Package 'clock'

December 2, 2021

Title Date-Time Types and Tools

Version 0.6.0

Description Provides a comprehensive library for date-time manipulations using a new family of orthogonal date-time classes (durations, time points, zoned-times, and calendars) that partition responsibilities so that the complexities of time zones are only considered when they are really needed. Capabilities include: date-time parsing, formatting, arithmetic, extraction and updating of components, and rounding.

License MIT + file LICENSE

URL https://clock.r-lib.org, https://github.com/r-lib/clock

BugReports https://github.com/r-lib/clock/issues

Depends R (>= 3.3)

Imports ellipsis (>= 0.3.1), rlang (>= 0.4.10), tzdb (>= 0.2.0), vctrs (>= 0.3.7)

Suggests covr, knitr, magrittr, pillar, rmarkdown, testthat (>= 3.0.0), withr

LinkingTo cpp11 (>= 0.4.0), tzdb (>= 0.2.0)

VignetteBuilder knitr

Config/Needs/website lubridate

Config/testthat/edition 3

Encoding UTF-8

LazyData true

RoxygenNote 7.1.2

SystemRequirements C++11

NeedsCompilation yes

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

Maintainer Davis Vaughan <davis@rstudio.com>

Repository CRAN

Date/Publication 2021-12-02 15:50:02 UTC

R topics documented:

as-zoned-time-Date . . . 5
as-zoned-time-naive-time . . . 7
as-zoned-time-posixt . . . 11
as-zoned-time-sys-time . . . 12
as_date . . . 13
as_date_time . . . 14
as_duration . . . 17
as_iso_year_week_day . . . 18
as_naive_time . . . 19
as_sys_time . . . 20
as_weekday . . . 21
as_year_day . . . 22
as_year_month_day . . . 22
as_year_month_weekday . . . 23
as_year_quarter_day . . . 24
as_zoned_time . . . 25
calendar-boundary . . . 26
calendar-count-between . . . 27
calendar_group . . . 28
calendar_leap_year . . . 29
calendar_month_factor . . . 30
calendar_narrow . . . 31
calendar_precision . . . 32
calendar_widen . . . 32
clock-arithmetic . . . 33
clock-codes . . . 35
clock-getters . . . 37
clock-invalid . . . 38
clock-setters . . . 40
clock_labels . . . 42
clock_locale . . . 43
date-and-date-time-boundary . . . 44
date-and-date-time-rounding . . . 45
date-and-date-time-shifting . . . 46
Date-arithmetic . . . 47
date-boundary . . . 49
date-count-between . . . 51
date-formatting . . . 53
Date-getters . . . 56
date-group . . . 57
date-rounding . . . 58
date-sequence . . . 60
Date-setters . . . 62
date-shifting . . . 64
date-time-parse . . . 65
date-today . . . 73

date-zone . . . 74
date_build . . . 75
date_count_between . . . 77
date_format . . . 78
date_group . . . 79
date_leap_year . . . 80
date_month_factor . . . 80
date_parse . . . 81
date_seq . . . 86
date_time_build . . . 87
date_weekday_factor . . . 90
duration-arithmetic . . . 91
duration-helper . . . 93
duration-rounding . . . 95
duration_cast . . . 97
duration_precision . . . 98
format.clock_zoned_time . . . 99
iso-year-week-day-arithmetic . . . 101
iso-year-week-day-boundary . . . 102
iso-year-week-day-count-between . . . 103
iso-year-week-day-getters . . . 104
iso-year-week-day-group . . . 106
iso-year-week-day-narrow . . . 107
iso-year-week-day-setters . . . 108
iso-year-week-day-widen . . . 109
iso_year_week_day . . . 110
is_duration . . . 112
is_iso_year_week_day . . . 113
is_naive_time . . . 113
is_sys_time . . . 114
is_weekday . . . 114
is_year_day . . . 115
is_year_month_day . . . 115
is_year_month_weekday . . . 116
is_year_quarter_day . . . 116
is_zoned_time . . . 117
naive_time_info . . . 117
naive_time_parse . . . 119
posixt-arithmetic . . . 124
posixt-boundary . . . 128
posixt-count-between . . . 131
posixt-formatting . . . 133
posixt-getters . . . 137
posixt-group . . . 138
posixt-rounding . . . 141
posixt-sequence . . . 144
posixt-setters . . . 149
posixt-shifting . . . 152

seq.clock_duration . . . 154
seq.clock_iso_year_week_day . . . 155
seq.clock_time_point . . . 157
seq.clock_year_day . . . 158
seq.clock_year_month_day . . . 160
seq.clock_year_month_weekday . . . 161
seq.clock_year_quarter_day . . . 162
sys-parsing . . . 163
sys_time_info . . . 169
sys_time_now . . . 171
time-point-arithmetic . . . 172
time-point-rounding . . . 174
time_point_cast . . . 176
time_point_count_between . . . 178
time_point_precision . . . 180
time_point_shift . . . 181
vec_arith.clock_year_day . . . 182
weekday . . . 183
weekday-arithmetic . . . 184
weekday_code . . . 185
weekday_factor . . . 186
year-day-arithmetic . . . 187
year-day-boundary . . . 188
year-day-count-between . . . 190
year-day-getters . . . 191
year-day-group . . . 192
year-day-narrow . . . 193
year-day-setters . . . 194
year-day-widen . . . 196
year-month-day-arithmetic . . . 197
year-month-day-boundary . . . 198
year-month-day-count-between . . . 199
year-month-day-getters . . . 200
year-month-day-group . . . 202
year-month-day-narrow . . . 203
year-month-day-setters . . . 204
year-month-day-widen . . . 206
year-month-weekday-arithmetic . . . 207
year-month-weekday-boundary . . . 208
year-month-weekday-count-between . . . 209
year-month-weekday-getters . . . 210
year-month-weekday-group . . . 212
year-month-weekday-narrow . . . 213
year-month-weekday-setters . . . 214
year-month-weekday-widen . . . 216
year-quarter-day-arithmetic . . . 217
year-quarter-day-boundary . . . 219
year-quarter-day-count-between . . . 220

year-quarter-day-getters . . . 221
year-quarter-day-group . . . 222
year-quarter-day-narrow . . . 223
year-quarter-day-setters . . . 224
year-quarter-day-widen . . . 226
year_day . . . 227
year_month_day . . . 229
year_month_day_parse . . . 230
year_month_weekday . . . 235
year_quarter_day . . . 237
zoned-parsing . . . 239
zoned-zone . . . 244
zoned_time_now . . . 245
zoned_time_precision . . . 246

Index **247**

`as-zoned-time-Date` *Convert to a zoned-time from a date*

Description

This is a Date method for the `as_zoned_time()` generic.

clock assumes that Dates are *naive* date-time types. Like naive-times, they have a yet-to-be-specified time zone. This method allows you to specify that time zone, keeping the printed time. If possible, the time will be set to midnight (see Details for the rare case in which this is not possible).

Usage

```
## S3 method for class 'Date'
as_zoned_time(x, zone, ..., nonexistent = NULL, ambiguous = NULL)
```

Arguments

`x`	[Date] A Date.
`zone`	[character(1)] The zone to convert to.
`...`	These dots are for future extensions and must be empty.
`nonexistent`	[character / NULL] One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input: • `"roll-forward"`: The next valid instant in time. • `"roll-backward"`: The previous valid instant in time. • `"shift-forward"`: Shift the nonexistent time forward by the size of the daylight saving time gap.

- "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap.
- "NA": Replace nonexistent times with NA.
- "error": Error on nonexistent times.

Using either "roll-forward" or "roll-backward" is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input.

If NULL, defaults to "error".

If getOption("clock.strict") is TRUE, nonexistent must be supplied and cannot be NULL. This is a convenient way to make production code robust to nonexistent times.

ambiguous [character / zoned_time / POSIXct / list(2) / NULL]

One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input:

- "earliest": Of the two possible times, choose the earliest one.
- "latest": Of the two possible times, choose the latest one.
- "NA": Replace ambiguous times with NA.
- "error": Error on ambiguous times.

Alternatively, ambiguous is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to NULL.

Finally, ambiguous is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If NULL, defaults to "error".

If getOption("clock.strict") is TRUE, ambiguous must be supplied and cannot be NULL. Additionally, ambiguous cannot be specified as a zoned_time on its own, as this implies NULL for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

Details

In the rare instance that the specified time zone does not contain a date-time at midnight due to daylight saving time, nonexistent can be used to resolve the issue. Similarly, if there are two possible midnight times due to a daylight saving time fallback, ambiguous can be used.

Value

A zoned-time.

Examples

```
x <- as.Date("2019-01-01")

# The resulting zoned-times have the same printed time, but are in
# different time zones
as_zoned_time(x, "UTC")
as_zoned_time(x, "America/New_York")

# Converting Date -> zoned-time is the same as naive-time -> zoned-time
x <- as_naive_time(year_month_day(2019, 1, 1))
as_zoned_time(x, "America/New_York")

# In Asia/Beirut, there was a DST gap from
# 2021-03-27 23:59:59 -> 2021-03-28 01:00:00,
# skipping the 0th hour entirely. This means there is no midnight value.
x <- as.Date("2021-03-28")
try(as_zoned_time(x, "Asia/Beirut"))

# To resolve this, set a `nonexistent` time resolution strategy
as_zoned_time(x, "Asia/Beirut", nonexistent = "roll-forward")
```

---

as-zoned-time-naive-time

Convert to a zoned-time from a naive-time

Description

This is a naive-time method for the `as_zoned_time()` generic.

Converting to a zoned-time from a naive-time retains the printed time, but changes the underlying duration, depending on the zone that you choose.

Naive-times are time points with a yet-to-be-determined time zone. By converting them to a zoned-time, all you are doing is specifying that time zone while attempting to keep all other printed information the same (if possible).

If you want to retain the underlying duration, try converting to a zoned-time from a sys-time, which is a time point interpreted as having a UTC time zone.

Usage

```
## S3 method for class 'clock_naive_time'
as_zoned_time(x, zone, ..., nonexistent = NULL, ambiguous = NULL)
```

Arguments

x	[clock_naive_time] A naive-time to convert to a zoned-time.
zone	[character(1)] The zone to convert to.

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

`nonexistent` [character / NULL]

One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"roll-forward"`: The next valid instant in time.
- `"roll-backward"`: The previous valid instant in time.
- `"shift-forward"`: Shift the nonexistent time forward by the size of the daylight saving time gap.
- "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap.
- `"NA"`: Replace nonexistent times with `NA`.
- `"error"`: Error on nonexistent times.

Using either `"roll-forward"` or `"roll-backward"` is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `nonexistent` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to nonexistent times.

`ambiguous` [character / zoned_time / POSIXct / list(2) / NULL]

One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"earliest"`: Of the two possible times, choose the earliest one.
- `"latest"`: Of the two possible times, choose the latest one.
- `"NA"`: Replace ambiguous times with `NA`.
- `"error"`: Error on ambiguous times.

Alternatively, `ambiguous` is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to `NULL`.

Finally, `ambiguous` is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `ambiguous` must be supplied and cannot be `NULL`. Additionally, `ambiguous` cannot be specified as a zoned_time on its own, as this implies `NULL` for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

Value

A zoned-time vector.

Daylight Saving Time

Converting from a naive-time to a zoned-time is not always possible due to daylight saving time issues. There are two types of these issues:

Nonexistent times are the result of daylight saving time "gaps". For example, in the America/New_York time zone, there was a daylight saving time gap 1 second after `"2020-03-08 01:59:59"`, where the clocks changed from `01:59:59 -> 03:00:00`, completely skipping the 2 o'clock hour. This means that if you had a naive time of `"2020-03-08 02:30:00"`, you couldn't convert that straight into a zoned-time with this time zone. To resolve these issues, the `nonexistent` argument can be used to specify one of many nonexistent time resolution strategies.

Ambiguous times are the result of daylight saving time "fallbacks". For example, in the America/New_York time zone, there was a daylight saving time fallback 1 second after `"2020-11-01 01:59:59 EDT"`, at which point the clocks "fell backwards" by 1 hour, resulting in a printed time of `"2020-11-01 01:00:00 EST"` (note the EDT->EST shift). This resulted in two 1 o'clock hours for this day, so if you had a naive time of `"2020-11-01 01:30:00"`, you wouldn't be able to convert that directly into a zoned-time with this time zone, as there is no way for clock to know which of the two ambiguous times you wanted. To resolve these issues, the `ambiguous` argument can be used to specify one of many ambiguous time resolution strategies.

Examples

```
library(magrittr)

x <- as_naive_time(year_month_day(2019, 1, 1))

# Converting a naive-time to a zoned-time generally retains the
# printed time, while changing the underlying duration.
as_zoned_time(x, "America/New_York")
as_zoned_time(x, "America/Los_Angeles")

# ------------------------------------------------------------------------------
# Nonexistent time:

new_york <- "America/New_York"

# There was a daylight saving gap in the America/New_York time zone on
# 2020-03-08 01:59:59 -> 03:00:00, which means that one of these
# naive-times don't exist in that time zone. By default, attempting to
# convert it to a zoned time will result in an error.
nonexistent_time <- year_month_day(2020, 03, 08, c(02, 03), c(45, 30), 00)
nonexistent_time <- as_naive_time(nonexistent_time)
try(as_zoned_time(nonexistent_time, new_york))

# Resolve this by specifying a nonexistent time resolution strategy
as_zoned_time(nonexistent_time, new_york, nonexistent = "roll-forward")
as_zoned_time(nonexistent_time, new_york, nonexistent = "roll-backward")
```

```
# Note that rolling backwards will choose the last possible moment in
# time at the current precision of the input
nonexistent_nanotime <- time_point_cast(nonexistent_time, "nanosecond")
nonexistent_nanotime
as_zoned_time(nonexistent_nanotime, new_york, nonexistent = "roll-backward")

# A word of caution - Shifting does not guarantee that the relative ordering
# of the input is maintained
shifted <- as_zoned_time(
  nonexistent_time,
  new_york,
  nonexistent = "shift-forward"
)
shifted

# 02:45:00 < 03:30:00
nonexistent_time[1] < nonexistent_time[2]
# 03:45:00 > 03:30:00 (relative ordering is lost)
shifted[1] < shifted[2]

# ------------------------------------------------------------------------
# Ambiguous time:

new_york <- "America/New_York"

# There was a daylight saving time fallback in the America/New_York time
# zone on 2020-11-01 01:59:59 EDT -> 2020-11-01 01:00:00 EST, resulting
# in two 1 o'clock hours. This means that the following naive time is
# ambiguous since we don't know which of the two 1 o'clocks it belongs to.
# By default, attempting to convert it to a zoned time will result in an
# error.
ambiguous_time <- year_month_day(2020, 11, 01, 01, 30, 00)
ambiguous_time <- as_naive_time(ambiguous_time)
try(as_zoned_time(ambiguous_time, new_york))

# Resolve this by specifying an ambiguous time resolution strategy
earliest <- as_zoned_time(ambiguous_time, new_york, ambiguous = "earliest")
latest <- as_zoned_time(ambiguous_time, new_york, ambiguous = "latest")
na <- as_zoned_time(ambiguous_time, new_york, ambiguous = "NA")
earliest
latest
na

# Now assume that you were given the following zoned-times, i.e.,
# you didn't build them from scratch so you already know their otherwise
# ambiguous offsets
x <- c(earliest, latest)
x

# To set the seconds to 5 in both, you might try:
x_naive <- x %>%
  as_naive_time() %>%
  as_year_month_day() %>%
```

```
  set_second(5) %>%
  as_naive_time()

x_naive

# But this fails because you've "lost" the information about which
# offsets these ambiguous times started in
try(as_zoned_time(x_naive, zoned_time_zone(x)))

# To get around this, you can use that information by specifying
# `ambiguous = x`, which will use the offset from `x` to resolve the
# ambiguity in `x_naive` as long as `x` is also an ambiguous time with the
# same daylight saving time transition point as `x_naive` (i.e. here
# everything has a transition point of `"2020-11-01 01:00:00 EST"`).
as_zoned_time(x_naive, zoned_time_zone(x), ambiguous = x)

# Say you added one more time to `x` that would not be considered ambiguous
# in naive-time
x <- c(x, as_zoned_time(as_sys_time(latest) + 3600, zoned_time_zone(latest)))
x

# Imagine you want to floor this vector to a multiple of 2 hours, with
# an origin of 1am that day. You can do this by subtracting the origin,
# flooring, then adding it back
origin <- year_month_day(2019, 11, 01, 01, 00, 00) %>%
  as_naive_time() %>%
  as_duration()

x_naive <- x %>%
  as_naive_time() %>%
  add_seconds(-origin) %>%
  time_point_floor("hour", n = 2) %>%
  add_seconds(origin)

x_naive

# You again have ambiguous naive-time points, so you might try using
# `ambiguous = x`. It looks like this took care of the first two problems,
# but we have an issue at location 3.
try(as_zoned_time(x_naive, zoned_time_zone(x), ambiguous = x))

# When we floored from 02:30:00 -> 01:00:00, we went from being
# unambiguous -> ambiguous. In clock, this is something you must handle
# explicitly, and cannot be handled by using information from `x`. You can
# handle this while still retaining the behavior for the other two
# time points that were ambiguous before and after the floor by passing a
# list containing `x` and an ambiguous time resolution strategy to use
# when information from `x` can't resolve ambiguities:
as_zoned_time(x_naive, zoned_time_zone(x), ambiguous = list(x, "latest"))
```

---

as-zoned-time-posixt *Convert to a zoned-time from a date-time*

Description

This is a POSIXct/POSIXlt method for the `as_zoned_time()` generic.

Converting from one of R's native date-time classes (POSIXct or POSIXlt) will retain the time zone of that object. There is no `zone` argument.

Usage

```
## S3 method for class 'POSIXt'
as_zoned_time(x, ...)
```

Arguments

`x`	[POSIXct / POSIXlt] A date-time.
`...`	These dots are for future extensions and must be empty.

Value

A zoned-time.

Examples

```
x <- as.POSIXct("2019-01-01", tz = "America/New_York")
as_zoned_time(x)
```

`as-zoned-time-sys-time`

Convert to a zoned-time from a sys-time

Description

This is a sys-time method for the `as_zoned_time()` generic.

Converting to a zoned-time from a sys-time retains the underlying duration, but changes the printed time, depending on the `zone` that you choose. Remember that sys-times are interpreted as UTC.

If you want to retain the printed time, try converting to a zoned-time from a naive-time, which is a time point with a yet-to-be-determined time zone.

Usage

```
## S3 method for class 'clock_sys_time'
as_zoned_time(x, zone, ...)
```

Arguments

x	[clock_sys_time] A sys-time to convert to a zoned-time.
zone	[character(1)] The zone to convert to.
...	These dots are for future extensions and must be empty.

Value

A zoned-time vector.

Examples

```
x <- as_sys_time(year_month_day(2019, 02, 01, 02, 30, 00))
x

# Since sys-time is interpreted as UTC, converting to a zoned-time with
# a zone of UTC retains the printed time
x_utc <- as_zoned_time(x, "UTC")
x_utc

# Converting to a different zone results in a different printed time,
# which corresponds to the exact same point in time, just in a different
# part of the work
x_ny <- as_zoned_time(x, "America/New_York")
x_ny
```

---

as_date *Convert to a date*

Description

`as_date()` is a generic function that converts its input to a date (Date).

There are methods for converting date-times (POSIXct), calendars, time points, and zoned-times to dates.

For converting to a date-time, see `as_date_time()`.

Usage

```
as_date(x)

## S3 method for class 'Date'
as_date(x)

## S3 method for class 'POSIXt'
as_date(x)
```

```
## S3 method for class 'clock_calendar'
as_date(x)

## S3 method for class 'clock_time_point'
as_date(x)

## S3 method for class 'clock_zoned_time'
as_date(x)
```

Arguments

x	[vector] A vector.

Details

Note that clock always assumes that R's Date class is naive, so converting a POSIXct to a Date will always retain the printed year, month, and day value.

This is not a drop-in replacement for `as.Date()`, as it only converts a limited set of types to Date. For parsing characters as dates, see `date_parse()`. For converting numerics to dates, see `vctrs::new_date()` or continue to use `as.Date()`.

Value

A date with the same length as `x`.

Examples

```
x <- date_time_parse("2019-01-01 23:02:03", "America/New_York")

# R's `as.Date.POSIXct()` method defaults to changing the printed time
# to UTC before converting, which can result in odd conversions like this:
as.Date(x)

# `as_date()` will never change the printed time before converting
as_date(x)

# Can also convert from other clock types
as_date(year_month_day(2019, 2, 5))
```

---

as_date_time *Convert to a date-time*

Description

`as_date_time()` is a generic function that converts its input to a date-time (POSIXct).

There are methods for converting dates (Date), calendars, time points, and zoned-times to date-times.

For converting to a date, see `as_date()`.

Usage

```
as_date_time(x, ...)

## S3 method for class 'POSIXt'
as_date_time(x, ...)

## S3 method for class 'Date'
as_date_time(x, zone, ..., nonexistent = NULL, ambiguous = NULL)

## S3 method for class 'clock_calendar'
as_date_time(x, zone, ..., nonexistent = NULL, ambiguous = NULL)

## S3 method for class 'clock_sys_time'
as_date_time(x, zone, ...)

## S3 method for class 'clock_naive_time'
as_date_time(x, zone, ..., nonexistent = NULL, ambiguous = NULL)

## S3 method for class 'clock_zoned_time'
as_date_time(x, ...)
```

Arguments

x	[vector] A vector.
...	These dots are for future extensions and must be empty.
zone	[character(1)] The zone to convert to.
nonexistent	[character / NULL] One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input: • "roll-forward": The next valid instant in time. • "roll-backward": The previous valid instant in time. • "shift-forward": Shift the nonexistent time forward by the size of the daylight saving time gap. • "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap. • "NA": Replace nonexistent times with NA. • "error": Error on nonexistent times. Using either "roll-forward" or "roll-backward" is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input. If NULL, defaults to "error". If getOption("clock.strict") is TRUE, nonexistent must be supplied and cannot be NULL. This is a convenient way to make production code robust to nonexistent times.

ambiguous [character / zoned_time / POSIXct / list(2) / NULL]

One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"earliest"`: Of the two possible times, choose the earliest one.
- `"latest"`: Of the two possible times, choose the latest one.
- `"NA"`: Replace ambiguous times with `NA`.
- `"error"`: Error on ambiguous times.

Alternatively, `ambiguous` is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to `NULL`.

Finally, `ambiguous` is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `ambiguous` must be supplied and cannot be `NULL`. Additionally, `ambiguous` cannot be specified as a zoned_time on its own, as this implies `NULL` for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

Details

Note that clock always assumes that R's Date class is naive, so converting a Date to a POSIXct will always attempt to retain the printed year, month, and day. Where possible, the resulting time will be at midnight (`00:00:00`), but in some rare cases this is not possible due to daylight saving time. If that issue ever arises, an error will be thrown, which can be resolved by explicitly supplying `nonexistent` or `ambiguous`.

This is not a drop-in replacement for `as.POSIXct()`, as it only converts a limited set of types to POSIXct. For parsing characters as date-times, see `date_time_parse()`. For converting numerics to date-times, see `vctrs::new_datetime()` or continue to use `as.POSIXct()`.

Value

A date-time with the same length as `x`.

Examples

```
x <- as.Date("2019-01-01")

# `as.POSIXct()` will always treat Date as UTC, but will show the result
```

```
# of the conversion in your system time zone, which can be somewhat confusing
if (rlang::is_installed("withr")) {
  withr::with_timezone("UTC", print(as.POSIXct(x)))
  withr::with_timezone("Europe/Paris", print(as.POSIXct(x)))
  withr::with_timezone("America/New_York", print(as.POSIXct(x)))
}

# `as_date_time()` will treat Date as naive, which means that the original
# printed date will attempt to be kept wherever possible, no matter the
# time zone. The time will be set to midnight.
as_date_time(x, "UTC")
as_date_time(x, "Europe/Paris")
as_date_time(x, "America/New_York")

# In some rare cases, this is not possible.
# For example, in Asia/Beirut, there was a DST gap from
# 2021-03-27 23:59:59 -> 2021-03-28 01:00:00,
# skipping the 0th hour entirely.
x <- as.Date("2021-03-28")
try(as_date_time(x, "Asia/Beirut"))

# To resolve this, set a `nonexistent` time resolution strategy
as_date_time(x, "Asia/Beirut", nonexistent = "roll-forward")


# You can also convert to date-time from other clock types
as_date_time(year_month_day(2019, 2, 3, 03), "America/New_York")
```

`as_duration` *Convert to a duration*

Description

You generally convert to a duration from either a sys-time or a naive-time. The precision of the input is retained in the returned duration.

To round an existing duration to another precision, see `duration_floor()`.

Usage

```
as_duration(x)
```

Arguments

`x`	[object] An object to convert to a duration.

Value

A duration with the same precision as `x`.

Examples

```
x <- as_sys_time(year_month_day(2019, 01, 01))

# The number of days since 1970-01-01 UTC
as_duration(x)

x <- x + duration_seconds(1)
x

# The number of seconds since 1970-01-01 00:00:00 UTC
as_duration(x)
```

---

as_iso_year_week_day *Convert to iso-year-week-day*

Description

`as_iso_year_week_day()` converts a vector to the iso-year-week-day calendar. Time points, Dates, POSIXct, and other calendars can all be converted to iso-year-week-day.

Usage

```
as_iso_year_week_day(x)
```

Arguments

x	[vector] A vector to convert to iso-year-week-day.

Value

A iso-year-week-day vector.

Examples

```
# From Date
as_iso_year_week_day(as.Date("2019-01-01"))

# From POSIXct, which assumes that the naive time is what should be converted
as_iso_year_week_day(as.POSIXct("2019-01-01 02:30:30", "America/New_York"))

# From other calendars
as_iso_year_week_day(year_quarter_day(2019, quarter = 2, day = 50))
```

as_naive_time *Convert to a naive-time*

Description

`as_naive_time()` converts `x` to a naive-time.

You can convert to a naive-time from any calendar type, as long as it has at least day precision. There also must not be any invalid dates. If invalid dates exist, they must first be resolved with `invalid_resolve()`.

Converting to a naive-time from a sys-time or zoned-time retains the printed time, but drops the assumption that the time should be interpreted with any specific time zone.

Converting to a naive-time from a duration just wraps the duration in a naive-time object, there is no assumption about the time zone. The duration must have at least day precision.

There are convenience methods for converting to a naive-time from R's native date and date-time types. Like converting from a zoned-time, these retain the printed time.

Usage

```
as_naive_time(x)
```

Arguments

`x`	[object] An object to convert to a naive-time.

Value

A naive-time vector.

Examples

```
x <- as.Date("2019-01-01")
as_naive_time(x)

ym <- year_month_day(2019, 02)

# A minimum of day precision is required
try(as_naive_time(ym))

ymd <- set_day(ym, 10)
as_naive_time(ymd)
```

`as_sys_time` *Convert to a sys-time*

Description

`as_sys_time()` converts `x` to a sys-time.

You can convert to a sys-time from any calendar type, as long as it has at least day precision. There also must not be any invalid dates. If invalid dates exist, they must first be resolved with `invalid_resolve()`.

Converting to a sys-time from a naive-time retains the printed time, but adds an assumption that the time should be interpreted in the UTC time zone.

Converting to a sys-time from a zoned-time retains the underlying duration, but the printed time is the equivalent UTC time to whatever the zoned-time's zone happened to be.

Converting to a sys-time from a duration just wraps the duration in a sys-time object, adding the assumption that the time should be interpreted in the UTC time zone. The duration must have at least day precision.

There are convenience methods for converting to a sys-time from R's native date and date-time types. Like converting from a zoned-time, these retain the underlying duration, but will change the printed time if the zone was not already UTC.

Usage

```
as_sys_time(x)
```

Arguments

`x`	[object] An object to convert to a sys-time.

Value

A sys-time vector.

Examples

```
x <- as.Date("2019-01-01")

# Dates are assumed to be naive, so the printed time is the same whether
# we convert it to sys-time or naive-time
as_sys_time(x)
as_naive_time(x)

y <- as.POSIXct("2019-01-01 01:00:00", tz = "America/New_York")

# The sys time displays the equivalent time in UTC (5 hours ahead of
# America/New_York at this point in the year)
as_sys_time(y)
```

```
ym <- year_month_day(2019, 02)

# A minimum of day precision is required
try(as_sys_time(ym))

ymd <- set_day(ym, 10)
as_sys_time(ymd)
```

---

as_weekday *Convert to a weekday*

Description

as_weekday() converts to a weekday type. This is normally useful for converting to a weekday from a sys-time or naive-time. You can use this function along with the *circular arithmetic* that weekday implements to easily get to the "next Monday" or "previous Sunday".

Usage

```
as_weekday(x)
```

Arguments

x	[object] An object to convert to a weekday. Usually a sys-time or naive-time.

Value

A weekday.

Examples

```
x <- as_naive_time(year_month_day(2019, 01, 05))

# This is a Saturday!
as_weekday(x)

# See the examples in `?weekday` for more usage.
```

as_year_day *Convert to year-day*

Description

`as_year_day()` converts a vector to the year-day calendar. Time points, Dates, POSIXct, and other calendars can all be converted to year-day.

Usage

```
as_year_day(x)
```

Arguments

x	[vector] A vector to convert to year-day.

Value

A year-day vector.

Examples

```
# From Date
as_year_day(as.Date("2019-05-01"))

# From POSIXct, which assumes that the naive time is what should be converted
as_year_day(as.POSIXct("2019-05-01 02:30:30", "America/New_York"))

# From other calendars
as_year_day(year_quarter_day(2019, quarter = 2, day = 50))
```

as_year_month_day *Convert to year-month-day*

Description

`as_year_month_day()` converts a vector to the year-month-day calendar. Time points, Dates, POSIXct, and other calendars can all be converted to year-month-day.

Usage

```
as_year_month_day(x)
```

Arguments

x	[vector] A vector to convert to year-month-day.

Value

A year-month-day vector.

Examples

```
# From Date
as_year_month_day(as.Date("2019-01-01"))

# From POSIXct, which assumes that the naive time is what should be converted
as_year_month_day(as.POSIXct("2019-01-01 02:30:30", "America/New_York"))

# From other calendars
as_year_month_day(year_quarter_day(2019, quarter = 2, day = 50))
```

---

as_year_month_weekday *Convert to year-month-weekday*

Description

`as_year_month_weekday()` converts a vector to the year-month-weekday calendar. Time points, Dates, POSIXct, and other calendars can all be converted to year-month-weekday.

Usage

```
as_year_month_weekday(x)
```

Arguments

x	[vector] A vector to convert to year-month-weekday.

Value

A year-month-weekday vector.

Examples

```
# From Date
as_year_month_weekday(as.Date("2019-01-01"))

# From POSIXct, which assumes that the naive time is what should be converted
as_year_month_weekday(as.POSIXct("2019-01-01 02:30:30", "America/New_York"))

# From other calendars
as_year_month_weekday(year_quarter_day(2019, quarter = 2, day = 50))
```

`as_year_quarter_day` *Convert to year-quarter-day*

Description

`as_year_quarter_day()` converts a vector to the year-quarter-day calendar. Time points, Dates, POSIXct, and other calendars can all be converted to year-quarter-day.

Usage

```
as_year_quarter_day(x, ..., start = NULL)
```

Arguments

`x`	[vector] A vector to convert to year-quarter-day.
`...`	These dots are for future extensions and must be empty.
`start`	[integer(1) / NULL] The month to start the fiscal year in. 1 = January and 12 = December. If `NULL`: • If `x` is a year-quarter-day, it will be returned as is. • Otherwise, a `start` of January will be used.

Value

A year-quarter-day vector.

Examples

```
# From Date
as_year_quarter_day(as.Date("2019-01-01"))
as_year_quarter_day(as.Date("2019-01-01"), start = 3)

# From POSIXct, which assumes that the naive time is what should be converted
as_year_quarter_day(as.POSIXct("2019-01-01 02:30:30", "America/New_York"))

# From other calendars
tuesday <- 3
as_year_quarter_day(year_month_weekday(2019, 2, tuesday, 2))

# Converting between `start`s
x <- year_quarter_day(2019, 01, 01, start = 2)
x

# Default keeps the same start
as_year_quarter_day(x)

# But you can change it
as_year_quarter_day(x, start = 1)
```

`as_zoned_time` *Convert to a zoned-time*

Description

`as_zoned_time()` converts `x` to a zoned-time. You generally convert to a zoned time from either a sys-time or a naive time. Each are documented on their own page:

- sys-time
- naive-time

There are also convenience methods for converting to a zoned time from native R date and date-time types:

- dates (Date)
- date-times (POSIXct / POSIXlt)

Usage

```
as_zoned_time(x, ...)
```

Arguments

`x`	[object] An object to convert to a zoned-time.
`...`	These dots are for future extensions and must be empty.

Value

A zoned-time vector.

Examples

```
x <- as.Date("2019-01-01")
as_zoned_time(x, "Europe/London")

y <- as_naive_time(year_month_day(2019, 2, 1))
as_zoned_time(y, zone = "America/New_York")
```

`calendar-boundary` *Boundaries: calendars*

Description

- `calendar_start()` computes the start of a calendar at a particular `precision`, such as the "start of the quarter".
- `calendar_end()` computes the end of a calendar at a particular `precision`, such as the "end of the month".

For both `calendar_start()` and `calendar_end()`, the precision of `x` is always retained.

Each calendar has its own help page describing the precisions that you can compute a boundary at:

- year-month-day
- year-month-weekday
- iso-year-week-day
- year-quarter-day
- year-day

Usage

```
calendar_start(x, precision)

calendar_end(x, precision)
```

Arguments

`x`	[calendar] A calendar vector.
`precision`	[character(1)] A precision. Allowed precisions are dependent on the calendar used.

Value

`x` at the same precision, but with some components altered to be at the boundary value.

Examples

```
# Hour precision
x <- year_month_day(2019, 2:4, 5, 6)
x

# Compute the start of the month
calendar_start(x, "month")

# Or the end of the month, notice that the hour value is adjusted as well
calendar_end(x, "month")
```

`calendar-count-between` *Counting: calendars*

Description

`calendar_count_between()` counts the number of `precision` units between `start` and `end` (i.e., the number of years or months). This count corresponds to the *whole number* of units, and will never return a fractional value.

This is suitable for, say, computing the whole number of years or months between two calendar dates, accounting for the day and time of day.

Each calendar has its own help page describing the precisions that you can count at:

- year-month-day
- year-month-weekday
- iso-year-week-day
- year-quarter-day
- year-day

Usage

```
calendar_count_between(start, end, precision, ..., n = 1L)
```

Arguments

`start, end`	[clock_calendar] A pair of calendar vectors. These will be recycled to their common size.
`precision`	[character(1)] A precision. Allowed precisions are dependent on the calendar used.
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Value

An integer representing the number of `precision` units between `start` and `end`.

Comparison Direction

The computed count has the property that if `start` <= `end`, then start + <count> <= end. Similarly, if `start` >= `end`, then start + <count> >= end. In other words, the comparison direction between `start` and `end` will never change after adding the count to `start`. This makes this function useful for repeated count computations at increasingly fine precisions.

Examples

```
# Number of whole years between these dates
x <- year_month_day(2000, 01, 05)
y <- year_month_day(2005, 01, 04:06)

# Note that `2000-01-05 -> 2005-01-04` is only 4 full years
calendar_count_between(x, y, "year")
```

`calendar_group` *Group calendar components*

Description

`calendar_group()` groups at a multiple of the specified precision. Grouping alters the value of a single component (i.e. the month component if grouping by month). Components that are more precise than the precision being grouped at are dropped altogether (i.e. the day component is dropped if grouping by month).

Each calendar has its own help page describing the grouping process in more detail:

- year-month-day
- year-month-weekday
- iso-year-week-day
- year-quarter-day
- year-day

Usage

```
calendar_group(x, precision, ..., n = 1L)
```

Arguments

`x`	[calendar] A calendar vector.
`precision`	[character(1)] A precision. Allowed precisions are dependent on the calendar used.
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Value

`x` grouped at the specified `precision`.

Examples

```
# See the calendar specific help pages for more examples
x <- year_month_day(2019, c(1, 1, 2, 2, 3, 3, 4, 4), 1:8)
x

# Group by two months
calendar_group(x, "month", n = 2)

# Group by two days of the month
calendar_group(x, "day", n = 2)
```

---

calendar_leap_year *Is the calendar year a leap year?*

Description

`calendar_leap_year()` detects if the year is a leap year according to the Gregorian calendar. It is only relevant for calendar types that use a Gregorian year, i.e. `year_month_day()`, `year_month_weekday()`, and `year_day()`.

Usage

```
calendar_leap_year(x)
```

Arguments

x	[calendar] A calendar type to detect leap years in.

Value

A logical vector the same size as `x`. Returns `TRUE` if in a leap year, `FALSE` if not in a leap year, and `NA` if `x` is `NA`.

Examples

```
x <- year_month_day(c(2019:2024, NA))
calendar_leap_year(x)
```

`calendar_month_factor` *Convert a calendar to an ordered factor of month names*

Description

`calendar_month_factor()` extracts the month values from a calendar and converts them to an ordered factor of month names. This can be useful in combination with ggplot2, or for modeling.

This function is only relevant for calendar types that use a month field, i.e. `year_month_day()` and `year_month_weekday()`. The calendar type must have at least month precision.

Usage

```
calendar_month_factor(x, ..., labels = "en", abbreviate = FALSE)
```

Arguments

`x`	[calendar] A calendar vector.
`...`	These dots are for future extensions and must be empty.
`labels`	[clock_labels / character(1)] Character representations of localized weekday names, month names, and AM/PM names. Either the language code as string (passed on to `clock_labels_lookup()`), or an object created by `clock_labels()`.
`abbreviate`	[logical(1)] If `TRUE`, the abbreviated month names from `labels` will be used. If `FALSE`, the full month names from `labels` will be used.

Value

An ordered factor representing the months.

Examples

```
x <- year_month_day(2019, 1:12)

calendar_month_factor(x)
calendar_month_factor(x, abbreviate = TRUE)
calendar_month_factor(x, labels = "fr")
```

`calendar_narrow` *Narrow a calendar to a less precise precision*

Description

`calendar_narrow()` narrows `x` to the specified `precision`. It does so by dropping components that represent a precision that is finer than `precision`.

Each calendar has its own help page describing the precisions that you can narrow to:

- year-month-day
- year-month-weekday
- iso-year-week-day
- year-quarter-day
- year-day

Usage

```
calendar_narrow(x, precision)
```

Arguments

`x`	[calendar] A calendar vector.
`precision`	[character(1)] A precision. Allowed precisions are dependent on the calendar used.

Details

A subsecond precision `x` cannot be narrowed to another subsecond precision. You cannot narrow from, say, `"nanosecond"` to `"millisecond"` precision. clock operates under the philosophy that once you have set the subsecond precision of a calendar, it is "locked in" at that precision. If you expected this to use integer division to divide the nanoseconds by 1e6 to get to millisecond precision, you probably want to convert to a time point first, and use `time_point_floor()`.

Value

`x` narrowed to the supplied `precision`.

