in locations 2 and 3
# but place the result in position 1 in the output.
slide(1:5, ~.x, .before = -1, .after = 2)

# `.after` can be negative as well to "look backwards"
slide(1:5, ~.x, .before = 2, .after = -1)

# ------------------------------------------------------------------------------
# Removing padding

# If you are looking for a way to remove the `NA` values from something like
# this, then it doesn't exist as a built in option.
x <- rnorm(10)
slide_dbl(x, mean, .before = 3, .step = 2, .complete = TRUE)

# Adding an option to `slide_dbl()` to remove the `NA` values would destroy
# its size stability. Instead, you can use a combination of `slide_dfr()`
# to get the start/stop indices with `hop_index_vec()`.
i <- seq_along(x)
idx <- slide_dfr(
  i,
  ~data.frame(start = .x[1], stop = .x[length(.x)]),
  .before = 3,
  .step = 2,
  .complete = TRUE
)

idx

hop_index_vec(x, i, idx$start, idx$stop, mean, .ptype = double())
```

---

slide2 *Slide over multiple inputs simultaneously*

Description

These are variants of slide() that iterate over multiple inputs in parallel. They are parallel in the sense that each input is processed in parallel with the others, not in the sense of multicore computing. These functions work similarly to map2() and pmap() from purrr.

Usage

```
slide2(
  .x,
  .y,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

slide2_vec(
  .x,
  .y,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE,
  .ptype = NULL
)

slide2_dbl(
  .x,
  .y,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

slide2_int(
  .x,
  .y,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
```

```
  .step = 1L,
  .complete = FALSE
)

slide2_lgl(
  .x,
  .y,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

slide2_chr(
  .x,
  .y,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

slide2_dfr(
  .x,
  .y,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE,
  .names_to = rlang::zap(),
  .name_repair = c("unique", "universal", "check_unique")
)

slide2_dfc(
  .x,
  .y,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE,
```

```
  .size = NULL,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)

pslide(.l, .f, ..., .before = 0L, .after = 0L, .step = 1L, .complete = FALSE)

pslide_vec(
  .l,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE,
  .ptype = NULL
)

pslide_dbl(
  .l,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

pslide_int(
  .l,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

pslide_lgl(
  .l,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

pslide_chr(
```

```
  .l,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE
)

pslide_dfr(
  .l,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE,
  .names_to = rlang::zap(),
  .name_repair = c("unique", "universal", "check_unique")
)

pslide_dfc(
  .l,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .step = 1L,
  .complete = FALSE,
  .size = NULL,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)
```

Arguments

`.x, .y`	[vector] Vectors to iterate over. Vectors of size 1 will be recycled.
`.f`	[function / formula] If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions.
`...`	Additional arguments passed on to the mapped function.

`.before, .after`
: [integer(1) / Inf]
The number of values before or after the current element to include in the sliding window. Set to `Inf` to select all elements before or after the current element. Negative values are allowed, which allows you to "look forward" from the current element if used as the `.before` value, or "look backwards" if used as `.after`.

`.step`
: [positive integer(1)]
The number of elements to shift the window forward between function calls.

`.complete`
: [logical(1)]
Should the function be evaluated on complete windows only? If `FALSE`, the default, then partial computations will be allowed.

`.ptype`
: [vector(0) / NULL]
A prototype corresponding to the type of the output.
If `NULL`, the default, the output type is determined by computing the common type across the results of the calls to `.f`.
If supplied, the result of each call to `.f` will be cast to that type, and the final output will have that type.
If `getOption("vctrs.no_guessing")` is `TRUE`, the `.ptype` must be supplied. This is a way to make production code demand fixed types.

`.names_to`
: This controls what to do with input names supplied in `...`.
- By default, input names are zapped.
- If a string, specifies a column where the input names will be copied. These names are often useful to identify rows with their original input. If a column name is supplied and `...` is not named, an integer column is used instead.
- If `NULL`, the input names are used as row names.

`.name_repair`
: One of `"unique"`, `"universal"`, or `"check_unique"`. See `vec_as_names()` for the meaning of these options.
With `vec_rbind()`, the repair function is applied to all inputs separately. This is because `vec_rbind()` needs to align their columns before binding the rows, and thus needs all inputs to have unique names. On the other hand, `vec_cbind()` applies the repair function after all inputs have been concatenated together in a final data frame. Hence `vec_cbind()` allows the more permissive minimal names repair.

`.size`
: If, `NULL`, the default, will determine the number of rows in `vec_cbind()` output by using the standard recycling rules.
Alternatively, specify the desired number of rows, and any inputs of length 1 will be recycled appropriately.

`.l`
: [list]
A list of vectors. The length of `.l` determines the number of arguments that `.f` will be called with. If `.l` has names, they will be used as named arguments to `.f`. Elements of `.l` with size 1 will be recycled.

Value

A vector fulfilling the following invariants:

`slide2()`:

- `vec_size(slide2(.x, .y)) == vec_size_common(.x, .y)`
- `vec_ptype(slide2(.x, .y)) == list()`

`slide2_vec()` **and** slide2_*() **variants:**

- `vec_size(slide2_vec(.x, .y)) == vec_size_common(.x, .y)`
- `vec_size(slide2_vec(.x, .y)[[1]]) == 1L`
- `vec_ptype(slide2_vec(.x, .y, .ptype = ptype)) == ptype`

`pslide()`:

- `vec_size(pslide(.l)) == vec_size_common(!!! .l)`
- `vec_ptype(pslide(.l)) == list()`

`pslide_vec()` **and** pslide_*() **variants:**

- `vec_size(pslide_vec(.l)) == vec_size_common(!!! .l)`
- `vec_size(pslide_vec(.l)[[1]]) == 1L`
- `vec_ptype(pslide_vec(.l, .ptype = ptype)) == ptype`

See Also

`slide()`, `slide_index2()`, `hop_index2()`

Examples

