# Package: warp (via r-universe)

June 10, 2024

**Title** Group Dates

**Version** 0.2.1.9000

**Description** Tooling to group dates by a variety of periods including:
yearly, monthly, by second, by week of the month, and more.
The groups are defined in such a way that they also represent
the distance between dates in terms of the period. This
extracts valuable information that can be used in further
calculations that rely on a specific temporal spacing between
observations.

**License** MIT + file LICENSE

**URL** https://github.com/DavisVaughan/warp,
https://davisvaughan.github.io/warp/

**BugReports** https://github.com/DavisVaughan/warp/issues

**Depends** R (>= 3.2)

**Suggests** covr, knitr, rmarkdown, testthat (>= 3.0.0)

**VignetteBuilder** knitr

**Config/Needs/website** tidyverse/tidytemplate

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.2.3

**Repository** https://davisvaughan.r-universe.dev

**RemoteUrl** https://github.com/davisvaughan/warp

**RemoteRef** HEAD

**RemoteSha** feb1eaaa3201c7d2eda1eb1cbfa53baa21c4104a

## Contents

---

`warp_boundary` *Locate period boundaries for a date vector*

---

## Description

`warp_boundary()` detects a change in time period along `x`, for example, rolling from one month to the next. It returns the start and stop positions for each contiguous period chunk in `x`.

## Usage

```
warp_boundary(x, period, ..., every = 1L, origin = NULL)
```

## Arguments

| | |
|---|---|
| `x` | `[Date / POSIXct / POSIXlt]`<br>A date time vector. |
| `period` | `[character(1)]`<br>A string defining the period to group by. Valid inputs can be roughly broken into:<br>• `"year"`, `"quarter"`, `"month"`, `"week"`, `"day"`<br>• `"hour"`, `"minute"`, `"second"`, `"millisecond"`<br>• `"yweek"`, `"mweek"`<br>• `"yday"`, `"mday"` |
| `...` | `[dots]`<br>These dots are for future extensions and must be empty. |
| `every` | `[positive integer(1)]`<br>The number of periods to group together.<br>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]`<br>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*.<br>This is generally used to define the anchor time to count from, which is relevant when the every value is `> 1`. |

## Details

The stop positions are just the `warp_change()` values, and the start positions are computed from these.

## Value

A two column data frame with the columns `start` and `stop`. Both are double vectors representing boundaries of the date time groups.

## Examples

```
x <- as.Date("1970-01-01") + -4:5
x

# Boundaries by month
warp_boundary(x, "month")

# Bound by every 5 days, relative to "1970-01-01"
# Creates boundaries of:
# [1969-12-27, 1970-01-01)
# [1970-01-01, 1970-01-06)
# [1970-01-06, 1970-01-11)
warp_boundary(x, "day", every = 5)

# Bound by every 5 days, relative to the smallest value in our vector
origin <- min(x)
origin

# Creates boundaries of:
# [1969-12-28, 1970-01-02)
# [1970-01-02, 1970-01-07)
warp_boundary(x, "day", every = 5, origin = origin)
```

---

## warp_change *Detect changes in a date time vector*

---

## Description

`warp_change()` detects changes at the `period` level.

If `last` = `TRUE`, it returns locations of the last value before a change, and the last location in `x` is always included. Additionally, if `endpoint` = `TRUE`, the first location in `x` will be included.

If `last` = `FALSE`, it returns locations of the first value after a change, and the first location in `x` is always included. Additionally, if `endpoint` = `TRUE`, the last location in `x` will be included.

## Usage

```
warp_change(
  x,
  period,
  ...,
  every = 1L,
  origin = NULL,
  last = TRUE,
  endpoint = FALSE
)
```

## Arguments

- `x`

  `[Date / POSIXct / POSIXlt]`

  A date time vector.

- `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"`

- `...`

  `[dots]`

  These dots are for future extensions and must be empty.

- `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`.

- `last`

  `[logical(1)]`

  If `TRUE`, the *last* location *before* a change is returned. The last location of the input is always returned.

  If `FALSE`, the *first* location *after* a change is returned. The first location of the input is always returned.

- `endpoint`

  `[logical(1)]`

  If `TRUE` and `last = TRUE`, will additionally return the first location of the input.

  If `TRUE` and `last = FALSE`, will additionally return the last location of the input.