Examples

```
# Hour precision
x <- year_month_day(2019, 1, 3, 4)
x

# Narrowed to day precision
calendar_narrow(x, "day")
```

```
# Or month precision
calendar_narrow(x, "month")
```

`calendar_precision` *Precision: calendar*

Description

`calendar_precision()` extracts the precision from a calendar object. It returns the precision as a single string.

Usage

```
calendar_precision(x)
```

Arguments

x	[clock_calendar] A calendar.

Value

A single string holding the precision of the calendar.

Examples

```
calendar_precision(year_month_day(2019))
calendar_precision(year_month_day(2019, 1, 1))
calendar_precision(year_quarter_day(2019, 3))
```

`calendar_widen` *Widen a calendar to a more precise precision*

Description

`calendar_widen()` widens `x` to the specified `precision`. It does so by setting new components to their smallest value.

Each calendar has its own help page describing the precisions that you can widen to:

- year-month-day
- year-month-weekday
- iso-year-week-day
- year-quarter-day
- year-day

Usage

```
calendar_widen(x, precision)
```

Arguments

x	[calendar]
	A calendar vector.
precision	[character(1)]
	A precision. Allowed precisions are dependent on the calendar used.

Details

A subsecond precision `x` cannot be widened. You cannot widen from, say, `"millisecond"` to `"nanosecond"` precision. clock operates under the philosophy that once you have set the subsecond precision of a calendar, it is "locked in" at that precision. If you expected this to multiply the milliseconds by 1e6 to get to nanosecond precision, you probably want to convert to a time point first, and use `time_point_cast()`.

Generally, clock treats calendars at a specific precision as a *range* of values. For example, a month precision year-month-day is treated as a range over [yyyy-mm-01, yyyy-mm-last], with no assumption about the day of the month. However, occasionally it is useful to quickly widen a calendar, assuming that you want the beginning of this range to be used for each component. This is where `calendar_widen()` can come in handy.

Value

`x` widened to the supplied `precision`.

Examples

```
# Month precision
x <- year_month_day(2019, 1)
x

# Widen to day precision
calendar_widen(x, "day")

# Or second precision
calendar_widen(x, "second")
```

`clock-arithmetic` *Clock arithmetic*

Description

This is the landing page for all clock arithmetic functions. There are specific sub-pages describing how arithmetic works for different calendars and time points, which is where you should look for more information.

Calendars are efficient at arithmetic with irregular units of time, such as month, quarters, or years.

- year-month-day
- year-month-weekday
- year-quarter-day
- iso-year-week-day
- year-day

Time points, such as naive-times and sys-times, are efficient at arithmetic with regular, well-defined units of time, such as days, hours, seconds, or nanoseconds.

- time-point

Durations can use any of these arithmetic functions, and return a new duration with a precision corresponding to the common type of the input and the function used.

- duration

Weekdays can perform day-based circular arithmetic.

- weekday

There are also convenience methods for doing arithmetic directly on a native R date or date-time type:

- dates (Date)
- date-times (POSIXct / POSIXlt)

Usage

```
add_years(x, n, ...)

add_quarters(x, n, ...)

add_months(x, n, ...)

add_weeks(x, n, ...)

add_days(x, n, ...)

add_hours(x, n, ...)

add_minutes(x, n, ...)

add_seconds(x, n, ...)
```

```
add_milliseconds(x, n, ...)

add_microseconds(x, n, ...)

add_nanoseconds(x, n, ...)
```

Arguments

x	[object] An object.
n	[integer / clock_duration] An integer vector to be converted to a duration, or a duration corresponding to the arithmetic function being used. This corresponds to the number of duration units to add. n may be negative to subtract units of duration.
...	These dots are for future extensions and must be empty.

Details

Months and years are considered "irregular" because some months have more days then others (28, 29, 30, or 31), and some years have more days than others (365 or 366).

Days are considered "regular" because they are defined as 86,400 seconds.

Value

x after performing the arithmetic.

Examples

```
# See each sub-page for more specific examples
x <- year_month_day(2019, 2, 1)
add_months(x, 1)
```

---

clock-codes *Integer codes*

Description

Objects with useful mappings from month names and weekday names to integer codes.

Month codes (clock_months)**:**

- january == 1
- february == 2
- march == 3
- april == 4
- may == 5

- `june == 6`
- `july == 7`
- `august == 8`
- `september == 9`
- `october == 10`
- `november == 11`
- `december == 12`

Weekday codes (`clock_weekdays`)**:**

- `sunday == 1`
- `monday == 2`
- `tuesday == 3`
- `wednesday == 4`
- `thursday == 5`
- `friday == 6`
- `saturday == 7`

ISO weekday codes (`clock_iso_weekdays`)**:**

- `monday == 1`
- `tuesday == 2`
- `wednesday == 3`
- `thursday == 4`
- `friday == 5`
- `saturday == 6`
- `sunday == 7`

Usage

```
clock_months

clock_weekdays

clock_iso_weekdays
```

Format

- `clock_months`: An environment containing month codes.
- `clock_weekdays`: An environment containing weekday codes.
- `clock_iso_weekdays`: An environment containing ISO weekday codes.

Examples

```
weekday(clock_weekdays$wednesday)

year_month_weekday(2019, clock_months$april, clock_weekdays$monday, 1:4)

iso_year_week_day(2020, 52, clock_iso_weekdays$thursday)
```

clock-getters *Calendar getters*

Description

This family of functions extract fields from a calendar vector. Each calendar has its own set of supported getters, which are documented on their own help page:

- year-month-day
- year-month-weekday
- iso-year-week-day
- year-quarter-day
- year-day

There are also convenience methods for extracting certain components directly from R's native date and date-time types.

- dates (Date)
- date-times (POSIXct / POSIXlt)

Usage

```
get_year(x)

get_quarter(x)

get_month(x)

get_week(x)

get_day(x)

get_hour(x)

get_minute(x)

get_second(x)

get_millisecond(x)

get_microsecond(x)

get_nanosecond(x)

get_index(x)
```

Arguments

x	[object] An object to get the component from.

Details

You cannot extract components directly from a time point type, such as sys-time or naive-time. Convert it to a calendar type first. Similarly, a zoned-time must be converted to either a sys-time or naive-time, and then to a calendar type, to be able to extract components from it.

Value

The component.

Examples

```
x <- year_month_day(2019, 1:3, 5:7, 1, 20, 30)
get_month(x)
get_day(x)
get_second(x)
```

---

clock-invalid *Invalid calendar dates*

Description

This family of functions is for working with *invalid* calendar dates.

Invalid dates represent dates made up of valid individual components, which taken as a whole don't represent valid calendar dates. For example, for `year_month_day()` the following component ranges are valid: year: [-32767, 32767], month: [1, 12], day: [1, 31]. However, the date `2019-02-31` doesn't exist even though it is made up of valid components. This is an example of an invalid date.

Invalid dates are allowed in clock, provided that they are eventually resolved by using `invalid_resolve()` or by manually resolving them through arithmetic or setter functions.

Usage

```
invalid_detect(x)

invalid_any(x)

invalid_count(x)

invalid_remove(x)

invalid_resolve(x, ..., invalid = NULL)
```

Arguments

x	[calendar] A calendar vector.
...	These dots are for future extensions and must be empty.
invalid	[character(1) / NULL] One of the following invalid date resolution strategies: • "previous": The previous valid instant in time. • "previous-day": The previous valid day in time, keeping the time of day. • "next": The next valid instant in time. • "next-day": The next valid day in time, keeping the time of day. • "overflow": Overflow by the number of days that the input is invalid by. Time of day is dropped. • "overflow-day": Overflow by the number of days that the input is invalid by. Time of day is kept. • "NA": Replace invalid dates with NA. • "error": Error on invalid dates. Using either "previous" or "next" is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input. If NULL, defaults to "error". If getOption("clock.strict") is TRUE, invalid must be supplied and cannot be NULL. This is a convenient way to make production code robust to invalid dates.

Details

Invalid dates must be resolved before converting them to a time point.

It is recommended to use "previous" or "next" for resolving invalid dates, as these ensure that *relative ordering* among x is maintained. This is a often a very important property to maintain when doing time series data analysis. See the examples for more information.

Value

- invalid_detect(): Returns a logical vector detecting invalid dates.
- invalid_any(): Returns TRUE if any invalid dates are detected.
- invalid_count(): Returns a single integer containing the number of invalid dates.
- invalid_remove(): Returns x with invalid dates removed.
- invalid_resolve(): Returns x with invalid dates resolved using the invalid strategy.

Examples

```
# Invalid date
x <- year_month_day(2019, 04, 30:31, c(3, 2), 30, 00)
x

invalid_detect(x)
```

```
# Previous valid moment in time
x_previous <- invalid_resolve(x, invalid = "previous")
x_previous

# Previous valid day, retaining time of day
x_previous_day <- invalid_resolve(x, invalid = "previous-day")
x_previous_day

# Note that `"previous"` retains the relative ordering in `x`
x[1] < x[2]
x_previous[1] < x_previous[2]

# But `"previous-day"` here does not!
x_previous_day[1] < x_previous_day[2]

# Remove invalid dates entirely
invalid_remove(x)

y <- year_quarter_day(2019, 1, 90:92)
y

# Overflow rolls forward by the number of days between `y` and the previous
# valid date
invalid_resolve(y, invalid = "overflow")
```

---

clock-setters *Calendar setters*

Description

This family of functions sets fields in a calendar vector. Each calendar has its own set of supported setters, which are documented on their own help page:

- year-month-day
- year-month-weekday
- iso-year-week-day
- year-quarter-day
- year-day

There are also convenience methods for setting certain components directly on R's native date and date-time types.

- dates (Date)
- date-times (POSIXct / POSIXlt)

Some general rules about setting components on calendar types:

- You can only set components that are relevant to the calendar type that you are working with. For example, you can't set the quarter of a year-month-day type. You'd have to convert to year-quarter-day first.
- You can set a component that is at the current precision, or one level of precision more precise than the current precision. For example, you can set the day field of a month precision year-month-day type, but not the hour field.
- Setting a component can result in an *invalid date*, such as `set_day(year_month_day(2019,02),31)`, as long as it is eventually resolved either manually or with a strategy from `invalid_resolve()`.
- With sub-second precisions, you can only set the component corresponding to the precision that you are at. For example, you can set the nanoseconds of the second while at nanosecond precision, but not milliseconds.

Usage

```
set_year(x, value, ...)

set_quarter(x, value, ...)

set_month(x, value, ...)

set_week(x, value, ...)

set_day(x, value, ...)

set_hour(x, value, ...)

set_minute(x, value, ...)

set_second(x, value, ...)

set_millisecond(x, value, ...)

set_microsecond(x, value, ...)

set_nanosecond(x, value, ...)

set_index(x, value, ...)
```

Arguments

`x`	[object] An object to set the component for.
`value`	[integer] The value to set the component to.
`...`	These dots are for future extensions and must be empty.

Details

You cannot set components directly on a time point type, such as sys-time or naive-time. Convert it to a calendar type first. Similarly, a zoned-time must be converted to either a sys-time or naive-time, and then to a calendar type, to be able to set components on it.

Value

`x` with the component set.

Examples

```
x <- year_month_day(2019, 1:3)

# Set the day
set_day(x, 12:14)

# Set to the "last" day of the month
set_day(x, "last")
```

---

clock_labels *Create or retrieve date related labels*

Description

When parsing and formatting dates, you often need to know how weekdays of the week and months are represented as text. These functions allow you to either create your own labels, or look them up from a standard set of language specific labels. The standard list is derived from ICU (https://unicode-org.github.io/icu/) via the stringi package.

- `clock_labels_lookup()` looks up a set of labels from a given language code.
- `clock_labels_languages()` lists the language codes that are accepted.
- `clock_labels()` lets you create your own set of labels. Use this if the currently supported languages don't meet your needs.

Usage

```
clock_labels(
  month,
  month_abbrev = month,
  weekday,
  weekday_abbrev = weekday,
  am_pm
)

clock_labels_lookup(language)

clock_labels_languages()
```

Arguments

`month, month_abbrev`
: [character(12)]
Full and abbreviated month names. Starts with January.

`weekday, weekday_abbrev`
: [character(7)]
Full and abbreviated weekday names. Starts with Sunday.

`am_pm`
: [character(2)]
Names used for AM and PM.

`language`
: [character(1)]
A BCP 47 locale, generally constructed from a two or three digit language code. See `clock_labels_languages()` for a complete list of available locales.

Value

A `"clock_labels"` object.

Examples

```
clock_labels_lookup("en")
clock_labels_lookup("ko")
clock_labels_lookup("fr")
```

`clock_locale` *Create a clock locale*

Description

A clock locale contains the information required to format and parse dates. The defaults have been chosen to match US English. A clock locale object can be provided to `format()` methods or parse functions (like `year_month_day_parse()`) to override the defaults.

Usage

```
clock_locale(labels = "en", decimal_mark = ".")
```

Arguments

`labels`
: [clock_labels / character(1)]
Character representations of localized weekday names, month names, and AM/PM names. Either the language code as string (passed on to `clock_labels_lookup()`), or an object created by `clock_labels()`.

`decimal_mark`
: [character(1)]
Symbol used for the decimal place when formatting sub-second date-times. Either `","` or `"."`.

Value

A "clock_locale" object.

Examples

```
clock_locale()
clock_locale(labels = "fr")
```

---

date-and-date-time-boundary

Boundaries: date and date-time

Description

- `date_start()` computes the date at the start of a particular `precision`, such as the "start of the year".
- `date_end()` computes the date at the end of a particular `precision`, such as the "end of the month".

There are separate help pages for computing boundaries for dates and date-times:

- dates (Date)
- date-times (POSIXct/POSIXlt)

Usage

```
date_start(x, precision, ...)

date_end(x, precision, ...)
```

Arguments

`x`	[Date / POSIXct / POSIXlt] A date or date-time vector.
`precision`	[character(1)] A precision. Allowed precisions are dependent on the input used.
`...`	These dots are for future extensions and must be empty.

Value

`x` but with some components altered to be at the boundary value.

Examples

```
# See type specific documentation for more examples

x <- date_build(2019, 2:4)

date_end(x, "month")

x <- date_time_build(2019, 2:4, 3:5, 4, 5, zone = "America/New_York")

# Note that the hour, minute, and second components are also adjusted
date_end(x, "month")
```

---

date-and-date-time-rounding

Date and date-time rounding

Description

- `date_floor()` rounds a date or date-time down to a multiple of the specified `precision`.
- `date_ceiling()` rounds a date or date-time up to a multiple of the specified `precision`.
- `date_round()` rounds up or down depending on what is closer, rounding up on ties.

There are separate help pages for rounding dates and date-times:

- dates (Date)
- date-times (POSIXct/POSIXlt)

These functions round the underlying duration itself, relative to an `origin`. For example, rounding to 15 hours will construct groups of 15 hours, starting from `origin`, which defaults to a naive time of 1970-01-01 00:00:00.

If you want to group by components, such as "day of the month", see `date_group()`.

Usage

```
date_floor(x, precision, ..., n = 1L, origin = NULL)

date_ceiling(x, precision, ..., n = 1L, origin = NULL)

date_round(x, precision, ..., n = 1L, origin = NULL)
```

Arguments

`x`	[Date / POSIXct / POSIXlt] A date or date-time vector.
`precision`	[character(1)] A precision. Allowed precisions are dependent on the input used.
`...`	These dots are for future extensions and must be empty.

n	[positive integer(1)] A single positive integer specifying a multiple of precision to use.
origin	[Date(1) / POSIXct(1) / POSIXlt(1) / NULL] An origin to start counting from. The default origin is midnight on 1970-01-01 in the time zone of x.

Value

x rounded to the specified precision.

Examples

```
# See the type specific documentation for more examples

x <- as.Date("2019-03-31") + 0:5
x

# Flooring by 2 days, note that this is not tied to the current month,
# and instead counts from the specified `origin`.
date_floor(x, "day", n = 2)
```

---

date-and-date-time-shifting

Shifting: date and date-time

Description

`date_shift()` shifts x to the target weekday. You can shift to the next or previous weekday. If x is currently on the target weekday, you can choose to leave it alone or advance it to the next instance of the target.

There are separate help pages for shifting dates and date-times:

- dates (Date)
- date-times (POSIXct/POSIXlt)

Usage

```
date_shift(x, target, ..., which = "next", boundary = "keep")
```

Arguments

x	[Date / POSIXct / POSIXlt] A date or date-time vector.
target	[weekday] A weekday created from `weekday()` to target. Generally this is length 1, but can also be the same length as x.
...	These dots are for future extensions and must be empty.

which [character(1)]
One of:
- "next": Shift to the next instance of the target weekday.
- "previous: Shift to the previous instance of the target weekday.

boundary [character(1)]
One of:
- "keep": If x is currently on the target weekday, return it.
- "advance": If x is currently on the target weekday, advance it anyways.

Value

x shifted to the target weekday.

Examples

```
# See the type specific documentation for more examples

x <- as.Date("2019-01-01") + 0:1

# A Tuesday and Wednesday
as_weekday(x)

monday <- weekday(clock_weekdays$monday)

# Shift to the next Monday
date_shift(x, monday)
```

Date-arithmetic *Arithmetic: date*

Description

These are Date methods for the arithmetic generics.

Calendrical based arithmetic:

These functions convert to a year-month-day calendar, perform the arithmetic, then convert back to a Date.

- add_years()
- add_quarters()
- add_months()

Time point based arithmetic:

These functions convert to a time point, perform the arithmetic, then convert back to a Date.

- add_weeks()
- add_days()

Usage

```
## S3 method for class 'Date'
add_years(x, n, ..., invalid = NULL)

## S3 method for class 'Date'
add_quarters(x, n, ..., invalid = NULL)

## S3 method for class 'Date'
add_months(x, n, ..., invalid = NULL)

## S3 method for class 'Date'
add_weeks(x, n, ...)

## S3 method for class 'Date'
add_days(x, n, ...)
```

Arguments

`x`	[Date] A Date vector.
`n`	[integer / clock_duration] An integer vector to be converted to a duration, or a duration corresponding to the arithmetic function being used. This corresponds to the number of duration units to add. `n` may be negative to subtract units of duration.
`...`	These dots are for future extensions and must be empty.
`invalid`	[character(1) / NULL] One of the following invalid date resolution strategies: • `"previous"`: The previous valid instant in time. • `"previous-day"`: The previous valid day in time, keeping the time of day. • `"next"`: The next valid instant in time. • `"next-day"`: The next valid day in time, keeping the time of day. • `"overflow"`: Overflow by the number of days that the input is invalid by. Time of day is dropped. • `"overflow-day"`: Overflow by the number of days that the input is invalid by. Time of day is kept. • `"NA"`: Replace invalid dates with `NA`. • `"error"`: Error on invalid dates. Using either `"previous"` or `"next"` is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input. If `NULL`, defaults to `"error"`. If `getOption("clock.strict")` is `TRUE`, `invalid` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to invalid dates.

Details

Adding a single quarter with `add_quarters()` is equivalent to adding 3 months.

`x` and `n` are recycled against each other.

Only calendrical based arithmetic has the potential to generate invalid dates. Time point based arithmetic, like adding days, will always generate a valid date.

Value

`x` after performing the arithmetic.

Examples

```
x <- as.Date("2019-01-01")

add_years(x, 1:5)

y <- as.Date("2019-01-31")

# Adding 1 month to `y` generates an invalid date. Unlike year-month-day
# types, R's native Date type cannot handle invalid dates, so you must
# resolve them immediately. If you don't you get an error:
try(add_months(y, 1:2))
add_months(as_year_month_day(y), 1:2)

# Resolve invalid dates by specifying an invalid date resolution strategy
# with the `invalid` argument. Using `"previous"` here sets the date to
# the previous valid date - i.e. the end of the month.
add_months(y, 1:2, invalid = "previous")
```

`date-boundary` *Boundaries: date*

Description

This is a Date method for the `date_start()` and `date_end()` generics.

Usage

```
## S3 method for class 'Date'
date_start(x, precision, ..., invalid = NULL)

## S3 method for class 'Date'
date_end(x, precision, ..., invalid = NULL)
```

Arguments

x	[Date] A date vector.
precision	[character(1)] One of: • "year" • "month" • "day"
...	These dots are for future extensions and must be empty.
invalid	[character(1) / NULL] One of the following invalid date resolution strategies: • "previous": The previous valid instant in time. • "previous-day": The previous valid day in time, keeping the time of day. • "next": The next valid instant in time. • "next-day": The next valid day in time, keeping the time of day. • "overflow": Overflow by the number of days that the input is invalid by. Time of day is dropped. • "overflow-day": Overflow by the number of days that the input is invalid by. Time of day is kept. • "NA": Replace invalid dates with NA. • "error": Error on invalid dates. Using either "previous" or "next" is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input. If NULL, defaults to "error". If getOption("clock.strict") is TRUE, invalid must be supplied and cannot be NULL. This is a convenient way to make production code robust to invalid dates.

Value

x but with some components altered to be at the boundary value.

Examples

```
x <- date_build(2019:2021, 2:4, 3:5)
x

# Last day of the month
date_end(x, "month")

# Last day of the year
date_end(x, "year")

# First day of the year
date_start(x, "year")
```

`date-count-between` *Counting: date*

Description

This is a Date method for the `date_count_between()` generic.

`date_count_between()` counts the number of `precision` units between `start` and `end` (i.e., the number of years or months). This count corresponds to the *whole number* of units, and will never return a fractional value.

This is suitable for, say, computing the whole number of years or months between two dates, accounting for the day of the month.

Calendrical based counting:

These precisions convert to a year-month-day calendar and count while in that type.

- `"year"`
- `"quarter"`
- `"month"`

Time point based counting:

These precisions convert to a time point and count while in that type.

- `"week"`
- `"day"`

For dates, whether a calendar or time point is used is not all that important, but is is fairly important for date-times.

Usage

```
## S3 method for class 'Date'
date_count_between(start, end, precision, ..., n = 1L)
```

Arguments

`start, end`	[Date] A pair of date vectors. These will be recycled to their common size.
`precision`	[character(1)] One of: • `"year"` • `"quarter"` • `"month"` • `"week"` • `"day"`
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Details

"quarter" is equivalent to "month" precision with n set to n * 3L.

Value

An integer representing the number of precision units between start and end.

Comparison Direction

The computed count has the property that if start <= end, then start + <count> <= end. Similarly, if start >= end, then start + <count> >= end. In other words, the comparison direction between start and end will never change after adding the count to start. This makes this function useful for repeated count computations at increasingly fine precisions.

Examples

```
start <- date_parse("2000-05-05")
end <- date_parse(c("2020-05-04", "2020-05-06"))

# Age in years
date_count_between(start, end, "year")

# Number of "whole" months between these dates. i.e.
# `2000-05-05 -> 2020-04-05` is 239 months
# `2000-05-05 -> 2020-05-05` is 240 months
# Since 2020-05-04 occurs before the 5th of that month,
# it gets a count of 239
date_count_between(start, end, "month")

# Number of "whole" quarters between (same as `"month"` with `n * 3`)
date_count_between(start, end, "quarter")
date_count_between(start, end, "month", n = 3)

# Number of days between
date_count_between(start, end, "day")

# Number of full 3 day periods between these two dates
date_count_between(start, end, "day", n = 3)

# Essentially the truncated value of this
date_count_between(start, end, "day") / 3

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

# Breakdown into full years, months, and days between
x <- start

years <- date_count_between(x, end, "year")
x <- add_years(x, years)

months <- date_count_between(x, end, "month")
x <- add_months(x, months)
```

```
days <- date_count_between(x, end, "day")
x <- add_days(x, days)

data.frame(
  start = start,
  end = end,
  years = years,
  months = months,
  days = days
)

# Note that when breaking down a date like that, you may need to
# set `invalid` during intermediate calculations
start <- date_build(2019, c(3, 3, 4), c(30, 31, 1))
end <- date_build(2019, 5, 05)

# These are 1 month apart (plus a few days)
months <- date_count_between(start, end, "month")

# But adding that 1 month to `start` results in an invalid date
try(add_months(start, months))

# You can choose various ways to resolve this
start_previous <- add_months(start, months, invalid = "previous")
start_next <- add_months(start, months, invalid = "next")

days_previous <- date_count_between(start_previous, end, "day")
days_next <- date_count_between(start_next, end, "day")

# Resulting in slightly different day values.
# No result is "perfect". Choosing "previous" or "next" both result
# in multiple `start` dates having the same month/day breakdown values.
data.frame(
  start = start,
  end = end,
  months = months,
  days_previous = days_previous,
  days_next = days_next
)
```

`date-formatting` *Formatting: date*

Description

This is a Date method for the `date_format()` generic.

`date_format()` formats a date (Date) using a `format` string.

If `format` is `NULL`, a default format of `"%Y-%m-%d"` is used.

Usage

```
## S3 method for class 'Date'
date_format(x, ..., format = NULL, locale = clock_locale())
```

Arguments

`x`	[Date] A date vector.
`...`	These dots are for future extensions and must be empty.
`format`	[character(1) / NULL] If `NULL`, a default format is used, which depends on the type of the input. Otherwise, a format string which is a combination of: **Year** • %C: The year divided by 100 using floored division. If the result is a single decimal digit, it is prefixed with `0`. • %y: The last two decimal digits of the year. If the result is a single digit it is prefixed by `0`. • %Y: The year as a decimal number. If the result is less than four digits it is left-padded with `0` to four digits. **Month** • %b, %h: The `locale`'s abbreviated month name. • %B: The `locale`'s full month name. • %m: The month as a decimal number. January is `01`. If the result is a single digit, it is prefixed with `0`. **Day** • %d: The day of month as a decimal number. If the result is a single decimal digit, it is prefixed with `0`. **Day of the week** • %a: The `locale`'s abbreviated weekday name. • %A: The `locale`'s full weekday name. • %w: The weekday as a decimal number (`0-6`), where Sunday is `0`. **ISO 8601 week-based year** • %g: The last two decimal digits of the ISO week-based year. If the result is a single digit it is prefixed by `0`. • %G: The ISO week-based year as a decimal number. If the result is less than four digits it is left-padded with `0` to four digits. • %V: The ISO week-based week number as a decimal number. If the result is a single digit, it is prefixed with `0`. • %u: The ISO weekday as a decimal number (`1-7`), where Monday is `1`. **Week of the year** • %U: The week number of the year as a decimal number. The first Sunday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. If the result is a single digit, it is prefixed with `0`.

- %W: The week number of the year as a decimal number. The first Monday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. If the result is a single digit, it is prefixed with `0`.

Day of the year

- %j: The day of the year as a decimal number. January 1 is `001`. If the result is less than three digits, it is left-padded with `0` to three digits.

Date

- %D, %x: Equivalent to %m/%d/%y.
- %F: Equivalent to %Y-%m-%d.

Time of day

- %H: The hour (24-hour clock) as a decimal number. If the result is a single digit, it is prefixed with `0`.
- %I: The hour (12-hour clock) as a decimal number. If the result is a single digit, it is prefixed with `0`.
- %M: The minute as a decimal number. If the result is a single digit, it is prefixed with `0`.
- %S: Seconds as a decimal number. Fractional seconds are printed at the precision of the input. The character for the decimal point is localized according to `locale`.
- %p: The `locale`'s equivalent of the AM/PM designations associated with a 12-hour clock.
- %R: Equivalent to %H:%M.
- %T, %X: Equivalent to %H:%M:%S.
- %r: Nearly equivalent to %I:%M:%S %p, but seconds are always printed at second precision.

Time zone

- %z: The offset from UTC in the ISO 8601 format. For example `-0430` refers to 4 hours 30 minutes behind UTC. If the offset is zero, `+0000` is used. The modified command %Ez inserts a `:` between the hour and minutes, like `-04:30`.
- %Z: The full time zone name. If `abbreviate_zone` is `TRUE`, the time zone abbreviation.

Miscellaneous

- %c: A date and time representation. Similar to, but not exactly the same as, %a %b %d %H:%M:%S %Y.
- `%%`: A % character.
- %n: A newline character.
- %t: A horizontal-tab character.

`locale` [clock_locale]

A locale object created from `clock_locale()`.

Details

Because a Date is considered to be a *naive* type in clock, meaning that it currently has no implied time zone, using the %z or %Z format commands is not allowed and will result in `NA`.

Value

A character vector of the formatted input.

Examples

```
x <- as.Date("2019-01-01")

# Default
date_format(x)

date_format(x, format = "year: %Y, month: %m, day: %d")

# With different locales
date_format(x, format = "%A, %B %d, %Y")
date_format(x, format = "%A, %B %d, %Y", locale = clock_locale("fr"))
```

Date-getters *Getters: date*

Description

These are Date methods for the getter generics.

- `get_year()` returns the Gregorian year.
- `get_month()` returns the month of the year.
- `get_day()` returns the day of the month.

For more advanced component extraction, convert to the calendar type that you are interested in.

Usage

```
## S3 method for class 'Date'
get_year(x)

## S3 method for class 'Date'
get_month(x)

## S3 method for class 'Date'
get_day(x)
```

Arguments

`x`	[Date] A Date to get the component from.

Value

The component.

Examples

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

`date-group` *Group date components*

Description

This is a Date method for the `date_group()` generic.

`date_group()` groups by a single component of a Date, such as month of the year, or day of the month.

If you need to group by more complex components, like ISO weeks, or quarters, convert to a calendar type that contains the component you are interested in grouping by.

Usage

```
## S3 method for class 'Date'
date_group(x, precision, ..., n = 1L, invalid = NULL)
```

Arguments

`x`	[Date] A date vector.
`precision`	[character(1)] One of: • `"year"` • `"month"` • `"day"`
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.
`invalid`	[character(1) / NULL] One of the following invalid date resolution strategies: • `"previous"`: The previous valid instant in time. • `"previous-day"`: The previous valid day in time, keeping the time of day. • `"next"`: The next valid instant in time. • `"next-day"`: The next valid day in time, keeping the time of day. • `"overflow"`: Overflow by the number of days that the input is invalid by. Time of day is dropped. • `"overflow-day"`: Overflow by the number of days that the input is invalid by. Time of day is kept. • `"NA"`: Replace invalid dates with `NA`.

- `"error"`: Error on invalid dates.

Using either `"previous"` or `"next"` is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `invalid` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to invalid dates.

Value

`x`, grouped at `precision`.

Examples

```
x <- as.Date("2019-01-01") + -3:5
x

# Group by 2 days of the current month.
# Note that this resets at the beginning of the month, creating day groups
# of [29, 30] [31] [01, 02] [03, 04].
date_group(x, "day", n = 2)

# Group by month
date_group(x, "month")
```

`date-rounding` *Rounding: date*

Description

These are Date methods for the rounding generics.

- `date_floor()` rounds a date down to a multiple of the specified `precision`.
- `date_ceiling()` rounds a date up to a multiple of the specified `precision`.
- `date_round()` rounds up or down depending on what is closer, rounding up on ties.

The only supported rounding `precisions` for Dates are `"day"` and `"week"`. You can group by irregular periods such as `"month"` or `"year"` by using `date_group()`.

Usage

```
## S3 method for class 'Date'
date_floor(x, precision, ..., n = 1L, origin = NULL)

## S3 method for class 'Date'
date_ceiling(x, precision, ..., n = 1L, origin = NULL)

## S3 method for class 'Date'
date_round(x, precision, ..., n = 1L, origin = NULL)
```

Arguments

x	[Date] A date vector.
precision	[character(1)] One of: • "week" • "day" "week" is an alias for "day" with n * 7.
...	These dots are for future extensions and must be empty.
n	[positive integer(1)] A single positive integer specifying a multiple of precision to use.
origin	[Date(1) / NULL] An origin to start counting from. The default origin is 1970-01-01.

Details

When rounding by "week", remember that the origin determines the "week start". By default, 1970-01-01 is the implicit origin, which is a Thursday. If you would like to round by weeks with a different week start, just supply an origin on the weekday you are interested in.

Value

x rounded to the specified precision.

Examples

```
x <- as.Date("2019-03-31") + 0:5
x

# Flooring by 2 days, note that this is not tied to the current month,
# and instead counts from the specified `origin`, so groups can cross
# the month boundary
date_floor(x, "day", n = 2)

# Compare to `date_group()`, which groups by the day of the month
date_group(x, "day", n = 2)

y <- as.Date("2019-01-01") + 0:20
y

# Flooring by week uses an implicit `origin` of 1970-01-01, which
# is a Thursday
date_floor(y, "week")
as_weekday(date_floor(y, "week"))

# If you want to round by weeks with a different week start, supply an
# `origin` that falls on the weekday you care about. This uses a Monday.
origin <- as.Date("1970-01-05")
as_weekday(origin)
```

```
date_floor(y, "week", origin = origin)
as_weekday(date_floor(y, "week", origin = origin))
```

`date-sequence` *Sequences: date*

Description

This is a Date method for the `date_seq()` generic.

`date_seq()` generates a date (Date) sequence.

When calling `date_seq()`, exactly two of the following must be specified:

- `to`
- `by`
- `total_size`

Usage

```
## S3 method for class 'Date'
date_seq(from, ..., to = NULL, by = NULL, total_size = NULL, invalid = NULL)
```

Arguments

`from`	[Date(1)] A date to start the sequence from. `from` is always included in the result.
`...`	These dots are for future extensions and must be empty.
`to`	[Date(1) / NULL] A date to stop the sequence at. `to` is only included in the result if the resulting sequence divides the distance between `from` and `to` exactly. If `to` is supplied along with `by`, all components of `to` more precise than the precision of `by` must match `from` exactly. For example, if `by = duration_months(1)`, the day component of `to` must match the day component of `from`. This ensures that the generated sequence is, at a minimum, a weakly monotonic sequence of dates.
`by`	[integer(1) / clock_duration(1) / NULL] The unit to increment the sequence by. If `to < from`, then `by` must be positive. If `to > from`, then `by` must be negative. If `by` is an integer, it is equivalent to `duration_days(by)`. If `by` is a duration, it is allowed to have a precision of: • year

- quarter
- month
- week
- day

`total_size` [positive integer(1) / NULL]

The size of the resulting sequence.

If specified alongside `to`, this must generate a non-fractional sequence between `from` and `to`.

`invalid` [character(1) / NULL]

One of the following invalid date resolution strategies:

- `"previous"`: The previous valid instant in time.
- `"previous-day"`: The previous valid day in time, keeping the time of day.
- `"next"`: The next valid instant in time.
- `"next-day"`: The next valid day in time, keeping the time of day.
- `"overflow"`: Overflow by the number of days that the input is invalid by. Time of day is dropped.
- `"overflow-day"`: Overflow by the number of days that the input is invalid by. Time of day is kept.
- `"NA"`: Replace invalid dates with `NA`.
- `"error"`: Error on invalid dates.

Using either `"previous"` or `"next"` is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `invalid` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to invalid dates.

Value

A date vector.

Examples

```
from <- date_build(2019, 1)
to <- date_build(2019, 4)

# Defaults to daily sequence
date_seq(from, to = to, by = 7)

# Use durations to change to monthly or yearly sequences
date_seq(from, to = to, by = duration_months(1))
date_seq(from, by = duration_years(-2), total_size = 3)

# Note that components of `to` more precise than the precision of `by`
# must match `from` exactly. For example, this is not well defined:
from <- date_build(2019, 5, 2)
to <- date_build(2025, 7, 5)
```

```
try(date_seq(from, to = to, by = duration_years(1)))

# The month and day components of `to` must match `from`
to <- date_build(2025, 5, 2)
date_seq(from, to = to, by = duration_years(1))

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

# Invalid dates must be resolved with the `invalid` argument
from <- date_build(2019, 1, 31)
to <- date_build(2019, 12, 31)

try(date_seq(from, to = to, by = duration_months(1)))
date_seq(from, to = to, by = duration_months(1), invalid = "previous")

# Compare this to the base R result, which is often a source of confusion
seq(from, to = to, by = "1 month")

# This is equivalent to the overflow invalid resolution strategy
date_seq(from, to = to, by = duration_months(1), invalid = "overflow")

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

# Usage of `to` and `total_size` must generate a non-fractional sequence
# between `from` and `to`
from <- date_build(2019, 1, 1)
to <- date_build(2019, 1, 4)

# These are fine
date_seq(from, to = to, total_size = 2)
date_seq(from, to = to, total_size = 4)

# But this is not!
try(date_seq(from, to = to, total_size = 3))
```

`Date-setters` *Setters: date*

Description

These are Date methods for the setter generics.

- `set_year()` sets the year.
- `set_month()` sets the month of the year. Valid values are in the range of [1, 12].
- `set_day()` sets the day of the month. Valid values are in the range of [1, 31].

Usage

```
## S3 method for class 'Date'
set_year(x, value, ..., invalid = NULL)
```

```
## S3 method for class 'Date'
set_month(x, value, ..., invalid = NULL)

## S3 method for class 'Date'
set_day(x, value, ..., invalid = NULL)
```

Arguments

x	[Date] A Date vector.
value	[integer / "last"] The value to set the component to. For `set_day()`, this can also be `"last"` to set the day to the last day of the month.
...	These dots are for future extensions and must be empty.
invalid	[character(1) / NULL] One of the following invalid date resolution strategies: • `"previous"`: The previous valid instant in time. • `"previous-day"`: The previous valid day in time, keeping the time of day. • `"next"`: The next valid instant in time. • `"next-day"`: The next valid day in time, keeping the time of day. • `"overflow"`: Overflow by the number of days that the input is invalid by. Time of day is dropped. • `"overflow-day"`: Overflow by the number of days that the input is invalid by. Time of day is kept. • `"NA"`: Replace invalid dates with `NA`. • `"error"`: Error on invalid dates. Using either `"previous"` or `"next"` is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input. If `NULL`, defaults to `"error"`. If `getOption("clock.strict")` is `TRUE`, `invalid` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to invalid dates.

Value

`x` with the component set.

Examples

```
x <- as.Date("2019-02-01")

# Set the day
set_day(x, 12:14)
```

```
# Set to the "last" day of the month
set_day(x, "last")

# You cannot set a Date to an invalid day like you can with
# a year-month-day. Instead, the default strategy is to error.
try(set_day(x, 31))
set_day(as_year_month_day(x), 31)

# You can resolve these issues while setting the day by specifying
# an invalid date resolution strategy with `invalid`
set_day(x, 31, invalid = "previous")
```

`date-shifting` *Shifting: date*

Description

`date_shift()` shifts `x` to the `target` weekday. You can shift to the next or previous weekday. If `x` is currently on the `target` weekday, you can choose to leave it alone or advance it to the next instance of the `target`.

Weekday shifting is one of the easiest ways to floor by week while controlling what is considered the first day of the week. You can also accomplish this with the `origin` argument of `date_floor()`, but this is slightly easier.

Usage

```
## S3 method for class 'Date'
date_shift(x, target, ..., which = "next", boundary = "keep")
```

Arguments

`x`	[Date] A date vector.
`target`	[weekday] A weekday created from `weekday()` to target. Generally this is length 1, but can also be the same length as `x`.
`...`	These dots are for future extensions and must be empty.
`which`	[character(1)] One of: • `"next"`: Shift to the next instance of the `target` weekday. • "previous: Shift to the previous instance of the `target` weekday.
`boundary`	[character(1)] One of: • `"keep"`: If `x` is currently on the `target` weekday, return it. • `"advance"`: If `x` is currently on the `target` weekday, advance it anyways.

Value

x shifted to the target weekday.