```
# Slide along two inputs at once
slide2(1:4, 5:8, ~list(.x, .y), .before = 2)

# Or, for more than two, use `pslide()`
pslide(list(1:4, 5:8, 9:12), ~list(.x, .y, ..3), .before = 2)

# You can even slide along the rows of multiple data frames of
# equal size at once
set.seed(16)
x <- data.frame(a = rnorm(5), b = rnorm(5))
y <- data.frame(c = letters[1:5], d = letters[6:10])

row_return <- function(x_rows, y_rows) {
  if (sum(x_rows$a) < 0) {
    x_rows
  } else {
    y_rows
  }
}

slide2(x, y, row_return, .before = 1, .after = 2)
```

slide_index *Slide relative to an index*

Description

slide_index() is similar to slide(), but allows a secondary .i-ndex vector to be provided.

This is often useful in business calculations, when you want to compute a rolling computation looking "3 months back", which is approximately but not equivalent to, 3 * 30 days. slide_index() allows for these irregular window sizes.

Usage

```
slide_index(.x, .i, .f, ..., .before = 0L, .after = 0L, .complete = FALSE)

slide_index_vec(
  .x,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .ptype = NULL
)

slide_index_dbl(.x, .i, .f, ..., .before = 0L, .after = 0L, .complete = FALSE)

slide_index_int(.x, .i, .f, ..., .before = 0L, .after = 0L, .complete = FALSE)

slide_index_lgl(.x, .i, .f, ..., .before = 0L, .after = 0L, .complete = FALSE)

slide_index_chr(.x, .i, .f, ..., .before = 0L, .after = 0L, .complete = FALSE)

slide_index_dfr(
  .x,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .names_to = rlang::zap(),
  .name_repair = c("unique", "universal", "check_unique")
)

slide_index_dfc(
```

```
  .x,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .size = NULL,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)
```

Arguments

`.x` [vector]

The vector to iterate over and apply `.f` to.

`.i` [vector]

The index vector that determines the window sizes. It is fairly common to supply a date vector as the index, but not required.

There are 3 restrictions on the index:

- The size of the index must match the size of `.x`, they will not be recycled to their common size.
- The index must be an *increasing* vector, but duplicate values are allowed.
- The index cannot have missing values.

`.f` [function / formula]

If a **function**, it is used as is.

If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:

- For a single argument function, use `.`
- For a two argument function, use `.x` and `.y`
- For more arguments, use `..1`, `..2`, `..3` etc

This syntax allows you to create very compact anonymous functions.

`...` Additional arguments passed on to the mapped function.

`.before, .after`

[vector(1) / function / Inf]

- If a vector of size 1, these represent the number of values before or after the current element of `.i` to include in the sliding window. Negative values are allowed, which allows you to "look forward" from the current element if used as the `.before` value, or "look backwards" if used as `.after`. Boundaries are computed from these elements as `.i - .before` and `.i + .after`. Any object that can be added or subtracted from `.i` with `+` and `-` can be used. For example, a lubridate period, such as `lubridate::weeks()`.
- If `Inf`, this selects all elements before or after the current element.
- If a function, or a one-sided formula which can be coerced to a function, it is applied to `.i` to compute the boundaries. Note that this function will only be applied to the *unique* values of `.i`, so it should not rely on the original length

of `.i` in any way. This is useful for applying a complex arithmetic operation that can't be expressed with a single `-` or `+` operation. One example would be to use `lubridate::add_with_rollback()` to avoid invalid dates at the end of the month.

The ranges that result from applying `.before` and `.after` have the same 3 restrictions as `.i` itself.

`.complete` [logical(1)]

Should the function be evaluated on complete windows only? If `FALSE`, the default, then partial computations will be allowed.

`.ptype` [vector(0) / NULL]

A prototype corresponding to the type of the output.

If `NULL`, the default, the output type is determined by computing the common type across the results of the calls to `.f`.

If supplied, the result of each call to `.f` will be cast to that type, and the final output will have that type.

If `getOption("vctrs.no_guessing")` is `TRUE`, the `.ptype` must be supplied. This is a way to make production code demand fixed types.

`.names_to` This controls what to do with input names supplied in `...`.

- By default, input names are zapped.
- If a string, specifies a column where the input names will be copied. These names are often useful to identify rows with their original input. If a column name is supplied and `...` is not named, an integer column is used instead.
- If `NULL`, the input names are used as row names.

`.name_repair` One of `"unique"`, `"universal"`, or `"check_unique"`. See `vec_as_names()` for the meaning of these options.

With `vec_rbind()`, the repair function is applied to all inputs separately. This is because `vec_rbind()` needs to align their columns before binding the rows, and thus needs all inputs to have unique names. On the other hand, `vec_cbind()` applies the repair function after all inputs have been concatenated together in a final data frame. Hence `vec_cbind()` allows the more permissive minimal names repair.

`.size` If, `NULL`, the default, will determine the number of rows in `vec_cbind()` output by using the standard recycling rules.

Alternatively, specify the desired number of rows, and any inputs of length 1 will be recycled appropriately.

Value

A vector fulfilling the following invariants:

`slide_index()`**:**

- `vec_size(slide_index(.x)) == vec_size(.x)`
- `vec_ptype(slide_index(.x)) == list()`

`slide_index_vec()` **and** slide_index_*() **variants:**

- `vec_size(slide_index_vec(.x)) == vec_size(.x)`
- `vec_size(slide_index_vec(.x)[[1]]) == 1L`
- `vec_ptype(slide_index_vec(.x, .ptype = ptype)) == ptype`

See Also

`slide()`, `hop_index()`, `slide_index2()`

Examples