  If `FALSE`, does nothing.

## Value

A double vector of locations.

## Examples

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

# Last location before a change, last location of `x` is always included
warp_change(x, period = "yday", every = 2, last = TRUE)

# Also include first location
warp_change(x, period = "yday", every = 2, last = TRUE, endpoint = TRUE)
```

```
# First location after a change, first location of `x` is always included
warp_change(x, period = "yday", every = 2, last = FALSE)

# Also include last location
warp_change(x, period = "yday", every = 2, last = FALSE, endpoint = TRUE)
```

## `warp_distance` *Compute distances from a date time origin*

### Description

`warp_distance()` is a low level engine for computing date time distances.

It returns the distance from `x` to the `origin` in units defined by the `period`.

For example, `period = "year"` would return the number of years from the `origin`. Setting `every = 2` would return the number of 2 year groups from the `origin`.

### Usage

```
warp_distance(x, period, ..., every = 1L, origin = NULL)
```

### Arguments

| | |
|---|---|
| `x` | `[Date / POSIXct / POSIXlt]`<br>A date time vector. |
| `period` | `[character(1)]`<br>A string defining the period to group by. Valid inputs can be roughly broken into:<br>• `"year"`, `"quarter"`, `"month"`, `"week"`, `"day"`<br>• `"hour"`, `"minute"`, `"second"`, `"millisecond"`<br>• `"yweek"`, `"mweek"`<br>• `"yday"`, `"mday"` |
| `...` | `[dots]`<br>These dots are for future extensions and must be empty. |
| `every` | `[positive integer(1)]`<br>The number of periods to group together.<br>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]`<br>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*.<br>This is generally used to define the anchor time to count from, which is relevant when the every value is `> 1`. |

## Details

The return value of `warp_distance()` has a variety of uses. It can be used for:

- A grouping column in a `dplyr::group_by()`. This is especially useful for grouping by a multitude of a particular period, such as "every 5 months".
- Computing distances between values in `x`, in units of the `period`. By returning the distances from the `origin`, `warp_distance()` has also implicitly computed the distances between values of `x`. This is used by `slide::block()` to break the input into time blocks.

When the time zone of `x` differs from the time zone of `origin`, a warning is issued, and `x` is coerced to the time zone of `origin` without changing the number of seconds of `x` from the epoch. In other words, the time zone of `x` is directly changed to the time zone of `origin` without changing the underlying numeric representation. **It is highly advised to specify an `origin` value with the same time zone as `x`.** If a `Date` is used for `x`, its time zone is assumed to be `"UTC"`.

## Value

A double vector containing the distances.

## Period

For `period` values of `"year"`, `"month"`, and `"day"`, the information provided in `origin` is truncated. Practically this means that if you specify:

```
warp_distance(period = "month", origin = as.Date("1970-01-15"))
```

then only `1970-01` will be used, and not the fact that the origin starts on the 15th of the month.

The `period` value of `"quarter"` is internally `period = "month", every = every * 3`. This means that for `"quarter"` the month specified for the `origin` will be used as the month to start counting from to generate the 3 month quarter.

To mimic the behavior of `lubridate::floor_date()`, use `period = "week"`. Internally this is just `period = "day", every = every * 7`. To mimic the `week_start` argument of `floor_date()`, set `origin` to a date with a week day identical to the one you want the week to start from. For example, the default origin of `1970-01-01` is a Thursday, so this would be generate groups identical to `floor_date(week_start = 4)`.

The `period` value of `"yday"` is computed as complete `every`-day periods from the `origin`, with a forced reset of the `every`-day counter every time you hit the month-day value of the `origin`. `"yweek"` is built on top of this internally as `period = "yday", every = every * 7`. This ends up using an algorithm very similar to `lubridate::week()`, with the added benefit of being able to control the `origin` date.

The `period` value of `"mday"` is computed as `every`-day periods within each month, with a forced reset of the `every`-day counter on the first day of each month. The most useful application of this is `"mweek"`, which is implemented as `period = "mday", every = every * 7`. This allows you to group by the "week of the month". For `"mday"` and `"mweek"`, only the year and month parts of the `origin` value are used. Because of this, the `origin` argument is not that interesting for these periods.

The `"hour"` period (and more granular frequencies) can produce results that might be surprising, even if they are technically correct. See the vignette at `vignette("hour", package = "warp")` for more information.

## Precision

With `POSIXct`, the limit of precision is approximately the microsecond level. Only dates that are very close to the unix origin of 1970-01-01 can possibly represent microsecond resolution correctly (close being within about 40 years on either side). Otherwise, the values past the microsecond resolution are essentially random, and can cause problems for the distance calculations. Because of this, decimal digits past the microsecond range are zeroed out, so please do not attempt to rely on them. It should still be safe to work with microseconds, by, say, bucketing them by millisecond distances.

## Examples