Examples

```
x <- as.Date("2019-01-01") + 0:1

# A Tuesday and Wednesday
as_weekday(x)

monday <- weekday(clock_weekdays$monday)

# Shift to the next Monday
date_shift(x, monday)

# Shift to the previous Monday
# This is an easy way to "floor by week" with a target weekday in mind
date_shift(x, monday, which = "previous")

# What about Tuesday?
tuesday <- weekday(clock_weekdays$tuesday)

# Notice that the day that was currently on a Tuesday was not shifted
date_shift(x, tuesday)

# You can force it to `"advance"`
date_shift(x, tuesday, boundary = "advance")
```

---

date-time-parse *Parsing: date-time*

Description

There are four parsers for parsing strings into POSIXct date-times, date_time_parse(), date_time_parse_complete(), date_time_parse_abbrev(), and date_time_parse_RFC_3339().

date_time_parse():

date_time_parse() is useful for strings like "2019-01-01 00:00:00", where the UTC offset and full time zone name are not present in the string. The string is first parsed as a naive-time without any time zone assumptions, and is then converted to a POSIXct with the supplied zone.

Because converting from naive-time to POSIXct may result in nonexistent or ambiguous times due to daylight saving time, these must be resolved explicitly with the nonexistent and ambiguous arguments.

date_time_parse() completely ignores the %z and %Z commands. The only time zone specific information that is used is the zone.

The default format used is "%Y-%m-%d %H:%M:%S". This matches the default result from calling format() on a POSIXct date-time.

date_time_parse_complete():

`date_time_parse_complete()` is a parser for *complete* date-time strings, like `"2019-01-01T00:00:00-05:00[America/`
A complete date-time string has both the time zone offset and full time zone name in the string, which is the only way for the string itself to contain all of the information required to unambiguously construct a zoned-time. Because of this, `date_time_parse_complete()` requires both the %z and %Z commands to be supplied in the `format` string.

The default `format` used is `"%Y-%m-%dT%H:%M:%S%Ez[%Z]"`. This matches the default result from calling `date_format()` on a POSIXct date-time. Additionally, this format matches the de-facto standard extension to RFC 3339 for creating completely unambiguous date-times.

date_time_parse_abbrev():

`date_time_parse_abbrev()` is a parser for date-time strings containing only a time zone abbreviation, like `"2019-01-01 00:00:00 EST"`. The time zone abbreviation is not enough to identify the full time zone name that the date-time belongs to, so the full time zone name must be supplied as the `zone` argument. However, the time zone abbreviation can help with resolving ambiguity around daylight saving time fallbacks.

For `date_time_parse_abbrev()`, %Z must be supplied and is interpreted as the time zone abbreviation rather than the full time zone name.

If used, the %z command must parse correctly, but its value will be completely ignored.

The default `format` used is `"%Y-%m-%d %H:%M:%S %Z"`. This matches the default result from calling `print()` or `format(usetz = TRUE)` on a POSIXct date-time.

date_time_parse_RFC_3339():

`date_time_parse_RFC_3339()` is a parser for date-time strings in the extremely common date-time format outlined by RFC 3339. This document outlines a profile of the ISO 8601 format that is even more restrictive, but corresponds to the most common formats that are likely to be used in internet protocols (i.e. through APIs).

In particular, this function is intended to parse the following three formats:

```
2019-01-01T00:00:00Z
2019-01-01T00:00:00+0430
2019-01-01T00:00:00+04:30
```

This function defaults to parsing the first of these formats by using a format string of `"%Y-%m-%dT%H:%M:%SZ"`.

If your date-time strings use offsets from UTC rather than `"Z"`, then set `offset` to one of the following:

- `"%z"` if the offset is of the form `"+0430"`.
- `"%Ez"` if the offset is of the form `"+04:30"`.

The RFC 3339 standard allows for replacing the `"T"` with a `"t"` or a space (`" "`). Set `separator` to adjust this as needed.

The date-times returned by this function will always be in the UTC time zone.

Usage

```
date_time_parse(
  x,
  zone,
  ...,
```

```
  format = NULL,
  locale = clock_locale(),
  nonexistent = NULL,
  ambiguous = NULL
)

date_time_parse_complete(x, ..., format = NULL, locale = clock_locale())

date_time_parse_abbrev(x, zone, ..., format = NULL, locale = clock_locale())

date_time_parse_RFC_3339(x, ..., separator = "T", offset = "Z")
```

Arguments

x	[character] A character vector to parse.
zone	[character(1)] A full time zone name.
...	These dots are for future extensions and must be empty.
format	[character / NULL] A format string. A combination of the following commands, or NULL, in which case a default format string is used. A vector of multiple format strings can be supplied. They will be tried in the order they are provided. **Year** • %C: The century as a decimal number. The modified command %NC where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required. • %y: The last two decimal digits of the year. If the century is not otherwise specified (e.g. with %C), values in the range [69 - 99] are presumed to refer to the years [1969 - 1999], and values in the range [00 - 68] are presumed to refer to the years [2000 - 2068]. The modified command %Ny, where N is a positive decimal integer, specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required. • %Y: The year as a decimal number. The modified command %NY where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 4. Leading zeroes are permitted but not required. **Month** • %b, %B, %h: The locale's full or abbreviated case-insensitive month name. • %m: The month as a decimal number. January is 1. The modified command %Nm where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.

Day

- %d, %e: The day of the month as a decimal number. The modified command %Nd where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.

Day of the week

- %a, %A: The `locale`'s full or abbreviated case-insensitive weekday name.
- %w: The weekday as a decimal number (`0`-`6`), where Sunday is `0`. The modified command %Nw where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `1`. Leading zeroes are permitted but not required.

ISO 8601 week-based year

- %g: The last two decimal digits of the ISO week-based year. The modified command %Ng where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %G: The ISO week-based year as a decimal number. The modified command %NG where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `4`. Leading zeroes are permitted but not required.
- %V: The ISO week-based week number as a decimal number. The modified command %NV where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %u: The ISO weekday as a decimal number (`1`-`7`), where Monday is `1`. The modified command %Nu where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `1`. Leading zeroes are permitted but not required.

Week of the year

- %U: The week number of the year as a decimal number. The first Sunday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. The modified command %NU where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %W: The week number of the year as a decimal number. The first Monday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. The modified command %NW where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.

Day of the year

- %j: The day of the year as a decimal number. January 1 is `1`. The modified command %Nj where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `3`. Leading zeroes are permitted but not required.

Date

- %D, %x: Equivalent to %m/%d/%y.
- %F: Equivalent to %Y-%m-%d. If modified with a width (like %NF), the width is applied to only %Y.

Time of day

- %H: The hour (24-hour clock) as a decimal number. The modified command %NH where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %I: The hour (12-hour clock) as a decimal number. The modified command %NI where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %M: The minutes as a decimal number. The modified command %NM where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %S: The seconds as a decimal number. Leading zeroes are permitted but not required. If encountered, the `locale` determines the decimal point character. Generally, the maximum number of characters to read is determined by the precision that you are parsing at. For example, a precision of `"second"` would read a maximum of 2 characters, while a precision of `"millisecond"` would read a maximum of 6 (2 for the values before the decimal point, 1 for the decimal point, and 3 for the values after it). The modified command %NS, where `N` is a positive decimal integer, can be used to exactly specify the maximum number of characters to read. This is only useful if you happen to have seconds with more than 1 leading zero.
- %p: The `locale`'s equivalent of the AM/PM designations associated with a 12-hour clock. The command %I must precede %p in the format string.
- %R: Equivalent to %H:%M.
- %T, %X: Equivalent to %H:%M:%S.
- %r: Equivalent to %I:%M:%S %p.

Time zone

- %z: The offset from UTC in the format [+|-]hh[mm]. For example `-0430` refers to 4 hours 30 minutes behind UTC. And `04` refers to 4 hours ahead of UTC. The modified command %Ez parses a `:` between the hours and minutes and leading zeroes on the hour field are optional: [+|-]h[h][:mm]. For example `-04:30` refers to 4 hours 30 minutes behind UTC. And `4` refers to 4 hours ahead of UTC.
- %Z: The full time zone name or the time zone abbreviation, depending on the function being used. A single word is parsed. This word can only contain characters that are alphanumeric, or one of `'_'`, `'/'`, `'-'` or `'+'`.

Miscellaneous

- %c: A date and time representation. Equivalent to %a %b %d %H:%M:%S %Y.
- %%: A % character.

- %n: Matches one white space character. %n, %t, and a space can be combined to match a wide range of white-space patterns. For example `"%n "` matches one or more white space characters, and `"%n%t%t"` matches one to three white space characters.
- %t: Matches zero or one white space characters.

`locale` [clock_locale]

A locale object created from `clock_locale()`.

`nonexistent` [character / NULL]

One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"roll-forward"`: The next valid instant in time.
- `"roll-backward"`: The previous valid instant in time.
- `"shift-forward"`: Shift the nonexistent time forward by the size of the daylight saving time gap.
- "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap.
- `"NA"`: Replace nonexistent times with `NA`.
- `"error"`: Error on nonexistent times.

Using either `"roll-forward"` or `"roll-backward"` is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `nonexistent` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to nonexistent times.

`ambiguous` [character / zoned_time / POSIXct / list(2) / NULL]

One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"earliest"`: Of the two possible times, choose the earliest one.
- `"latest"`: Of the two possible times, choose the latest one.
- `"NA"`: Replace ambiguous times with `NA`.
- `"error"`: Error on ambiguous times.

Alternatively, `ambiguous` is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to `NULL`.

Finally, `ambiguous` is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If NULL, defaults to "error".

If getOption("clock.strict") is TRUE, ambiguous must be supplied and cannot be NULL. Additionally, ambiguous cannot be specified as a zoned_time on its own, as this implies NULL for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

separator [character(1)]

The separator between the date and time components of the string. One of:

- "T"
- "t"
- " "

offset [character(1)]

The format of the offset from UTC contained in the string. One of:

- "Z"
- "z"
- "%z" to parse a numeric offset of the form "+0430"
- "%Ez" to parse a numeric offset of the form "+04:30"

Details

If date_time_parse_complete() is given input that is length zero, all NAs, or completely fails to parse, then no time zone will be able to be determined. In that case, the result will use "UTC".

If you have strings with sub-second components, then these date-time parsers are not appropriate for you. Remember that clock treats POSIXct as a second precision type, so parsing a string with fractional seconds directly into a POSIXct is ambiguous and undefined. Instead, fully parse the string, including its fractional seconds, into a clock type that can handle it, such as a naive-time with naive_time_parse(), then round to seconds with whatever rounding convention is appropriate for your use case, such as time_point_floor(), and finally convert that to POSIXct with as_date_time(). This gives you complete control over how the fractional seconds are handled when converting to POSIXct.

Value

A POSIXct.

Examples

```
# Parse with a known `zone`, even though that information isn't in the string
date_time_parse("2020-01-01 05:06:07", "America/New_York")

# Same time as above, except this is a completely unambiguous parse that
# doesn't require a `zone` argument, because the zone name and offset are
# both present in the string
date_time_parse_complete("2020-01-01T05:06:07-05:00[America/New_York]")

# Only day components
date_time_parse("2020-01-01", "America/New_York", format = "%Y-%m-%d")
```

```
# `date_time_parse()` may have issues with ambiguous times due to daylight
# saving time fallbacks. For example, there were two 1'oclock hours here:
x <- date_time_parse("1970-10-25 00:59:59", "America/New_York")

# First (earliest) 1'oclock hour
add_seconds(x, 1)
# Second (latest) 1'oclock hour
add_seconds(x, 3601)

# If you try to parse this ambiguous time directly, you'll get an error:
ambiguous_time <- "1970-10-25 01:00:00"
try(date_time_parse(ambiguous_time, "America/New_York"))

# Resolve it by specifying whether you'd like to use the
# `earliest` or `latest` of the two possible times
date_time_parse(ambiguous_time, "America/New_York", ambiguous = "earliest")
date_time_parse(ambiguous_time, "America/New_York", ambiguous = "latest")

# `date_time_parse_complete()` doesn't have these issues, as it requires
# that the offset and zone name are both in the string, which resolves
# the ambiguity
complete_times <- c(
  "1970-10-25T01:00:00-04:00[America/New_York]",
  "1970-10-25T01:00:00-05:00[America/New_York]"
)
date_time_parse_complete(complete_times)

# `date_time_parse_abbrev()` also doesn't have these issues, since it
# uses the time zone abbreviation name to resolve the ambiguity
abbrev_times <- c(
  "1970-10-25 01:00:00 EDT",
  "1970-10-25 01:00:00 EST"
)
date_time_parse_abbrev(abbrev_times, "America/New_York")

# ---------------------------------------------------------------------------
# RFC 3339

# Typical UTC format
x <- "2019-01-01T00:01:02Z"
date_time_parse_RFC_3339(x)

# With a UTC offset containing a `:`
x <- "2019-01-01T00:01:02+02:30"
date_time_parse_RFC_3339(x, offset = "%Ez")

# With a space between the date and time and no `:` in the offset
x <- "2019-01-01 00:01:02+0230"
date_time_parse_RFC_3339(x, separator = " ", offset = "%z")

# ---------------------------------------------------------------------------
# Sub-second components
```

```
# If you have a string with sub-second components, but only require up to
# seconds, first parse them into a clock type that can handle sub-seconds to
# fully capture that information, then round using whatever convention is
# required for your use case before converting to a date-time.
x <- c("2019-01-01T00:00:01.1", "2019-01-01T00:00:01.78")

x <- naive_time_parse(x, precision = "millisecond")
x

time_point_floor(x, "second")
time_point_round(x, "second")

as_date_time(time_point_round(x, "second"), "America/New_York")
```

`date-today` *Current date and date-time*

Description

- `date_today()` returns the current date in the specified `zone` as a Date.
- `date_now()` returns the current date-time in the specified `zone` as a POSIXct.

Usage

```
date_today(zone)

date_now(zone)
```

Arguments

`zone`	[character(1)] A time zone to get the current time for.

Details

clock assumes that Date is a *naive* type, like naive-time. This means that `date_today()` first looks up the current date-time in the specified `zone`, then converts that to a Date, retaining the printed time while dropping any information about that time zone.

Value

- `date_today()` a single Date.
- `date_now()` a single POSIXct.

Examples

```
# Current date in the local time zone
date_today("")

# Current date in a specified time zone
date_today("Europe/London")

# Current date-time in that same time zone
date_now("Europe/London")
```

`date-zone` *Get or set the time zone*

Description

- `date_zone()` gets the time zone.
- `date_set_zone()` sets the time zone. This retains the *underlying duration*, but changes the *printed time* depending on the zone that is chosen.

Usage

```
date_zone(x)

date_set_zone(x, zone)
```

Arguments

`x`	[POSIXct / POSIXlt]
	A date-time vector.
`zone`	[character(1)]
	A valid time zone to switch to.

Details

This function is only valid for date-times, as clock treats R's Date class as a *naive* type, which always has a yet-to-be-specified time zone.

Value

- `date_zone()` returns a string containing the time zone.
- `date_set_zone()` returns `x` with an altered printed time. The underlying duration is not changed.

Examples

```
library(magrittr)

# Cannot set or get the zone of Date.
# clock assumes that Dates are naive types, like naive-time.
x <- as.Date("2019-01-01")
try(date_zone(x))
try(date_set_zone(x, "America/New_York"))

x <- as.POSIXct("2019-01-02 01:30:00", tz = "America/New_York")
x

date_zone(x)

# If it is 1:30am in New York, what time is it in Los Angeles?
# Same underlying duration, new printed time
date_set_zone(x, "America/Los_Angeles")

# If you want to retain the printed time, but change the underlying duration,
# convert to a naive-time to drop the time zone, then convert back to a
# date-time. Be aware that this requires that you handle daylight saving time
# irregularities with the `nonexistent` and `ambiguous` arguments to
# `as.POSIXct()`!
x %>%
  as_naive_time() %>%
  as.POSIXct("America/Los_Angeles")

y <- as.POSIXct("2021-03-28 03:30:00", "America/New_York")
y

y_nt <- as_naive_time(y)
y_nt

# Helsinki had a daylight saving time gap where they jumped from
# 02:59:59 -> 04:00:00
try(as.POSIXct(y_nt, "Europe/Helsinki"))

as.POSIXct(y_nt, "Europe/Helsinki", nonexistent = "roll-forward")
as.POSIXct(y_nt, "Europe/Helsinki", nonexistent = "roll-backward")
```

---

date_build *Building: date*

Description

`date_build()` builds a Date from it's individual components.

Usage

```
date_build(year, month = 1L, day = 1L, ..., invalid = NULL)
```

Arguments

year	[integer] The year. Values [-32767, 32767] are generally allowed.
month	[integer] The month. Values [1, 12] are allowed.
day	[integer / "last"] The day of the month. Values [1, 31] are allowed. If "last", then the last day of the month is returned.
...	These dots are for future extensions and must be empty.
invalid	[character(1) / NULL] One of the following invalid date resolution strategies: • "previous": The previous valid instant in time. • "previous-day": The previous valid day in time, keeping the time of day. • "next": The next valid instant in time. • "next-day": The next valid day in time, keeping the time of day. • "overflow": Overflow by the number of days that the input is invalid by. Time of day is dropped. • "overflow-day": Overflow by the number of days that the input is invalid by. Time of day is kept. • "NA": Replace invalid dates with NA. • "error": Error on invalid dates. Using either "previous" or "next" is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input. If NULL, defaults to "error". If getOption("clock.strict") is TRUE, invalid must be supplied and cannot be NULL. This is a convenient way to make production code robust to invalid dates.

Details

Components are recycled against each other.

Value

A Date.

Examples

```
date_build(2019)
date_build(2019, 1:3)

# Generating invalid dates will trigger an error
try(date_build(2019, 1:12, 31))

# You can resolve this with `invalid`
date_build(2019, 1:12, 31, invalid = "previous")
```

```
# But this particular case (the last day of the month) is better
# specified as:
date_build(2019, 1:12, "last")
```

`date_count_between` *Counting: date and date-time*

Description

`date_count_between()` counts the number of `precision` units between `start` and `end` (i.e., the number of years or months or hours). This count corresponds to the *whole number* of units, and will never return a fractional value.

This is suitable for, say, computing the whole number of years or months between two dates, accounting for the day and time of day.

There are separate help pages for counting for dates and date-times:

- dates (Date)
- date-times (POSIXct/POSIXlt)

Usage

```
date_count_between(start, end, precision, ..., n = 1L)
```

Arguments

`start, end`	[Date / POSIXct / POSIXlt] A pair of date or date-time vectors. These will be recycled to their common size.
`precision`	[character(1)] A precision. Allowed precisions are dependent on the calendar used.
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Value

An integer representing the number of `precision` units between `start` and `end`.

Comparison Direction

The computed count has the property that if `start` <= `end`, then start + <count> <= end. Similarly, if `start` >= `end`, then start + <count> >= end. In other words, the comparison direction between `start` and `end` will never change after adding the count to `start`. This makes this function useful for repeated count computations at increasingly fine precisions.

Examples

```
# See method specific documentation for more examples

start <- date_parse("2000-05-05")
end <- date_parse(c("2020-05-04", "2020-05-06"))

# Age in years
date_count_between(start, end, "year")

# Number of "whole" months between these dates
date_count_between(start, end, "month")
```

`date_format` *Formatting: date and date-time*

Description

`date_format()` formats a date (Date) or date-time (POSIXct/POSIXlt) using a `format` string.

There are separate help pages for formatting dates and date-times:

- dates (Date)
- date-times (POSIXct/POSIXlt)

Usage

```
date_format(x, ...)
```

Arguments

`x`	[Date / POSIXct / POSIXlt] A date or date-time vector.
`...`	These dots are for future extensions and must be empty.

Value

A character vector of the formatted input.

Examples

```
# See method specific documentation for more examples

x <- as.Date("2019-01-01")
date_format(x, format = "year: %Y, month: %m, day: %d")
```

`date_group` *Group date and date-time components*

Description

`date_group()` groups by a single component of a date-time, such as month of the year, or day of the month.

There are separate help pages for grouping dates and date-times:

- dates (Date)
- date-times (POSIXct/POSIXlt)

Usage

```
date_group(x, precision, ..., n = 1L)
```

Arguments

`x`	[Date / POSIXct / POSIXlt] A date or date-time vector.
`precision`	[character(1)] A precision. Allowed precisions are dependent on the input used.
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Value

`x`, grouped at `precision`.

Examples

```
# See type specific documentation for more examples
date_group(as.Date("2019-01-01") + 0:5, "day", n = 2)
```

date_leap_year *Is the year a leap year?*

Description

`date_leap_year()` detects if the year is a leap year.

Usage

```
date_leap_year(x)
```

Arguments

`x`	[Date / POSIXct / POSIXlt] A date or date-time to detect leap years in.

Value

A logical vector the same size as `x`. Returns `TRUE` if in a leap year, `FALSE` if not in a leap year, and `NA` if `x` is `NA`.

Examples

```
x <- as.Date("2019-01-01")
x <- add_years(x, 0:5)
date_leap_year(x)

y <- as.POSIXct("2019-01-01", "America/New_York")
y <- add_years(y, 0:5)
date_leap_year(y)
```

date_month_factor *Convert a date or date-time to an ordered factor of month names*

Description

`date_month_factor()` extracts the month values from a date or date-time and converts them to an ordered factor of month names. This can be useful in combination with ggplot2, or for modeling.

Usage

```
date_month_factor(x, ..., labels = "en", abbreviate = FALSE)
```

Arguments

x	[Date / POSIXct / POSIXlt] A date or date-time vector.
...	These dots are for future extensions and must be empty.
labels	[clock_labels / character(1)] Character representations of localized weekday names, month names, and AM/PM names. Either the language code as string (passed on to clock_labels_lookup()), or an object created by clock_labels().
abbreviate	[logical(1)] If TRUE, the abbreviated month names from labels will be used. If FALSE, the full month names from labels will be used.

Value

An ordered factor representing the months.

Examples

```
x <- add_months(as.Date("2019-01-01"), 0:11)

date_month_factor(x)
date_month_factor(x, abbreviate = TRUE)
date_month_factor(x, labels = "fr")
```

---

date_parse *Parsing: date*

Description

`date_parse()` parses strings into a Date.

The default `format` used is `"%Y-%m-%d"`. This matches the default result from calling `print()` or `format()` on a Date.

Usage

```
date_parse(x, ..., format = NULL, locale = clock_locale())
```

Arguments

x	[character] A character vector to parse.
...	These dots are for future extensions and must be empty.

format [character / NULL]

A format string. A combination of the following commands, or NULL, in which case a default format string is used.

A vector of multiple format strings can be supplied. They will be tried in the order they are provided.

Year

- %C: The century as a decimal number. The modified command %NC where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
- %y: The last two decimal digits of the year. If the century is not otherwise specified (e.g. with %C), values in the range [69 - 99] are presumed to refer to the years [1969 - 1999], and values in the range [00 - 68] are presumed to refer to the years [2000 - 2068]. The modified command %Ny, where N is a positive decimal integer, specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
- %Y: The year as a decimal number. The modified command %NY where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 4. Leading zeroes are permitted but not required.

Month

- %b, %B, %h: The locale's full or abbreviated case-insensitive month name.
- %m: The month as a decimal number. January is 1. The modified command %Nm where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.

Day

- %d, %e: The day of the month as a decimal number. The modified command %Nd where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.

Day of the week

- %a, %A: The locale's full or abbreviated case-insensitive weekday name.
- %w: The weekday as a decimal number (0-6), where Sunday is 0. The modified command %Nw where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 1. Leading zeroes are permitted but not required.

ISO 8601 week-based year

- %g: The last two decimal digits of the ISO week-based year. The modified command %Ng where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.

- %G: The ISO week-based year as a decimal number. The modified command %NG where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 4. Leading zeroes are permitted but not required.
- %V: The ISO week-based week number as a decimal number. The modified command %NV where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
- %u: The ISO weekday as a decimal number (1-7), where Monday is 1. The modified command %Nu where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 1. Leading zeroes are permitted but not required.

Week of the year

- %U: The week number of the year as a decimal number. The first Sunday of the year is the first day of week 01. Days of the same year prior to that are in week 00. The modified command %NU where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
- %W: The week number of the year as a decimal number. The first Monday of the year is the first day of week 01. Days of the same year prior to that are in week 00. The modified command %NW where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.

Day of the year

- %j: The day of the year as a decimal number. January 1 is 1. The modified command %Nj where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 3. Leading zeroes are permitted but not required.

Date

- %D, %x: Equivalent to %m/%d/%y.
- %F: Equivalent to %Y-%m-%d. If modified with a width (like %NF), the width is applied to only %Y.

Time of day

- %H: The hour (24-hour clock) as a decimal number. The modified command %NH where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
- %I: The hour (12-hour clock) as a decimal number. The modified command %NI where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
- %M: The minutes as a decimal number. The modified command %NM where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.

- %S: The seconds as a decimal number. Leading zeroes are permitted but not required. If encountered, the `locale` determines the decimal point character. Generally, the maximum number of characters to read is determined by the precision that you are parsing at. For example, a precision of `"second"` would read a maximum of 2 characters, while a precision of `"millisecond"` would read a maximum of 6 (2 for the values before the decimal point, 1 for the decimal point, and 3 for the values after it). The modified command %NS, where `N` is a positive decimal integer, can be used to exactly specify the maximum number of characters to read. This is only useful if you happen to have seconds with more than 1 leading zero.
- %p: The `locale`'s equivalent of the AM/PM designations associated with a 12-hour clock. The command %I must precede %p in the format string.
- %R: Equivalent to %H:%M.
- %T, %X: Equivalent to %H:%M:%S.
- %r: Equivalent to %I:%M:%S %p.

Time zone

- %z: The offset from UTC in the format [+|-]hh[mm]. For example `-0430` refers to 4 hours 30 minutes behind UTC. And `04` refers to 4 hours ahead of UTC. The modified command %Ez parses a `:` between the hours and minutes and leading zeroes on the hour field are optional: [+|-]h[h][:mm]. For example `-04:30` refers to 4 hours 30 minutes behind UTC. And `4` refers to 4 hours ahead of UTC.
- %Z: The full time zone name or the time zone abbreviation, depending on the function being used. A single word is parsed. This word can only contain characters that are alphanumeric, or one of `'_'`, `'/'`, `'-'` or `'+'`.

Miscellaneous

- %c: A date and time representation. Equivalent to %a %b %d %H:%M:%S %Y.
- `%%`: A % character.
- %n: Matches one white space character. %n, %t, and a space can be combined to match a wide range of white-space patterns. For example `"%n "` matches one or more white space characters, and `"%n%t%t"` matches one to three white space characters.
- %t: Matches zero or one white space characters.

`locale` [clock_locale]

A locale object created from `clock_locale()`.

Details

`date_parse()` *ignores both the* %z *and* %Z *commands,* as clock treats Date as a *naive* type, with a yet-to-be-specified time zone.

Parsing strings with sub-daily components, such as hours, minutes, or seconds, should generally be done with `date_time_parse()`. If you only need the date components from a string with sub-daily components, choose one of the following:

- If the date components are at the front of the string, and you don't want the time components to affect the date in any way, you can use `date_parse()` to parse only the date components. For

example, `date_parse("2019-01-05 00:01:02", format = "%Y-%m-%d")` will parse through `05` and then stop.

- If you want the time components to influence the date, then parse the full string with `date_time_parse()`, round to day precision with a rounding function like `date_round()`, and cast to date with `as_date()`.

Attempting to directly parse all components of a sub-daily string into a Date is ambiguous and undefined, and is unlikely to work as you might expect. For example, `date_parse("2019-01-05 00:01:02", format = "%Y-%m-%d %H:%M:%S")` is not officially supported, even if it works in some cases.

Value

A Date.

Examples

```
date_parse("2020-01-01")

date_parse(
  "January 5, 2020",
  format = "%B %d, %Y"
)

# With a different locale
date_parse(
  "janvier 5, 2020",
  format = "%B %d, %Y",
  locale = clock_locale("fr")
)

# A neat feature of `date_parse()` is the ability to parse
# the ISO year-week-day format
date_parse("2020-W01-2", format = "%G-W%V-%u")

# ------------------------------------------------------------------------
# Sub-daily components

# If you have a string with sub-daily components, but only require the date,
# first parse them as date-times to fully parse the sub-daily components,
# then round using whatever convention is required for your use case before
# converting to date.
x <- c("2019-01-01 11", "2019-01-01 12")

x <- date_time_parse(x, zone = "UTC", format = "%Y-%m-%d %H")
x

date_floor(x, "day")
date_round(x, "day")

as_date(date_round(x, "day"))
```

`date_seq` *Sequences: date and date-time*

Description

`date_seq()` generates a date (Date) or date-time (POSIXct/POSIXlt) sequence.

There are separate help pages for generating sequences for dates and date-times:

- dates (Date)
- date-times (POSIXct/POSIXlt)

Usage

```
date_seq(from, ..., to = NULL, by = NULL, total_size = NULL)
```

Arguments

`from`	[Date(1) / POSIXct(1) / POSIXlt(1)] A date or date-time to start the sequence from. `from` is always included in the result.
`...`	These dots are for future extensions and must be empty.
`to`	[Date(1) / POSIXct(1) / POSIXlt(1) / NULL] A date or date-time to stop the sequence at. `to` is only included in the result if the resulting sequence divides the distance between `from` and `to` exactly.
`by`	[integer(1) / clock_duration(1) / NULL] The unit to increment the sequence by. If `to` < `from`, then `by` must be positive. If `to` > `from`, then `by` must be negative.
`total_size`	[positive integer(1) / NULL] The size of the resulting sequence. If specified alongside `to`, this must generate a non-fractional sequence between `from` and `to`.

Value

A date or date-time vector.

Examples

```
# See method specific documentation for more examples

x <- as.Date("2019-01-01")
date_seq(x, by = duration_months(2), total_size = 20)
```

`date_time_build` *Building: date-time*

Description

`date_time_build()` builds a POSIXct from it's individual components.

To build a POSIXct, it is required that you specify the `zone`.

Usage

```
date_time_build(
  year,
  month = 1L,
  day = 1L,
  hour = 0L,
  minute = 0L,
  second = 0L,
  ...,
  zone,
  invalid = NULL,
  nonexistent = NULL,
  ambiguous = NULL
)
```

Arguments

`year`	[integer] The year. Values [-32767, 32767] are generally allowed.
`month`	[integer] The month. Values [1, 12] are allowed.
`day`	[integer / "last"] The day of the month. Values [1, 31] are allowed. If `"last"`, then the last day of the month is returned.
`hour`	[integer] The hour. Values [0, 23] are allowed.
`minute`	[integer] The minute. Values [0, 59] are allowed.
`second`	[integer] The second. Values [0, 59] are allowed.
`...`	These dots are for future extensions and must be empty.
`zone`	[character(1)] A valid time zone name. This argument is required, and must be specified by name.

`invalid` [character(1) / NULL]

One of the following invalid date resolution strategies:

- `"previous"`: The previous valid instant in time.
- `"previous-day"`: The previous valid day in time, keeping the time of day.
- `"next"`: The next valid instant in time.
- `"next-day"`: The next valid day in time, keeping the time of day.
- `"overflow"`: Overflow by the number of days that the input is invalid by. Time of day is dropped.
- `"overflow-day"`: Overflow by the number of days that the input is invalid by. Time of day is kept.
- `"NA"`: Replace invalid dates with `NA`.
- `"error"`: Error on invalid dates.

Using either `"previous"` or `"next"` is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `invalid` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to invalid dates.

`nonexistent` [character / NULL]

One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"roll-forward"`: The next valid instant in time.
- `"roll-backward"`: The previous valid instant in time.
- `"shift-forward"`: Shift the nonexistent time forward by the size of the daylight saving time gap.
- "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap.
- `"NA"`: Replace nonexistent times with `NA`.
- `"error"`: Error on nonexistent times.

Using either `"roll-forward"` or `"roll-backward"` is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `nonexistent` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to nonexistent times.

`ambiguous` [character / zoned_time / POSIXct / list(2) / NULL]

One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"earliest"`: Of the two possible times, choose the earliest one.
- `"latest"`: Of the two possible times, choose the latest one.
- `"NA"`: Replace ambiguous times with `NA`.
- `"error"`: Error on ambiguous times.

Alternatively, ambiguous is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to NULL.

Finally, ambiguous is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If NULL, defaults to "error".

If getOption("clock.strict") is TRUE, ambiguous must be supplied and cannot be NULL. Additionally, ambiguous cannot be specified as a zoned_time on its own, as this implies NULL for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

Details

Components are recycled against each other.

Value

A POSIXct.

Examples

```
# The zone argument is required!
# clock always requires you to be explicit about your choice of `zone`.
try(date_time_build(2020))

date_time_build(2020, zone = "America/New_York")

# Nonexistent time due to daylight saving time gap from 01:59:59 -> 03:00:00
try(date_time_build(1970, 4, 26, 1:12, 30, zone = "America/New_York"))

# Resolve with a nonexistent time resolution strategy
date_time_build(
  1970, 4, 26, 1:12, 30,
  zone = "America/New_York",
  nonexistent = "roll-forward"
)
```

`date_weekday_factor` *Convert a date or date-time to a weekday factor*

Description

`date_weekday_factor()` converts a date or date-time to an ordered factor with levels representing the weekday. This can be useful in combination with ggplot2, or for modeling.

Usage

```
date_weekday_factor(
  x,
  ...,
  labels = "en",
  abbreviate = TRUE,
  encoding = "western"
)
```

Arguments

`x`	[Date / POSIXct / POSIXlt] A date or date-time vector.
`...`	These dots are for future extensions and must be empty.
`labels`	[clock_labels / character(1)] Character representations of localized weekday names, month names, and AM/PM names. Either the language code as string (passed on to `clock_labels_lookup()`), or an object created by `clock_labels()`.
`abbreviate`	[logical(1)] If `TRUE`, the abbreviated weekday names from `labels` will be used. If `FALSE`, the full weekday names from `labels` will be used.
`encoding`	[character(1)] One of: • `"western"`: Encode the weekdays as an ordered factor with levels from Sunday -> Saturday. • `"iso"`: Encode the weekdays as an ordered factor with levels from Monday -> Sunday.

Value

An ordered factor representing the weekdays.

Examples

```
x <- as.Date("2019-01-01") + 0:6

# Default to Sunday -> Saturday
date_weekday_factor(x)

# ISO encoding is Monday -> Sunday
date_weekday_factor(x, encoding = "iso")

# With full names
date_weekday_factor(x, abbreviate = FALSE)

# Or a different language
date_weekday_factor(x, labels = "fr")
```

---

duration-arithmetic *Arithmetic: duration*

Description

These are duration methods for the arithmetic generics.

- `add_years()`
- `add_quarters()`
- `add_months()`
- `add_weeks()`
- `add_days()`
- `add_hours()`
- `add_minutes()`
- `add_seconds()`
- `add_milliseconds()`
- `add_microseconds()`
- `add_nanoseconds()`

When adding to a duration using one of these functions, a second duration is created based on the function name and `n`. The two durations are then added together, and the precision of the result is determined as the *more precise precision* of the two durations.

Usage

```
## S3 method for class 'clock_duration'
add_years(x, n, ...)

## S3 method for class 'clock_duration'
add_quarters(x, n, ...)
```

```
## S3 method for class 'clock_duration'
add_months(x, n, ...)

## S3 method for class 'clock_duration'
add_weeks(x, n, ...)

## S3 method for class 'clock_duration'
add_days(x, n, ...)

## S3 method for class 'clock_duration'
add_hours(x, n, ...)

## S3 method for class 'clock_duration'
add_minutes(x, n, ...)

## S3 method for class 'clock_duration'
add_seconds(x, n, ...)

## S3 method for class 'clock_duration'
add_milliseconds(x, n, ...)

## S3 method for class 'clock_duration'
add_microseconds(x, n, ...)

## S3 method for class 'clock_duration'
add_nanoseconds(x, n, ...)
```

Arguments

x	[clock_duration] A duration vector.
n	[integer / clock_duration] An integer vector to be converted to a duration, or a duration corresponding to the arithmetic function being used. This corresponds to the number of duration units to add. n may be negative to subtract units of duration.
...	These dots are for future extensions and must be empty.

Details

You can add calendrical durations to other calendrical durations, and chronological durations to other chronological durations, but you can't add a chronological duration to a calendrical duration (such as adding days and months). For more information, see the documentation on the duration helper page.

x and n are recycled against each other.

Value

x after performing the arithmetic, possibly with a more precise precision.

Examples

```
x <- duration_seconds(5)

# Addition in the same precision
add_seconds(x, 1:10)

# Addition with days, defined as 86400 seconds
add_days(x, 1)

# Similarly, if you start with days and add seconds, you get the common
# precision of the two back, which is seconds
y <- duration_days(1)
add_seconds(y, 5)

# But you can't add a chronological duration (days) and
# a calendrical duration (months)
try(add_months(y, 1))

# You can add years to a duration of months, which adds
# an additional 12 months / year
z <- duration_months(5)
add_years(z, 1)
```

---

duration-helper *Construct a duration*

Description

These helpers construct durations of the specified precision. Durations represent units of time.

Durations are separated into two categories:

Calendrical

- year
- quarter
- month

Chronological

- week
- day
- hour
- minute
- second
- millisecond
- microsecond
- nanosecond

Calendrical durations are generally used when manipulating calendar types, like year-month-day. Chronological durations are generally used when working with time points, like sys-time or naive-time.

Usage

```
duration_years(n = integer())

duration_quarters(n = integer())

duration_months(n = integer())

duration_weeks(n = integer())

duration_days(n = integer())

duration_hours(n = integer())

duration_minutes(n = integer())

duration_seconds(n = integer())

duration_milliseconds(n = integer())

duration_microseconds(n = integer())

duration_nanoseconds(n = integer())
```

Arguments

n	[integer] The number of units of time to use when creating the duration.

Value

A duration of the specified precision.

Internal Representation

Durations are internally represented as an integer number of "ticks" along with a ratio describing how it converts to a number of seconds. The following duration ratios are used in clock:

- 1 year == 31556952 seconds
- 1 quarter == 7889238 seconds
- 1 month == 2629746 seconds
- 1 week == 604800 seconds
- 1 day == 86400 seconds
- 1 hour == 3600 seconds
- 1 minute == 60 seconds

- 1 second == 1 second
- 1 millisecond == 1 / 1000 seconds
- 1 microsecond == 1 / 1000000 seconds
- 1 nanosecond == 1 / 1000000000 seconds

A duration of 1 year is defined to correspond to the average length of a proleptic Gregorian year, i.e. 365.2425 days.

A duration of 1 month is defined as exactly 1/12 of a year.

A duration of 1 quarter is defined as exactly 1/4 of a year.

A duration of 1 week is defined as exactly 7 days.

These conversions come into play when doing operations like adding or flooring durations. Generally, you add two calendrical durations together to get a new calendrical duration, rather than adding a calendrical and a chronological duration together. The one exception is `duration_cast()`, which can cast durations to any other precision, with a potential loss of information.

Examples

```
duration_years(1:5)
duration_nanoseconds(1:5)
```

`duration-rounding` *Duration rounding*

Description

- `duration_floor()` rounds a duration down to a multiple of the specified `precision`.
- `duration_ceiling()` rounds a duration up to a multiple of the specified `precision`.
- `duration_round()` rounds up or down depending on what is closer, rounding up on ties.