```
library(lubridate)

x <- 1:5

# In some cases, sliding over `x` with a strict window size of 2
# will fit your use case.
slide(x, ~.x, .before = 1)

# However, if this `i` is a date vector paired with `x`, when computing
# rolling calculations you might want to iterate over `x` while
# respecting the fact that `i` is an irregular sequence.
i <- as.Date("2019-08-15") + c(0:1, 4, 6, 7)

# For example, a "2 day" window should not pair `"2019-08-19"` and
# `"2019-08-21"` together, even though they are next to each other in `x`.
# `slide_index()` computes the lookback value from the current date in `.i`,
# meaning that if you are currently on `"2019-08-21"` and look back 1 day,
# it will correctly not include `"2019-08-19"`.
slide_index(i, i, ~.x, .before = 1)

# We could have equivalently used a lubridate period object for this as well,
# since `i - lubridate::days(1)` is allowed
slide_index(i, i, ~.x, .before = lubridate::days(1))

# ------------------------------------------------------------------------------
# Functions for `.before` and `.after`

# In some cases, it might not be appropriate to compute
# `.i - .before` or `.i + .after`, either because there isn't a `-` or `+`
# method defined, or because there is an alternative way to perform the
# arithmetic. For example, subtracting 1 month with `- months(1)` (using
# lubridate) can sometimes land you on an invalid date that doesn't exist.
i <- as.Date(c("2019-01-31", "2019-02-28", "2019-03-31"))

# 2019-03-31 - months(1) = 2019-02-31, which doesn't exist
i - months(1)

# These NAs create problems with `slide_index()`, which doesn't allow
# missing values in the computed endpoints
try(slide_index(i, i, identity, .before = months(1)))

# In these cases, it is more appropriate to use `%m-%`,
```

```
# which will snap to the end of the month, at least giving you something
# to work with.
i %m-% months(1)

# To use this as your `.before` or `.after`, supply an anonymous function of
# 1 argument that performs the computation
slide_index(i, i, identity, .before = ~.x %m-% months(1))

# Notice that in the `.after` case, `2019-02-28 %m+% months(1)` doesn't
# capture the end of March, so it isn't included in the 2nd result
slide_index(i, i, identity, .after = ~.x %m+% months(1))

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

# When `.i` has repeated values, they are always grouped together.
i <- c(2017, 2017, 2018, 2019, 2020, 2020)
slide_index(i, i, ~.x)
slide_index(i, i, ~.x, .after = 1)

# ------------------------------------------------------------------------------
# Rolling regressions

# Rolling regressions are easy with `slide_index()` because:
# - Data frame `.x` values are iterated over rowwise
# - The index is respected by using `.i`
set.seed(123)

df <- data.frame(
  y = rnorm(100),
  x = rnorm(100),
  i = as.Date("2019-08-15") + c(0, 2, 4, 6:102) # <- irregular
)

# 20 day rolling regression. Current day + 19 days back.
# Additionally, set `.complete = TRUE` to not compute partial results.
regr <- slide_index(df, df$i, ~lm(y ~ x, .x), .before = 19, .complete = TRUE)

regr[16:18]

# The first 16 slots are NULL because there is no possible way to
# look back 19 days from the 16th index position and construct a full
# window. But on the 17th index position, `""2019-09-03"`, if we look
# back 19 days we get to `""2019-08-15"`, which is the same value as
# `i[1]` so a full window can be constructed.
df$i[16] - 19 >= df$i[1] # FALSE
df$i[17] - 19 >= df$i[1] # TRUE

# ------------------------------------------------------------------------------
# Accessing the current index value

# A very simplistic version of `purrr::map2()`
fake_map2 <- function(.x, .y, .f, ...) {
  Map(.f, .x, .y, ...)
```

```
}

# Occasionally you need to access the index value that you are currently on.
# This is generally not possible with a single call to `slide_index()`, but
# can be easily accomplished by following up a `slide_index()` call with a
# `purrr::map2()`. In this example, we want to use the distance from the
# current index value (in days) as a multiplier on `x`. Values further
# away from the current date get a higher multiplier.
set.seed(123)

# 25 random days past 2000-01-01
i <- sort(as.Date("2000-01-01") + sample(100, 25))

df <- data.frame(i = i, x = rnorm(25))

weight_by_distance <- function(df, i) {
  df$weight = abs(as.integer(df$i - i))
  df$x_weighted = df$x * df$weight
  df
}

# Use `slide_index()` to just generate the rolling data.
# Here we take the current date + 5 days before + 5 days after.
dfs <- slide_index(df, df$i, ~.x, .before = 5, .after = 5)

# Follow up with a `map2()` with `i` as the second input.
# This allows you to track the current `i` value and weight accordingly.
result <- fake_map2(dfs, df$i, weight_by_distance)

head(result)
```

`slide_index2` *Slide along multiples inputs simultaneously relative to an index*

Description

`slide_index2()` and `pslide_index()` represent the combination of `slide2()` and `pslide()` with `slide_index()`, allowing you to iterate over multiple vectors at once relative to an `.i`-ndex.

Usage

```
slide_index2(.x, .y, .i, .f, ..., .before = 0L, .after = 0L, .complete = FALSE)

slide_index2_vec(
  .x,
  .y,
  .i,
  .f,
```

```
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .ptype = NULL
)

slide_index2_dbl(
  .x,
  .y,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_index2_int(
  .x,
  .y,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_index2_lgl(
  .x,
  .y,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_index2_chr(
  .x,
  .y,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
```

```
  .complete = FALSE
)

slide_index2_dfr(
  .x,
  .y,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .names_to = rlang::zap(),
  .name_repair = c("unique", "universal", "check_unique")
)

slide_index2_dfc(
  .x,
  .y,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .size = NULL,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)

pslide_index(.l, .i, .f, ..., .before = 0L, .after = 0L, .complete = FALSE)

pslide_index_vec(
  .l,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .ptype = NULL
)

pslide_index_dbl(.l, .i, .f, ..., .before = 0L, .after = 0L, .complete = FALSE)

pslide_index_int(.l, .i, .f, ..., .before = 0L, .after = 0L, .complete = FALSE)

pslide_index_lgl(.l, .i, .f, ..., .before = 0L, .after = 0L, .complete = FALSE)
```