```
x <- as.Date("1970-01-01") + -4:4
x

# Compute monthly distances (really, year + month)
warp_distance(x, "month")

# Compute distances every 2 days, relative to "1970-01-01"
warp_distance(x, "day", every = 2)

# Compute distances every 2 days, this time relative to "1970-01-02"
warp_distance(x, "day", every = 2, origin = as.Date("1970-01-02"))

y <- as.POSIXct("1970-01-01 00:00:01", "UTC") + c(0, 2, 3, 4, 5, 6, 10)

# Compute distances every 5 seconds, starting from the unix epoch of
# 1970-01-01 00:00:00
# So this buckets:
# [1970-01-01 00:00:00, 1970-01-01 00:00:05) = 0
# [1970-01-01 00:00:05, 1970-01-01 00:00:10) = 1
# [1970-01-01 00:00:10, 1970-01-01 00:00:15) = 2
warp_distance(y, "second", every = 5)

# Compute distances every 5 seconds, starting from the minimum of `x`
# 1970-01-01 00:00:01
# So this buckets:
# [1970-01-01 00:00:01, 1970-01-01 00:00:06) = 0
# [1970-01-01 00:00:06, 1970-01-01 00:00:11) = 1
# [1970-01-01 00:00:11, 1970-01-01 00:00:16) = 2
origin <- as.POSIXct("1970-01-01 00:00:01", "UTC")
warp_distance(y, "second", every = 5, origin = origin)

# ---------------------------------------------------------------------------
# Time zones

# When `x` is not UTC and `origin` is left as `NULL`, the origin is set as
# 1970-01-01 00:00:00 in the time zone of `x`. This seems to be the most
# practically useful default.
z <- as.POSIXct("1969-12-31 23:00:00", "UTC")
z_in_nyc <- as.POSIXct("1969-12-31 23:00:00", "America/New_York")
```

```
# Practically this means that these give the same result, because their
# `origin` values are defined in their respective time zones.
warp_distance(z, "year")
warp_distance(z_in_nyc, "year")

# Compare that to what would happen if we used a static `origin` of
# 1970-01-01 00:00:00 UTC.
# America/New_York is 5 hours behind UTC, so when `z_in_nyc` is converted to
# UTC the value becomes `1970-01-01 04:00:00 UTC`, a different year. Because
# this is generally surprising, a warning is thrown.
origin <- as.POSIXct("1970-01-01 00:00:00", tz = "UTC")
warp_distance(z, "year", origin = origin)
warp_distance(z_in_nyc, "year", origin = origin)

# ---------------------------------------------------------------------------
# `period = "yweek"`

x <- as.Date("2019-12-23") + 0:16
origin <- as.Date("1970-01-01")

# `"week"` counts the number of 7 day periods from the `origin`
# `"yweek"` restarts the 7 day counter every time you hit the month-day
# value of the `origin`. Notice how, for the `yweek` column, only 1 day was
# in the week starting with `2019-12-31`. This is because the next day is
# `2020-01-01`, which aligns with the month-day value of the `origin`.
data.frame(
  x = x,
  week = warp_distance(x, "week", origin = origin),
  yweek = warp_distance(x, "yweek", origin = origin)
)

# ---------------------------------------------------------------------------
# `period = "mweek"`

x <- as.Date("2019-12-23") + 0:16

# `"mweek"` breaks `x` up into weeks of the month. Notice how days 1-7
# of 2020-01 all have the same distance value. A forced reset of the 7 day
# counter is done at the 1st of every month. This results in the 3 day
# week of the month at the end of 2019-12, from 29-31.
data.frame(
  x = x,
  mweek = warp_distance(x, "mweek")
)
```

# Index