Usage

```
duration_floor(x, precision, ..., n = 1L)

duration_ceiling(x, precision, ..., n = 1L)

duration_round(x, precision, ..., n = 1L)
```

Arguments

`x`	[clock_duration] A duration.
`precision`	[character(1)] A precision. One of: • `"year"` • `"quarter"`

- "month"
- "week"
- "day"
- "hour"
- "minute"
- "second"
- "millisecond"
- "microsecond"
- "nanosecond"

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

n [positive integer(1)]
A positive integer specifying the multiple of precision to use.

Details

You can floor calendrical durations to other calendrical durations, and chronological durations to other chronological durations, but you can't floor a chronological duration to a calendrical duration (such as flooring from day to month). For more information, see the documentation on the duration helper page.

Value

x rounded to the precision.

Examples

```
x <- duration_seconds(c(86399, 86401))

duration_floor(x, "day")
duration_ceiling(x, "day")

# Can't floor from a chronological duration (seconds)
# to a calendrical duration (months)
try(duration_floor(x, "month"))

# Every 2 days, using an origin of day 0
y <- duration_seconds(c(0, 86400, 86400 * 2, 86400 * 3))
duration_floor(y, "day", n = 2)

# Shifting the origin to be day 1
origin <- duration_days(1)
duration_floor(y - origin, "day", n = 2) + origin

# Rounding will round ties up
half_day <- 86400 / 2
half_day_durations <- duration_seconds(c(half_day - 1, half_day, half_day + 1))
duration_round(half_day_durations, "day")

# With larger units
```

```
x <- duration_months(c(0, 15, 24))
duration_floor(x, "year")
duration_floor(x, "quarter")
```

`duration_cast` *Cast a duration between precisions*

Description

Casting is one way to change a duration's precision.

Casting to a less precise precision will completely drop information that is more precise than the precision that you are casting to. It does so in a way that makes it round towards zero.

Casting to a more precise precision is done through a multiplication by a conversion factor between the current precision and the new precision.

Usage

```
duration_cast(x, precision)
```

Arguments

`x`	[clock_duration] A duration.
`precision`	[character(1)] A precision. One of: • `"year"` • `"quarter"` • `"month"` • `"week"` • `"day"` • `"hour"` • `"minute"` • `"second"` • `"millisecond"` • `"microsecond"` • `"nanosecond"`

Details

When you want to change to a less precise precision, you often want `duration_floor()` instead of `duration_cast()`, as that rounds towards negative infinity, which is generally the desired behavior when working with time points (especially ones pre-1970, which are stored as negative durations).

Value

`x` cast to the new `precision`.

Examples

```
x <- duration_seconds(c(86401, -86401))

# Casting rounds towards 0
cast <- duration_cast(x, "day")
cast

# Flooring rounds towards negative infinity
floor <- duration_floor(x, "day")
floor

# Flooring is generally more useful when working with time points,
# note that the cast ends up rounding the pre-1970 date up to the next
# day, while the post-1970 date is rounded down.
as_sys_time(x)
as_sys_time(cast)
as_sys_time(floor)

# Casting to a more precise precision
duration_cast(x, "millisecond")
```

---

duration_precision *Precision: duration*

Description

`duration_precision()` extracts the precision from a duration object. It returns the precision as a single string.

Usage

```
duration_precision(x)
```

Arguments

x	[clock_duration] A duration.

Value

A single string holding the precision of the duration.

Examples

```
duration_precision(duration_seconds(1))
duration_precision(duration_nanoseconds(2))
duration_precision(duration_quarters(1:5))
```

format.clock_zoned_time
Formatting: zoned-time

Description

This is a zoned-time method for the `format()` generic.

This function allows you to format a zoned-time using a flexible `format` string.

If `format` is `NULL`, a default format of `"%Y-%m-%dT%H:%M:%S%Ez[%Z]"` is used. This matches the default format that `zoned_time_parse_complete()` parses. Additionally, this format matches the de-facto standard extension to RFC 3339 for creating completely unambiguous date-times.

Usage

```
## S3 method for class 'clock_zoned_time'
format(x, ..., format = NULL, locale = clock_locale(), abbreviate_zone = FALSE)
```

Arguments

`x`	[clock_zoned_time] A zoned-time.
`...`	[dots] Not used, but no error will be thrown if not empty to remain compatible with usage of the `format()` generic.
`format`	[character(1) / NULL] If `NULL`, a default format is used, which depends on the type of the input. Otherwise, a format string which is a combination of: **Year** • %C: The year divided by 100 using floored division. If the result is a single decimal digit, it is prefixed with `0`. • %y: The last two decimal digits of the year. If the result is a single digit it is prefixed by `0`. • %Y: The year as a decimal number. If the result is less than four digits it is left-padded with `0` to four digits. **Month** • %b, %h: The `locale`'s abbreviated month name. • %B: The `locale`'s full month name. • %m: The month as a decimal number. January is `01`. If the result is a single digit, it is prefixed with `0`. **Day** • %d: The day of month as a decimal number. If the result is a single decimal digit, it is prefixed with `0`. **Day of the week**

- %a: The `locale`'s abbreviated weekday name.
- %A: The `locale`'s full weekday name.
- %w: The weekday as a decimal number (`0`-`6`), where Sunday is `0`.

ISO 8601 week-based year

- %g: The last two decimal digits of the ISO week-based year. If the result is a single digit it is prefixed by `0`.
- %G: The ISO week-based year as a decimal number. If the result is less than four digits it is left-padded with `0` to four digits.
- %V: The ISO week-based week number as a decimal number. If the result is a single digit, it is prefixed with `0`.
- %u: The ISO weekday as a decimal number (`1`-`7`), where Monday is `1`.

Week of the year

- %U: The week number of the year as a decimal number. The first Sunday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. If the result is a single digit, it is prefixed with `0`.
- %W: The week number of the year as a decimal number. The first Monday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. If the result is a single digit, it is prefixed with `0`.

Day of the year

- %j: The day of the year as a decimal number. January 1 is `001`. If the result is less than three digits, it is left-padded with `0` to three digits.

Date

- %D, %x: Equivalent to %m/%d/%y.
- %F: Equivalent to %Y-%m-%d.

Time of day

- %H: The hour (24-hour clock) as a decimal number. If the result is a single digit, it is prefixed with `0`.
- %I: The hour (12-hour clock) as a decimal number. If the result is a single digit, it is prefixed with `0`.
- %M: The minute as a decimal number. If the result is a single digit, it is prefixed with `0`.
- %S: Seconds as a decimal number. Fractional seconds are printed at the precision of the input. The character for the decimal point is localized according to `locale`.
- %p: The `locale`'s equivalent of the AM/PM designations associated with a 12-hour clock.
- %R: Equivalent to %H:%M.
- %T, %X: Equivalent to %H:%M:%S.
- %r: Nearly equivalent to %I:%M:%S %p, but seconds are always printed at second precision.

Time zone

- %z: The offset from UTC in the ISO 8601 format. For example `-0430` refers to 4 hours 30 minutes behind UTC. If the offset is zero, `+0000` is used. The modified command %Ez inserts a `:` between the hour and minutes, like `-04:30`.
- %Z: The full time zone name. If `abbreviate_zone` is `TRUE`, the time zone abbreviation.

Miscellaneous

- %c: A date and time representation. Similar to, but not exactly the same as, %a %b %d %H:%M:%S %Y.
- %%: A % character.
- %n: A newline character.
- %t: A horizontal-tab character.

`locale` [clock_locale]

A locale object created from `clock_locale()`.

`abbreviate_zone`

[logical(1)]

If `TRUE`, %Z returns an abbreviated time zone name.

If `FALSE`, %Z returns the full time zone name.

Value

A character vector of the formatted input.

Examples

```
x <- year_month_day(2019, 1, 1)
x <- as_zoned_time(as_naive_time(x), "America/New_York")

format(x)
format(x, format = "%B %d, %Y")
format(x, format = "%B %d, %Y", locale = clock_locale("fr"))
```

---

iso-year-week-day-arithmetic

Arithmetic: iso-year-week-day

Description

These are iso-year-week-day methods for the arithmetic generics.

- `add_years()`

You cannot add weeks or days to an iso-year-week-day calendar. Adding days is much more efficiently done by converting to a time point first by using `as_naive_time()` or `as_sys_time()`. Adding weeks is equally as efficient as adding 7 days. Additionally, adding weeks to an invalid iso-year-week object containing `iso_year_week_day(2019,53)` would be undefined, as the 53rd ISO week of 2019 doesn't exist to begin with.

Usage

```
## S3 method for class 'clock_iso_year_week_day'
add_years(x, n, ...)
```

Arguments

x	[clock_iso_year_week_day] A iso-year-week-day vector.
n	[integer / clock_duration] An integer vector to be converted to a duration, or a duration corresponding to the arithmetic function being used. This corresponds to the number of duration units to add. n may be negative to subtract units of duration.
...	These dots are for future extensions and must be empty.

Details

x and n are recycled against each other.

Value

x after performing the arithmetic.

Examples

```
x <- iso_year_week_day(2019, 1, 1)
add_years(x, 1:2)
```

---

iso-year-week-day-boundary

Boundaries: iso-year-week-day

Description

This is an iso-year-week-day method for the calendar_start() and calendar_end() generics. They adjust components of a calendar to the start or end of a specified precision.

Usage

```
## S3 method for class 'clock_iso_year_week_day'
calendar_start(x, precision)

## S3 method for class 'clock_iso_year_week_day'
calendar_end(x, precision)
```

Arguments

x	[clock_iso_year_week_day] A iso-year-week-day vector.
precision	[character(1)] One of: • "year" • "week" • "day" • "hour" • "minute" • "second" • "millisecond" • "microsecond" • "nanosecond"

Value

x at the same precision, but with some components altered to be at the boundary value.

Examples

```
x <- iso_year_week_day(2019:2020, 5, 6, 10)
x

# Compute the last moment of the last iso week of the year
calendar_end(x, "year")

# Compare that to just setting the week to `"last"`,
# which doesn't affect the other components
set_week(x, "last")
```

---

iso-year-week-day-count-between

Counting: iso-year-week-day

Description

This is an iso-year-week-day method for the calendar_count_between() generic. It counts the number of precision units between start and end (i.e., the number of ISO years).

Usage

```
## S3 method for class 'clock_iso_year_week_day'
calendar_count_between(start, end, precision, ..., n = 1L)
```

Arguments

`start, end`	[clock_iso_year_week_day] A pair of iso-year-week-day vectors. These will be recycled to their common size.
`precision`	[character(1)] One of: • `"year"`
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Value

An integer representing the number of `precision` units between `start` and `end`.

Examples

```
# Compute the number of whole ISO years between two dates
x <- iso_year_week_day(2001, 1, 2)
y <- iso_year_week_day(2021, 1, c(1, 3))
calendar_count_between(x, y, "year")
```

---

iso-year-week-day-getters

Getters: iso-year-week-day

Description

These are iso-year-week-day methods for the getter generics.

- `get_year()` returns the ISO year. Note that this can differ from the Gregorian year.
- `get_week()` returns the ISO week of the current ISO year.
- `get_day()` returns a value between 1-7 indicating the weekday of the current ISO week, where 1 = Monday and 7 = Sunday, in line with the ISO standard.
- There are sub-daily getters for extracting more precise components.

Usage

```
## S3 method for class 'clock_iso_year_week_day'
get_year(x)

## S3 method for class 'clock_iso_year_week_day'
get_week(x)

## S3 method for class 'clock_iso_year_week_day'
```

```
get_day(x)

## S3 method for class 'clock_iso_year_week_day'
get_hour(x)

## S3 method for class 'clock_iso_year_week_day'
get_minute(x)

## S3 method for class 'clock_iso_year_week_day'
get_second(x)

## S3 method for class 'clock_iso_year_week_day'
get_millisecond(x)

## S3 method for class 'clock_iso_year_week_day'
get_microsecond(x)

## S3 method for class 'clock_iso_year_week_day'
get_nanosecond(x)
```

Arguments

x	[clock_iso_year_week_day] A iso-year-week-day to get the component from.

Value

The component.

Examples

```
x <- iso_year_week_day(2019, 50:52, 1:3)
x

# Get the ISO week
get_week(x)

# Gets the weekday, 1 = Monday, 7 = Sunday
get_day(x)

# Note that the ISO year can differ from the Gregorian year
iso <- iso_year_week_day(2019, 1, 1)
ymd <- as_year_month_day(iso)

get_year(iso)
get_year(ymd)
```

`iso-year-week-day-group` *Grouping: iso-year-week-day*

Description

This is a iso-year-week-day method for the `calendar_group()` generic.

Grouping for a iso-year-week-day object can be done at any precision, as long as `x` is at least as precise as `precision`.

Usage

```
## S3 method for class 'clock_iso_year_week_day'
calendar_group(x, precision, ..., n = 1L)
```

Arguments

`x`	[clock_iso_year_week_day] A iso-year-week-day vector.
`precision`	[character(1)] One of: • `"year"` • `"week"` • `"day"` • `"hour"` • `"minute"` • `"second"` • `"millisecond"` • `"microsecond"` • `"nanosecond"`
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Value

`x` grouped at the specified `precision`.

Examples

```
x <- iso_year_week_day(2019, 1:52)

# Group by 3 ISO weeks
calendar_group(x, "week", n = 3)
```

```
y <- iso_year_week_day(2000:2020, 1, 1)

# Group by 2 ISO years
calendar_group(y, "year", n = 2)
```

`iso-year-week-day-narrow`

Narrow: iso-year-week-day

Description

This is a iso-year-week-day method for the `calendar_narrow()` generic. It narrows a iso-year-week-day vector to the specified `precision`.

Usage

```
## S3 method for class 'clock_iso_year_week_day'
calendar_narrow(x, precision)
```

Arguments

`x`	[clock_iso_year_week_day] A iso-year-week-day vector.
`precision`	[character(1)] One of: • `"year"` • `"week"` • `"day"` • `"hour"` • `"minute"` • `"second"` • `"millisecond"` • `"microsecond"` • `"nanosecond"`

Value

`x` narrowed to the supplied `precision`.

Examples

```
# Day precision
x <- iso_year_week_day(2019, 1, 5)
x

# Narrowed to week precision
calendar_narrow(x, "week")
```

`iso-year-week-day-setters`
Setters: iso-year-week-day

Description

These are iso-year-week-day methods for the setter generics.

- `set_year()` sets the ISO year.
- `set_week()` sets the ISO week of the year. Valid values are in the range of [1, 53].
- `set_day()` sets the day of the week. Valid values are in the range of [1, 7], with 1 = Monday, and 7 = Sunday.
- There are sub-daily setters for setting more precise components.

Usage

```
## S3 method for class 'clock_iso_year_week_day'
set_year(x, value, ...)

## S3 method for class 'clock_iso_year_week_day'
set_week(x, value, ...)

## S3 method for class 'clock_iso_year_week_day'
set_day(x, value, ...)

## S3 method for class 'clock_iso_year_week_day'
set_hour(x, value, ...)

## S3 method for class 'clock_iso_year_week_day'
set_minute(x, value, ...)

## S3 method for class 'clock_iso_year_week_day'
set_second(x, value, ...)

## S3 method for class 'clock_iso_year_week_day'
set_millisecond(x, value, ...)

## S3 method for class 'clock_iso_year_week_day'
set_microsecond(x, value, ...)

## S3 method for class 'clock_iso_year_week_day'
set_nanosecond(x, value, ...)
```

Arguments

x	[clock_iso_year_week_day] A iso-year-week-day vector.

value	[integer / "last"] The value to set the component to. For set_week(), this can also be "last" to adjust to the last week of the current ISO year.
...	These dots are for future extensions and must be empty.

Value

x with the component set.

Examples

```
# Year precision vector
x <- iso_year_week_day(2019:2023)

# Promote to week precision by setting the week
# (Note that some ISO weeks have 52 weeks, and others have 53)
x <- set_week(x, "last")
x

# Set to an invalid week
invalid <- set_week(x, 53)
invalid

# Here are the invalid ones (they only have 52 weeks)
invalid[invalid_detect(invalid)]

# Resolve the invalid dates by choosing the previous/next valid moment
invalid_resolve(invalid, invalid = "previous")
invalid_resolve(invalid, invalid = "next")
```

---

iso-year-week-day-widen

Widen: iso-year-week-day

Description

This is a iso-year-week-day method for the calendar_widen() generic. It widens a iso-year-week-day vector to the specified precision.

Usage

```
## S3 method for class 'clock_iso_year_week_day'
calendar_widen(x, precision)
```

Arguments

x	[clock_iso_year_week_day] A iso-year-week-day vector.
precision	[character(1)] One of: • "year" • "week" • "day" • "hour" • "minute" • "second" • "millisecond" • "microsecond" • "nanosecond"

Value

x widened to the supplied precision.

Examples

```
# Week precision
x <- iso_year_week_day(2019, 1)
x

# Widen to day precision
# In the ISO calendar, the first day of the week is a Monday
calendar_widen(x, "day")

# Or second precision
sec <- calendar_widen(x, "second")
sec
```

---

iso_year_week_day *Calendar: iso-year-week-day*

Description

iso_year_week_day() constructs a calendar from the ISO year, week number, and week day.

Usage

```
iso_year_week_day(
  year,
  week = NULL,
  day = NULL,
  hour = NULL,
  minute = NULL,
  second = NULL,
  subsecond = NULL,
  ...,
  subsecond_precision = NULL
)
```

Arguments

`year`	[integer] The ISO year. Values [-32767, 32767] are generally allowed.
`week`	[integer / "last" / NULL] The ISO week. Values [1, 53] are allowed. If `"last"`, then the last week of the ISO year is returned.
`day`	[integer / NULL] The day of the week. Values [1, 7] are allowed, with 1 = Monday and 7 = Sunday, in accordance with the ISO specifications.
`hour`	[integer / NULL] The hour. Values [0, 23] are allowed.
`minute`	[integer / NULL] The minute. Values [0, 59] are allowed.
`second`	[integer / NULL] The second. Values [0, 59] are allowed.
`subsecond`	[integer / NULL] The subsecond. If specified, `subsecond_precision` must also be specified to determine how to interpret the `subsecond`. If using milliseconds, values [0, 999] are allowed. If using microseconds, values [0, 999999] are allowed. If using nanoseconds, values [0, 999999999] are allowed.
`...`	These dots are for future extensions and must be empty.
`subsecond_precision`	[character(1) / NULL] The precision to interpret `subsecond` as. One of: `"millisecond"`, `"microsecond"`, or `"nanosecond"`.

Details

Fields are recycled against each other.

Fields are collected in order until the first `NULL` field is located. No fields after the first `NULL` field are used.

Value

A iso-year-week-day calendar vector.

Examples

```
# Year-week
x <- iso_year_week_day(2019:2025, 1)
x

# 2nd day of the first ISO week in multiple years
iso_days <- set_day(x, clock_iso_weekdays$tuesday)
iso_days

# What year-month-day is this?
as_year_month_day(iso_days)
```

---

is_duration *Is* x *a duration?*

Description

This function determines if the input is a duration object.

Usage

```
is_duration(x)
```

Arguments

x	[object] An object.

Value

TRUE if x inherits from "clock_duration", otherwise FALSE.

Examples

```
is_duration(1)
is_duration(duration_days(1))
```

is_iso_year_week_day *Is* x *a iso-year-week-day?*

Description

Check if x is a iso-year-week-day.

Usage

```
is_iso_year_week_day(x)
```

Arguments

x	[object] An object.

Value

Returns TRUE if x inherits from "clock_iso_year_week_day", otherwise returns FALSE.

Examples

```
is_iso_year_week_day(iso_year_week_day(2019))
```

is_naive_time *Is* x *a naive-time?*

Description

This function determines if the input is a naive-time object.

Usage

```
is_naive_time(x)
```

Arguments

x	[object] An object.

Value

TRUE if x inherits from "clock_naive_time", otherwise FALSE.

Examples

```
is_naive_time(1)
is_naive_time(as_naive_time(duration_days(1)))
```

is_sys_time *Is* x *a sys-time?*

Description

This function determines if the input is a sys-time object.

Usage

```
is_sys_time(x)
```

Arguments

x	[object] An object.

Value

TRUE if x inherits from "clock_sys_time", otherwise FALSE.

Examples

```
is_sys_time(1)
is_sys_time(as_sys_time(duration_days(1)))
```

is_weekday *Is* x *a weekday?*

Description

This function determines if the input is a weekday object.

Usage

```
is_weekday(x)
```

Arguments

x	[object] An object.

Value

TRUE if x inherits from "clock_weekday", otherwise FALSE.

Examples

```
is_weekday(1)
is_weekday(weekday(1))
```

is_year_day *Is* x *a year-day?*

Description

Check if x is a year-day.

Usage

```
is_year_day(x)
```

Arguments

x	[object] An object.

Value

Returns TRUE if x inherits from "clock_year_day", otherwise returns FALSE.

Examples

```
is_year_day(year_day(2019))
```

is_year_month_day *Is* x *a year-month-day?*

Description

Check if x is a year-month-day.

Usage

```
is_year_month_day(x)
```

Arguments

x	[object] An object.

Value

Returns TRUE if x inherits from "clock_year_month_day", otherwise returns FALSE.

Examples

```
is_year_month_day(year_month_day(2019))
```

is_year_month_weekday *Is* x *a year-month-weekday?*

Description

Check if x is a year-month-weekday.

Usage

```
is_year_month_weekday(x)
```

Arguments

x	[object] An object.

Value

Returns TRUE if x inherits from "clock_year_month_weekday", otherwise returns FALSE.

Examples

```
is_year_month_weekday(year_month_weekday(2019))
```

is_year_quarter_day *Is* x *a year-quarter-day?*

Description

Check if x is a year-quarter-day.

Usage

```
is_year_quarter_day(x)
```

Arguments

x	[object] An object.

Value

Returns TRUE if x inherits from "clock_year_quarter_day", otherwise returns FALSE.

Examples

```
is_year_quarter_day(year_quarter_day(2019))
```

is_zoned_time *Is* x *a zoned-time?*

Description

This function determines if the input is a zoned-time object.

Usage

```
is_zoned_time(x)
```

Arguments

x	[object] An object.

Value

TRUE if x inherits from "clock_zoned_time", otherwise FALSE.

Examples

```
is_zoned_time(1)
is_zoned_time(zoned_time_now("America/New_York"))
```

naive_time_info *Info: naive-time*

Description

naive_time_info() retrieves a set of low-level information generally not required for most date-time manipulations. It is used implicitly by as_zoned_time() when converting from a naive-time.

It returns a data frame with the following columns:

- type: A character vector containing one of:
  - "unique": The naive-time maps uniquely to a zoned-time that can be created with zone.
  - "nonexistent": The naive-time does not exist as a zoned-time that can be created with zone.
  - "ambiguous": The naive-time exists twice as a zoned-time that can be created with zone.
- first: A sys_time_info() data frame.
- second: A sys_time_info() data frame.

type == "unique":

- first will be filled out with sys-info representing daylight saving time information for that time point in zone.

- second will contain only NA values, as there is no ambiguity to represent information for.

type == "nonexistent":

- first will be filled out with the sys-info that ends just prior to x.
- second will be filled out with the sys-info that begins just after x.

type == "ambiguous":

- first will be filled out with the sys-info that ends just after x.
- second will be filled out with the sys-info that starts just before x.

Usage

```
naive_time_info(x, zone)
```

Arguments

x	[clock_naive_time] A naive-time.
zone	[character] A valid time zone name. Unlike most functions in clock, in naive_time_info() zone is vectorized and is recycled against x.

Details

If the tibble package is installed, it is recommended to convert the output to a tibble with as_tibble(), as that will print the df-cols much nicer.

Value

A data frame of low level information.

Examples

```
library(vctrs)

x <- year_month_day(1970, 04, 26, 02, 30, 00)
x <- as_naive_time(x)

# Maps uniquely to a time in London
naive_time_info(x, "Europe/London")

# This naive-time never existed in New York!
# A DST gap jumped the time from 01:59:59 -> 03:00:00,
# skipping the 2 o'clock hour
zone <- "America/New_York"
info <- naive_time_info(x, zone)
info

# You can recreate various `nonexistent` strategies with this info
```

```
as_zoned_time(x, zone, nonexistent = "roll-forward")
as_zoned_time(info$first$end, zone)

as_zoned_time(x, zone, nonexistent = "roll-backward")
as_zoned_time(info$first$end - 1, zone)

as_zoned_time(x, zone, nonexistent = "shift-forward")
as_zoned_time(as_sys_time(x) - info$first$offset, zone)

as_zoned_time(x, zone, nonexistent = "shift-backward")
as_zoned_time(as_sys_time(x) - info$second$offset, zone)

# ------------------------------------------------------------------------
# Normalizing to UTC

# Imagine you had the following printed times, and knowledge that they
# are to be interpreted as in the corresponding time zones
df <- data_frame(
  x = c("2020-01-05 02:30:00", "2020-06-03 12:20:05"),
  zone = c("America/Los_Angeles", "Europe/London")
)

# The times are assumed to be naive-times, i.e. if you lived in the `zone`
# at the moment the time was recorded, then you would have seen that time
# printed on the clock. Currently, these are strings. To convert them to
# a time based type, you'll have to acknowledge that R only lets you have
# 1 time zone in a vector of date-times at a time. So you'll need to
# normalize these naive-times. The easiest thing to normalize them to
# is UTC.
df$naive <- naive_time_parse(df$x)

# Get info about the naive times using a vector of zones
info <- naive_time_info(df$naive, df$zone)
info

# We'll assume that some system generated these naive-times with no
# chance of them ever being nonexistent or ambiguous. So now all we have
# to do is use the offset to convert the naive-time to a sys-time. The
# relationship used is:
# offset = naive_time - sys_time
df$sys <- as_sys_time(df$naive) - info$first$offset
df

# At this point, both times are in UTC. From here, you can convert them
# both to either America/Los_Angeles or Europe/London as required.
as_zoned_time(df$sys, "America/Los_Angeles")
as_zoned_time(df$sys, "Europe/London")
```

---

naive_time_parse *Parsing: naive-time*

Description

`naive_time_parse()` is a parser into a naive-time.

`naive_time_parse()` is useful when you have date-time strings like `"2020-01-01T01:04:30"`. If there is no attached UTC offset or time zone name, then parsing this string as a naive-time is your best option. If you know that this string should be interpreted in a specific time zone, parse as a naive-time, then use `as_zoned_time()`.

The default options assume that `x` should be parsed at second precision, using a `format` string of `"%Y-%m-%dT%H:%M:%S"`. This matches the default result from calling `format()` on a naive-time.

`naive_time_parse()` *ignores both the* %z *and* %Z *commands.*

If your date-time strings contain a full time zone name and a UTC offset, use `zoned_time_parse_complete()`. If they contain a time zone abbreviation, use `zoned_time_parse_abbrev()`.

If your date-time strings contain a UTC offset, but not a full time zone name, use `sys_time_parse()`.

Usage

```
naive_time_parse(
  x,
  ...,
  format = NULL,
  precision = "second",
  locale = clock_locale()
)
```

Arguments

`x`	[character] A character vector to parse.
`...`	These dots are for future extensions and must be empty.
`format`	[character / NULL] A format string. A combination of the following commands, or `NULL`, in which case a default format string is used. A vector of multiple format strings can be supplied. They will be tried in the order they are provided. **Year** • %C: The century as a decimal number. The modified command %NC where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required. • %y: The last two decimal digits of the year. If the century is not otherwise specified (e.g. with %C), values in the range [69 - 99] are presumed to refer to the years [1969 - 1999], and values in the range [00 - 68] are presumed to refer to the years [2000 - 2068]. The modified command %Ny, where `N` is a positive decimal integer, specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.

- %Y: The year as a decimal number. The modified command %NY where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 4. Leading zeroes are permitted but not required.

Month

- %b, %B, %h: The locale's full or abbreviated case-insensitive month name.
- %m: The month as a decimal number. January is 1. The modified command %Nm where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.

Day

- %d, %e: The day of the month as a decimal number. The modified command %Nd where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.

Day of the week

- %a, %A: The locale's full or abbreviated case-insensitive weekday name.
- %w: The weekday as a decimal number (0-6), where Sunday is 0. The modified command %Nw where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 1. Leading zeroes are permitted but not required.

ISO 8601 week-based year

- %g: The last two decimal digits of the ISO week-based year. The modified command %Ng where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
- %G: The ISO week-based year as a decimal number. The modified command %NG where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 4. Leading zeroes are permitted but not required.
- %V: The ISO week-based week number as a decimal number. The modified command %NV where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
- %u: The ISO weekday as a decimal number (1-7), where Monday is 1. The modified command %Nu where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 1. Leading zeroes are permitted but not required.

Week of the year

- %U: The week number of the year as a decimal number. The first Sunday of the year is the first day of week 01. Days of the same year prior to that are in week 00. The modified command %NU where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.

- %W: The week number of the year as a decimal number. The first Monday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. The modified command %NW where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.

Day of the year

- %j: The day of the year as a decimal number. January 1 is `1`. The modified command %Nj where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `3`. Leading zeroes are permitted but not required.

Date

- %D, %x: Equivalent to %m/%d/%y.
- %F: Equivalent to %Y-%m-%d. If modified with a width (like %NF), the width is applied to only %Y.

Time of day

- %H: The hour (24-hour clock) as a decimal number. The modified command %NH where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %I: The hour (12-hour clock) as a decimal number. The modified command %NI where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %M: The minutes as a decimal number. The modified command %NM where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %S: The seconds as a decimal number. Leading zeroes are permitted but not required. If encountered, the `locale` determines the decimal point character. Generally, the maximum number of characters to read is determined by the precision that you are parsing at. For example, a precision of `"second"` would read a maximum of 2 characters, while a precision of `"millisecond"` would read a maximum of 6 (2 for the values before the decimal point, 1 for the decimal point, and 3 for the values after it). The modified command %NS, where `N` is a positive decimal integer, can be used to exactly specify the maximum number of characters to read. This is only useful if you happen to have seconds with more than 1 leading zero.
- %p: The `locale`'s equivalent of the AM/PM designations associated with a 12-hour clock. The command %I must precede %p in the format string.
- %R: Equivalent to %H:%M.
- %T, %X: Equivalent to %H:%M:%S.
- %r: Equivalent to %I:%M:%S %p.

Time zone

- %z: The offset from UTC in the format [+|-]hh[mm]. For example `-0430` refers to 4 hours 30 minutes behind UTC. And `04` refers to 4 hours ahead

of UTC. The modified command %Ez parses a : between the hours and minutes and leading zeroes on the hour field are optional: [+|-]h[h][:mm]. For example `-04:30` refers to 4 hours 30 minutes behind UTC. And 4 refers to 4 hours ahead of UTC.

- %Z: The full time zone name or the time zone abbreviation, depending on the function being used. A single word is parsed. This word can only contain characters that are alphanumeric, or one of `'_'`, `'/'`, `'-'` or `'+'`.

Miscellaneous

- %c: A date and time representation. Equivalent to %a %b %d %H:%M:%S %Y.
- `%%`: A % character.
- %n: Matches one white space character. %n, %t, and a space can be combined to match a wide range of white-space patterns. For example `"%n "` matches one or more white space characters, and `"%n%t%t"` matches one to three white space characters.
- %t: Matches zero or one white space characters.

`precision` [character(1)]

A precision for the resulting time point. One of:

- `"day"`
- `"hour"`
- `"minute"`
- `"second"`
- `"millisecond"`
- `"microsecond"`
- `"nanosecond"`

Setting the `precision` determines how much information %S attempts to parse.

`locale` [clock_locale]

A locale object created from `clock_locale()`.

Value

A naive-time.

Full Precision Parsing

It is highly recommended to parse all of the information in the date-time string into a type at least as precise as the string. For example, if your string has fractional seconds, but you only require seconds, specify a sub-second `precision`, then round to seconds manually using whatever convention is appropriate for your use case. Parsing such a string directly into a second precision result is ambiguous and undefined, and is unlikely to work as you might expect.

Examples

```
naive_time_parse("2020-01-01T05:06:07")

# Day precision
naive_time_parse("2020-01-01", precision = "day")
```

```
# Nanosecond precision, but using a day based format
naive_time_parse("2020-01-01", format = "%Y-%m-%d", precision = "nanosecond")

# Remember that the `%z` and `%Z` commands are ignored entirely!
naive_time_parse(
  "2020-01-01 -4000 America/New_York",
  format = "%Y-%m-%d %z %Z"
)

# ---------------------------------------------------------------------------
# Fractional seconds and POSIXct

# If you have a string with fractional seconds and want to convert it to
# a POSIXct, remember that clock treats POSIXct as a second precision type.
# Ideally, you'd use a clock type that can support fractional seconds, but
# if you really want to parse it into a POSIXct, the correct way to do so
# is to parse the full fractional time point with the correct `precision`,
# then round to seconds using whatever convention you require, and finally
# convert that to POSIXct.
x <- c("2020-01-01T00:00:00.123", "2020-01-01T00:00:00.555")

# First, parse string with full precision
x <- naive_time_parse(x, precision = "millisecond")
x

# Then round to second with a floor, ceiling, or round to nearest
time_point_floor(x, "second")
time_point_round(x, "second")

# Finally, convert to POSIXct
as_date_time(time_point_round(x, "second"), zone = "UTC")
```

`posixt-arithmetic` *Arithmetic: date-time*

Description

These are POSIXct/POSIXlt methods for the arithmetic generics.

Calendrical based arithmetic:

These functions convert to a naive-time, then to a year-month-day, perform the arithmetic, then convert back to a date-time.

- `add_years()`
- `add_quarters()`
- `add_months()`

Naive-time based arithmetic:

These functions convert to a naive-time, perform the arithmetic, then convert back to a date-time.

- add_weeks()
- add_days()

Sys-time based arithmetic:

These functions convert to a sys-time, perform the arithmetic, then convert back to a date-time.

- add_hours()
- add_minutes()
- add_seconds()

Usage

```
## S3 method for class 'POSIXt'
add_years(x, n, ..., invalid = NULL, nonexistent = NULL, ambiguous = x)

## S3 method for class 'POSIXt'
add_quarters(x, n, ..., invalid = NULL, nonexistent = NULL, ambiguous = x)

## S3 method for class 'POSIXt'
add_months(x, n, ..., invalid = NULL, nonexistent = NULL, ambiguous = x)

## S3 method for class 'POSIXt'
add_weeks(x, n, ..., nonexistent = NULL, ambiguous = x)

## S3 method for class 'POSIXt'
add_days(x, n, ..., nonexistent = NULL, ambiguous = x)

## S3 method for class 'POSIXt'
add_hours(x, n, ...)

## S3 method for class 'POSIXt'
add_minutes(x, n, ...)

## S3 method for class 'POSIXt'
add_seconds(x, n, ...)
```

Arguments

x	[POSIXct / POSIXlt] A date-time vector.
n	[integer / clock_duration] An integer vector to be converted to a duration, or a duration corresponding to the arithmetic function being used. This corresponds to the number of duration units to add. n may be negative to subtract units of duration.
...	These dots are for future extensions and must be empty.
invalid	[character(1) / NULL] One of the following invalid date resolution strategies:

- "previous": The previous valid instant in time.
- "previous-day": The previous valid day in time, keeping the time of day.
- "next": The next valid instant in time.
- "next-day": The next valid day in time, keeping the time of day.
- "overflow": Overflow by the number of days that the input is invalid by. Time of day is dropped.
- "overflow-day": Overflow by the number of days that the input is invalid by. Time of day is kept.
- "NA": Replace invalid dates with NA.
- "error": Error on invalid dates.

Using either "previous" or "next" is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input.

If NULL, defaults to "error".

If getOption("clock.strict") is TRUE, invalid must be supplied and cannot be NULL. This is a convenient way to make production code robust to invalid dates.

nonexistent [character / NULL]

One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input:

- "roll-forward": The next valid instant in time.
- "roll-backward": The previous valid instant in time.
- "shift-forward": Shift the nonexistent time forward by the size of the daylight saving time gap.
- "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap.
- "NA": Replace nonexistent times with NA.
- "error": Error on nonexistent times.

Using either "roll-forward" or "roll-backward" is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input.

If NULL, defaults to "error".

If getOption("clock.strict") is TRUE, nonexistent must be supplied and cannot be NULL. This is a convenient way to make production code robust to nonexistent times.

ambiguous [character / zoned_time / POSIXct / list(2) / NULL]

One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input:

- "earliest": Of the two possible times, choose the earliest one.
- "latest": Of the two possible times, choose the latest one.
- "NA": Replace ambiguous times with NA.
- "error": Error on ambiguous times.

Alternatively, ambiguous is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a

naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to `NULL`.

Finally, `ambiguous` is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `ambiguous` must be supplied and cannot be `NULL`. Additionally, `ambiguous` cannot be specified as a zoned_time on its own, as this implies `NULL` for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

Details

Adding a single quarter with `add_quarters()` is equivalent to adding 3 months.

`x` and `n` are recycled against each other.

Calendrical based arithmetic has the potential to generate invalid dates (like the 31st of February), nonexistent times (due to daylight saving time gaps), and ambiguous times (due to daylight saving time fallbacks).

Naive-time based arithmetic will never generate an invalid date, but may generate a nonexistent or ambiguous time (i.e. you added 1 day and landed in a daylight saving time gap).

Sys-time based arithmetic operates in the UTC time zone, which means that it will never generate any invalid dates or nonexistent / ambiguous times.

The conversion from POSIXct/POSIXlt to the corresponding clock type uses a "best guess" about whether you want to do the arithmetic using a naive-time or a sys-time. For example, when adding months, you probably want to retain the printed time when converting to a year-month-day to perform the arithmetic, so the conversion goes through naive-time. However, when adding smaller units like seconds, you probably want "2020-03-08 01:59:59" + 1 second in the America/New_York time zone to return `"2020-03-08 03:00:00"`, taking into account the fact that there was a daylight saving time gap. This requires doing the arithmetic in sys-time, so that is what clock converts to. If you disagree with this heuristic for any reason, you can take control and perform the conversions yourself. For example, you could convert the previous example to a naive-time instead of a sys-time manually with `as_naive_time()`, add 1 second giving `"2020-03-08 02:00:00"`, then convert back to a POSIXct/POSIXlt, dealing with the nonexistent time that gets created by using the `nonexistent` argument of `as.POSIXct()`.

Value

`x` after performing the arithmetic.

Examples

```
x <- as.POSIXct("2019-01-01", tz = "America/New_York")

add_years(x, 1:5)

y <- as.POSIXct("2019-01-31 00:30:00", tz = "America/New_York")

# Adding 1 month to `y` generates an invalid date. Unlike year-month-day
# types, R's native date-time types cannot handle invalid dates, so you must
# resolve them immediately. If you don't you get an error:
try(add_months(y, 1:2))
add_months(as_year_month_day(y), 1:2)

# Resolve invalid dates by specifying an invalid date resolution strategy
# with the `invalid` argument. Using `"previous"` here sets the date-time to
# the previous valid moment in time - i.e. the end of the month. The
# time is set to the last moment in the day to retain the relative ordering
# within your input. If you are okay with potentially losing this, and
# want to retain your time of day, you can use `"previous-day"` to set the
# date-time to the previous valid day, while keeping the time of day.
add_months(y, 1:2, invalid = "previous")
add_months(y, 1:2, invalid = "previous-day")
```

---

posixt-boundary *Boundaries: date-time*

Description

This is a POSIXct/POSIXlt method for the `date_start()` and `date_end()` generics.