```
pslide_index_chr(.l, .i, .f, ..., .before = 0L, .after = 0L, .complete = FALSE)

pslide_index_dfr(
  .l,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .names_to = rlang::zap(),
  .name_repair = c("unique", "universal", "check_unique")
)

pslide_index_dfc(
  .l,
  .i,
  .f,
  ...,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .size = NULL,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)
```

Arguments

`.x, .y` [vector]

Vectors to iterate over. Vectors of size 1 will be recycled.

`.i` [vector]

The index vector that determines the window sizes. It is fairly common to supply a date vector as the index, but not required.

There are 3 restrictions on the index:

- The size of the index must match the size of `.x`, they will not be recycled to their common size.
- The index must be an *increasing* vector, but duplicate values are allowed.
- The index cannot have missing values.

`.f` [function / formula]

If a **function**, it is used as is.

If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:

- For a single argument function, use `.`
- For a two argument function, use `.x` and `.y`
- For more arguments, use `..1`, `..2`, `..3` etc

This syntax allows you to create very compact anonymous functions.

`...` Additional arguments passed on to the mapped function.

`.before, .after`

[vector(1) / function / Inf]

- If a vector of size 1, these represent the number of values before or after the current element of `.i` to include in the sliding window. Negative values are allowed, which allows you to "look forward" from the current element if used as the `.before` value, or "look backwards" if used as `.after`. Boundaries are computed from these elements as `.i - .before` and `.i + .after`. Any object that can be added or subtracted from `.i` with `+` and `-` can be used. For example, a lubridate period, such as `lubridate::weeks()`.
- If `Inf`, this selects all elements before or after the current element.
- If a function, or a one-sided formula which can be coerced to a function, it is applied to `.i` to compute the boundaries. Note that this function will only be applied to the *unique* values of `.i`, so it should not rely on the original length of `.i` in any way. This is useful for applying a complex arithmetic operation that can't be expressed with a single `-` or `+` operation. One example would be to use `lubridate::add_with_rollback()` to avoid invalid dates at the end of the month.

The ranges that result from applying `.before` and `.after` have the same 3 restrictions as `.i` itself.

`.complete` [logical(1)]

Should the function be evaluated on complete windows only? If `FALSE`, the default, then partial computations will be allowed.

`.ptype` [vector(0) / NULL]

A prototype corresponding to the type of the output.

If `NULL`, the default, the output type is determined by computing the common type across the results of the calls to `.f`.

If supplied, the result of each call to `.f` will be cast to that type, and the final output will have that type.

If `getOption("vctrs.no_guessing")` is `TRUE`, the `.ptype` must be supplied. This is a way to make production code demand fixed types.

`.names_to` This controls what to do with input names supplied in `...`.

- By default, input names are zapped.
- If a string, specifies a column where the input names will be copied. These names are often useful to identify rows with their original input. If a column name is supplied and `...` is not named, an integer column is used instead.
- If `NULL`, the input names are used as row names.

`.name_repair` One of `"unique"`, `"universal"`, or `"check_unique"`. See `vec_as_names()` for the meaning of these options.

With `vec_rbind()`, the repair function is applied to all inputs separately. This is because `vec_rbind()` needs to align their columns before binding the rows, and thus needs all inputs to have unique names. On the other hand, `vec_cbind()` applies the repair function after all inputs have been concatenated together in a final data frame. Hence `vec_cbind()` allows the more permissive minimal names repair.

.size	If, NULL, the default, will determine the number of rows in vec_cbind() output by using the standard recycling rules. Alternatively, specify the desired number of rows, and any inputs of length 1 will be recycled appropriately.
.l	[list] A list of vectors. The length of .l determines the number of arguments that .f will be called with. If .l has names, they will be used as named arguments to .f. Elements of .l with size 1 will be recycled.

Value

A vector fulfilling the following invariants:

`slide_index2()`**:**

- `vec_size(slide_index2(.x, .y)) == vec_size_common(.x, .y)`
- `vec_ptype(slide_index2(.x, .y)) == list()`

`slide_index2_vec()` **and** slide_index2_*() **variants:**

- `vec_size(slide_index2_vec(.x, .y)) == vec_size_common(.x, .y)`
- `vec_size(slide_index2_vec(.x, .y)[[1]]) == 1L`
- `vec_ptype(slide_index2_vec(.x, .y, .ptype = ptype)) == ptype`

`pslide_index()`**:**

- `vec_size(pslide_index(.l)) == vec_size_common(!!! .l)`
- `vec_ptype(pslide_index(.l)) == list()`

`pslide_index_vec()` **and** pslide_index_*() **variants:**

- `vec_size(pslide_index_vec(.l)) == vec_size_common(!!! .l)`
- `vec_size(pslide_index_vec(.l)[[1]]) == 1L`
- `vec_ptype(pslide_index_vec(.l, .ptype = ptype)) == ptype`

See Also

slide2(), hop_index2(), slide_index()

Examples

```
# Notice that `i` is an irregular index!
x <- 1:5
y <- 6:10
i <- as.Date("2019-08-15") + c(0:1, 4, 6, 7)

# When we slide over `i` looking back 1 day, the irregularity is respected.
# When there is a gap in dates, only 2 values are returned (one from
# `x` and one from `y`), otherwise, 4 values are returned.
slide_index2(x, y, i, ~c(.x, .y), .before = 1)
```

slide_period *Slide relative to an index chunked by period*

Description

`slide_period()` breaks up the `.i`-ndex by `.period`, and then uses that to define the indices to slide over `.x` with.

It can be useful for, say, sliding over daily data in monthly chunks.

The underlying engine for breaking up `.i` is `warp::warp_distance()`. If you need more information about the `.period` types, that is the best place to look.

Usage