Usage

```
## S3 method for class 'POSIXt'
date_start(
  x,
  precision,
  ...,
  invalid = NULL,
  nonexistent = NULL,
  ambiguous = x
)

## S3 method for class 'POSIXt'
date_end(x, precision, ..., invalid = NULL, nonexistent = NULL, ambiguous = x)
```

Arguments

x	[POSIXct / POSIXlt] A date-time vector.

`precision` [character(1)]

One of:

- `"year"`
- `"month"`
- `"day"`
- `"hour"`
- `"minute"`
- `"second"`

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

`invalid` [character(1) / NULL]

One of the following invalid date resolution strategies:

- `"previous"`: The previous valid instant in time.
- `"previous-day"`: The previous valid day in time, keeping the time of day.
- `"next"`: The next valid instant in time.
- `"next-day"`: The next valid day in time, keeping the time of day.
- `"overflow"`: Overflow by the number of days that the input is invalid by. Time of day is dropped.
- `"overflow-day"`: Overflow by the number of days that the input is invalid by. Time of day is kept.
- `"NA"`: Replace invalid dates with `NA`.
- `"error"`: Error on invalid dates.

Using either `"previous"` or `"next"` is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `invalid` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to invalid dates.

`nonexistent` [character / NULL]

One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"roll-forward"`: The next valid instant in time.
- `"roll-backward"`: The previous valid instant in time.
- `"shift-forward"`: Shift the nonexistent time forward by the size of the daylight saving time gap.
- "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap.
- `"NA"`: Replace nonexistent times with `NA`.
- `"error"`: Error on nonexistent times.

Using either `"roll-forward"` or `"roll-backward"` is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `nonexistent` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to nonexistent times.

ambiguous [character / zoned_time / POSIXct / list(2) / NULL]

One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"earliest"`: Of the two possible times, choose the earliest one.
- `"latest"`: Of the two possible times, choose the latest one.
- `"NA"`: Replace ambiguous times with `NA`.
- `"error"`: Error on ambiguous times.

Alternatively, `ambiguous` is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to `NULL`.

Finally, `ambiguous` is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `ambiguous` must be supplied and cannot be `NULL`. Additionally, `ambiguous` cannot be specified as a zoned_time on its own, as this implies `NULL` for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

Value

`x` but with some components altered to be at the boundary value.

Examples

```
x <- date_time_build(2019:2021, 2:4, 3:5, 4, 5, 6, zone = "America/New_York")
x

# Last moment of the month
date_end(x, "month")

# Notice that this is different from just setting the day to `"last"`
set_day(x, "last")

# Last moment of the year
date_end(x, "year")

# First moment of the hour
date_start(x, "hour")
```

`posixt-count-between` *Counting: date-times*

Description

This is a POSIXct/POSIXlt method for the `date_count_between()` generic.

`date_count_between()` counts the number of `precision` units between `start` and `end` (i.e., the number of years or months). This count corresponds to the *whole number* of units, and will never return a fractional value.

This is suitable for, say, computing the whole number of years or months between two dates, accounting for the day of the month and the time of day.

Internally, the date-time is converted to one of the following three clock types, and the counting is done directly on that type. The choice of type is based on the most common interpretation of each precision, but is ultimately a heuristic. See the examples for more information.

Calendrical based counting:

These precisions convert to a year-month-day calendar and count while in that type.

- `"year"`
- `"quarter"`
- `"month"`

Naive-time based counting:

These precisions convert to a naive-time and count while in that type.

- `"week"`
- `"day"`

Sys-time based counting:

These precisions convert to a sys-time and count while in that type.

- `"hour"`
- `"minute"`
- `"second"`

Usage

```
## S3 method for class 'POSIXt'
date_count_between(start, end, precision, ..., n = 1L)
```

Arguments

start, end	[POSIXct / POSIXlt] A pair of date-time vectors. These will be recycled to their common size.
precision	[character(1)] One of: • "year" • "quarter" • "month" • "week" • "day" • "hour" • "minute" • "second"
...	These dots are for future extensions and must be empty.
n	[positive integer(1)] A single positive integer specifying a multiple of precision to use.

Details

"quarter" is equivalent to "month" precision with n set to n * 3L.

Value

An integer representing the number of precision units between start and end.

Comparison Direction

The computed count has the property that if start <= end, then start + <count> <= end. Similarly, if start >= end, then start + <count> >= end. In other words, the comparison direction between start and end will never change after adding the count to start. This makes this function useful for repeated count computations at increasingly fine precisions.

Examples

```
start <- date_time_parse("2000-05-05 02:00:00", zone = "America/New_York")
end <- date_time_parse(
  c("2020-05-05 01:00:00", "2020-05-05 03:00:00"),
  zone = "America/New_York"
)

# Age in years
date_count_between(start, end, "year")

# Number of "whole" months between these dates. i.e.
# `2000-05-05 02:00:00 -> 2020-04-05 02:00:00` is 239 months
# `2000-05-05 02:00:00 -> 2020-05-05 02:00:00` is 240 months
# Since `2020-05-05 01:00:00` occurs before the 2nd hour,
# it gets a count of 239
```

```
date_count_between(start, end, "month")

# Number of seconds between
date_count_between(start, end, "second")

# ----------------------------------------------------------------------------
# Naive-time VS Sys-time interpretation

# The difference between whether `start` and `end` are converted to a
# naive-time vs a sys-time comes into play when dealing with daylight
# savings.

# Here are two times around a 1 hour DST gap where clocks jumped from
# 01:59:59 -> 03:00:00
x <- date_time_build(1970, 4, 26, 1, 50, 00, zone = "America/New_York")
y <- date_time_build(1970, 4, 26, 3, 00, 00, zone = "America/New_York")

# When treated like sys-times, these are considered to be 10 minutes apart,
# which is the amount of time that would have elapsed if you were watching
# a clock as it changed between these two times.
date_count_between(x, y, "minute")

# Lets add a 3rd date that is ~1 day ahead of these
z <- date_time_build(1970, 4, 27, 1, 55, 00, zone = "America/New_York")

# When treated like naive-times, `z` is considered to be at least 1 day ahead
# of `x`, because `01:55:00` is after `01:50:00`. This is probably what you
# expected.
date_count_between(x, z, "day")

# If these were interpreted like sys-times, then `z` would not be considered
# to be 1 day ahead. That would look something like this:
date_count_between(x, z, "second")
trunc(date_count_between(x, z, "second") / 86400)

# This is because there have only been 83,100 elapsed seconds since `x`,
# which isn't a full day's worth (86,400 seconds). But we'd generally
# consider `z` to be 1 day ahead of `x` (and ignore the DST gap), so that is
# how it is implemented.

# You can override this by converting directly to sys-time, then using
# `time_point_count_between()`
x_st <- as_sys_time(x)
x_st

z_st <- as_sys_time(z)
z_st

time_point_count_between(x_st, z_st, "day")
```

`posixt-formatting` *Formatting: date-time*

Description

This is a POSIXct method for the `date_format()` generic.

`date_format()` formats a date-time (POSIXct) using a `format` string.

If `format` is `NULL`, a default format of `"%Y-%m-%dT%H:%M:%S%Ez[%Z]"` is used. This matches the default format that `date_time_parse_complete()` parses. Additionally, this format matches the de-facto standard extension to RFC 3339 for creating completely unambiguous date-times.

Usage

```
## S3 method for class 'POSIXt'
date_format(
  x,
  ...,
  format = NULL,
  locale = clock_locale(),
  abbreviate_zone = FALSE
)
```

Arguments

`x`	[POSIXct / POSIXlt] A date-time vector.
`...`	These dots are for future extensions and must be empty.
`format`	[character(1) / NULL] If `NULL`, a default format is used, which depends on the type of the input. Otherwise, a format string which is a combination of: **Year** • %C: The year divided by 100 using floored division. If the result is a single decimal digit, it is prefixed with `0`. • %y: The last two decimal digits of the year. If the result is a single digit it is prefixed by `0`. • %Y: The year as a decimal number. If the result is less than four digits it is left-padded with `0` to four digits. **Month** • %b, %h: The `locale`'s abbreviated month name. • %B: The `locale`'s full month name. • %m: The month as a decimal number. January is `01`. If the result is a single digit, it is prefixed with `0`. **Day** • %d: The day of month as a decimal number. If the result is a single decimal digit, it is prefixed with `0`.

Day of the week

- %a: The `locale`'s abbreviated weekday name.
- %A: The `locale`'s full weekday name.
- %w: The weekday as a decimal number (`0`-`6`), where Sunday is `0`.

ISO 8601 week-based year

- %g: The last two decimal digits of the ISO week-based year. If the result is a single digit it is prefixed by `0`.
- %G: The ISO week-based year as a decimal number. If the result is less than four digits it is left-padded with `0` to four digits.
- %V: The ISO week-based week number as a decimal number. If the result is a single digit, it is prefixed with `0`.
- %u: The ISO weekday as a decimal number (`1`-`7`), where Monday is `1`.

Week of the year

- %U: The week number of the year as a decimal number. The first Sunday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. If the result is a single digit, it is prefixed with `0`.
- %W: The week number of the year as a decimal number. The first Monday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. If the result is a single digit, it is prefixed with `0`.

Day of the year

- %j: The day of the year as a decimal number. January 1 is `001`. If the result is less than three digits, it is left-padded with `0` to three digits.

Date

- %D, %x: Equivalent to %m/%d/%y.
- %F: Equivalent to %Y-%m-%d.

Time of day

- %H: The hour (24-hour clock) as a decimal number. If the result is a single digit, it is prefixed with `0`.
- %I: The hour (12-hour clock) as a decimal number. If the result is a single digit, it is prefixed with `0`.
- %M: The minute as a decimal number. If the result is a single digit, it is prefixed with `0`.
- %S: Seconds as a decimal number. Fractional seconds are printed at the precision of the input. The character for the decimal point is localized according to `locale`.
- %p: The `locale`'s equivalent of the AM/PM designations associated with a 12-hour clock.
- %R: Equivalent to %H:%M.
- %T, %X: Equivalent to %H:%M:%S.
- %r: Nearly equivalent to %I:%M:%S %p, but seconds are always printed at second precision.

Time zone

- %z: The offset from UTC in the ISO 8601 format. For example `-0430` refers to 4 hours 30 minutes behind UTC. If the offset is zero, `+0000` is used. The modified command %Ez inserts a `:` between the hour and minutes, like `-04:30`.
- %Z: The full time zone name. If `abbreviate_zone` is `TRUE`, the time zone abbreviation.

Miscellaneous

- %c: A date and time representation. Similar to, but not exactly the same as, %a %b %d %H:%M:%S %Y.
- `%%`: A % character.
- %n: A newline character.
- %t: A horizontal-tab character.

`locale` [clock_locale]

A locale object created from `clock_locale()`.

`abbreviate_zone`

[logical(1)]

If `TRUE`, %Z returns an abbreviated time zone name.

If `FALSE`, %Z returns the full time zone name.

Value

A character vector of the formatted input.

Examples

```
x <- date_time_parse(
  c("1970-04-26 01:30:00", "1970-04-26 03:30:00"),
  zone = "America/New_York"
)

# Default
date_format(x)

# Which is parseable by `date_time_parse_complete()`
date_time_parse_complete(date_format(x))

date_format(x, format = "%B %d, %Y %H:%M:%S")

# By default, `%Z` uses the full zone name, but you can switch to the
# abbreviated name
date_format(x, format = "%z %Z")
date_format(x, format = "%z %Z", abbreviate_zone = TRUE)
```

`posixt-getters` *Getters: date-time*

Description

These are POSIXct/POSIXlt methods for the getter generics.

- `get_year()` returns the Gregorian year.
- `get_month()` returns the month of the year.
- `get_day()` returns the day of the month.
- There are sub-daily getters for extracting more precise components, up to a precision of seconds.

For more advanced component extraction, convert to the calendar type that you are interested in.

Usage

```
## S3 method for class 'POSIXt'
get_year(x)

## S3 method for class 'POSIXt'
get_month(x)

## S3 method for class 'POSIXt'
get_day(x)

## S3 method for class 'POSIXt'
get_hour(x)

## S3 method for class 'POSIXt'
get_minute(x)

## S3 method for class 'POSIXt'
get_second(x)
```

Arguments

`x`	[POSIXct / POSIXlt] A date-time type to get the component from.

Value

The component.

Examples

```
x <- as.POSIXct("2019-01-01", tz = "America/New_York")

x <- add_days(x, 0:5)
x <- set_second(x, 10:15)

get_day(x)
get_second(x)
```

`posixt-group` *Group date-time components*

Description

This is a POSIXct/POSIXlt method for the `date_group()` generic.

`date_group()` groups by a single component of a date-time, such as month of the year, day of the month, or hour of the day.

If you need to group by more complex components, like ISO weeks, or quarters, convert to a calendar type that contains the component you are interested in grouping by.

Usage

```
## S3 method for class 'POSIXt'
date_group(
  x,
  precision,
  ...,
  n = 1L,
  invalid = NULL,
  nonexistent = NULL,
  ambiguous = x
)
```

Arguments

`x`	[POSIXct / POSIXlt] A date-time vector.
`precision`	[character(1)] One of: • `"year"` • `"month"` • `"day"` • `"hour"` • `"minute"` • `"second"`

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

`n` [positive integer(1)]

A single positive integer specifying a multiple of `precision` to use.

`invalid` [character(1) / NULL]

One of the following invalid date resolution strategies:

- `"previous"`: The previous valid instant in time.
- `"previous-day"`: The previous valid day in time, keeping the time of day.
- `"next"`: The next valid instant in time.
- `"next-day"`: The next valid day in time, keeping the time of day.
- `"overflow"`: Overflow by the number of days that the input is invalid by. Time of day is dropped.
- `"overflow-day"`: Overflow by the number of days that the input is invalid by. Time of day is kept.
- `"NA"`: Replace invalid dates with `NA`.
- `"error"`: Error on invalid dates.

Using either `"previous"` or `"next"` is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `invalid` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to invalid dates.

`nonexistent` [character / NULL]

One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"roll-forward"`: The next valid instant in time.
- `"roll-backward"`: The previous valid instant in time.
- `"shift-forward"`: Shift the nonexistent time forward by the size of the daylight saving time gap.
- "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap.
- `"NA"`: Replace nonexistent times with `NA`.
- `"error"`: Error on nonexistent times.

Using either `"roll-forward"` or `"roll-backward"` is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `nonexistent` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to nonexistent times.

`ambiguous` [character / zoned_time / POSIXct / list(2) / NULL]

One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"earliest"`: Of the two possible times, choose the earliest one.

- `"latest"`: Of the two possible times, choose the latest one.
- `"NA"`: Replace ambiguous times with `NA`.
- `"error"`: Error on ambiguous times.

Alternatively, `ambiguous` is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to `NULL`.

Finally, `ambiguous` is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `ambiguous` must be supplied and cannot be `NULL`. Additionally, `ambiguous` cannot be specified as a zoned_time on its own, as this implies `NULL` for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

Value

`x`, grouped at `precision`.

Examples

```
x <- as.POSIXct("2019-01-01", "America/New_York")
x <- add_days(x, -3:5)

# Group by 2 days of the current month.
# Note that this resets at the beginning of the month, creating day groups
# of [29, 30] [31] [01, 02] [03, 04].
date_group(x, "day", n = 2)

# Group by month
date_group(x, "month")

# Group by hour of the day
y <- as.POSIXct("2019-12-30", "America/New_York")
y <- add_hours(y, 0:12)
y

date_group(y, "hour", n = 3)
```

`posixt-rounding` *Rounding: date-time*

Description

These are POSIXct/POSIXlt methods for the rounding generics.

- `date_floor()` rounds a date-time down to a multiple of the specified `precision`.
- `date_ceiling()` rounds a date-time up to a multiple of the specified `precision`.
- `date_round()` rounds up or down depending on what is closer, rounding up on ties.

You can group by irregular periods such as `"month"` or `"year"` by using `date_group()`.

Usage

```
## S3 method for class 'POSIXt'
date_floor(
  x,
  precision,
  ...,
  n = 1L,
  origin = NULL,
  nonexistent = NULL,
  ambiguous = x
)

## S3 method for class 'POSIXt'
date_ceiling(
  x,
  precision,
  ...,
  n = 1L,
  origin = NULL,
  nonexistent = NULL,
  ambiguous = x
)

## S3 method for class 'POSIXt'
date_round(
  x,
  precision,
  ...,
  n = 1L,
  origin = NULL,
  nonexistent = NULL,
  ambiguous = x
)
```

Arguments

x	[POSIXct / POSIXlt] A date-time vector.
precision	[character(1)] One of: • "week" • "day" • "hour" • "minute" • "second" "week" is an alias for "day" with n * 7.
...	These dots are for future extensions and must be empty.
n	[positive integer(1)] A single positive integer specifying a multiple of precision to use.
origin	[POSIXct(1) / POSIXlt(1) / NULL] An origin to start counting from. origin must have exactly the same time zone as x. origin will be floored to precision. If information is lost when flooring, a warning will be thrown. If NULL, defaults to midnight on 1970-01-01 in the time zone of x.
nonexistent	[character / NULL] One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input: • "roll-forward": The next valid instant in time. • "roll-backward": The previous valid instant in time. • "shift-forward": Shift the nonexistent time forward by the size of the daylight saving time gap. • "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap. • "NA": Replace nonexistent times with NA. • "error": Error on nonexistent times. Using either "roll-forward" or "roll-backward" is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input. If NULL, defaults to "error". If getOption("clock.strict") is TRUE, nonexistent must be supplied and cannot be NULL. This is a convenient way to make production code robust to nonexistent times.
ambiguous	[character / zoned_time / POSIXct / list(2) / NULL] One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input: • "earliest": Of the two possible times, choose the earliest one.

- `"latest"`: Of the two possible times, choose the latest one.
- `"NA"`: Replace ambiguous times with `NA`.
- `"error"`: Error on ambiguous times.

Alternatively, `ambiguous` is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to `NULL`.

Finally, `ambiguous` is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `ambiguous` must be supplied and cannot be `NULL`. Additionally, `ambiguous` cannot be specified as a zoned_time on its own, as this implies `NULL` for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

Details

When rounding by `"week"`, remember that the `origin` determines the "week start". By default, 1970-01-01 is the implicit origin, which is a Thursday. If you would like to round by weeks with a different week start, just supply an origin on the weekday you are interested in.

Value

`x` rounded to the specified `precision`.

Examples

```
x <- as.POSIXct("2019-03-31", "America/New_York")
x <- add_days(x, 0:5)

# Flooring by 2 days, note that this is not tied to the current month,
# and instead counts from the specified `origin`, so groups can cross
# the month boundary
date_floor(x, "day", n = 2)

# Compare to `date_group()`, which groups by the day of the month
date_group(x, "day", n = 2)

# Note that daylight saving time gaps can throw off rounding
x <- as.POSIXct("1970-04-26 01:59:59", "America/New_York") + c(0, 1)
x
```

```
# Rounding is done in naive-time, which means that rounding by 2 hours
# will attempt to generate a time of 1970-04-26 02:00:00, which doesn't
# exist in this time zone
try(date_floor(x, "hour", n = 2))

# You can handle this by specifying a nonexistent time resolution strategy
date_floor(x, "hour", n = 2, nonexistent = "roll-forward")
```

`posixt-sequence` *Sequences: date-time*

Description

This is a POSIXct method for the `date_seq()` generic.

`date_seq()` generates a date-time (POSIXct) sequence.

When calling `date_seq()`, exactly two of the following must be specified:

- `to`
- `by`
- `total_size`

Usage

```
## S3 method for class 'POSIXt'
date_seq(
  from,
  ...,
  to = NULL,
  by = NULL,
  total_size = NULL,
  invalid = NULL,
  nonexistent = NULL,
  ambiguous = NULL
)
```

Arguments

`from`	[POSIXct(1) / POSIXlt(1)] A date-time to start the sequence from. `from` is always included in the result.
`...`	These dots are for future extensions and must be empty.
`to`	[POSIXct(1) / POSIXlt(1) / NULL] A date-time to stop the sequence at. `to` is only included in the result if the resulting sequence divides the distance between `from` and `to` exactly.

If `to` is supplied along with `by`, all components of `to` more precise than the precision of `by` must match `from` exactly. For example, if `by = duration_months(1)`, the day, hour, minute, and second components of `to` must match the corresponding components of `from`. This ensures that the generated sequence is, at a minimum, a weakly monotonic sequence of date-times.

The time zone of `to` must match the time zone of `from` exactly.

`by` [integer(1) / clock_duration(1) / NULL]

The unit to increment the sequence by.

If `to < from`, then `by` must be positive.

If `to > from`, then `by` must be negative.

If `by` is an integer, it is equivalent to `duration_seconds(by)`.

If `by` is a duration, it is allowed to have a precision of:

- year
- quarter
- month
- week
- day
- hour
- minute
- second

`total_size` [positive integer(1) / NULL]

The size of the resulting sequence.

If specified alongside `to`, this must generate a non-fractional sequence between `from` and `to`.

`invalid` [character(1) / NULL]

One of the following invalid date resolution strategies:

- `"previous"`: The previous valid instant in time.
- `"previous-day"`: The previous valid day in time, keeping the time of day.
- `"next"`: The next valid instant in time.
- `"next-day"`: The next valid day in time, keeping the time of day.
- `"overflow"`: Overflow by the number of days that the input is invalid by. Time of day is dropped.
- `"overflow-day"`: Overflow by the number of days that the input is invalid by. Time of day is kept.
- `"NA"`: Replace invalid dates with `NA`.
- `"error"`: Error on invalid dates.

Using either `"previous"` or `"next"` is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `invalid` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to invalid dates.

`nonexistent` [character / NULL]

One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"roll-forward"`: The next valid instant in time.
- `"roll-backward"`: The previous valid instant in time.
- `"shift-forward"`: Shift the nonexistent time forward by the size of the daylight saving time gap.
- "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap.
- `"NA"`: Replace nonexistent times with `NA`.
- `"error"`: Error on nonexistent times.

Using either `"roll-forward"` or `"roll-backward"` is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `nonexistent` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to nonexistent times.

`ambiguous` [character / zoned_time / POSIXct / list(2) / NULL]

One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"earliest"`: Of the two possible times, choose the earliest one.
- `"latest"`: Of the two possible times, choose the latest one.
- `"NA"`: Replace ambiguous times with `NA`.
- `"error"`: Error on ambiguous times.

Alternatively, `ambiguous` is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to `NULL`.

Finally, `ambiguous` is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `ambiguous` must be supplied and cannot be `NULL`. Additionally, `ambiguous` cannot be specified as a zoned_time on its own, as this implies `NULL` for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

Value

A date-time vector.

Sequence Generation

Different methods are used to generate the sequences, depending on the precision implied by by. They are intended to generate the most intuitive sequences, especially around daylight saving time gaps and fallbacks.

See the examples for more details.

Calendrical based sequences::

These convert to a naive-time, then to a year-month-day, generate the sequence, then convert back to a date-time.

- by = `duration_years()`
- by = `duration_quarters()`
- by = `duration_months()`

Naive-time based sequences::

These convert to a naive-time, generate the sequence, then convert back to a date-time.

- by = `duration_weeks()`
- by = `duration_days()`

Sys-time based sequences::

These convert to a sys-time, generate the sequence, then convert back to a date-time.

- by = `duration_hours()`
- by = `duration_minutes()`
- by = `duration_seconds()`

Examples

```
zone <- "America/New_York"

from <- date_time_build(2019, 1, zone = zone)
to <- date_time_build(2019, 1, second = 50, zone = zone)

# Defaults to second precision sequence
date_seq(from, to = to, by = 7)

to <- date_time_build(2019, 1, 5, zone = zone)

# Use durations to change to alternative precisions
date_seq(from, to = to, by = duration_days(1))
date_seq(from, to = to, by = duration_hours(10))
date_seq(from, by = duration_minutes(-2), total_size = 3)

# Note that components of `to` more precise than the precision of `by`
# must match `from` exactly. For example, this is not well defined:
from <- date_time_build(2019, 1, 1, 0, 1, 30, zone = zone)
```

```
to <- date_time_build(2019, 1, 1, 5, 2, 20, zone = zone)
try(date_seq(from, to = to, by = duration_hours(1)))

# The minute and second components of `to` must match `from`
to <- date_time_build(2019, 1, 1, 5, 1, 30, zone = zone)
date_seq(from, to = to, by = duration_hours(1))

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

# Invalid dates must be resolved with the `invalid` argument
from <- date_time_build(2019, 1, 31, zone = zone)
to <- date_time_build(2019, 12, 31, zone = zone)

try(date_seq(from, to = to, by = duration_months(1)))
date_seq(from, to = to, by = duration_months(1), invalid = "previous-day")

# Compare this to the base R result, which is often a source of confusion
seq(from, to = to, by = "1 month")

# This is equivalent to the overflow invalid resolution strategy
date_seq(from, to = to, by = duration_months(1), invalid = "overflow")

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

# This date-time is 2 days before a daylight saving time gap that occurred
# on 2021-03-14 between 01:59:59 -> 03:00:00
from <- as.POSIXct("2021-03-12 02:30:00", "America/New_York")

# So creating a daily sequence lands us in that daylight saving time gap,
# creating a nonexistent time
try(date_seq(from, by = duration_days(1), total_size = 5))

# Resolve the nonexistent time with `nonexistent`. Note that this importantly
# allows times after the gap to retain the `02:30:00` time.
date_seq(from, by = duration_days(1), total_size = 5, nonexistent = "roll-forward")

# Compare this to the base R behavior, where the hour is adjusted from 2->3
# as you cross the daylight saving time gap, and is never restored. This is
# equivalent to always using sys-time (rather than naive-time, like clock
# uses for daily sequences).
seq(from, by = "1 day", length.out = 5)

# You can replicate this behavior by generating a second precision sequence
# of 86,400 seconds. Seconds always add in sys-time.
date_seq(from, by = duration_seconds(86400), total_size = 5)

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

# Usage of `to` and `total_size` must generate a non-fractional sequence
# between `from` and `to`
from <- date_time_build(2019, 1, 1, 0, 0, 0, zone = "America/New_York")
to <- date_time_build(2019, 1, 1, 0, 0, 3, zone = "America/New_York")
```

```
# These are fine
date_seq(from, to = to, total_size = 2)
date_seq(from, to = to, total_size = 4)

# But this is not!
try(date_seq(from, to = to, total_size = 3))
```

`posixt-setters` *Setters: date-time*

Description

These are POSIXct/POSIXlt methods for the setter generics.

- `set_year()` sets the year.
- `set_month()` sets the month of the year. Valid values are in the range of [1, 12].
- `set_day()` sets the day of the month. Valid values are in the range of [1, 31].
- There are sub-daily setters for setting more precise components, up to a precision of seconds.

Usage

```
## S3 method for class 'POSIXt'
set_year(x, value, ..., invalid = NULL, nonexistent = NULL, ambiguous = x)

## S3 method for class 'POSIXt'
set_month(x, value, ..., invalid = NULL, nonexistent = NULL, ambiguous = x)

## S3 method for class 'POSIXt'
set_day(x, value, ..., invalid = NULL, nonexistent = NULL, ambiguous = x)

## S3 method for class 'POSIXt'
set_hour(x, value, ..., invalid = NULL, nonexistent = NULL, ambiguous = x)

## S3 method for class 'POSIXt'
set_minute(x, value, ..., invalid = NULL, nonexistent = NULL, ambiguous = x)

## S3 method for class 'POSIXt'
set_second(x, value, ..., invalid = NULL, nonexistent = NULL, ambiguous = x)
```

Arguments

`x`	[POSIXct / POSIXlt] A date-time vector.
`value`	[integer / "last"] The value to set the component to. For `set_day()`, this can also be `"last"` to set the day to the last day of the month.

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

`invalid` [character(1) / NULL]

One of the following invalid date resolution strategies:

- `"previous"`: The previous valid instant in time.
- `"previous-day"`: The previous valid day in time, keeping the time of day.
- `"next"`: The next valid instant in time.
- `"next-day"`: The next valid day in time, keeping the time of day.
- `"overflow"`: Overflow by the number of days that the input is invalid by. Time of day is dropped.
- `"overflow-day"`: Overflow by the number of days that the input is invalid by. Time of day is kept.
- `"NA"`: Replace invalid dates with `NA`.
- `"error"`: Error on invalid dates.

Using either `"previous"` or `"next"` is generally recommended, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `invalid` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to invalid dates.

`nonexistent` [character / NULL]

One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"roll-forward"`: The next valid instant in time.
- `"roll-backward"`: The previous valid instant in time.
- `"shift-forward"`: Shift the nonexistent time forward by the size of the daylight saving time gap.
- "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap.
- `"NA"`: Replace nonexistent times with `NA`.
- `"error"`: Error on nonexistent times.

Using either `"roll-forward"` or `"roll-backward"` is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input.

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `nonexistent` must be supplied and cannot be `NULL`. This is a convenient way to make production code robust to nonexistent times.

`ambiguous` [character / zoned_time / POSIXct / list(2) / NULL]

One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input:

- `"earliest"`: Of the two possible times, choose the earliest one.
- `"latest"`: Of the two possible times, choose the latest one.
- `"NA"`: Replace ambiguous times with `NA`.

- `"error"`: Error on ambiguous times.

Alternatively, `ambiguous` is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to `NULL`.

Finally, `ambiguous` is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If `NULL`, defaults to `"error"`.

If `getOption("clock.strict")` is `TRUE`, `ambiguous` must be supplied and cannot be `NULL`. Additionally, `ambiguous` cannot be specified as a zoned_time on its own, as this implies `NULL` for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

Value

`x` with the component set.

Examples

```
x <- as.POSIXct("2019-02-01", tz = "America/New_York")

# Set the day
set_day(x, 12:14)

# Set to the "last" day of the month
set_day(x, "last")

# You cannot set a date-time to an invalid date like you can with
# a year-month-day. Instead, the default strategy is to error.
try(set_day(x, 31))
set_day(as_year_month_day(x), 31)

# You can resolve these issues while setting the day by specifying
# an invalid date resolution strategy with `invalid`
set_day(x, 31, invalid = "previous")

y <- as.POSIXct("2020-03-08 01:30:00", tz = "America/New_York")

# Nonexistent and ambiguous times must be resolved immediately when
# working with R's native date-time types. An error is thrown by default.
try(set_hour(y, 2))
set_hour(y, 2, nonexistent = "roll-forward")
```

```
set_hour(y, 2, nonexistent = "roll-backward")
```

`posixt-shifting` *Shifting: date and date-time*

Description

`date_shift()` shifts `x` to the `target` weekday. You can shift to the next or previous weekday. If `x` is currently on the `target` weekday, you can choose to leave it alone or advance it to the next instance of the `target`.

Shifting with date-times retains the time of day where possible. Be aware that you can run into daylight saving time issues if you shift into a daylight saving time gap or fallback period.

Usage

```
## S3 method for class 'POSIXt'
date_shift(
  x,
  target,
  ...,
  which = "next",
  boundary = "keep",
  nonexistent = NULL,
  ambiguous = x
)
```

Arguments

`x`	[POSIXct / POSIXlt] A date-time vector.
`target`	[weekday] A weekday created from `weekday()` to target. Generally this is length 1, but can also be the same length as `x`.
`...`	These dots are for future extensions and must be empty.
`which`	[character(1)] One of: • `"next"`: Shift to the next instance of the `target` weekday. • "previous: Shift to the previous instance of the `target` weekday.
`boundary`	[character(1)] One of: • `"keep"`: If `x` is currently on the `target` weekday, return it. • `"advance"`: If `x` is currently on the `target` weekday, advance it anyways.
`nonexistent`	[character / NULL] One of the following nonexistent time resolution strategies, allowed to be either length 1, or the same length as the input:

- "roll-forward": The next valid instant in time.
- "roll-backward": The previous valid instant in time.
- "shift-forward": Shift the nonexistent time forward by the size of the daylight saving time gap.
- "shift-backward: Shift the nonexistent time backward by the size of the daylight saving time gap.
- "NA": Replace nonexistent times with NA.
- "error": Error on nonexistent times.

Using either "roll-forward" or "roll-backward" is generally recommended over shifting, as these two strategies maintain the *relative ordering* between elements of the input.

If NULL, defaults to "error".

If getOption("clock.strict") is TRUE, nonexistent must be supplied and cannot be NULL. This is a convenient way to make production code robust to nonexistent times.

ambiguous [character / zoned_time / POSIXct / list(2) / NULL]

One of the following ambiguous time resolution strategies, allowed to be either length 1, or the same length as the input:

- "earliest": Of the two possible times, choose the earliest one.
- "latest": Of the two possible times, choose the latest one.
- "NA": Replace ambiguous times with NA.
- "error": Error on ambiguous times.

Alternatively, ambiguous is allowed to be a zoned_time (or POSIXct) that is either length 1, or the same length as the input. If an ambiguous time is encountered, the zoned_time is consulted. If the zoned_time corresponds to a naive_time that is also ambiguous *and* uses the same daylight saving time transition point as the original ambiguous time, then the offset of the zoned_time is used to resolve the ambiguity. If the ambiguity cannot be resolved by consulting the zoned_time, then this method falls back to NULL.

Finally, ambiguous is allowed to be a list of size 2, where the first element of the list is a zoned_time (as described above), and the second element of the list is an ambiguous time resolution strategy to use when the ambiguous time cannot be resolved by consulting the zoned_time. Specifying a zoned_time on its own is identical to list(<zoned_time>, NULL).

If NULL, defaults to "error".

If getOption("clock.strict") is TRUE, ambiguous must be supplied and cannot be NULL. Additionally, ambiguous cannot be specified as a zoned_time on its own, as this implies NULL for ambiguous times that the zoned_time cannot resolve. Instead, it must be specified as a list alongside an ambiguous time resolution strategy as described above. This is a convenient way to make production code robust to ambiguous times.

Value

x shifted to the target weekday.

Examples

```
tuesday <- weekday(clock_weekdays$tuesday)

x <- as.POSIXct("1970-04-22 02:30:00", "America/New_York")

# Shift to the next Tuesday
date_shift(x, tuesday)

# Be aware that you can run into daylight saving time issues!
# Here we shift directly into a daylight saving time gap
# from 01:59:59 -> 03:00:00
sunday <- weekday(clock_weekdays$sunday)
try(date_shift(x, sunday))

# You can resolve this with the `nonexistent` argument
date_shift(x, sunday, nonexistent = "roll-forward")
```

---

seq.clock_duration *Sequences: duration*

Description

This is a duration method for the `seq()` generic.

Using `seq()` on duration objects always retains the type of `from`.

When calling `seq()`, exactly two of the following must be specified:

- `to`
- `by`
- Either `length.out` or `along.with`

Usage

```
## S3 method for class 'clock_duration'
seq(from, to = NULL, by = NULL, length.out = NULL, along.with = NULL, ...)
```

Arguments

`from`	[clock_duration(1)] A duration to start the sequence from. `from` is always included in the result.
`to`	[clock_duration(1) / NULL] A duration to stop the sequence at. `to` is cast to the type of `from`. `to` is only included in the result if the resulting sequence divides the distance between `from` and `to` exactly.

by	[integer(1) / clock_duration(1) / NULL] The unit to increment the sequence by. If to < from, then by must be positive. If to > from, then by must be negative. If by is an integer, it is transformed into a duration with the precision of from. If by is a duration, it is cast to the type of from.
length.out	[positive integer(1) / NULL] The length of the resulting sequence. If specified, along.with must be NULL.
along.with	[vector / NULL] A vector who's length determines the length of the resulting sequence. Equivalent to length.out = vec_size(along.with). If specified, length.out must be NULL.
...	These dots are for future extensions and must be empty.

Value

A sequence with the type of from.

Examples

```
seq(duration_days(0), duration_days(100), by = 5)

# Using a duration `by`. Note that `by` is cast to the type of `from`.
seq(duration_days(0), duration_days(100), by = duration_weeks(1))

# `to` is cast from 5 years to 60 months
# `by` is cast from 1 quarter to 4 months
seq(duration_months(0), duration_years(5), by = duration_quarters(1))

seq(duration_days(20), by = 2, length.out = 5)
```

---

seq.clock_iso_year_week_day

Sequences: iso-year-week-day

Description

This is a iso-year-week-day method for the seq() generic.

Sequences can only be generated for "year" precision iso-year-week-day vectors. If you need to generate week-based sequences, you'll have to convert to a time point first.

When calling seq(), exactly two of the following must be specified:

- to
- by
- Either length.out or along.with

Usage

```
## S3 method for class 'clock_iso_year_week_day'
seq(from, to = NULL, by = NULL, length.out = NULL, along.with = NULL, ...)
```

Arguments

from	[clock_iso_year_week_day(1)] A "year" precision iso-year-week-day to start the sequence from. from is always included in the result.
to	[clock_iso_year_week_day(1) / NULL] A "year" precision iso-year-week-day to stop the sequence at. to is cast to the type of from. to is only included in the result if the resulting sequence divides the distance between from and to exactly.
by	[integer(1) / clock_duration(1) / NULL] The unit to increment the sequence by. If to < from, then by must be positive. If to > from, then by must be negative. If by is an integer, it is transformed into a duration with the precision of from. If by is a duration, it is cast to the type of from.
length.out	[positive integer(1) / NULL] The length of the resulting sequence. If specified, along.with must be NULL.
along.with	[vector / NULL] A vector who's length determines the length of the resulting sequence. Equivalent to length.out = vec_size(along.with). If specified, length.out must be NULL.
...	These dots are for future extensions and must be empty.

Value

A sequence with the type of from.

Examples

```
# Yearly sequence
x <- seq(iso_year_week_day(2020), iso_year_week_day(2026), by = 2)
x

# Which we can then set the week of.
# Some years have 53 ISO weeks, some have 52.
set_week(x, "last")
```

`seq.clock_time_point` *Sequences: time points*

Description

This is a time point method for the `seq()` generic. It works for sys-time and naive-time vectors.

Sequences can be generated for all valid time point precisions (daily through nanosecond).

When calling `seq()`, exactly two of the following must be specified:

- `to`
- `by`
- Either `length.out` or `along.with`

Usage

```
## S3 method for class 'clock_time_point'
seq(from, to = NULL, by = NULL, length.out = NULL, along.with = NULL, ...)
```

Arguments

`from`	[clock_sys_time(1) / clock_naive_time(1)] A time point to start the sequence from. `from` is always included in the result.
`to`	[clock_sys_time(1) / clock_naive_time(1) / NULL] A time point to stop the sequence at. `to` is cast to the type of `from`. `to` is only included in the result if the resulting sequence divides the distance between `from` and `to` exactly.
`by`	[integer(1) / clock_duration(1) / NULL] The unit to increment the sequence by. If `to` < `from`, then `by` must be positive. If `to` > `from`, then `by` must be negative. If `by` is an integer, it is transformed into a duration with the precision of `from`. If `by` is a duration, it is cast to the type of `from`.
`length.out`	[positive integer(1) / NULL] The length of the resulting sequence. If specified, `along.with` must be `NULL`.
`along.with`	[vector / NULL] A vector who's length determines the length of the resulting sequence. Equivalent to `length.out = vec_size(along.with)`. If specified, `length.out` must be `NULL`.
`...`	These dots are for future extensions and must be empty.

Value

A sequence with the type of `from`.

Examples

```
# Daily sequence
seq(
  as_naive_time(year_month_day(2019, 1, 1)),
  as_naive_time(year_month_day(2019, 2, 4)),
  by = 5
)

# Minutely sequence using minute precision naive-time
x <- as_naive_time(year_month_day(2019, 1, 2, 3, 3))
x

seq(x, by = 4, length.out = 10)

# You can use larger step sizes by using a duration-based `by`
seq(x, by = duration_days(1), length.out = 5)

# Nanosecond sequence
from <- as_naive_time(year_month_day(2019, 1, 1))
from <- time_point_cast(from, "nanosecond")
to <- from + 100
seq(from, to, by = 10)
```

`seq.clock_year_day` *Sequences: year-day*

Description

This is a year-day method for the `seq()` generic.

Sequences can only be generated for `"year"` precision year-day vectors.

When calling `seq()`, exactly two of the following must be specified:

- `to`
- `by`
- Either `length.out` or `along.with`

Usage

```
## S3 method for class 'clock_year_day'
seq(from, to = NULL, by = NULL, length.out = NULL, along.with = NULL, ...)
```

Arguments

from	[clock_year_day(1)]
	A "year" precision year-day to start the sequence from.
	from is always included in the result.
to	[clock_year_day(1) / NULL]
	A "year" precision year-day to stop the sequence at.
	to is cast to the type of from.
	to is only included in the result if the resulting sequence divides the distance between from and to exactly.
by	[integer(1) / clock_duration(1) / NULL]
	The unit to increment the sequence by.
	If to < from, then by must be positive.
	If to > from, then by must be negative.
	If by is an integer, it is transformed into a duration with the precision of from.
	If by is a duration, it is cast to the type of from.
length.out	[positive integer(1) / NULL]
	The length of the resulting sequence.
	If specified, along.with must be NULL.
along.with	[vector / NULL]
	A vector who's length determines the length of the resulting sequence.
	Equivalent to length.out = vec_size(along.with).
	If specified, length.out must be NULL.
...	These dots are for future extensions and must be empty.

Value

A sequence with the type of from.

Examples

```
# Yearly sequence
x <- seq(year_day(2020), year_day(2040), by = 2)
x

# Which we can then set the day of to get a sequence of end-of-year values
set_day(x, "last")

# Daily sequences are not allowed. Use a naive-time for this instead.
try(seq(year_day(2019, 1), by = 2, length.out = 2))
as_year_day(seq(as_naive_time(year_day(2019, 1)), by = 2, length.out = 2))
```

`seq.clock_year_month_day` *Sequences: year-month-day*

Description

This is a year-month-day method for the `seq()` generic.

Sequences can only be generated for `"year"` and `"month"` precision year-month-day vectors.

When calling `seq()`, exactly two of the following must be specified:

- `to`
- `by`
- Either `length.out` or `along.with`

Usage

```
## S3 method for class 'clock_year_month_day'
seq(from, to = NULL, by = NULL, length.out = NULL, along.with = NULL, ...)
```

Arguments

`from`	[clock_year_month_day(1)] A `"year"` or `"month"` precision year-month-day to start the sequence from. `from` is always included in the result.
`to`	[clock_year_month_day(1) / NULL] A `"year"` or `"month"` precision year-month-day to stop the sequence at. `to` is cast to the type of `from`. `to` is only included in the result if the resulting sequence divides the distance between `from` and `to` exactly.
`by`	[integer(1) / clock_duration(1) / NULL] The unit to increment the sequence by. If `to` < `from`, then `by` must be positive. If `to` > `from`, then `by` must be negative. If `by` is an integer, it is transformed into a duration with the precision of `from`. If `by` is a duration, it is cast to the type of `from`.
`length.out`	[positive integer(1) / NULL] The length of the resulting sequence. If specified, `along.with` must be `NULL`.
`along.with`	[vector / NULL] A vector who's length determines the length of the resulting sequence. Equivalent to `length.out = vec_size(along.with)`. If specified, `length.out` must be `NULL`.
`...`	These dots are for future extensions and must be empty.

Value

A sequence with the type of `from`.

Examples

```
# Monthly sequence
x <- seq(year_month_day(2019, 1), year_month_day(2020, 12), by = 1)
x

# Which we can then set the day of to get a sequence of end-of-month values
set_day(x, "last")

# Daily sequences are not allowed. Use a naive-time for this instead.
try(seq(year_month_day(2019, 1, 1), by = 2, length.out = 2))
seq(as_naive_time(year_month_day(2019, 1, 1)), by = 2, length.out = 2)
```

---

seq.clock_year_month_weekday

Sequences: year-month-weekday

Description

This is a year-month-weekday method for the `seq()` generic.

Sequences can only be generated for `"year"` and `"month"` precision year-month-weekday vectors.

When calling `seq()`, exactly two of the following must be specified:

- `to`
- `by`
- Either `length.out` or `along.with`

Usage

```
## S3 method for class 'clock_year_month_weekday'
seq(from, to = NULL, by = NULL, length.out = NULL, along.with = NULL, ...)
```

Arguments

`from`	[clock_year_month_weekday(1)] A `"year"` or `"month"` precision year-month-weekday to start the sequence from. `from` is always included in the result.
`to`	[clock_year_month_weekday(1) / NULL] A `"year"` or `"month"` precision year-month-weekday to stop the sequence at. `to` is cast to the type of `from`. `to` is only included in the result if the resulting sequence divides the distance between `from` and `to` exactly.

by	[integer(1) / clock_duration(1) / NULL] The unit to increment the sequence by. If to < from, then by must be positive. If to > from, then by must be negative. If by is an integer, it is transformed into a duration with the precision of from. If by is a duration, it is cast to the type of from.
length.out	[positive integer(1) / NULL] The length of the resulting sequence. If specified, along.with must be NULL.
along.with	[vector / NULL] A vector who's length determines the length of the resulting sequence. Equivalent to length.out = vec_size(along.with). If specified, length.out must be NULL.
...	These dots are for future extensions and must be empty.

Value

A sequence with the type of from.

Examples

```
# Monthly sequence
x <- seq(year_month_weekday(2019, 1), year_month_weekday(2020, 12), by = 1)
x

# Which we can then set the indexed weekday of
set_day(x, clock_weekdays$sunday, index = "last")
```

---

seq.clock_year_quarter_day

Sequences: year-quarter-day

Description

This is a year-quarter-day method for the seq() generic.

Sequences can only be generated for "year" and "quarter" precision year-quarter-day vectors.

When calling seq(), exactly two of the following must be specified:

- to
- by
- Either length.out or along.with

Usage

```
## S3 method for class 'clock_year_quarter_day'
seq(from, to = NULL, by = NULL, length.out = NULL, along.with = NULL, ...)
```

Arguments

from	[clock_year_quarter_day(1)]
	A "year" or "quarter" precision year-quarter-day to start the sequence from.
	from is always included in the result.
to	[clock_year_quarter_day(1) / NULL]
	A "year" or "quarter" precision year-quarter-day to stop the sequence at.
	to is cast to the type of from.
	to is only included in the result if the resulting sequence divides the distance between from and to exactly.
by	[integer(1) / clock_duration(1) / NULL]
	The unit to increment the sequence by.
	If to < from, then by must be positive.
	If to > from, then by must be negative.
	If by is an integer, it is transformed into a duration with the precision of from.
	If by is a duration, it is cast to the type of from.
length.out	[positive integer(1) / NULL]
	The length of the resulting sequence.
	If specified, along.with must be NULL.
along.with	[vector / NULL]
	A vector who's length determines the length of the resulting sequence.
	Equivalent to length.out = vec_size(along.with).
	If specified, length.out must be NULL.
...	These dots are for future extensions and must be empty.

Value

A sequence with the type of from.

Examples

```
# Quarterly sequence
x <- seq(year_quarter_day(2020, 1), year_quarter_day(2026, 3), by = 2)
x

# Which we can then set the day of the quarter of
set_day(x, "last")
```

---

sys-parsing *Parsing: sys-time*

Description

There are two parsers into a sys-time, `sys_time_parse()` and `sys_time_parse_RFC_3339()`.

sys_time_parse():

`sys_time_parse()` is useful when you have date-time strings like `"2020-01-01T01:04:30"` that you know should be interpreted as UTC, or like `"2020-01-01T01:04:30-04:00"` with a UTC offset but no zone name. If you find yourself in the latter situation, then parsing this string as a sys-time using the %Ez command to capture the offset is probably your best option. If you know that this string should be interpreted in a specific time zone, parse as a sys-time to get the UTC equivalent, then use `as_zoned_time()`.

The default options assume that `x` should be parsed at second precision, using a `format` string of `"%Y-%m-%dT%H:%M:%S"`. This matches the default result from calling `format()` on a sys-time.

`sys_time_parse()` is nearly equivalent to `naive_time_parse()`, except for the fact that the %z command is actually used. Using %z assumes that the rest of the date-time string should be interpreted as a naive-time, which is then shifted by the UTC offset found in %z. The returned time can then be validly interpreted as UTC.

`sys_time_parse()` *ignores the %Z command.*

sys_time_parse_RFC_3339():

`sys_time_parse_RFC_3339()` is a wrapper around `sys_time_parse()` that is intended to parse the extremely common date-time format outlined by RFC 3339. This document outlines a profile of the ISO 8601 format that is even more restrictive.

In particular, this function is intended to parse the following three formats:

```
2019-01-01T00:00:00Z
2019-01-01T00:00:00+0430
2019-01-01T00:00:00+04:30
```

This function defaults to parsing the first of these formats by using a format string of `"%Y-%m-%dT%H:%M:%SZ"`.

If your date-time strings use offsets from UTC rather than `"Z"`, then set `offset` to one of the following:

- `"%z"` if the offset is of the form `"+0430"`.
- `"%Ez"` if the offset is of the form `"+04:30"`.

The RFC 3339 standard allows for replacing the `"T"` with a `"t"` or a space (`" "`). Set `separator` to adjust this as needed.

For this function, the `precision` must be at least `"second"`.

Usage

```
sys_time_parse(
  x,
  ...,
  format = NULL,
  precision = "second",
  locale = clock_locale()
)

sys_time_parse_RFC_3339(
```

```
  x,
  ...,
  separator = "T",
  offset = "Z",
  precision = "second"
)
```

Arguments

`x`	[character] A character vector to parse.
`...`	These dots are for future extensions and must be empty.
`format`	[character / NULL] A format string. A combination of the following commands, or `NULL`, in which case a default format string is used. A vector of multiple format strings can be supplied. They will be tried in the order they are provided. **Year** • %C: The century as a decimal number. The modified command %NC where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required. • %y: The last two decimal digits of the year. If the century is not otherwise specified (e.g. with %C), values in the range [69 - 99] are presumed to refer to the years [1969 - 1999], and values in the range [00 - 68] are presumed to refer to the years [2000 - 2068]. The modified command %Ny, where `N` is a positive decimal integer, specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required. • %Y: The year as a decimal number. The modified command %NY where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `4`. Leading zeroes are permitted but not required. **Month** • %b, %B, %h: The `locale`'s full or abbreviated case-insensitive month name. • %m: The month as a decimal number. January is `1`. The modified command %Nm where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required. **Day** • %d, %e: The day of the month as a decimal number. The modified command %Nd where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required. **Day of the week**

- %a, %A: The `locale`'s full or abbreviated case-insensitive weekday name.
- %w: The weekday as a decimal number (`0`-`6`), where Sunday is `0`. The modified command %Nw where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `1`. Leading zeroes are permitted but not required.

ISO 8601 week-based year

- %g: The last two decimal digits of the ISO week-based year. The modified command %Ng where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %G: The ISO week-based year as a decimal number. The modified command %NG where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `4`. Leading zeroes are permitted but not required.
- %V: The ISO week-based week number as a decimal number. The modified command %NV where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %u: The ISO weekday as a decimal number (`1`-`7`), where Monday is `1`. The modified command %Nu where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `1`. Leading zeroes are permitted but not required.

Week of the year

- %U: The week number of the year as a decimal number. The first Sunday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. The modified command %NU where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %W: The week number of the year as a decimal number. The first Monday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. The modified command %NW where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.

Day of the year

- %j: The day of the year as a decimal number. January 1 is `1`. The modified command %Nj where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `3`. Leading zeroes are permitted but not required.

Date

- %D, %x: Equivalent to %m/%d/%y.
- %F: Equivalent to %Y-%m-%d. If modified with a width (like %NF), the width is applied to only %Y.

Time of day

- %H: The hour (24-hour clock) as a decimal number. The modified command %NH where `N` is a positive decimal integer specifies the maximum

number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.

- %I: The hour (12-hour clock) as a decimal number. The modified command %NI where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %M: The minutes as a decimal number. The modified command %NM where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %S: The seconds as a decimal number. Leading zeroes are permitted but not required. If encountered, the `locale` determines the decimal point character. Generally, the maximum number of characters to read is determined by the precision that you are parsing at. For example, a precision of `"second"` would read a maximum of 2 characters, while a precision of `"millisecond"` would read a maximum of 6 (2 for the values before the decimal point, 1 for the decimal point, and 3 for the values after it). The modified command %NS, where `N` is a positive decimal integer, can be used to exactly specify the maximum number of characters to read. This is only useful if you happen to have seconds with more than 1 leading zero.
- %p: The `locale`'s equivalent of the AM/PM designations associated with a 12-hour clock. The command %I must precede %p in the format string.
- %R: Equivalent to %H:%M.
- %T, %X: Equivalent to %H:%M:%S.
- %r: Equivalent to %I:%M:%S %p.

Time zone

- %z: The offset from UTC in the format [+|-]hh[mm]. For example `-0430` refers to 4 hours 30 minutes behind UTC. And `04` refers to 4 hours ahead of UTC. The modified command %Ez parses a `:` between the hours and minutes and leading zeroes on the hour field are optional: [+|-]h[h][:mm]. For example `-04:30` refers to 4 hours 30 minutes behind UTC. And `4` refers to 4 hours ahead of UTC.
- %Z: The full time zone name or the time zone abbreviation, depending on the function being used. A single word is parsed. This word can only contain characters that are alphanumeric, or one of `'_'`, `'/'`, `'-'` or `'+'`.

Miscellaneous

- %c: A date and time representation. Equivalent to %a %b %d %H:%M:%S %Y.
- `%%`: A % character.
- %n: Matches one white space character. %n, %t, and a space can be combined to match a wide range of white-space patterns. For example `"%n "` matches one or more white space characters, and `"%n%t%t"` matches one to three white space characters.
- %t: Matches zero or one white space characters.

`precision` [character(1)]

A precision for the resulting time point. One of:

- "day"
- "hour"
- "minute"
- "second"
- "millisecond"
- "microsecond"
- "nanosecond"

Setting the precision determines how much information %S attempts to parse.

locale [clock_locale]

A locale object created from clock_locale().

separator [character(1)]

The separator between the date and time components of the string. One of:

- "T"
- "t"
- " "

offset [character(1)]

The format of the offset from UTC contained in the string. One of:

- "Z"
- "z"
- "%z" to parse a numeric offset of the form "+0430"
- "%Ez" to parse a numeric offset of the form "+04:30"

Details

If your date-time strings contain a full time zone name and a UTC offset, use zoned_time_parse_complete(). If they contain a time zone abbreviation, use zoned_time_parse_abbrev().

If your date-time strings don't contain an offset from UTC and you aren't sure if they should be treated as UTC or not, you might consider using naive_time_parse(), since the resulting naive-time doesn't come with an assumption of a UTC time zone.

Value

A sys-time.

Full Precision Parsing

It is highly recommended to parse all of the information in the date-time string into a type at least as precise as the string. For example, if your string has fractional seconds, but you only require seconds, specify a sub-second precision, then round to seconds manually using whatever convention is appropriate for your use case. Parsing such a string directly into a second precision result is ambiguous and undefined, and is unlikely to work as you might expect.

Examples

```
sys_time_parse("2020-01-01T05:06:07")

# Day precision
sys_time_parse("2020-01-01", precision = "day")

# Nanosecond precision, but using a day based format
sys_time_parse("2020-01-01", format = "%Y-%m-%d", precision = "nanosecond")

# Multiple format strings are allowed for heterogeneous times
sys_time_parse(
  c("2019-01-01", "2019/1/1"),
  format = c("%Y/%m/%d", "%Y-%m-%d"),
  precision = "day"
)

# The `%z` command shifts the date-time by subtracting the UTC offset so
# that the returned sys-time can be interpreted as UTC
sys_time_parse(
  "2020-01-01 02:00:00 -0400",
  format = "%Y-%m-%d %H:%M:%S %z"
)

# Remember that the `%Z` command is ignored entirely!
sys_time_parse("2020-01-01 America/New_York", format = "%Y-%m-%d %Z")

# ---------------------------------------------------------------------------
# RFC 3339

# Typical UTC format
x <- "2019-01-01T00:01:02Z"
sys_time_parse_RFC_3339(x)

# With a UTC offset containing a `:`
x <- "2019-01-01T00:01:02+02:30"
sys_time_parse_RFC_3339(x, offset = "%Ez")

# With a space between the date and time and no `:` in the offset
x <- "2019-01-01 00:01:02+0230"
sys_time_parse_RFC_3339(x, separator = " ", offset = "%z")
```

---

sys_time_info *Info: sys-time*

Description

`sys_time_info()` retrieves a set of low-level information generally not required for most date-time manipulations. It returns a data frame with the following columns:

- `begin`, `end`: Second precision sys-times specifying the range of the current daylight saving time rule. The range is a half-open interval of [begin, end).
- `offset`: A second precision `duration` specifying the offset from UTC.
- `dst`: A logical vector specifying if daylight saving time is currently active.
- `abbreviation`: The time zone abbreviation in use throughout this `begin` to `end` range.

Usage

```
sys_time_info(x, zone)
```

Arguments

`x`	[clock_sys_time] A sys-time.
`zone`	[character] A valid time zone name. Unlike most functions in clock, in `sys_time_info()` `zone` is vectorized and is recycled against `x`.

Details

If there have never been any daylight saving time transitions, the minimum supported year value is returned for `begin` (typically, a year value of `-32767`).

If daylight saving time is no longer used in a time zone, the maximum supported year value is returned for `end` (typically, a year value of `32767`).

The `offset` is the bridge between sys-time and naive-time for the `zone` being used. The relationship of the three values is:

```
offset = naive_time - sys_time
```

Value

A data frame of low level information.

Examples

```
library(vctrs)

x <- year_month_day(2021, 03, 14, c(01, 03), c(59, 00), c(59, 00))
x <- as_naive_time(x)
x <- as_zoned_time(x, "America/New_York")

# x[1] is in EST, x[2] is in EDT
x

x_sys <- as_sys_time(x)

info <- sys_time_info(x_sys, zoned_time_zone(x))
info
```

```
# Convert `begin` and `end` to zoned-times to see the previous and
# next daylight saving time transitions
data_frame(
  x = x,
  begin = as_zoned_time(info$begin, zoned_time_zone(x)),
  end = as_zoned_time(info$end, zoned_time_zone(x))
)

# `end` can be used to iterate through daylight saving time transitions
# by repeatedly calling `sys_time_info()`
sys_time_info(info$end, zoned_time_zone(x))

# Multiple `zone`s can be supplied to look up daylight saving time
# information in different time zones
zones <- c("America/New_York", "America/Los_Angeles")

info2 <- sys_time_info(x_sys[1], zones)
info2

# The offset can be used to display the naive-time (i.e. the printed time)
# in both of those time zones
data_frame(
  zone = zones,
  naive_time = x_sys[1] + info2$offset
)
```

`sys_time_now` *What is the current sys-time?*

Description

`sys_time_now()` returns the current time in UTC.

Usage

```
sys_time_now()
```

Details

The time is returned with a nanosecond precision, but the actual amount of data returned is OS dependent. Usually, information at at least the microsecond level is returned, with some platforms returning nanosecond information.

Value

A sys-time of the current time in UTC.

Examples

```
x <- sys_time_now()
```

`time-point-arithmetic` *Arithmetic: Time points*

Description

These are naive-time and sys-time methods for the arithmetic generics.

- `add_weeks()`
- `add_days()`
- `add_hours()`
- `add_minutes()`
- `add_seconds()`
- `add_milliseconds()`
- `add_microseconds()`
- `add_nanoseconds()`

When working with zoned times, generally you convert to either sys-time or naive-time, add the duration, then convert back to zoned time. Typically, *weeks and days* are added in *naive-time*, and *hours, minutes, seconds, and subseconds* are added in *sys-time*.

If you aren't using zoned times, arithmetic on sys-times and naive-time is equivalent.

If you need to add larger irregular units of time, such as months, quarters, or years, convert to a calendar type with a converter like `as_year_month_day()`.

Usage

```
## S3 method for class 'clock_time_point'
add_weeks(x, n, ...)

## S3 method for class 'clock_time_point'
add_days(x, n, ...)

## S3 method for class 'clock_time_point'
add_hours(x, n, ...)

## S3 method for class 'clock_time_point'
add_minutes(x, n, ...)

## S3 method for class 'clock_time_point'
add_seconds(x, n, ...)

## S3 method for class 'clock_time_point'
add_milliseconds(x, n, ...)

## S3 method for class 'clock_time_point'
```

```
add_microseconds(x, n, ...)

## S3 method for class 'clock_time_point'
add_nanoseconds(x, n, ...)
```

Arguments

x	[clock_sys_time / clock_naive_time] A time point vector.
n	[integer / clock_duration] An integer vector to be converted to a duration, or a duration corresponding to the arithmetic function being used. This corresponds to the number of duration units to add. n may be negative to subtract units of duration.
...	These dots are for future extensions and must be empty.

Details

x and n are recycled against each other.

Value

x after performing the arithmetic.

Examples

```
library(magrittr)

# Say you started with this zoned time, and you want to add 1 day to it
x <- as_naive_time(year_month_day(1970, 04, 25, 02, 30, 00))
x <- as_zoned_time(x, "America/New_York")
x

# Note that there was a daylight saving time gap on 1970-04-26 where
# we jumped from 01:59:59 -> 03:00:00.

# You can choose to add 1 day in "system time", by first converting to
# sys-time (the equivalent UTC time), adding the day, then converting back to
# zoned time. If you sat still for exactly 86,400 seconds, this is the
# time that you would see after daylight saving time adjusted the clock
# (note that the hour field is shifted forward by the size of the gap)
as_sys_time(x)

x %>%
  as_sys_time() %>%
  add_days(1) %>%
  as_zoned_time(zoned_time_zone(x))

# Alternatively, you can add 1 day in "naive time". Naive time represents
# a clock time with a yet-to-be-specified time zone. It tries to maintain
# smaller units where possible, so adding 1 day would attempt to return
# "1970-04-26T02:30:00" in the America/New_York time zone, but...
```

```
as_naive_time(x)

try({
x %>%
  as_naive_time() %>%
  add_days(1) %>%
  as_zoned_time(zoned_time_zone(x))
})

# ...this time doesn't exist in that time zone! It is "nonexistent".
# You can resolve nonexistent times by setting the `nonexistent` argument
# when converting to zoned time. Let's roll forward to the next available
# moment in time.
x %>%
  as_naive_time() %>%
  add_days(1) %>%
  as_zoned_time(zoned_time_zone(x), nonexistent = "roll-forward")
```

`time-point-rounding` *Time point rounding*

Description

- `time_point_floor()` rounds a sys-time or naive-time down to a multiple of the specified `precision`.
- `time_point_ceiling()` rounds a sys-time or naive-time up to a multiple of the specified `precision`.
- `time_point_round()` rounds up or down depending on what is closer, rounding up on ties.

Rounding time points is mainly useful for rounding sub-daily time points up to daily time points.

It can also be useful for flooring by a set number of days (like 20) with respect to some origin. By default, the origin is 1970-01-01 00:00:00.

If you want to group by components, such as "day of the month", rather than by "n days", see `calendar_group()`.

Usage

```
time_point_floor(x, precision, ..., n = 1L, origin = NULL)

time_point_ceiling(x, precision, ..., n = 1L, origin = NULL)

time_point_round(x, precision, ..., n = 1L, origin = NULL)
```

Arguments

x	[clock_sys_time / clock_naive_time] A sys-time or naive-time.

`precision` [character(1)]

A time point precision. One of:

- `"day"`
- `"hour"`
- `"minute"`
- `"second"`
- `"millisecond"`
- `"microsecond"`
- `"nanosecond"`

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

`n` [positive integer(1)]

A positive integer specifying the multiple of `precision` to use.

`origin` [clock_sys_time(1) / clock_naive_time(1) / NULL]

An origin to begin counting from. Mostly useful when `n > 1` and you want to control how the rounding groups are created.

If `x` is a sys-time, `origin` must be a sys-time.

If `x` is a naive-time, `origin` must be a naive-time.

The precision of `origin` must be equally precise as or less precise than `precision`.

If `NULL`, a default origin of midnight on 1970-01-01 is used.

Value

`x` rounded to the new `precision`.

Boundary Handling

To understand how flooring and ceiling work, you need to know how they create their intervals for rounding.

- `time_point_floor()` constructs intervals of `[lower,upper)` that bound each element of `x`, then always chooses the *left-hand side*.
- `time_point_ceiling()` constructs intervals of `(lower,upper]` that bound each element of `x`, then always chooses the *right-hand side*.

As an easy example, consider 2020-01-02 00:00:05.

To floor this to the nearest day, the following interval is constructed, and the left-hand side is returned at day precision:

`[2020-01-02 00:00:00,2020-01-03 00:00:00)`

To ceiling this to the nearest day, the following interval is constructed, and the right-hand side is returned at day precision:

`(2020-01-02 00:00:00,2020-01-03 00:00:00]`

Here is another example, this time with a time point on a boundary, 2020-01-02 00:00:00.

To floor this to the nearest day, the following interval is constructed, and the left-hand side is returned at day precision:

[2020-01-02 00:00:00,2020-01-03 00:00:00)

To ceiling this to the nearest day, the following interval is constructed, and the right-hand side is returned at day precision:

(2020-01-01 00:00:00,2020-01-02 00:00:00]

Notice that, regardless of whether you are doing a floor or ceiling, if the input falls on a boundary then it will be returned as is.

Examples

```
library(magrittr)

x <- as_naive_time(year_month_day(2019, 01, 01))
x <- add_days(x, 0:40)
head(x)

# Floor by sets of 20 days
# The implicit origin to start the 20 day counter is 1970-01-01
time_point_floor(x, "day", n = 20)

# You can easily customize the origin by creating a duration out of the
# origin date of interest...
origin <- year_month_day(2019, 01, 01) %>%
  as_naive_time() %>%
  as_duration()

# Which you can subtract from, floor, and then add to your input
time_point_floor(x - origin, "day", n = 20) + origin

# For times on the boundary, floor and ceiling both return the input
# at the new precision. Notice how the first element is on the boundary,
# and the second is 1 second after the boundary.
y <- as_naive_time(year_month_day(2020, 01, 02, 00, 00, c(00, 01)))
time_point_floor(y, "day")
time_point_ceiling(y, "day")
```

---

time_point_cast *Cast a time point between precisions*

Description

Casting is one way to change a time point's precision.

Casting to a less precise precision will completely drop information that is more precise than the precision that you are casting to. It does so in a way that makes it round towards zero. When converting time points to a less precise precision, you often want `time_point_floor()` instead of `time_point_cast()`, as that handles pre-1970 dates (which are stored as negative durations) in a more intuitive manner.

Casting to a more precise precision is done through a multiplication by a conversion factor between the current precision and the new precision.

Usage

```
time_point_cast(x, precision)
```

Arguments

x	[clock_sys_time / clock_naive_time] A sys-time or naive-time.
precision	[character(1)] A time point precision. One of: • "day" • "hour" • "minute" • "second" • "millisecond" • "microsecond" • "nanosecond"

Value

x cast to the new precision.

Examples

```
# Hour precision time points
# One is pre-1970, one is post-1970
x <- duration_hours(c(25, -25))
x <- as_naive_time(x)
x

# Casting rounds the underlying duration towards 0
cast <- time_point_cast(x, "day")
cast

# Flooring rounds the underlying duration towards negative infinity,
# which is often more intuitive for time points.
# Note that the cast ends up rounding the pre-1970 date up to the next
# day, while the post-1970 date is rounded down.
floor <- time_point_floor(x, "day")
floor

# Casting to a more precise precision, hour->millisecond
time_point_cast(x, "millisecond")
```

`time_point_count_between` *Counting: time point*

Description

`time_point_count_between()` counts the number of `precision` units between `start` and `end` (i.e., the number of days or hours). This count corresponds to the *whole number* of units, and will never return a fractional value.

This is suitable for, say, computing the whole number of days between two time points, accounting for the time of day.

Usage

```
time_point_count_between(start, end, precision, ..., n = 1L)
```

Arguments

`start, end`	[clock_time_point] A pair of time points. These will be recycled to their common size.
`precision`	[character(1)] One of: • `"week"` • `"day"` • `"hour"` • `"minute"` • `"second"` • `"millisecond"` • `"microsecond"` • `"nanosecond"`
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Details

Remember that `time_point_count_between()` returns an integer vector. With extremely fine precisions, such as nanoseconds, the count can quickly exceed the maximum value that is allowed in an integer. In this case, an `NA` will be returned with a warning.

Value

An integer representing the number of `precision` units between `start` and `end`.

Comparison Direction

The computed count has the property that if `start` <= end, then start + <count> <= end. Similarly, if `start` >= end, then start + <count> >= end. In other words, the comparison direction between `start` and end will never change after adding the count to `start`. This makes this function useful for repeated count computations at increasingly fine precisions.

Examples

```
x <- as_naive_time(year_month_day(2019, 2, 3))
y <- as_naive_time(year_month_day(2019, 2, 10))

# Whole number of days or hours between two time points
time_point_count_between(x, y, "day")
time_point_count_between(x, y, "hour")

# Whole number of 2-day units
time_point_count_between(x, y, "day", n = 2)

# Leap years are taken into account
x <- as_naive_time(year_month_day(c(2020, 2021), 2, 28))
y <- as_naive_time(year_month_day(c(2020, 2021), 3, 01))
time_point_count_between(x, y, "day")

# Time of day is taken into account.
# `2020-02-02T04 -> 2020-02-03T03` is not a whole day (because of the hour)
# `2020-02-02T04 -> 2020-02-03T05` is a whole day
x <- as_naive_time(year_month_day(2020, 2, 2, 4))
y <- as_naive_time(year_month_day(2020, 2, 3, c(3, 5)))
time_point_count_between(x, y, "day")
time_point_count_between(x, y, "hour")

# Can compute negative counts (using the same example from above)
time_point_count_between(y, x, "day")
time_point_count_between(y, x, "hour")

# Repeated computation at increasingly fine precisions
x <- as_naive_time(year_month_day(
  2020, 2, 2, 4, 5, 6, 200,
  subsecond_precision = "microsecond"
))
y <- as_naive_time(year_month_day(
  2020, 3, 1, 8, 9, 10, 100,
  subsecond_precision = "microsecond"
))

days <- time_point_count_between(x, y, "day")
x <- x + duration_days(days)

hours <- time_point_count_between(x, y, "hour")
x <- x + duration_hours(hours)

minutes <- time_point_count_between(x, y, "minute")
```

```
x <- x + duration_minutes(minutes)

seconds <- time_point_count_between(x, y, "second")
x <- x + duration_seconds(seconds)

microseconds <- time_point_count_between(x, y, "microsecond")
x <- x + duration_microseconds(microseconds)

data.frame(
  days = days,
  hours = hours,
  minutes = minutes,
  seconds = seconds,
  microseconds = microseconds
)
```

`time_point_precision` *Precision: time point*

Description

`time_point_precision()` extracts the precision from a time point, such as a sys-time or naive-time. It returns the precision as a single string.

Usage

```
time_point_precision(x)
```

Arguments

`x`	[clock_time_point] A time point.

Value

A single string holding the precision of the time point.

Examples

```
time_point_precision(sys_time_now())
time_point_precision(as_naive_time(duration_days(1)))
```

`time_point_shift` *Shifting: time point*

Description

`time_point_shift()` shifts `x` to the `target` weekday. You can shift to the next or previous weekday. If `x` is currently on the `target` weekday, you can choose to leave it alone or advance it to the next instance of the `target`.

Weekday shifting is one of the easiest ways to floor by week while controlling what is considered the first day of the week. You can also accomplish this with the `origin` argument of `time_point_floor()`, but this is slightly easier.

Usage

```
time_point_shift(x, target, ..., which = "next", boundary = "keep")
```

Arguments

`x`	[clock_time_point] A time point.
`target`	[weekday] A weekday created from `weekday()` to target. Generally this is length 1, but can also be the same length as `x`.
`...`	These dots are for future extensions and must be empty.
`which`	[character(1)] One of: • `"next"`: Shift to the next instance of the `target` weekday. • "previous: Shift to the previous instance of the `target` weekday.
`boundary`	[character(1)] One of: • `"keep"`: If `x` is currently on the `target` weekday, return it. • `"advance"`: If `x` is currently on the `target` weekday, advance it anyways.

Value

`x` shifted to the `target` weekday.

Examples

```
x <- as_naive_time(year_month_day(2019, 1, 1:2))

# A Tuesday and Wednesday
as_weekday(x)

monday <- weekday(clock_weekdays$monday)
```

```
# Shift to the next Monday
time_point_shift(x, monday)

# Shift to the previous Monday
# This is an easy way to "floor by week" with a target weekday in mind
time_point_shift(x, monday, which = "previous")

# What about Tuesday?
tuesday <- weekday(clock_weekdays$tuesday)

# Notice that the day that was currently on a Tuesday was not shifted
time_point_shift(x, tuesday)

# You can force it to `"advance"`
time_point_shift(x, tuesday, boundary = "advance")
```

`vec_arith.clock_year_day` *Support for vctrs arithmetic*

Description

Support for vctrs arithmetic

Usage

```
## S3 method for class 'clock_year_day'
vec_arith(op, x, y, ...)

## S3 method for class 'clock_year_month_day'
vec_arith(op, x, y, ...)

## S3 method for class 'clock_year_month_weekday'
vec_arith(op, x, y, ...)

## S3 method for class 'clock_iso_year_week_day'
vec_arith(op, x, y, ...)

## S3 method for class 'clock_naive_time'
vec_arith(op, x, y, ...)

## S3 method for class 'clock_year_quarter_day'
vec_arith(op, x, y, ...)

## S3 method for class 'clock_sys_time'
vec_arith(op, x, y, ...)
```

```
## S3 method for class 'clock_weekday'
vec_arith(op, x, y, ...)
```

Arguments

op	An arithmetic operator as a string
x	A pair of vectors. For !, unary + and unary -, y will be a sentinel object of class MISSING, as created by MISSING().
y	A pair of vectors. For !, unary + and unary -, y will be a sentinel object of class MISSING, as created by MISSING().
...	These dots are for future extensions and must be empty.

Value

The result of the arithmetic operation.

Examples

```
vctrs::vec_arith("+", year_month_day(2019), 1)
```

---

weekday *Construct a weekday vector*

Description

A weekday is a simple type that represents a day of the week.

The most interesting thing about the weekday type is that it implements *circular arithmetic*, which makes determining the "next Monday" or "previous Tuesday" from a sys-time or naive-time easy to compute. See the examples.

Usage

```
weekday(code = integer(), ..., encoding = "western")
```

Arguments

code	[integer] Integer codes between [1, 7] representing days of the week. The interpretation of these values depends on encoding.
...	These dots are for future extensions and must be empty.
encoding	[character(1)] One of: • "western": Encode weekdays assuming 1 == Sunday and 7 == Saturday. • "iso": Encode weekdays assuming 1 == Monday and 7 == Sunday. This is in line with the ISO standard.

Value

A weekday vector.

Examples

```
x <- as_naive_time(year_month_day(2019, 01, 05))

# This is a Saturday!
as_weekday(x)

# Adjust to the next Wednesday
wednesday <- weekday(clock_weekdays$wednesday)

# This returns the number of days until the next Wednesday using
# circular arithmetic
# "Wednesday - Saturday = 4 days until next Wednesday"
wednesday - as_weekday(x)

# Advance to the next Wednesday
x_next_wednesday <- x + (wednesday - as_weekday(x))

as_weekday(x_next_wednesday)

# What about the previous Tuesday?
tuesday <- weekday(clock_weekdays$tuesday)
x - (as_weekday(x) - tuesday)

# What about the next Saturday?
# With an additional condition that if today is a Saturday,
# then advance to the next one.
saturday <- weekday(clock_weekdays$saturday)
x + 1L + (saturday - as_weekday(x + 1L))

# You can supply an ISO coding for `code` as well, where 1 == Monday.
weekday(1:7, encoding = "western")
weekday(1:7, encoding = "iso")
```

---

weekday-arithmetic *Arithmetic: weekday*

Description

These are weekday methods for the arithmetic generics.

- `add_days()`

Also check out the examples on the `weekday()` page for more advanced usage.

Usage

```
## S3 method for class 'clock_weekday'
add_days(x, n, ...)
```

Arguments

x	[clock_weekday] A weekday vector.
n	[integer / clock_duration] An integer vector to be converted to a duration, or a duration corresponding to the arithmetic function being used. This corresponds to the number of duration units to add. n may be negative to subtract units of duration.
...	These dots are for future extensions and must be empty.

Details

x and n are recycled against each other.

Value

x after performing the arithmetic.

Examples

```
saturday <- weekday(clock_weekdays$saturday)
saturday

add_days(saturday, 1)
add_days(saturday, 2)
```

---

weekday_code *Extract underlying weekday codes*

Description

weekday_code() extracts out the integer code for the weekday.

Usage

```
weekday_code(x, ..., encoding = "western")
```

Arguments

x	[weekday] A weekday vector.
...	These dots are for future extensions and must be empty.
encoding	[character(1)] One of: • "western": Encode weekdays assuming 1 == Sunday and 7 == Saturday. • "iso": Encode weekdays assuming 1 == Monday and 7 == Sunday. This is in line with the ISO standard.

Value

An integer vector of codes.

Examples

```
# Here we supply a western encoding to `weekday()`
x <- weekday(1:7)
x

# We can extract out the codes using different encodings
weekday_code(x, encoding = "western")
weekday_code(x, encoding = "iso")
```

---

weekday_factor *Convert a weekday to an ordered factor*

Description

weekday_factor() converts a weekday object to an ordered factor. This can be useful in combination with ggplot2, or for modeling.

Usage

```
weekday_factor(x, ..., labels = "en", abbreviate = TRUE, encoding = "western")
```

Arguments

x	[weekday] A weekday vector.
...	These dots are for future extensions and must be empty.
labels	[clock_labels / character(1)] Character representations of localized weekday names, month names, and AM/PM names. Either the language code as string (passed on to clock_labels_lookup()), or an object created by clock_labels().

abbreviate [logical(1)]
If TRUE, the abbreviated weekday names from labels will be used.
If FALSE, the full weekday names from labels will be used.

encoding [character(1)]
One of:

- "western": Encode the weekdays as an ordered factor with levels from Sunday -> Saturday.
- "iso": Encode the weekdays as an ordered factor with levels from Monday -> Sunday.