```
slide_period(
  .x,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_period_vec(
  .x,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .ptype = NULL
)

slide_period_dbl(
  .x,
  .i,
  .period,
  .f,
  ...,
```

```
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_period_int(
  .x,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_period_lgl(
  .x,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_period_chr(
  .x,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_period_dfr(
  .x,
```

```
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .names_to = rlang::zap(),
  .name_repair = c("unique", "universal", "check_unique")
)

slide_period_dfc(
  .x,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .size = NULL,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)
```

Arguments

`.x` [vector]

The vector to iterate over and apply `.f` to.

`.i` [Date / POSIXct / POSIXlt]

A datetime index to break into periods.

There are 3 restrictions on the index:

- The size of the index must match the size of `.x`, they will not be recycled to their common size.
- The index must be an *increasing* vector, but duplicate values are allowed.
- The index cannot have missing values.

`.period` [character(1)]

A string defining the period to group by. Valid inputs can be roughly broken into:

- `"year"`, `"quarter"`, `"month"`, `"week"`, `"day"`
- `"hour"`, `"minute"`, `"second"`, `"millisecond"`
- `"yweek"`, `"mweek"`
- `"yday"`, `"mday"`

`.f` [function / formula]

If a **function**, it is used as is.

If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:

- For a single argument function, use `.`
- For a two argument function, use `.x` and `.y`
- For more arguments, use `..1`, `..2`, `..3` etc

This syntax allows you to create very compact anonymous functions.

`...` Additional arguments passed on to the mapped function.

`.every` [positive integer(1)]

The number of periods to group together.

For example, if the period was set to `"year"` with an every value of `2`, then the years 1970 and 1971 would be placed in the same group.

`.origin` [Date(1) / POSIXct(1) / POSIXlt(1) / NULL]

The reference date time value. The default when left as `NULL` is the epoch time of 1970-01-01 00:00:00, *in the time zone of the index*.

This is generally used to define the anchor time to count from, which is relevant when the every value is > 1.

`.before, .after`

[integer(1) / Inf]

The number of values before or after the current element to include in the sliding window. Set to `Inf` to select all elements before or after the current element. Negative values are allowed, which allows you to "look forward" from the current element if used as the `.before` value, or "look backwards" if used as `.after`.

`.complete` [logical(1)]

Should the function be evaluated on complete windows only? If `FALSE`, the default, then partial computations will be allowed.

`.ptype` [vector(0) / NULL]

A prototype corresponding to the type of the output.

If `NULL`, the default, the output type is determined by computing the common type across the results of the calls to `.f`.

If supplied, the result of each call to `.f` will be cast to that type, and the final output will have that type.

If `getOption("vctrs.no_guessing")` is `TRUE`, the `.ptype` must be supplied. This is a way to make production code demand fixed types.

`.names_to` This controls what to do with input names supplied in `...`.

- By default, input names are zapped.
- If a string, specifies a column where the input names will be copied. These names are often useful to identify rows with their original input. If a column name is supplied and `...` is not named, an integer column is used instead.
- If `NULL`, the input names are used as row names.

`.name_repair` One of `"unique"`, `"universal"`, or `"check_unique"`. See `vec_as_names()` for the meaning of these options.

With `vec_rbind()`, the repair function is applied to all inputs separately. This is because `vec_rbind()` needs to align their columns before binding the rows, and thus needs all inputs to have unique names. On the other hand, `vec_cbind()` applies the repair function after all inputs have been concatenated together in a final data frame. Hence `vec_cbind()` allows the more permissive minimal names repair.

`.size` If, `NULL`, the default, will determine the number of rows in `vec_cbind()` output by using the standard recycling rules.

Alternatively, specify the desired number of rows, and any inputs of length 1 will be recycled appropriately.

Value

A vector fulfilling the following invariants:

`slide_period()`**:**

- `vec_size(slide_period(.x)) == vec_size(unique(warp::warp_distance(.i)))`
- `vec_ptype(slide_period(.x)) == list()`

`slide_period_vec()` **and** slide_period_*() **variants:**

- `vec_size(slide_period_vec(.x)) == vec_size(unique(warp::warp_distance(.i)))`
- `vec_size(slide_period_vec(.x)[[1]]) == 1L`
- `vec_ptype(slide_period_vec(.x, .ptype = ptype)) == ptype`

See Also

`block()`, `slide()`, `slide_index()`

Examples

```
i <- as.Date("2019-01-28") + 0:5

# Split `i` into 2-day periods to apply `.f` to
slide_period(i, i, "day", identity, .every = 2)

# Or into 1-month periods
slide_period(i, i, "month", identity)

# Now select:
# - The current 2-day period
# - Plus 1 2-day period before the current one
slide_period(i, i, "day", identity, .every = 2, .before = 1)

# Alter the `origin` to control the reference date for
# how the 2-day groups are formed
origin <- as.Date("2019-01-29")
slide_period(i, i, "day", identity, .every = 2, .origin = origin)
```

```
# This can be useful for, say, monthly averages
daily_sales <- c(2, 5, 3, 6, 9, 4)
slide_period_dbl(daily_sales, i, "month", mean)

# If you need the index, slide over and return a data frame
sales_df <- data.frame(i = i, sales = daily_sales)

slide_period_dfr(
  sales_df,
  sales_df$i,
  "month",
  ~data.frame(
     i = max(.x$i),
     sales = mean(.x$sales)
   )
)

# One of the most unique features about `slide_period()` is that it is
# aware of how far apart elements of `.i` are in the `.period` you are
# interested in. For example, if you do a monthly slide with `i2`, selecting
# the current month and 1 month before it, then it will recognize that
# `2019-02-01` and `2019-04-01` are not beside each other, and it won't
# group them together.
i2 <- as.Date(c("2019-01-01", "2019-02-01", "2019-04-01", "2019-05-01"))

slide_period(i2, i2, "month", identity, .before = 1)
```

`slide_period2` *Slide along multiple inputs simultaneously relative to an index chunked by period*

Description

`slide_period2()` and `pslide_period()` represent the combination of `slide2()` and `pslide()` with `slide_period()`, allowing you to slide over multiple vectors at once, using indices defined by breaking up the `.i`-ndex by `.period`.

Usage

```
slide_period2(
  .x,
  .y,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
```

```
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_period2_vec(
  .x,
  .y,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .ptype = NULL
)

slide_period2_dbl(
  .x,
  .y,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_period2_int(
  .x,
  .y,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)
```

```
slide_period2_lgl(
  .x,
  .y,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_period2_chr(
  .x,
  .y,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

slide_period2_dfr(
  .x,
  .y,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .names_to = rlang::zap(),
  .name_repair = c("unique", "universal", "check_unique")
)

slide_period2_dfc(
  .x,
  .y,
```

```
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .size = NULL,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)

pslide_period(
  .l,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

pslide_period_vec(
  .l,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .ptype = NULL
)

pslide_period_dbl(
  .l,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
```

```
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

pslide_period_int(
  .l,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

pslide_period_lgl(
  .l,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

pslide_period_chr(
  .l,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE
)

pslide_period_dfr(
  .l,
  .i,
  .period,
```

```
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .names_to = rlang::zap(),
  .name_repair = c("unique", "universal", "check_unique")
)

pslide_period_dfc(
  .l,
  .i,
  .period,
  .f,
  ...,
  .every = 1L,
  .origin = NULL,
  .before = 0L,
  .after = 0L,
  .complete = FALSE,
  .size = NULL,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)
```

Arguments

`.x, .y`	[vector] Vectors to iterate over. Vectors of size 1 will be recycled.
`.i`	[Date / POSIXct / POSIXlt] A datetime index to break into periods. There are 3 restrictions on the index: • The size of the index must match the size of `.x`, they will not be recycled to their common size. • The index must be an *increasing* vector, but duplicate values are allowed. • The index cannot have missing values.
`.period`	[character(1)] A string defining the period to group by. Valid inputs can be roughly broken into: • `"year"`, `"quarter"`, `"month"`, `"week"`, `"day"` • `"hour"`, `"minute"`, `"second"`, `"millisecond"` • `"yweek"`, `"mweek"` • `"yday"`, `"mday"`
`.f`	[function / formula] If a **function**, it is used as is.

If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:

- For a single argument function, use `.`
- For a two argument function, use `.x` and `.y`
- For more arguments, use `..1`, `..2`, `..3` etc

This syntax allows you to create very compact anonymous functions.

`...` Additional arguments passed on to the mapped function.

`.every` [positive integer(1)]

The number of periods to group together.

For example, if the period was set to `"year"` with an every value of `2`, then the years 1970 and 1971 would be placed in the same group.

`.origin` [Date(1) / POSIXct(1) / POSIXlt(1) / NULL]

The reference date time value. The default when left as `NULL` is the epoch time of 1970-01-01 00:00:00, *in the time zone of the index*.

This is generally used to define the anchor time to count from, which is relevant when the every value is > 1.

`.before` [integer(1) / Inf]

The number of values before or after the current element to include in the sliding window. Set to `Inf` to select all elements before or after the current element. Negative values are allowed, which allows you to "look forward" from the current element if used as the `.before` value, or "look backwards" if used as `.after`.

`.after` [integer(1) / Inf]

The number of values before or after the current element to include in the sliding window. Set to `Inf` to select all elements before or after the current element. Negative values are allowed, which allows you to "look forward" from the current element if used as the `.before` value, or "look backwards" if used as `.after`.

`.complete` [logical(1)]

Should the function be evaluated on complete windows only? If `FALSE`, the default, then partial computations will be allowed.

`.ptype` [vector(0) / NULL]

A prototype corresponding to the type of the output.

If `NULL`, the default, the output type is determined by computing the common type across the results of the calls to `.f`.

If supplied, the result of each call to `.f` will be cast to that type, and the final output will have that type.

If `getOption("vctrs.no_guessing")` is `TRUE`, the `.ptype` must be supplied. This is a way to make production code demand fixed types.

`.names_to` This controls what to do with input names supplied in `...`.

- By default, input names are zapped.
- If a string, specifies a column where the input names will be copied. These names are often useful to identify rows with their original input. If a column name is supplied and `...` is not named, an integer column is used instead.

- If NULL, the input names are used as row names.

`.name_repair` One of "unique", "universal", or "check_unique". See vec_as_names() for the meaning of these options.

With vec_rbind(), the repair function is applied to all inputs separately. This is because vec_rbind() needs to align their columns before binding the rows, and thus needs all inputs to have unique names. On the other hand, vec_cbind() applies the repair function after all inputs have been concatenated together in a final data frame. Hence vec_cbind() allows the more permissive minimal names repair.

`.size` If, NULL, the default, will determine the number of rows in vec_cbind() output by using the standard recycling rules.

Alternatively, specify the desired number of rows, and any inputs of length 1 will be recycled appropriately.

`.l` [list]

A list of vectors. The length of .l determines the number of arguments that .f will be called with. If .l has names, they will be used as named arguments to .f. Elements of .l with size 1 will be recycled.

Value

A vector fulfilling the following invariants:

`slide_period2()`:

- `vec_size(slide_period2(.x, .y)) == vec_size(unique(warp::warp_distance(.i)))`
- `vec_ptype(slide_period2(.x, .y)) == list()`

`slide_period2_vec()` **and** slide_period2_*() **variants:**

- `vec_size(slide_period2_vec(.x, .y)) == vec_size(unique(warp::warp_distance(.i)))`
- `vec_size(slide_period2_vec(.x, .y)[[1]]) == 1L`
- `vec_ptype(slide_period2_vec(.x, .y, .ptype = ptype)) == ptype`

`pslide_period()`:

- `vec_size(pslide_period(.l)) == vec_size(unique(warp::warp_distance(.i)))`
- `vec_ptype(pslide_period(.l)) == list()`

`pslide_period_vec()` **and** pslide_period_*() **variants:**

- `vec_size(pslide_period_vec(.l)) == vec_size(unique(warp::warp_distance(.i)))`
- `vec_size(pslide_period_vec(.l)[[1]]) == 1L`
- `vec_ptype(pslide_period_vec(.l, .ptype = ptype)) == ptype`

See Also

slide2(), slide_index2(), slide_period()

Examples