Value

An ordered factor representing the weekdays.

Examples

```
x <- weekday(1:7)

# Default to Sunday -> Saturday
weekday_factor(x)

# ISO encoding is Monday -> Sunday
weekday_factor(x, encoding = "iso")

# With full names
weekday_factor(x, abbreviate = FALSE)

# Or a different language
weekday_factor(x, labels = "fr")
```

---

year-day-arithmetic *Arithmetic: year-day*

Description

These are year-day methods for the arithmetic generics.

- add_years()

Notably, *you cannot add days to a year-day*. For day-based arithmetic, first convert to a time point with as_naive_time() or as_sys_time().

Usage

```
## S3 method for class 'clock_year_day'
add_years(x, n, ...)
```

Arguments

x	[clock_year_day] A year-day vector.
n	[integer / clock_duration] An integer vector to be converted to a duration, or a duration corresponding to the arithmetic function being used. This corresponds to the number of duration units to add. n may be negative to subtract units of duration.
...	These dots are for future extensions and must be empty.

Details

x and n are recycled against each other.

Value

x after performing the arithmetic.

Examples

```
x <- year_day(2019, 10)

add_years(x, 1:5)

# A valid day in a leap year
y <- year_day(2020, 366)
y

# Adding 1 year to `y` generates an invalid date
y_plus <- add_years(y, 1)
y_plus

# Invalid dates are fine, as long as they are eventually resolved
# by either manually resolving, or by calling `invalid_resolve()`

# Resolve by returning the previous / next valid moment in time
invalid_resolve(y_plus, invalid = "previous")
invalid_resolve(y_plus, invalid = "next")

# Manually resolve by setting to the last day of the year
invalid <- invalid_detect(y_plus)
y_plus[invalid] <- set_day(y_plus[invalid], "last")
y_plus
```

---

year-day-boundary *Boundaries: year-day*

Description

This is a year-day method for the `calendar_start()` and `calendar_end()` generics. They adjust components of a calendar to the start or end of a specified `precision`.

Usage

```
## S3 method for class 'clock_year_day'
calendar_start(x, precision)

## S3 method for class 'clock_year_day'
calendar_end(x, precision)
```

Arguments

`x`	[clock_year_day] A year-day vector.
`precision`	[character(1)] One of: • `"year"` • `"day"` • `"hour"` • `"minute"` • `"second"` • `"millisecond"` • `"microsecond"` • `"nanosecond"`

Value

`x` at the same precision, but with some components altered to be at the boundary value.

Examples

```
# Day precision
x <- year_day(2019:2020, 5)
x

# Compute the last day of the year
calendar_end(x, "year")
```

year-day-count-between *Counting: year-day*

Description

This is a year-day method for the `calendar_count_between()` generic. It counts the number of `precision` units between `start` and `end` (i.e., the number of years).

Usage

```
## S3 method for class 'clock_year_day'
calendar_count_between(start, end, precision, ..., n = 1L)
```

Arguments

`start`, `end`	[clock_year_day] A pair of year-day vectors. These will be recycled to their common size.
`precision`	[character(1)] One of: • `"year"`
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Value

An integer representing the number of `precision` units between `start` and `end`.

Examples

```
# Compute an individual's age in years
x <- year_day(2001, 100)
y <- year_day(2021, c(99, 101))

calendar_count_between(x, y, "year")

# Or in a whole number multiple of years
calendar_count_between(x, y, "year", n = 3)
```

`year-day-getters` *Getters: year-day*

Description

These are year-day methods for the getter generics.

- `get_year()` returns the Gregorian year.
- `get_day()` returns the day of the year.
- There are sub-daily getters for extracting more precise components.

Usage

```
## S3 method for class 'clock_year_day'
get_year(x)

## S3 method for class 'clock_year_day'
get_day(x)

## S3 method for class 'clock_year_day'
get_hour(x)

## S3 method for class 'clock_year_day'
get_minute(x)

## S3 method for class 'clock_year_day'
get_second(x)

## S3 method for class 'clock_year_day'
get_millisecond(x)

## S3 method for class 'clock_year_day'
get_microsecond(x)

## S3 method for class 'clock_year_day'
get_nanosecond(x)
```

Arguments

`x`	[clock_year_day] A year-day to get the component from.

Value

The component.

Examples

```
x <- year_day(2019, 101:105, 1, 20, 30)

get_day(x)
get_second(x)

# Cannot extract more precise components
y <- year_day(2019, 1)
try(get_hour(y))

# Cannot extract components that don't exist for this calendar
try(get_quarter(x))
```

---

year-day-group *Grouping: year-day*

Description

This is a year-day method for the `calendar_group()` generic.

Grouping for a year-day object can be done at any precision, as long as `x` is at least as precise as `precision`.

Usage

```
## S3 method for class 'clock_year_day'
calendar_group(x, precision, ..., n = 1L)
```

Arguments

`x`	[clock_year_day] A year-day vector.
`precision`	[character(1)] One of: • `"year"` • `"day"` • `"hour"` • `"minute"` • `"second"` • `"millisecond"` • `"microsecond"` • `"nanosecond"`
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Value

x grouped at the specified precision.

Examples

```
x <- seq(as_naive_time(year_month_day(2019, 1, 1)), by = 5, length.out = 20)
x <- as_year_day(x)
x

# Group by day of the current year
calendar_group(x, "day", n = 20)
```

`year-day-narrow` *Narrow: year-day*

Description

This is a year-day method for the `calendar_narrow()` generic. It narrows a year-day vector to the specified precision.

Usage

```
## S3 method for class 'clock_year_day'
calendar_narrow(x, precision)
```

Arguments

x	[clock_year_day] A year-day vector.
precision	[character(1)] One of: • "year" • "day" • "hour" • "minute" • "second" • "millisecond" • "microsecond" • "nanosecond"

Value

x narrowed to the supplied precision.

Examples

```
# Hour precision
x <- year_day(2019, 3, 4)
x

# Narrowed to day precision
calendar_narrow(x, "day")

# Or year precision
calendar_narrow(x, "year")

# Subsecond precision can be narrowed to second precision
milli <- calendar_widen(x, "millisecond")
micro <- calendar_widen(x, "microsecond")
milli
micro

calendar_narrow(milli, "second")
calendar_narrow(micro, "second")

# But once you have "locked in" a subsecond precision, it can't be
# narrowed to another subsecond precision
try(calendar_narrow(micro, "millisecond"))
```

---

year-day-setters *Setters: year-day*

Description

These are year-day methods for the setter generics.

- `set_year()` sets the Gregorian year.
- `set_day()` sets the day of the year. Valid values are in the range of [1, 366].
- There are sub-daily setters for setting more precise components.

Usage

```
## S3 method for class 'clock_year_day'
set_year(x, value, ...)

## S3 method for class 'clock_year_day'
set_day(x, value, ...)

## S3 method for class 'clock_year_day'
set_hour(x, value, ...)

## S3 method for class 'clock_year_day'
set_minute(x, value, ...)
```

```
## S3 method for class 'clock_year_day'
set_second(x, value, ...)

## S3 method for class 'clock_year_day'
set_millisecond(x, value, ...)

## S3 method for class 'clock_year_day'
set_microsecond(x, value, ...)

## S3 method for class 'clock_year_day'
set_nanosecond(x, value, ...)
```

Arguments

x	[clock_year_day] A year-day vector.
value	[integer / "last"] The value to set the component to. For `set_day()`, this can also be `"last"` to set the day to the last day of the year.
...	These dots are for future extensions and must be empty.

Value

x with the component set.

Examples

```
x <- year_day(2019)

# Set the day
set_day(x, 12:14)

# Set to the "last" day of the year
set_day(x, "last")

# Set to an invalid day of the year
invalid <- set_day(x, 366)
invalid

# Then resolve the invalid day by choosing the next valid day
invalid_resolve(invalid, invalid = "next")

# Cannot set a component two levels more precise than where you currently are
try(set_hour(x, 5))
```

`year-day-widen` *Widen: year-day*

Description

This is a year-day method for the `calendar_widen()` generic. It widens a year-day vector to the specified `precision`.

Usage

```
## S3 method for class 'clock_year_day'
calendar_widen(x, precision)
```

Arguments

`x`	[clock_year_day] A year-day vector.
`precision`	[character(1)] One of: • `"year"` • `"day"` • `"hour"` • `"minute"` • `"second"` • `"millisecond"` • `"microsecond"` • `"nanosecond"`

Value

`x` widened to the supplied `precision`.

Examples

```
# Year precision
x <- year_day(2019)
x

# Widen to day precision
calendar_widen(x, "day")

# Or second precision
sec <- calendar_widen(x, "second")
sec

# Second precision can be widened to subsecond precision
milli <- calendar_widen(sec, "millisecond")
```

```
micro <- calendar_widen(sec, "microsecond")
milli
micro

# But once you have "locked in" a subsecond precision, it can't
# be widened again
try(calendar_widen(milli, "microsecond"))
```

`year-month-day-arithmetic`
Arithmetic: year-month-day

Description

These are year-month-day methods for the arithmetic generics.

- `add_years()`
- `add_quarters()`
- `add_months()`

Notably, *you cannot add days to a year-month-day*. For day-based arithmetic, first convert to a time point with `as_naive_time()` or `as_sys_time()`.

Usage

```
## S3 method for class 'clock_year_month_day'
add_years(x, n, ...)

## S3 method for class 'clock_year_month_day'
add_quarters(x, n, ...)

## S3 method for class 'clock_year_month_day'
add_months(x, n, ...)
```

Arguments

`x`	[clock_year_month_day] A year-month-day vector.
`n`	[integer / clock_duration] An integer vector to be converted to a duration, or a duration corresponding to the arithmetic function being used. This corresponds to the number of duration units to add. `n` may be negative to subtract units of duration.
`...`	These dots are for future extensions and must be empty.

Details

Adding a single quarter with `add_quarters()` is equivalent to adding 3 months.

`x` and `n` are recycled against each other.

Value

x after performing the arithmetic.

Examples

```
x <- year_month_day(2019, 1, 1)

add_years(x, 1:5)

y <- year_month_day(2019, 1, 31)

# Adding 1 month to `y` generates an invalid date
y_plus <- add_months(y, 1:2)
y_plus

# Invalid dates are fine, as long as they are eventually resolved
# by either manually resolving, or by calling `invalid_resolve()`

# Resolve by returning the previous / next valid moment in time
invalid_resolve(y_plus, invalid = "previous")
invalid_resolve(y_plus, invalid = "next")

# Manually resolve by setting to the last day of the month
invalid <- invalid_detect(y_plus)
y_plus[invalid] <- set_day(y_plus[invalid], "last")
y_plus
```

---

year-month-day-boundary

Boundaries: year-month-day

Description

This is a year-month-day method for the `calendar_start()` and `calendar_end()` generics. They adjust components of a calendar to the start or end of a specified `precision`.

Usage

```
## S3 method for class 'clock_year_month_day'
calendar_start(x, precision)

## S3 method for class 'clock_year_month_day'
calendar_end(x, precision)
```

Arguments

x	[clock_year_month_day] A year-month-day vector.

precision [character(1)]

One of:

- "year"
- "month"
- "day"
- "hour"
- "minute"
- "second"
- "millisecond"
- "microsecond"
- "nanosecond"

Value

x at the same precision, but with some components altered to be at the boundary value.

Examples

```
# Hour precision
x <- year_month_day(2019, 2:4, 5, 6)
x

# Compute the start of the month
calendar_start(x, "month")

# Or the end of the month, notice that the hour value is adjusted as well
calendar_end(x, "month")

# Compare that with just setting the day of the month to `"last"`, which
# doesn't adjust any other components
set_day(x, "last")

# You can't compute the start / end at a more precise precision than
# the input is at
try(calendar_start(x, "second"))
```

---

year-month-day-count-between

Counting: year-month-day

Description

This is a year-month-day method for the calendar_count_between() generic. It counts the number of precision units between start and end (i.e., the number of years or months).

Usage

```
## S3 method for class 'clock_year_month_day'
calendar_count_between(start, end, precision, ..., n = 1L)
```

Arguments

`start`, `end`	[clock_year_month_day] A pair of year-month-day vectors. These will be recycled to their common size.
`precision`	[character(1)] One of: • `"year"` • `"quarter"` • `"month"`
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Details

`"quarter"` is equivalent to `"month"` precision with `n` set to `n * 3L`.

Value

An integer representing the number of `precision` units between `start` and `end`.

Examples

```
# Compute an individual's age in years
x <- year_month_day(2001, 2, 4)
today <- year_month_day(2021, 11, 30)
calendar_count_between(x, today, "year")

# Compute the number of months between two dates, taking
# into account the day of the month and time of day
x <- year_month_day(2000, 4, 2, 5)
y <- year_month_day(2000, 7, c(1, 2, 2), c(3, 4, 6))
calendar_count_between(x, y, "month")
```

`year-month-day-getters` *Getters: year-month-day*

Description

These are year-month-day methods for the getter generics.

- `get_year()` returns the Gregorian year.
- `get_month()` returns the month of the year.
- `get_day()` returns the day of the month.
- There are sub-daily getters for extracting more precise components.

Usage

```
## S3 method for class 'clock_year_month_day'
get_year(x)

## S3 method for class 'clock_year_month_day'
get_month(x)

## S3 method for class 'clock_year_month_day'
get_day(x)

## S3 method for class 'clock_year_month_day'
get_hour(x)

## S3 method for class 'clock_year_month_day'
get_minute(x)

## S3 method for class 'clock_year_month_day'
get_second(x)

## S3 method for class 'clock_year_month_day'
get_millisecond(x)

## S3 method for class 'clock_year_month_day'
get_microsecond(x)

## S3 method for class 'clock_year_month_day'
get_nanosecond(x)
```

Arguments

x	[clock_year_month_day] A year-month-day to get the component from.

Value

The component.

Examples

```
x <- year_month_day(2019, 1:3, 5:7, 1, 20, 30)

get_month(x)
get_day(x)
get_second(x)

# Cannot extract more precise components
y <- year_month_day(2019, 1)
try(get_day(y))

# Cannot extract components that don't exist for this calendar
try(get_quarter(x))
```

---

year-month-day-group *Grouping: year-month-day*

Description

This is a year-month-day method for the `calendar_group()` generic.

Grouping for a year-month-day object can be done at any precision, as long as `x` is at least as precise as `precision`.

Usage

```
## S3 method for class 'clock_year_month_day'
calendar_group(x, precision, ..., n = 1L)
```

Arguments

`x`	[clock_year_month_day] A year-month-day vector.
`precision`	[character(1)] One of: • `"year"` • `"month"` • `"day"` • `"hour"` • `"minute"` • `"second"` • `"millisecond"` • `"microsecond"` • `"nanosecond"`
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Value

x grouped at the specified precision.

Examples

```
steps <- duration_days(seq(0, 100, by = 5))
x <- year_month_day(2019, 1, 1)
x <- as_naive_time(x) + steps
x <- as_year_month_day(x)
x

# Group by a single month
calendar_group(x, "month")

# Or multiple months
calendar_group(x, "month", n = 2)

# Group 3 days of the month together
y <- year_month_day(2019, 1, 1:12)
calendar_group(y, "day", n = 3)

# Group by 5 nanosecond of the current second
z <- year_month_day(
  2019, 1, 2, 1, 5, 20, 1:20,
  subsecond_precision = "nanosecond"
)
calendar_group(z, "nanosecond", n = 5)
```

---

year-month-day-narrow *Narrow: year-month-day*

Description

This is a year-month-day method for the calendar_narrow() generic. It narrows a year-month-day vector to the specified precision.

Usage

```
## S3 method for class 'clock_year_month_day'
calendar_narrow(x, precision)
```

Arguments

x	[clock_year_month_day] A year-month-day vector.
precision	[character(1)] One of: • "year"

- "month"
- "day"
- "hour"
- "minute"
- "second"
- "millisecond"
- "microsecond"
- "nanosecond"

Value

x narrowed to the supplied precision.

Examples

```
# Hour precision
x <- year_month_day(2019, 1, 3, 4)
x

# Narrowed to day precision
calendar_narrow(x, "day")

# Or month precision
calendar_narrow(x, "month")

# Subsecond precision can be narrowed to second precision
milli <- calendar_widen(x, "millisecond")
micro <- calendar_widen(x, "microsecond")
milli
micro

calendar_narrow(milli, "second")
calendar_narrow(micro, "second")

# But once you have "locked in" a subsecond precision, it can't be
# narrowed to another subsecond precision
try(calendar_narrow(micro, "millisecond"))
```

---

year-month-day-setters

Setters: year-month-day

Description

These are year-month-day methods for the setter generics.

- set_year() sets the Gregorian year.
- set_month() sets the month of the year. Valid values are in the range of [1, 12].
- set_day() sets the day of the month. Valid values are in the range of [1, 31].
- There are sub-daily setters for setting more precise components.

Usage

```
## S3 method for class 'clock_year_month_day'
set_year(x, value, ...)

## S3 method for class 'clock_year_month_day'
set_month(x, value, ...)

## S3 method for class 'clock_year_month_day'
set_day(x, value, ...)

## S3 method for class 'clock_year_month_day'
set_hour(x, value, ...)

## S3 method for class 'clock_year_month_day'
set_minute(x, value, ...)

## S3 method for class 'clock_year_month_day'
set_second(x, value, ...)

## S3 method for class 'clock_year_month_day'
set_millisecond(x, value, ...)

## S3 method for class 'clock_year_month_day'
set_microsecond(x, value, ...)

## S3 method for class 'clock_year_month_day'
set_nanosecond(x, value, ...)
```

Arguments

x	[clock_year_month_day] A year-month-day vector.
value	[integer / "last"] The value to set the component to. For `set_day()`, this can also be `"last"` to set the day to the last day of the month.
...	These dots are for future extensions and must be empty.

Value

`x` with the component set.

Examples

```
x <- year_month_day(2019, 1:3)

# Set the day
set_day(x, 12:14)
```

```
# Set to the "last" day of the month
set_day(x, "last")

# Set to an invalid day of the month
invalid <- set_day(x, 31)
invalid

# Then resolve the invalid day by choosing the next valid day
invalid_resolve(invalid, invalid = "next")

# Cannot set a component two levels more precise than where you currently are
try(set_hour(x, 5))
```

`year-month-day-widen` *Widen: year-month-day*

Description

This is a year-month-day method for the `calendar_widen()` generic. It widens a year-month-day vector to the specified `precision`.

Usage

```
## S3 method for class 'clock_year_month_day'
calendar_widen(x, precision)
```

Arguments

`x`	[clock_year_month_day] A year-month-day vector.
`precision`	[character(1)] One of: • `"year"` • `"month"` • `"day"` • `"hour"` • `"minute"` • `"second"` • `"millisecond"` • `"microsecond"` • `"nanosecond"`

Value

`x` widened to the supplied `precision`.

Examples

```
# Month precision
x <- year_month_day(2019, 1)
x

# Widen to day precision
calendar_widen(x, "day")

# Or second precision
sec <- calendar_widen(x, "second")
sec

# Second precision can be widened to subsecond precision
milli <- calendar_widen(sec, "millisecond")
micro <- calendar_widen(sec, "microsecond")
milli
micro

# But once you have "locked in" a subsecond precision, it can't
# be widened again
try(calendar_widen(milli, "microsecond"))
```

---

year-month-weekday-arithmetic

Arithmetic: year-month-weekday

Description

These are year-month-weekday methods for the arithmetic generics.

- `add_years()`
- `add_quarters()`
- `add_months()`

Notably, *you cannot add days to a year-month-weekday*. For day-based arithmetic, first convert to a time point with `as_naive_time()` or `as_sys_time()`.

Usage

```
## S3 method for class 'clock_year_month_weekday'
add_years(x, n, ...)

## S3 method for class 'clock_year_month_weekday'
add_quarters(x, n, ...)

## S3 method for class 'clock_year_month_weekday'
add_months(x, n, ...)
```

Arguments

x	[clock_year_month_weekday] A year-month-weekday vector.
n	[integer / clock_duration] An integer vector to be converted to a duration, or a duration corresponding to the arithmetic function being used. This corresponds to the number of duration units to add. n may be negative to subtract units of duration.
...	These dots are for future extensions and must be empty.

Details

Adding a single quarter with `add_quarters()` is equivalent to adding 3 months.

x and n are recycled against each other.

Value

x after performing the arithmetic.

Examples

```
# 2nd Friday in January, 2019
x <- year_month_weekday(2019, 1, clock_weekdays$friday, 2)
x

add_months(x, 1:5)

# These don't necessarily correspond to the same day of the month
as_year_month_day(add_months(x, 1:5))
```

---

year-month-weekday-boundary

Boundaries: year-month-weekday

Description

This is a year-month-weekday method for the `calendar_start()` and `calendar_end()` generics. They adjust components of a calendar to the start or end of a specified `precision`.

This method is restricted to only `"year"` and `"month"` `precisions`, and `x` can't be more precise than month precision. Computing the "start" of a day precision year-month-weekday object isn't defined because a year-month-weekday with day = 1, index = 1 doesn't necessarily occur earlier (chronologically) than day = 2, index = 1. Because of these restrictions, this method isn't particularly useful, but is included for completeness.

Usage

```
## S3 method for class 'clock_year_month_weekday'
calendar_start(x, precision)

## S3 method for class 'clock_year_month_weekday'
calendar_end(x, precision)
```

Arguments

x	[clock_year_month_weekday] A year-month-weekday vector.
precision	[character(1)] One of: • "year" • "month"

Value

x at the same precision, but with some components altered to be at the boundary value.

Examples

```
# Month precision
x <- year_month_weekday(2019, 1:5)
x

# Compute the last month of the year
calendar_end(x, "year")
```

---

year-month-weekday-count-between

Counting: year-month-weekday

Description

This is a year-month-weekday method for the calendar_count_between() generic. It counts the number of precision units between start and end (i.e., the number of years or months).

Usage

```
## S3 method for class 'clock_year_month_weekday'
calendar_count_between(start, end, precision, ..., n = 1L)
```

Arguments

start, end	[clock_year_month_weekday] A pair of year-month-weekday vectors. These will be recycled to their common size.
precision	[character(1)] One of: • "year" • "quarter" • "month"
...	These dots are for future extensions and must be empty.
n	[positive integer(1)] A single positive integer specifying a multiple of precision to use.

Details

Remember that year-month-weekday is not comparable when it is "day" precision or finer, so this method is only defined for "year" and "month" precision year-month-weekday objects.

"quarter" is equivalent to "month" precision with n set to n * 3L.

Value

An integer representing the number of precision units between start and end.

Examples

```
# Compute the number of months between two dates
x <- year_month_weekday(2001, 2)
y <- year_month_weekday(2021, c(1, 3))

calendar_count_between(x, y, "month")

# Remember that day precision or finer year-month-weekday objects
# are not comparable, so this won't work
x <- year_month_weekday(2001, 2, 1, 1)
try(calendar_count_between(x, x, "month"))
```

---

year-month-weekday-getters

Getters: year-month-weekday

Description

These are year-month-weekday methods for the getter generics.

- `get_year()` returns the Gregorian year.
- `get_month()` returns the month of the year.
- `get_day()` returns the day of the week encoded from 1-7, where 1 = Sunday and 7 = Saturday.
- `get_index()` returns a value from 1-5 indicating that the corresponding weekday is the n-th instance of that weekday in the current month.
- There are sub-daily getters for extracting more precise components.

Usage

```
## S3 method for class 'clock_year_month_weekday'
get_year(x)

## S3 method for class 'clock_year_month_weekday'
get_month(x)

## S3 method for class 'clock_year_month_weekday'
get_day(x)

## S3 method for class 'clock_year_month_weekday'
get_index(x)

## S3 method for class 'clock_year_month_weekday'
get_hour(x)

## S3 method for class 'clock_year_month_weekday'
get_minute(x)

## S3 method for class 'clock_year_month_weekday'
get_second(x)

## S3 method for class 'clock_year_month_weekday'
get_millisecond(x)

## S3 method for class 'clock_year_month_weekday'
get_microsecond(x)

## S3 method for class 'clock_year_month_weekday'
get_nanosecond(x)
```

Arguments

x	[clock_year_month_weekday] A year-month-weekday to get the component from.

Value

The component.

Examples

```
monday <- clock_weekdays$monday
thursday <- clock_weekdays$thursday

x <- year_month_weekday(2019, 1, monday:thursday, 1:4)
x

# Gets the weekday, 1 = Sunday, 7 = Saturday
get_day(x)

# Gets the index indicating which instance of that particular weekday
# it is in the current month (i.e. the "1st Monday of January, 2019")
get_index(x)
```

---

year-month-weekday-group

Grouping: year-month-weekday

Description

This is a year-month-weekday method for the `calendar_group()` generic.

Grouping for a year-month-weekday object can be done at any precision except for `"day"`, as long as `x` is at least as precise as `precision`.

Usage

```
## S3 method for class 'clock_year_month_weekday'
calendar_group(x, precision, ..., n = 1L)
```

Arguments

`x`	[clock_year_month_weekday] A year-month-weekday vector.
`precision`	[character(1)] One of: • `"year"` • `"month"` • `"hour"` • `"minute"` • `"second"` • `"millisecond"` • `"microsecond"`

- "nanosecond"

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

`n` [positive integer(1)]
A single positive integer specifying a multiple of `precision` to use.

Details

Grouping by "day" is undefined for a year-month-weekday because there are two day fields, the weekday and the index, and there is no clear way to define how to group by that.

Value

`x` grouped at the specified `precision`.

Examples

```
x <- year_month_weekday(2019, 1:12, clock_weekdays$sunday, 1, 00, 05, 05)
x

# Group by 3 months - drops more precise components!
calendar_group(x, "month", n = 3)
```

`year-month-weekday-narrow`
Narrow: year-month-weekday

Description

This is a year-month-weekday method for the `calendar_narrow()` generic. It narrows a year-month-weekday vector to the specified `precision`.

Usage

```
## S3 method for class 'clock_year_month_weekday'
calendar_narrow(x, precision)
```

Arguments

`x` [clock_year_month_weekday]
A year-month-weekday vector.

`precision` [character(1)]
One of:

- "year"
- "month"
- "day"
- "hour"

- "minute"
- "second"
- "millisecond"
- "microsecond"
- "nanosecond"

Value

x narrowed to the supplied precision.

Examples

```
# Day precision
x <- year_month_weekday(2019, 1, 1, 2)
x

# Narrowed to month precision
calendar_narrow(x, "month")
```

---

year-month-weekday-setters

Setters: year-month-weekday

Description

These are year-month-weekday methods for the setter generics.

- set_year() sets the Gregorian year.
- set_month() sets the month of the year. Valid values are in the range of [1, 12].
- set_day() sets the day of the week. Valid values are in the range of [1, 7], with 1 = Sunday, and 7 = Saturday.
- set_index() sets the index indicating that the corresponding weekday is the n-th instance of that weekday in the current month. Valid values are in the range of [1, 5].
- There are sub-daily setters for setting more precise components.

Usage

```
## S3 method for class 'clock_year_month_weekday'
set_year(x, value, ...)

## S3 method for class 'clock_year_month_weekday'
set_month(x, value, ...)

## S3 method for class 'clock_year_month_weekday'
set_day(x, value, ..., index = NULL)
```

```
## S3 method for class 'clock_year_month_weekday'
set_index(x, value, ...)

## S3 method for class 'clock_year_month_weekday'
set_hour(x, value, ...)

## S3 method for class 'clock_year_month_weekday'
set_minute(x, value, ...)

## S3 method for class 'clock_year_month_weekday'
set_second(x, value, ...)

## S3 method for class 'clock_year_month_weekday'
set_millisecond(x, value, ...)

## S3 method for class 'clock_year_month_weekday'
set_microsecond(x, value, ...)

## S3 method for class 'clock_year_month_weekday'
set_nanosecond(x, value, ...)
```

Arguments

x	[clock_year_month_weekday] A year-month-weekday vector.
value	[integer / "last"] The value to set the component to. For `set_index()`, this can also be `"last"` to adjust to the last instance of the corresponding weekday in that month.
...	These dots are for future extensions and must be empty.
index	[NULL / integer / "last"] This argument is only used with `set_day()`, and allows you to set the index while also setting the weekday. If `x` is a month precision year-month-weekday, `index` is required to be set, as you must specify the weekday and the index simultaneously to promote from month to day precision.

Value

`x` with the component set.

Examples

```
x <- year_month_weekday(2019, 1:3)

set_year(x, 2020:2022)

# Setting the weekday on a month precision year-month-weekday requires
```

```
# also setting the `index` to fully specify the day information
x <- set_day(x, clock_weekdays$sunday, index = 1)
x

# Once you have at least day precision, you can set the weekday and
# the index separately
set_day(x, clock_weekdays$monday)
set_index(x, 3)

# Set to the "last" instance of the corresponding weekday in this month
# (Note that some months have 4 Sundays, and others have 5)
set_index(x, "last")

# Set to an invalid index
# January and February of 2019 don't have 5 Sundays!
invalid <- set_index(x, 5)
invalid

# Resolve the invalid dates by choosing the previous/next valid moment
invalid_resolve(invalid, invalid = "previous")
invalid_resolve(invalid, invalid = "next")

# You can also "overflow" the index. This keeps the weekday, but resets
# the index to 1 and increments the month value by 1.
invalid_resolve(invalid, invalid = "overflow")
```

`year-month-weekday-widen`

Widen: year-month-weekday

Description

This is a year-month-weekday method for the `calendar_widen()` generic. It widens a year-month-weekday vector to the specified `precision`.

Usage

```
## S3 method for class 'clock_year_month_weekday'
calendar_widen(x, precision)
```

Arguments

`x`	[clock_year_month_weekday] A year-month-weekday vector.
`precision`	[character(1)] One of: • `"year"` • `"month"`

- "day"
- "hour"
- "minute"
- "second"
- "millisecond"
- "microsecond"
- "nanosecond"

Details

Widening a month precision year-month-weekday to day precision will set the day and the index to 1. This sets the weekday components to the first Sunday of the month.

Value

x widened to the supplied precision.

Examples

```
# Month precision
x <- year_month_weekday(2019, 1)
x

# Widen to day precision
# Note that this sets both the day and index to 1,
# i.e. the first Sunday of the month.
calendar_widen(x, "day")

# Or second precision
sec <- calendar_widen(x, "second")
sec
```

---

year-quarter-day-arithmetic

Arithmetic: year-quarter-day

Description

These are year-quarter-day methods for the arithmetic generics.

- add_years()
- add_quarters()

Notably, *you cannot add days to a year-quarter-day*. For day-based arithmetic, first convert to a time point with as_naive_time() or as_sys_time().

Usage

```
## S3 method for class 'clock_year_quarter_day'
add_years(x, n, ...)

## S3 method for class 'clock_year_quarter_day'
add_quarters(x, n, ...)
```

Arguments

x	[clock_year_quarter_day] A year-quarter-day vector.
n	[integer / clock_duration] An integer vector to be converted to a duration, or a duration corresponding to the arithmetic function being used. This corresponds to the number of duration units to add. n may be negative to subtract units of duration.
...	These dots are for future extensions and must be empty.

Details

x and n are recycled against each other.

Value

x after performing the arithmetic.

Examples

```
x <- year_quarter_day(2019, 1:3)
x

add_quarters(x, 2)

# Make the fiscal year start in March
y <- year_quarter_day(2019, 1:2, 1, start = 3)
y

add_quarters(y, 1)

# What year-month-day does this correspond to?
# Note that the fiscal year doesn't necessarily align with the Gregorian
# year!
as_year_month_day(add_quarters(y, 1))
```

year-quarter-day-boundary
Boundaries: year-quarter-day

Description

This is a year-quarter-day method for the `calendar_start()` and `calendar_end()` generics. They adjust components of a calendar to the start or end of a specified `precision`.

Usage

```
## S3 method for class 'clock_year_quarter_day'
calendar_start(x, precision)

## S3 method for class 'clock_year_quarter_day'
calendar_end(x, precision)
```

Arguments

`x`	[clock_year_quarter_day] A year-quarter-day vector.
`precision`	[character(1)] One of: • `"year"` • `"quarter"` • `"day"` • `"hour"` • `"minute"` • `"second"` • `"millisecond"` • `"microsecond"` • `"nanosecond"`

Value

`x` at the same precision, but with some components altered to be at the boundary value.

Examples

```
x <- year_quarter_day(2019:2020, 2:3, 5, 6, 7, 8, start = clock_months$march)
x

# Compute the last moment of the fiscal quarter
calendar_end(x, "quarter")

# Compare that to just setting the day to `"last"`,
```

```
# which doesn't affect the other components
set_day(x, "last")

# Compute the start of the fiscal year
calendar_start(x, "year")

as_date(calendar_start(x, "year"))
```

`year-quarter-day-count-between`
Counting: year-quarter-day

Description

This is a year-quarter-day method for the `calendar_count_between()` generic. It counts the number of `precision` units between `start` and `end` (i.e., the number of years or quarters).

Usage

```
## S3 method for class 'clock_year_quarter_day'
calendar_count_between(start, end, precision, ..., n = 1L)
```

Arguments

`start, end`	[clock_year_quarter_day] A pair of year-quarter-day vectors. These will be recycled to their common size.
`precision`	[character(1)] One of: • `"year"` • `"quarter"`
`...`	These dots are for future extensions and must be empty.
`n`	[positive integer(1)] A single positive integer specifying a multiple of `precision` to use.

Value

An integer representing the number of `precision` units between `start` and `end`.

Examples

```
# Compute the number of whole quarters between two dates
x <- year_quarter_day(2020, 3, 91)
y <- year_quarter_day(2025, 4, c(90, 92))
calendar_count_between(x, y, "quarter")

# Note that this is not always the same as the number of whole 3 month
# periods between two dates
```

```
x <- as_year_month_day(x)
y <- as_year_month_day(y)
calendar_count_between(x, y, "month", n = 3)
```

`year-quarter-day-getters` *Getters: year-quarter-day*

Description

These are year-quarter-day methods for the getter generics.

- `get_year()` returns the fiscal year. Note that this can differ from the Gregorian year if `start != 1L`.
- `get_quarter()` returns the fiscal quarter as a value between 1-4.
- `get_day()` returns the day of the fiscal quarter as a value between 1-92.
- There are sub-daily getters for extracting more precise components.

Usage

```
## S3 method for class 'clock_year_quarter_day'
get_year(x)

## S3 method for class 'clock_year_quarter_day'
get_quarter(x)

## S3 method for class 'clock_year_quarter_day'
get_day(x)

## S3 method for class 'clock_year_quarter_day'
get_hour(x)

## S3 method for class 'clock_year_quarter_day'
get_minute(x)

## S3 method for class 'clock_year_quarter_day'
get_second(x)

## S3 method for class 'clock_year_quarter_day'
get_millisecond(x)

## S3 method for class 'clock_year_quarter_day'
get_microsecond(x)

## S3 method for class 'clock_year_quarter_day'
get_nanosecond(x)
```

Arguments

x	[clock_year_quarter_day] A year-quarter-day to get the component from.

Value

The component.

Examples

```
x <- year_quarter_day(2020, 1:4)

get_quarter(x)

# Set and then get the last day of the quarter
x <- set_day(x, "last")
get_day(x)

# Start the fiscal year in November and choose the 50th day in
# each quarter of 2020
november <- 11
y <- year_quarter_day(2020, 1:4, 50, start = 11)
y

get_day(y)

# What does that map to in year-month-day?
as_year_month_day(y)
```

---

year-quarter-day-group

Grouping: year-quarter-day

Description

This is a year-quarter-day method for the `calendar_group()` generic.

Grouping for a year-quarter-day object can be done at any precision, as long as `x` is at least as precise as `precision`.

Usage

```
## S3 method for class 'clock_year_quarter_day'
calendar_group(x, precision, ..., n = 1L)
```

Arguments

x	[clock_year_quarter_day] A year-quarter-day vector.
precision	[character(1)] One of: • "year" • "quarter" • "day" • "hour" • "minute" • "second" • "millisecond" • "microsecond" • "nanosecond"
...	These dots are for future extensions and must be empty.
n	[positive integer(1)] A single positive integer specifying a multiple of precision to use.

Value

x grouped at the specified precision.

Examples

```
x <- year_quarter_day(2019, 1:4)
x <- c(x, set_year(x, 2020))

# Group by 3 quarters
# Note that this is a grouping of 3 quarters of the current year
# (i.e. the count resets at the beginning of the next year)
calendar_group(x, "quarter", n = 3)

# Group by 5 days of the current quarter
y <- year_quarter_day(2019, 1, 1:90)
calendar_group(y, "day", n = 5)
```

---

year-quarter-day-narrow

Narrow: year-quarter-day

Description

This is a year-quarter-day method for the calendar_narrow() generic. It narrows a year-quarter-day vector to the specified precision.

Usage

```
## S3 method for class 'clock_year_quarter_day'
calendar_narrow(x, precision)
```

Arguments

x	[clock_year_quarter_day] A year-quarter-day vector.
precision	[character(1)] One of: • "year" • "quarter" • "day" • "hour" • "minute" • "second" • "millisecond" • "microsecond" • "nanosecond"

Value

x narrowed to the supplied precision.

Examples

```
# Day precision
x <- year_quarter_day(2019, 1, 5)
x

# Narrow to quarter precision
calendar_narrow(x, "quarter")
```

---

year-quarter-day-setters

Setters: year-quarter-day

Description

These are year-quarter-day methods for the setter generics.

- set_year() sets the fiscal year.
- set_quarter() sets the fiscal quarter of the year. Valid values are in the range of [1, 4].
- set_day() sets the day of the fiscal quarter. Valid values are in the range of [1, 92].
- There are sub-daily setters for setting more precise components.

Usage

```
## S3 method for class 'clock_year_quarter_day'
set_year(x, value, ...)

## S3 method for class 'clock_year_quarter_day'
set_quarter(x, value, ...)

## S3 method for class 'clock_year_quarter_day'
set_day(x, value, ...)

## S3 method for class 'clock_year_quarter_day'
set_hour(x, value, ...)

## S3 method for class 'clock_year_quarter_day'
set_minute(x, value, ...)

## S3 method for class 'clock_year_quarter_day'
set_second(x, value, ...)

## S3 method for class 'clock_year_quarter_day'
set_millisecond(x, value, ...)

## S3 method for class 'clock_year_quarter_day'
set_microsecond(x, value, ...)

## S3 method for class 'clock_year_quarter_day'
set_nanosecond(x, value, ...)
```

Arguments

x	[clock_year_quarter_day] A year-quarter-day vector.
value	[integer / "last"] The value to set the component to. For `set_day()`, this can also be `"last"` to adjust to the last day of the current fiscal quarter.
...	These dots are for future extensions and must be empty.

Value

`x` with the component set.