```
i <- as.Date("2019-01-28") + 0:5

slide_period2(
  .x = 1:6,
  .y = i,
  .i = i,
  .period = "month",
  .f = ~data.frame(x = .x, i = .y)
)

pslide_period(
  .l = list(1:6, 7:12, i),
  .i = i,
  .period = "month",
  .f = ~data.frame(x = .x, y = .y, i = ..3)
)
```

---

summary-index *Specialized sliding functions relative to an index*

Description

These functions are specialized variants of the most common ways that `slide_index()` is generally used. Notably, `slide_index_sum()` can be used for rolling sums relative to an index (like a Date column), and `slide_index_mean()` can be used for rolling averages.

These specialized variants are *much* faster and more memory efficient than using an otherwise equivalent call constructed with `slide_index_dbl()` or `slide_index_lgl()`, especially with a very wide window.

Usage

```
slide_index_sum(
  x,
  i,
  ...,
  before = 0L,
  after = 0L,
  complete = FALSE,
  na_rm = FALSE
)

slide_index_prod(
  x,
  i,
  ...,
  before = 0L,
```

```
  after = 0L,
  complete = FALSE,
  na_rm = FALSE
)

slide_index_mean(
  x,
  i,
  ...,
  before = 0L,
  after = 0L,
  complete = FALSE,
  na_rm = FALSE
)

slide_index_min(
  x,
  i,
  ...,
  before = 0L,
  after = 0L,
  complete = FALSE,
  na_rm = FALSE
)

slide_index_max(
  x,
  i,
  ...,
  before = 0L,
  after = 0L,
  complete = FALSE,
  na_rm = FALSE
)

slide_index_all(
  x,
  i,
  ...,
  before = 0L,
  after = 0L,
  complete = FALSE,
  na_rm = FALSE
)

slide_index_any(
  x,
  i,
```

```
  ...,
  before = 0L,
  after = 0L,
  complete = FALSE,
  na_rm = FALSE
)
```

Arguments

x	[vector] A vector to compute the sliding function on. • For sliding sum, mean, prod, min, and max, x will be cast to a double vector with vctrs::vec_cast(). • For sliding any and all, x will be cast to a logical vector with vctrs::vec_cast().
i	[vector] The index vector that determines the window sizes. It is fairly common to supply a date vector as the index, but not required. There are 3 restrictions on the index: • The size of the index must match the size of .x, they will not be recycled to their common size. • The index must be an *increasing* vector, but duplicate values are allowed. • The index cannot have missing values.
...	These dots are for future extensions and must be empty.
before	[vector(1) / function / Inf] • If a vector of size 1, these represent the number of values before or after the current element of .i to include in the sliding window. Negative values are allowed, which allows you to "look forward" from the current element if used as the .before value, or "look backwards" if used as .after. Boundaries are computed from these elements as .i - .before and .i + .after. Any object that can be added or subtracted from .i with + and - can be used. For example, a lubridate period, such as lubridate::weeks(). • If Inf, this selects all elements before or after the current element. • If a function, or a one-sided formula which can be coerced to a function, it is applied to .i to compute the boundaries. Note that this function will only be applied to the *unique* values of .i, so it should not rely on the original length of .i in any way. This is useful for applying a complex arithmetic operation that can't be expressed with a single - or + operation. One example would be to use lubridate::add_with_rollback() to avoid invalid dates at the end of the month. The ranges that result from applying .before and .after have the same 3 restrictions as .i itself.
after	[vector(1) / function / Inf] • If a vector of size 1, these represent the number of values before or after the current element of .i to include in the sliding window. Negative values are allowed, which allows you to "look forward" from the current element if

used as the .before value, or "look backwards" if used as .after. Boundaries are computed from these elements as .i - .before and .i + .after. Any object that can be added or subtracted from .i with + and - can be used. For example, a lubridate period, such as lubridate::weeks().

- If Inf, this selects all elements before or after the current element.
- If a function, or a one-sided formula which can be coerced to a function, it is applied to .i to compute the boundaries. Note that this function will only be applied to the *unique* values of .i, so it should not rely on the original length of .i in any way. This is useful for applying a complex arithmetic operation that can't be expressed with a single - or + operation. One example would be to use lubridate::add_with_rollback() to avoid invalid dates at the end of the month.

The ranges that result from applying .before and .after have the same 3 restrictions as .i itself.

complete [logical(1)]

Should the function be evaluated on complete windows only? If FALSE, the default, then partial computations will be allowed.

na_rm [logical(1)]

Should missing values be removed from the computation?

Details

For more details about the implementation, see the help page of slide_sum().

Value

A vector the same size as x containing the result of applying the summary function over the sliding windows.

- For sliding sum, mean, prod, min, and max, a double vector will be returned.
- For sliding any and all, a logical vector will be returned.

See Also

slide_sum()

Examples

```
x <- c(1, 5, 3, 2, 6, 10)
i <- as.Date("2019-01-01") + c(0, 1, 3, 4, 6, 8)

# `slide_index_sum()` can be used for rolling sums relative to an index,
# allowing you to "respect gaps" in your series. Notice that the rolling
# sum in row 3 is only computed from `2019-01-04` and `2019-01-02` since
# `2019-01-01` is more than two days before the current date.
data.frame(
  i = i,
  x = x,
  roll = slide_index_sum(x, i, before = 2)
```

```
)

# `slide_index_mean()` can be used for rolling averages
slide_index_mean(x, i, before = 2)

# Only evaluate the sum on windows that have the potential to be complete
slide_index_sum(x, i, before = 2, after = 1, complete = TRUE)
```

---

summary-slide *Specialized sliding functions*

Description

These functions are specialized variants of the most common ways that `slide()` is generally used. Notably, `slide_sum()` can be used for rolling sums, and `slide_mean()` can be used for rolling averages.

These specialized variants are *much* faster and more memory efficient than using an otherwise equivalent call constructed with `slide_dbl()` or `slide_lgl()`, especially with a very wide window.

Usage

```
slide_sum(
  x,
  ...,
  before = 0L,
  after = 0L,
  step = 1L,
  complete = FALSE,
  na_rm = FALSE
)

slide_prod(
  x,
  ...,
  before = 0L,
  after = 0L,
  step = 1L,
  complete = FALSE,
  na_rm = FALSE
)

slide_mean(
  x,
  ...,
  before = 0L,
  after = 0L,
```

```
  step = 1L,
  complete = FALSE,
  na_rm = FALSE
)

slide_min(
  x,
  ...,
  before = 0L,
  after = 0L,
  step = 1L,
  complete = FALSE,
  na_rm = FALSE
)

slide_max(
  x,
  ...,
  before = 0L,
  after = 0L,
  step = 1L,
  complete = FALSE,
  na_rm = FALSE
)

slide_all(
  x,
  ...,
  before = 0L,
  after = 0L,
  step = 1L,
  complete = FALSE,
  na_rm = FALSE
)

slide_any(
  x,
  ...,
  before = 0L,
  after = 0L,
  step = 1L,
  complete = FALSE,
  na_rm = FALSE
)
```

Arguments

x	[vector] A vector to compute the sliding function on.

- For sliding sum, mean, prod, min, and max, x will be cast to a double vector with vctrs::vec_cast().
- For sliding any and all, x will be cast to a logical vector with vctrs::vec_cast().

`...` These dots are for future extensions and must be empty.

`before` [integer(1) / Inf]

The number of values before or after the current element to include in the sliding window. Set to `Inf` to select all elements before or after the current element. Negative values are allowed, which allows you to "look forward" from the current element if used as the `.before` value, or "look backwards" if used as `.after`.

`after` [integer(1) / Inf]

The number of values before or after the current element to include in the sliding window. Set to `Inf` to select all elements before or after the current element. Negative values are allowed, which allows you to "look forward" from the current element if used as the `.before` value, or "look backwards" if used as `.after`.

`step` [positive integer(1)]

The number of elements to shift the window forward between function calls.

`complete` [logical(1)]

Should the function be evaluated on complete windows only? If `FALSE`, the default, then partial computations will be allowed.

`na_rm` [logical(1)]

Should missing values be removed from the computation?

Details

Note that these functions are *not* generic and do not respect method dispatch of the corresponding summary function (i.e. base::sum(), base::mean()). Input will always be cast to a double or logical vector using vctrs::vec_cast(), and an internal method for computing the summary function will be used.

Due to the structure of segment trees, `slide_mean()` does not perform the same "two pass" mean that `mean()` does (the intention of the second pass is to perform a floating point error correction). Because of this, there may be small differences between `slide_mean(x)` and `slide_dbl(x,mean)` in some cases.

Value

A vector the same size as x containing the result of applying the summary function over the sliding windows.

- For sliding sum, mean, prod, min, and max, a double vector will be returned.
- For sliding any and all, a logical vector will be returned.

Implementation

These variants are implemented using a data structure known as a *segment tree*, which allows for extremely fast repeated range queries without loss of precision.

One alternative to segment trees is to directly recompute the summary function on each full window. This is what is done by using, for example, `slide_dbl(x,sum)`. This is extremely slow with large window sizes and wastes a lot of effort recomputing nearly the same information on each window. It can be made slightly faster by moving the sum to C to avoid intermediate allocations, but it still fairly slow.

A second alternative is to use an *online* algorithm, which uses information from the previous window to compute the next window. These are extremely fast, only requiring a single pass through the data, but often suffer from numerical instability issues.

Segment trees are an attempt to reconcile the performance issues of the direct approach with the numerical issues of the online approach. The performance of segment trees isn't quite as fast as online algorithms, but is close enough that it should be usable on most large data sets without any issues. Unlike online algorithms, segment trees don't suffer from any extra numerical instability issues.

References

Leis, Kundhikanjana, Kemper, and Neumann (2015). "Efficient Processing of Window Functions in Analytical SQL Queries". https://dl.acm.org/doi/10.14778/2794367.2794375

See Also

`slide_index_sum()`

Examples

```
x <- c(1, 5, 3, 2, 6, 10)

# `slide_sum()` can be used for rolling sums.
# The following are equivalent, but `slide_sum()` is much faster.
slide_sum(x, before = 2)
slide_dbl(x, sum, .before = 2)

# `slide_mean()` can be used for rolling averages
slide_mean(x, before = 2)

# Only evaluate the sum on complete windows
slide_sum(x, before = 2, after = 1, complete = TRUE)

# Skip every other calculation
slide_sum(x, before = 2, step = 2)
```

Index