Examples

```
library(magrittr)

# Quarter precision vector
x <- year_quarter_day(2019, 1:4)
```

```
x

# Promote to day precision by setting the day
x <- set_day(x, 1)
x

# Or set to the last day of the quarter
x <- set_day(x, "last")
x

# What year-month-day is this?
as_year_month_day(x)

# Set to an invalid day of the quarter
# (not all quarters have 92 days)
invalid <- set_day(x, 92)
invalid

# Here are the invalid ones
invalid[invalid_detect(invalid)]

# Resolve the invalid dates by choosing the previous/next valid moment
invalid_resolve(invalid, invalid = "previous")
invalid_resolve(invalid, invalid = "next")

# Or resolve by "overflowing" by the number of days that you have
# gone past the last valid day
invalid_resolve(invalid, invalid = "overflow")

# This is similar to
days <- get_day(invalid) - 1L
invalid %>%
  set_day(1) %>%
  as_naive_time() %>%
  add_days(days) %>%
  as_year_quarter_day()
```

---

year-quarter-day-widen

Widen: year-quarter-day

Description

This is a year-quarter-day method for the `calendar_widen()` generic. It widens a year-quarter-day vector to the specified `precision`.

Usage

```
## S3 method for class 'clock_year_quarter_day'
calendar_widen(x, precision)
```

Arguments

x	[clock_year_quarter_day] A year-quarter-day vector.
precision	[character(1)] One of: • "year" • "quarter" • "day" • "hour" • "minute" • "second" • "millisecond" • "microsecond" • "nanosecond"

Value

x widened to the supplied precision.

Examples

```
# Quarter precision
x <- year_quarter_day(2019, 1)
x

# Widen to day precision
calendar_widen(x, "day")

# Or second precision
sec <- calendar_widen(x, "second")
sec
```

---

year_day *Calendar: year-day*

Description

year_day() constructs a calendar vector from the Gregorian year and day of the year.

Usage

```
year_day(
  year,
  day = NULL,
  hour = NULL,
  minute = NULL,
```

```
  second = NULL,
  subsecond = NULL,
  ...,
  subsecond_precision = NULL
)
```

Arguments

year	[integer] The year. Values [-32767, 32767] are generally allowed.
day	[integer / NULL] The day of the year. Values [1, 366] are allowed.
hour	[integer / NULL] The hour. Values [0, 23] are allowed.
minute	[integer / NULL] The minute. Values [0, 59] are allowed.
second	[integer / NULL] The second. Values [0, 59] are allowed.
subsecond	[integer / NULL] The subsecond. If specified, `subsecond_precision` must also be specified to determine how to interpret the `subsecond`. If using milliseconds, values [0, 999] are allowed. If using microseconds, values [0, 999999] are allowed. If using nanoseconds, values [0, 999999999] are allowed.
...	These dots are for future extensions and must be empty.
subsecond_precision	[character(1) / NULL] The precision to interpret `subsecond` as. One of: `"millisecond"`, `"microsecond"`, or `"nanosecond"`.

Details

Fields are recycled against each other.

Fields are collected in order until the first `NULL` field is located. No fields after the first `NULL` field are used.

Value

A year-day calendar vector.

Examples

```
# Just the year
x <- year_day(2019:2025)
x

year_day(2020, 1:10)
```

```
# Last day of the year, accounting for leap years
year_day(2019:2021, "last")

# Precision can go all the way out to nanosecond
year_day(2019, 100, 2, 40, 45, 200, subsecond_precision = "nanosecond")
```

`year_month_day` *Calendar: year-month-day*

Description

`year_month_day()` constructs the most common calendar type using the Gregorian year, month, day, and time of day components.

Usage

```
year_month_day(
  year,
  month = NULL,
  day = NULL,
  hour = NULL,
  minute = NULL,
  second = NULL,
  subsecond = NULL,
  ...,
  subsecond_precision = NULL
)
```

Arguments

`year`	[integer] The year. Values [-32767, 32767] are generally allowed.
`month`	[integer / NULL] The month. Values [1, 12] are allowed.
`day`	[integer / "last" / NULL] The day of the month. Values [1, 31] are allowed. If `"last"`, then the last day of the month is returned.
`hour`	[integer / NULL] The hour. Values [0, 23] are allowed.
`minute`	[integer / NULL] The minute. Values [0, 59] are allowed.
`second`	[integer / NULL] The second. Values [0, 59] are allowed.

subsecond [integer / NULL]
The subsecond. If specified, subsecond_precision must also be specified to determine how to interpret the subsecond.
If using milliseconds, values [0, 999] are allowed.
If using microseconds, values [0, 999999] are allowed.
If using nanoseconds, values [0, 999999999] are allowed.

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

subsecond_precision
[character(1) / NULL]
The precision to interpret subsecond as. One of: "millisecond", "microsecond", or "nanosecond".

Details

Fields are recycled against each other.

Fields are collected in order until the first NULL field is located. No fields after the first NULL field are used.

Value

A year-month-day calendar vector.

Examples

```
# Just the year
x <- year_month_day(2019:2025)

# Year-month type
year_month_day(2020, 1:12)

# The most common use case involves year, month, and day fields
x <- year_month_day(2020, clock_months$january, 1:5)
x

# Precision can go all the way out to nanosecond
year_month_day(2019, 1, 2, 2, 40, 45, 200, subsecond_precision = "nanosecond")
```

---

year_month_day_parse *Parsing: year-month-day*

Description

year_month_day_parse() parses strings into a year-month-day.

The default options assume x should be parsed at day precision, using a format string of "%Y-%m-%d".

If a more precise precision than day is used, then time components will also be parsed. The default format separates date and time components by a "T" and the time components by a ":". For example, setting the precision to "second" will use a default format of "%Y-%m-%dT%H:%M:%S". This is aligned with the format() method for year-month-day, and with the RFC 3339 standard.

Usage

```
year_month_day_parse(
  x,
  ...,
  format = NULL,
  precision = "day",
  locale = clock_locale()
)
```

Arguments

`x`	[character] A character vector to parse.
`...`	These dots are for future extensions and must be empty.
`format`	[character / NULL] A format string. A combination of the following commands, or `NULL`, in which case a default format string is used. A vector of multiple format strings can be supplied. They will be tried in the order they are provided.

Year

- %C: The century as a decimal number. The modified command %NC where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %y: The last two decimal digits of the year. If the century is not otherwise specified (e.g. with %C), values in the range [69 - 99] are presumed to refer to the years [1969 - 1999], and values in the range [00 - 68] are presumed to refer to the years [2000 - 2068]. The modified command %Ny, where `N` is a positive decimal integer, specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %Y: The year as a decimal number. The modified command %NY where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `4`. Leading zeroes are permitted but not required.

Month

- %b, %B, %h: The `locale`'s full or abbreviated case-insensitive month name.
- %m: The month as a decimal number. January is `1`. The modified command %Nm where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.

Day

- %d, %e: The day of the month as a decimal number. The modified command %Nd where `N` is a positive decimal integer specifies the maximum

number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.

Day of the week

- %a, %A: The locale's full or abbreviated case-insensitive weekday name.
- %w: The weekday as a decimal number (0-6), where Sunday is 0. The modified command %Nw where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 1. Leading zeroes are permitted but not required.

ISO 8601 week-based year

- %g: The last two decimal digits of the ISO week-based year. The modified command %Ng where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
- %G: The ISO week-based year as a decimal number. The modified command %NG where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 4. Leading zeroes are permitted but not required.
- %V: The ISO week-based week number as a decimal number. The modified command %NV where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
- %u: The ISO weekday as a decimal number (1-7), where Monday is 1. The modified command %Nu where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 1. Leading zeroes are permitted but not required.

Week of the year

- %U: The week number of the year as a decimal number. The first Sunday of the year is the first day of week 01. Days of the same year prior to that are in week 00. The modified command %NU where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
- %W: The week number of the year as a decimal number. The first Monday of the year is the first day of week 01. Days of the same year prior to that are in week 00. The modified command %NW where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.

Day of the year

- %j: The day of the year as a decimal number. January 1 is 1. The modified command %Nj where N is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 3. Leading zeroes are permitted but not required.

Date

- %D, %x: Equivalent to %m/%d/%y.
- %F: Equivalent to %Y-%m-%d. If modified with a width (like %NF), the width is applied to only %Y.

Time of day

- %H: The hour (24-hour clock) as a decimal number. The modified command %NH where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %I: The hour (12-hour clock) as a decimal number. The modified command %NI where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %M: The minutes as a decimal number. The modified command %NM where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %S: The seconds as a decimal number. Leading zeroes are permitted but not required. If encountered, the `locale` determines the decimal point character. Generally, the maximum number of characters to read is determined by the precision that you are parsing at. For example, a precision of `"second"` would read a maximum of 2 characters, while a precision of `"millisecond"` would read a maximum of 6 (2 for the values before the decimal point, 1 for the decimal point, and 3 for the values after it). The modified command %NS, where `N` is a positive decimal integer, can be used to exactly specify the maximum number of characters to read. This is only useful if you happen to have seconds with more than 1 leading zero.
- %p: The `locale`'s equivalent of the AM/PM designations associated with a 12-hour clock. The command %I must precede %p in the format string.
- %R: Equivalent to %H:%M.
- %T, %X: Equivalent to %H:%M:%S.
- %r: Equivalent to %I:%M:%S %p.

Time zone

- %z: The offset from UTC in the format [+|-]hh[mm]. For example `-0430` refers to 4 hours 30 minutes behind UTC. And `04` refers to 4 hours ahead of UTC. The modified command %Ez parses a `:` between the hours and minutes and leading zeroes on the hour field are optional: [+|-]h[h][:mm]. For example `-04:30` refers to 4 hours 30 minutes behind UTC. And `4` refers to 4 hours ahead of UTC.
- %Z: The full time zone name or the time zone abbreviation, depending on the function being used. A single word is parsed. This word can only contain characters that are alphanumeric, or one of `'_'`, `'/'`, `'-'` or `'+'`.

Miscellaneous

- %c: A date and time representation. Equivalent to %a %b %d %H:%M:%S %Y.
- `%%`: A % character.
- %n: Matches one white space character. %n, %t, and a space can be combined to match a wide range of white-space patterns. For example `"%n "` matches one or more white space characters, and `"%n%t%t"` matches one to three white space characters.

- %t: Matches zero or one white space characters.

`precision` [character(1)]

A precision for the resulting year-month-day. One of:

- `"year"`
- `"month"`
- `"day"`
- `"hour"`
- `"minute"`
- `"second"`
- `"millisecond"`
- `"microsecond"`
- `"nanosecond"`

Setting the `precision` determines how much information %S attempts to parse.

`locale` [clock_locale]

A locale object created by `clock_locale()`.

Details

`year_month_day_parse()` completely ignores the %z and %Z commands.

Value

A year-month-day calendar vector. If a parsing fails, `NA` is returned.

Full Precision Parsing

It is highly recommended to parse all of the information in the date-time string into a type at least as precise as the string. For example, if your string has fractional seconds, but you only require seconds, specify a sub-second `precision`, then round to seconds manually using whatever convention is appropriate for your use case. Parsing such a string directly into a second precision result is ambiguous and undefined, and is unlikely to work as you might expect.

Examples

```
x <- "2019-01-01"

# Default parses at day precision
year_month_day_parse(x)

# Can parse at less precise precisions too
year_month_day_parse(x, precision = "month")
year_month_day_parse(x, precision = "year")

# Even invalid dates can be round-tripped through format<->parse calls
invalid <- year_month_day(2019, 2, 30)
year_month_day_parse(format(invalid))

# Can parse with time of day
```

```
year_month_day_parse(
  "2019-01-30T02:30:00.123456789",
  precision = "nanosecond"
)

# Can parse using multiple format strings, which will be tried
# in the order they are provided
x <- c("2019-01-01", "2020-01-01", "2021/2/3")
formats <- c("%Y-%m-%d", "%Y/%m/%d")
year_month_day_parse(x, format = formats)

# Can parse using other format tokens as well
year_month_day_parse(
  "January, 2019",
  format = "%B, %Y",
  precision = "month"
)

# Parsing a French year-month-day
year_month_day_parse(
  "octobre 1, 2000",
  format = "%B %d, %Y",
  locale = clock_locale("fr")
)
```

`year_month_weekday` *Calendar: year-month-weekday*

Description

`year_month_weekday()` constructs a calendar vector from the Gregorian year, month, weekday, and index specifying that this is the n-th weekday of the month.

Usage

```
year_month_weekday(
  year,
  month = NULL,
  day = NULL,
  index = NULL,
  hour = NULL,
  minute = NULL,
  second = NULL,
  subsecond = NULL,
  ...,
  subsecond_precision = NULL
)
```

Arguments

year	[integer] The year. Values [-32767, 32767] are generally allowed.
month	[integer / NULL] The month. Values [1, 12] are allowed.
day	[integer / NULL] The weekday of the month. Values [1, 7] are allowed, where 1 is Sunday and 7 is Saturday.
index	[integer / "last" / NULL] The index specifying that day is the n-th weekday of the month. Values [1, 5] are allowed. If "last", then the last instance of day in the current month is returned.
hour	[integer / NULL] The hour. Values [0, 23] are allowed.
minute	[integer / NULL] The minute. Values [0, 59] are allowed.
second	[integer / NULL] The second. Values [0, 59] are allowed.
subsecond	[integer / NULL] The subsecond. If specified, subsecond_precision must also be specified to determine how to interpret the subsecond. If using milliseconds, values [0, 999] are allowed. If using microseconds, values [0, 999999] are allowed. If using nanoseconds, values [0, 999999999] are allowed.
...	These dots are for future extensions and must be empty.
subsecond_precision	[character(1) / NULL] The precision to interpret subsecond as. One of: "millisecond", "microsecond", or "nanosecond".

Details

Fields are recycled against each other.

Fields are collected in order until the first NULL field is located. No fields after the first NULL field are used.

Value

A year-month-weekday calendar vector.

Examples

```
# All Fridays in January, 2019
# Note that there was no 5th Friday in January
x <- year_month_weekday(
```

```
  2019,
  clock_months$january,
  clock_weekdays$friday,
  1:5
)
x

invalid_detect(x)

# Resolve this invalid date by using the previous valid date
invalid_resolve(x, invalid = "previous")
```

`year_quarter_day` *Calendar: year-quarter-day*

Description

`year_quarter_day()` constructs a calendar from the fiscal year, fiscal quarter, and day of the quarter, along with a value determining which month the fiscal year `starts` in.

Usage

```
year_quarter_day(
  year,
  quarter = NULL,
  day = NULL,
  hour = NULL,
  minute = NULL,
  second = NULL,
  subsecond = NULL,
  ...,
  start = NULL,
  subsecond_precision = NULL
)
```

Arguments

`year`	[integer] The fiscal year. Values [-32767, 32767] are generally allowed.
`quarter`	[integer / NULL] The fiscal quarter. Values [1, 4] are allowed.
`day`	[integer / "last" / NULL] The day of the quarter. Values [1, 92] are allowed. If `"last"`, the last day of the quarter is returned.
`hour`	[integer / NULL] The hour. Values [0, 23] are allowed.

`minute` [integer / NULL]
The minute. Values [0, 59] are allowed.

`second` [integer / NULL]
The second. Values [0, 59] are allowed.

`subsecond` [integer / NULL]
The subsecond. If specified, `subsecond_precision` must also be specified to determine how to interpret the `subsecond`.
If using milliseconds, values [0, 999] are allowed.
If using microseconds, values [0, 999999] are allowed.
If using nanoseconds, values [0, 999999999] are allowed.

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

`start` [integer(1) / NULL]
The month to start the fiscal year in. 1 = January and 12 = December.
If `NULL`, a `start` of January will be used.

`subsecond_precision`
[character(1) / NULL]
The precision to interpret `subsecond` as. One of: `"millisecond"`, `"microsecond"`, or `"nanosecond"`.

Details

Fields are recycled against each other.

Fields are collected in order until the first `NULL` field is located. No fields after the first `NULL` field are used.

Value

A year-quarter-day calendar vector.

Examples

```
# Year-quarter type
x <- year_quarter_day(2019, 1:4)
x

add_quarters(x, 2)

# Set the day to the last day of the quarter
x <- set_day(x, "last")
x

# Start the fiscal year in June
june <- 6L
y <- year_quarter_day(2019, 1:4, "last", start = june)

# Compare the year-month-day values that result from having different
# fiscal year start months
as_year_month_day(x)
as_year_month_day(y)
```

zoned-parsing *Parsing: zoned-time*

Description

There are two parsers into a zoned-time, `zoned_time_parse_complete()` and `zoned_time_parse_abbrev()`.

zoned_time_parse_complete():

`zoned_time_parse_complete()` is a parser for *complete* date-time strings, like `"2019-01-01T00:00:00-05:00[America`. A complete date-time string has both the time zone offset and full time zone name in the string, which is the only way for the string itself to contain all of the information required to construct a zoned-time. Because of this, `zoned_time_parse_complete()` requires both the %z and %Z commands to be supplied in the `format` string.

The default options assume that `x` should be parsed at second precision, using a `format` string of `"%Y-%m-%dT%H:%M:%S%Ez[%Z]"`. This matches the default result from calling `format()` on a zoned-time. Additionally, this format matches the de-facto standard extension to RFC 3339 for creating completely unambiguous date-times.

zoned_time_parse_abbrev():

`zoned_time_parse_abbrev()` is a parser for date-time strings containing only a time zone abbreviation, like `"2019-01-01 00:00:00 EST"`. The time zone abbreviation is not enough to identify the full time zone name that the date-time belongs to, so the full time zone name must be supplied as the `zone` argument. However, the time zone abbreviation can help with resolving ambiguity around daylight saving time fallbacks.

For `zoned_time_parse_abbrev()`, %Z must be supplied and is interpreted as the time zone abbreviation rather than the full time zone name.

If used, the %z command must parse correctly, but its value will be completely ignored.

The default options assume that `x` should be parsed at second precision, using a `format` string of `"%Y-%m-%d %H:%M:%S %Z"`. This matches the default result from calling `print()` or `format(usetz = TRUE)` on a POSIXct date-time.

Usage

```
zoned_time_parse_complete(
  x,
  ...,
  format = NULL,
  precision = "second",
  locale = clock_locale()
)

zoned_time_parse_abbrev(
  x,
  zone,
  ...,
  format = NULL,
```

```
  precision = "second",
  locale = clock_locale()
)
```

Arguments

`x`	[character] A character vector to parse.
`...`	These dots are for future extensions and must be empty.
`format`	[character / NULL] A format string. A combination of the following commands, or `NULL`, in which case a default format string is used. A vector of multiple format strings can be supplied. They will be tried in the order they are provided.

Year

- %C: The century as a decimal number. The modified command %NC where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %y: The last two decimal digits of the year. If the century is not otherwise specified (e.g. with %C), values in the range [69 - 99] are presumed to refer to the years [1969 - 1999], and values in the range [00 - 68] are presumed to refer to the years [2000 - 2068]. The modified command %Ny, where `N` is a positive decimal integer, specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %Y: The year as a decimal number. The modified command %NY where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `4`. Leading zeroes are permitted but not required.

Month

- %b, %B, %h: The `locale`'s full or abbreviated case-insensitive month name.
- %m: The month as a decimal number. January is `1`. The modified command %Nm where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.

Day

- %d, %e: The day of the month as a decimal number. The modified command %Nd where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.

Day of the week

- %a, %A: The `locale`'s full or abbreviated case-insensitive weekday name.

- %w: The weekday as a decimal number (`0-6`), where Sunday is `0`. The modified command %Nw where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `1`. Leading zeroes are permitted but not required.

ISO 8601 week-based year

- %g: The last two decimal digits of the ISO week-based year. The modified command %Ng where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %G: The ISO week-based year as a decimal number. The modified command %NG where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `4`. Leading zeroes are permitted but not required.
- %V: The ISO week-based week number as a decimal number. The modified command %NV where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %u: The ISO weekday as a decimal number (`1-7`), where Monday is `1`. The modified command %Nu where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `1`. Leading zeroes are permitted but not required.

Week of the year

- %U: The week number of the year as a decimal number. The first Sunday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. The modified command %NU where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %W: The week number of the year as a decimal number. The first Monday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. The modified command %NW where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.

Day of the year

- %j: The day of the year as a decimal number. January 1 is `1`. The modified command %Nj where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `3`. Leading zeroes are permitted but not required.

Date

- %D, %x: Equivalent to %m/%d/%y.
- %F: Equivalent to %Y-%m-%d. If modified with a width (like %NF), the width is applied to only %Y.

Time of day

- %H: The hour (24-hour clock) as a decimal number. The modified command %NH where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.

- %I: The hour (12-hour clock) as a decimal number. The modified command %NI where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %M: The minutes as a decimal number. The modified command %NM where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is `2`. Leading zeroes are permitted but not required.
- %S: The seconds as a decimal number. Leading zeroes are permitted but not required. If encountered, the `locale` determines the decimal point character. Generally, the maximum number of characters to read is determined by the precision that you are parsing at. For example, a precision of `"second"` would read a maximum of 2 characters, while a precision of `"millisecond"` would read a maximum of 6 (2 for the values before the decimal point, 1 for the decimal point, and 3 for the values after it). The modified command %NS, where `N` is a positive decimal integer, can be used to exactly specify the maximum number of characters to read. This is only useful if you happen to have seconds with more than 1 leading zero.
- %p: The `locale`'s equivalent of the AM/PM designations associated with a 12-hour clock. The command %I must precede %p in the format string.
- %R: Equivalent to %H:%M.
- %T, %X: Equivalent to %H:%M:%S.
- %r: Equivalent to %I:%M:%S %p.

Time zone

- %z: The offset from UTC in the format [+|-]hh[mm]. For example `-0430` refers to 4 hours 30 minutes behind UTC. And `04` refers to 4 hours ahead of UTC. The modified command %Ez parses a `:` between the hours and minutes and leading zeroes on the hour field are optional: [+|-]h[h][:mm]. For example `-04:30` refers to 4 hours 30 minutes behind UTC. And `4` refers to 4 hours ahead of UTC.
- %Z: The full time zone name or the time zone abbreviation, depending on the function being used. A single word is parsed. This word can only contain characters that are alphanumeric, or one of `'_'`, `'/'`, `'-'` or `'+'`.

Miscellaneous

- %c: A date and time representation. Equivalent to %a %b %d %H:%M:%S %Y.
- `%%`: A % character.
- %n: Matches one white space character. %n, %t, and a space can be combined to match a wide range of white-space patterns. For example `"%n "` matches one or more white space characters, and `"%n%t%t"` matches one to three white space characters.
- %t: Matches zero or one white space characters.

`precision` [character(1)]

A precision for the resulting zoned-time. One of:

- `"second"`
- `"millisecond"`

- "microsecond"
- "nanosecond"

Setting the precision determines how much information %S attempts to parse.

locale [clock_locale]

A locale object created from clock_locale().

zone [character(1)]

A full time zone name.

Details

If zoned_time_parse_complete() is given input that is length zero, all NAs, or completely fails to parse, then no time zone will be able to be determined. In that case, the result will use "UTC".

If your date-time strings contain time zone offsets (like -04:00), but not the full time zone name, you might need sys_time_parse().

If your date-time strings don't contain time zone offsets or the full time zone name, you might need to use naive_time_parse(). From there, if you know the time zone that the date-times are supposed to be in, you can convert to a zoned-time with as_zoned_time().

Value

A zoned-time.

Full Precision Parsing

It is highly recommended to parse all of the information in the date-time string into a type at least as precise as the string. For example, if your string has fractional seconds, but you only require seconds, specify a sub-second precision, then round to seconds manually using whatever convention is appropriate for your use case. Parsing such a string directly into a second precision result is ambiguous and undefined, and is unlikely to work as you might expect.

Examples

```
library(magrittr)

zoned_time_parse_complete("2019-01-01T01:02:03-05:00[America/New_York]")

zoned_time_parse_complete(
  "January 21, 2019 -0500 America/New_York",
  format = "%B %d, %Y %z %Z"
)

# Nanosecond precision
x <- "2019/12/31 01:05:05.123456700-05:00[America/New_York]"
zoned_time_parse_complete(
  x,
  format = "%Y/%m/%d %H:%M:%S%Ez[%Z]",
  precision = "nanosecond"
)
```

```
# The `%z` offset must correspond to the true offset that would be used
# if the input was parsed as a naive-time and then converted to a zoned-time
# with `as_zoned_time()`. For example, the time that was parsed above used an
# offset of `-05:00`. We can confirm that this is correct with:
year_month_day(2019, 1, 1, 1, 2, 3) %>%
  as_naive_time() %>%
  as_zoned_time("America/New_York")

# So the following would not parse correctly
zoned_time_parse_complete("2019-01-01T01:02:03-04:00[America/New_York]")

# `%z` is useful for breaking ties in otherwise ambiguous times. For example,
# these two times are on either side of a daylight saving time fallback.
# Without the `%z` offset, you wouldn't be able to tell them apart!
x <- c(
  "1970-10-25T01:30:00-04:00[America/New_York]",
  "1970-10-25T01:30:00-05:00[America/New_York]"
)

zoned_time_parse_complete(x)

# If you have date-time strings with time zone abbreviations,
# `zoned_time_parse_abbrev()` should be able to help. The `zone` must be
# provided, because multiple countries may use the same time zone
# abbreviation. For example:
x <- "1970-01-01 02:30:30 IST"

# IST = India Standard Time
zoned_time_parse_abbrev(x, "Asia/Kolkata")

# IST = Israel Standard Time
zoned_time_parse_abbrev(x, "Asia/Jerusalem")

# The time zone abbreviation is mainly useful for resolving ambiguity
# around daylight saving time fallbacks. Without the abbreviation, these
# date-times would be ambiguous.
x <- c(
  "1970-10-25 01:30:00 EDT",
  "1970-10-25 01:30:00 EST"
)
zoned_time_parse_abbrev(x, "America/New_York")
```

---

zoned-zone *Get or set the time zone*

Description

`zoned_time_zone()` gets the time zone.

`zoned_time_set_zone()` sets the time zone *without changing the underlying instant*. This means that the result will represent the equivalent time in the new time zone.

Usage

```
zoned_time_zone(x)

zoned_time_set_zone(x, zone)
```

Arguments

x	[zoned_time / Date / POSIXt] A zoned time to get or set the time zone of.
zone	[character(1)] A valid time zone to switch to.

Value

`zoned_time_zone()` returns a string containing the time zone.

`zoned_time_set_zone()` returns `x` with an altered time zone attribute. The underlying instant is *not* changed.

Examples

```
x <- year_month_day(2019, 1, 1)
x <- as_zoned_time(as_naive_time(x), "America/New_York")
x

zoned_time_zone(x)

# Equivalent UTC time
zoned_time_set_zone(x, "UTC")

# To force a new time zone with the same printed time,
# convert to a naive time that has no implied time zone,
# then convert back to a zoned time in the new time zone.
nt <- as_naive_time(x)
nt
as_zoned_time(nt, "UTC")
```

---

zoned_time_now *What is the current zoned-time?*

Description

`zoned_time_now()` returns the current time in the corresponding zone. It is a wrapper around `sys_time_now()` that attaches the time zone.

Usage

```
zoned_time_now(zone)
```

Arguments

zone	[character(1)] A time zone to get the current time for.

Details

The time is returned with a nanosecond precision, but the actual amount of data returned is OS dependent. Usually, information at at least the microsecond level is returned, with some platforms returning nanosecond information.

Value

A zoned-time of the current time.

Examples

```
x <- zoned_time_now("America/New_York")
```

---

zoned_time_precision *Precision: zoned-time*

Description

`zoned_time_precision()` extracts the precision from a zoned-time. It returns the precision as a single string.

Usage

```
zoned_time_precision(x)
```

Arguments

x	[clock_zoned_time] A zoned-time.

Value

A single string holding the precision of the zoned-time.

Examples

```
zoned_time_precision(zoned_time_now("America/New_York"))
```

Index

∗ **datasets**
clock-codes, 35

add_days *(clock-arithmetic)*, 33
add_days.clock_duration *(duration-arithmetic)*, 91
add_days.clock_time_point *(time-point-arithmetic)*, 172
add_days.clock_weekday *(weekday-arithmetic)*, 184
add_days.Date *(Date-arithmetic)*, 47
add_days.POSIXt *(posixt-arithmetic)*, 124
add_hours *(clock-arithmetic)*, 33
add_hours.clock_duration *(duration-arithmetic)*, 91
add_hours.clock_time_point *(time-point-arithmetic)*, 172
add_hours.POSIXt *(posixt-arithmetic)*, 124
add_microseconds *(clock-arithmetic)*, 33
add_microseconds.clock_duration *(duration-arithmetic)*, 91
add_microseconds.clock_time_point *(time-point-arithmetic)*, 172
add_milliseconds *(clock-arithmetic)*, 33
add_milliseconds.clock_duration *(duration-arithmetic)*, 91
add_milliseconds.clock_time_point *(time-point-arithmetic)*, 172
add_minutes *(clock-arithmetic)*, 33
add_minutes.clock_duration *(duration-arithmetic)*, 91
add_minutes.clock_time_point *(time-point-arithmetic)*, 172
add_minutes.POSIXt *(posixt-arithmetic)*, 124
add_months *(clock-arithmetic)*, 33
add_months.clock_duration *(duration-arithmetic)*, 91
add_months.clock_year_month_day *(year-month-day-arithmetic)*, 197
add_months.clock_year_month_weekday *(year-month-weekday-arithmetic)*, 207
add_months.Date *(Date-arithmetic)*, 47
add_months.POSIXt *(posixt-arithmetic)*, 124
add_nanoseconds *(clock-arithmetic)*, 33
add_nanoseconds.clock_duration *(duration-arithmetic)*, 91
add_nanoseconds.clock_time_point *(time-point-arithmetic)*, 172
add_quarters *(clock-arithmetic)*, 33
add_quarters.clock_duration *(duration-arithmetic)*, 91
add_quarters.clock_year_month_day *(year-month-day-arithmetic)*, 197
add_quarters.clock_year_month_weekday *(year-month-weekday-arithmetic)*, 207
add_quarters.clock_year_quarter_day *(year-quarter-day-arithmetic)*, 217
add_quarters.Date *(Date-arithmetic)*, 47
add_quarters.POSIXt *(posixt-arithmetic)*, 124
add_seconds *(clock-arithmetic)*, 33
add_seconds.clock_duration *(duration-arithmetic)*, 91
add_seconds.clock_time_point *(time-point-arithmetic)*, 172
add_seconds.POSIXt *(posixt-arithmetic)*, 124
add_weeks *(clock-arithmetic)*, 33
add_weeks.clock_duration *(duration-arithmetic)*, 91

add_weeks.clock_time_point
(time-point-arithmetic), 172
add_weeks.Date (Date-arithmetic), 47
add_weeks.POSIXt (posixt-arithmetic),
124
add_years (clock-arithmetic), 33
add_years.clock_duration
(duration-arithmetic), 91
add_years.clock_iso_year_week_day
(iso-year-week-day-arithmetic),
101
add_years.clock_year_day
(year-day-arithmetic), 187
add_years.clock_year_month_day
(year-month-day-arithmetic),
197
add_years.clock_year_month_weekday
(year-month-weekday-arithmetic),
207
add_years.clock_year_quarter_day
(year-quarter-day-arithmetic),
217
add_years.Date (Date-arithmetic), 47
add_years.POSIXt (posixt-arithmetic),
124
arithmetic generics, *47*, *91*, *101*, *124*, *172*,
184, *187*, *197*, *207*, *217*
as-zoned-time-Date, 5
as-zoned-time-naive-time, 7
as-zoned-time-posixt, 11
as-zoned-time-sys-time, 12
as_date, 13
as_date(), *14*, *85*
as_date_time, 14
as_date_time(), *13*, *71*
as_duration, 17
as_iso_year_week_day, 18
as_naive_time, 19
as_naive_time(), *101*, *127*, *187*, *197*, *207*,
217
as_sys_time, 20
as_sys_time(), *101*, *187*, *197*, *207*, *217*
as_weekday, 21
as_year_day, 22
as_year_month_day, 22
as_year_month_day(), *172*
as_year_month_weekday, 23
as_year_quarter_day, 24
as_zoned_time, 25
as_zoned_time(), *5*, *7*, *12*, *120*, *164*, *243*
as_zoned_time.clock_naive_time
(as-zoned-time-naive-time), 7
as_zoned_time.clock_sys_time
(as-zoned-time-sys-time), 12
as_zoned_time.Date
(as-zoned-time-Date), 5
as_zoned_time.POSIXt
(as-zoned-time-posixt), 12

calendar-boundary, 26
calendar-count-between, 27
calendar_count_between
(calendar-count-between), 27
calendar_count_between(), *103*, *190*, *199*,
209, *220*
calendar_count_between.clock_iso_year_week_day
(iso-year-week-day-count-between),
103
calendar_count_between.clock_year_day
(year-day-count-between), 190
calendar_count_between.clock_year_month_day
(year-month-day-count-between),
199
calendar_count_between.clock_year_month_weekday
(year-month-weekday-count-between),
209
calendar_count_between.clock_year_quarter_day
(year-quarter-day-count-between),
220
calendar_end (calendar-boundary), 26
calendar_end(), *102*, *189*, *198*, *208*, *219*
calendar_end.clock_iso_year_week_day
(iso-year-week-day-boundary),
102
calendar_end.clock_year_day
(year-day-boundary), 188
calendar_end.clock_year_month_day
(year-month-day-boundary), 198
calendar_end.clock_year_month_weekday
(year-month-weekday-boundary),
208
calendar_end.clock_year_quarter_day
(year-quarter-day-boundary),
219
calendar_group, 28
calendar_group(), *106*, *174*, *192*, *202*, *212*,
222

calendar_group.clock_iso_year_week_day (iso-year-week-day-group), 106
calendar_group.clock_year_day (year-day-group), 192
calendar_group.clock_year_month_day (year-month-day-group), 202
calendar_group.clock_year_month_weekday (year-month-weekday-group), 212
calendar_group.clock_year_quarter_day (year-quarter-day-group), 222
calendar_leap_year, 29
calendar_month_factor, 30
calendar_narrow, 31
calendar_narrow(), *107*, *193*, *203*, *213*, *223*
calendar_narrow.clock_iso_year_week_day (iso-year-week-day-narrow), 107
calendar_narrow.clock_year_day (year-day-narrow), 193
calendar_narrow.clock_year_month_day (year-month-day-narrow), 203
calendar_narrow.clock_year_month_weekday (year-month-weekday-narrow), 213
calendar_narrow.clock_year_quarter_day (year-quarter-day-narrow), 223
calendar_precision, 32
calendar_start (calendar-boundary), 26
calendar_start(), *102*, *189*, *198*, *208*, *219*
calendar_start.clock_iso_year_week_day (iso-year-week-day-boundary), 102
calendar_start.clock_year_day (year-day-boundary), 188
calendar_start.clock_year_month_day (year-month-day-boundary), 198
calendar_start.clock_year_month_weekday (year-month-weekday-boundary), 208
calendar_start.clock_year_quarter_day (year-quarter-day-boundary), 219
calendar_widen, 32
calendar_widen(), *109*, *196*, *206*, *216*, *226*
calendar_widen.clock_iso_year_week_day (iso-year-week-day-widen), 109
calendar_widen.clock_year_day (year-day-widen), 196
calendar_widen.clock_year_month_day (year-month-day-widen), 206
calendar_widen.clock_year_month_weekday (year-month-weekday-widen), 216
calendar_widen.clock_year_quarter_day (year-quarter-day-widen), 226
clock-arith (vec_arith.clock_year_day), 182
clock-arithmetic, 33
clock-codes, 35
clock-getters, 37
clock-invalid, 38
clock-setters, 40
clock_iso_weekdays (clock-codes), 35
clock_labels, 42
clock_labels(), *30*, *43*, *81*, *90*, *186*
clock_labels_languages (clock_labels), 42
clock_labels_lookup (clock_labels), 42
clock_labels_lookup(), *30*, *43*, *81*, *90*, *186*
clock_locale, 43
clock_locale(), *55*, *70*, *84*, *101*, *123*, *136*, *168*, *234*, *243*
clock_months (clock-codes), 35
clock_weekdays (clock-codes), 35

date-and-date-time-boundary, 44
date-and-date-time-rounding, 45
date-and-date-time-shifting, 46
Date-arithmetic, 47
date-boundary, 49
date-count-between, 51
date-formatting, 53
Date-getters, 56
date-group, 57
date-rounding, 58
date-sequence, 60
Date-setters, 62
date-shifting, 64
date-time-parse, 65
date-times (POSIXct / POSIXlt), *25*, *34*, *37*, *40*
date-times (POSIXct/POSIXlt), *44–46*, *77–79*, *86*
date-today, 73
date-zone, 74
date_build, 75
date_ceiling (date-and-date-time-rounding), 45

date_ceiling.Date (date-rounding), 58
date_ceiling.POSIXt (posixt-rounding), 141
date_count_between, 77
date_count_between(), *51*, *131*
date_count_between.Date (date-count-between), 51
date_count_between.POSIXt (posixt-count-between), 131
date_end (date-and-date-time-boundary), 44
date_end(), *49*, *128*
date_end.Date (date-boundary), 49
date_end.POSIXt (posixt-boundary), 128
date_floor (date-and-date-time-rounding), 45
date_floor(), *64*
date_floor.Date (date-rounding), 58
date_floor.POSIXt (posixt-rounding), 141
date_format, 78
date_format(), *53*, *134*
date_format.Date (date-formatting), 53
date_format.POSIXt (posixt-formatting), 134
date_group, 79
date_group(), *45*, *57*, *58*, *138*, *141*
date_group.Date (date-group), 57
date_group.POSIXt (posixt-group), 138
date_leap_year, 80
date_month_factor, 80
date_now (date-today), 73
date_parse, 81
date_parse(), *14*, *84*
date_round (date-and-date-time-rounding), 45
date_round(), *85*
date_round.Date (date-rounding), 58
date_round.POSIXt (posixt-rounding), 141
date_seq, 86
date_seq(), *60*, *144*
date_seq.Date (date-sequence), 60
date_seq.POSIXt (posixt-sequence), 144
date_set_zone (date-zone), 74
date_shift (date-and-date-time-shifting), 46
date_shift.Date (date-shifting), 64
date_shift.POSIXt (posixt-shifting), 152
date_start (date-and-date-time-boundary), 44
date_start(), *49*, *128*
date_start.Date (date-boundary), 49
date_start.POSIXt (posixt-boundary), 128
date_time_build, 87
date_time_parse (date-time-parse), 65
date_time_parse(), *16*, *84*, *85*
date_time_parse_abbrev (date-time-parse), 65
date_time_parse_complete (date-time-parse), 65
date_time_parse_complete(), *134*
date_time_parse_RFC_3339 (date-time-parse), 65
date_today (date-today), 73
date_weekday_factor, 90
date_zone (date-zone), 74
dates (Date), *25*, *34*, *37*, *40*, *44–46*, *77–79*, *86*
duration, *34*
duration helper, *92*, *96*
duration-arithmetic, 91
duration-helper, 93
duration-rounding, 95
duration_cast, 97
duration_cast(), *95*
duration_ceiling (duration-rounding), 95
duration_days (duration-helper), 93
duration_floor (duration-rounding), 95
duration_floor(), *17*, *97*
duration_hours (duration-helper), 93
duration_microseconds (duration-helper), 93
duration_milliseconds (duration-helper), 93
duration_minutes (duration-helper), 93
duration_months (duration-helper), 93
duration_nanoseconds (duration-helper), 93
duration_precision, 98
duration_quarters (duration-helper), 93
duration_round (duration-rounding), 95
duration_seconds (duration-helper), 93
duration_weeks (duration-helper), 93

duration_years (duration-helper), 93

format(), *99*, *230*
format.clock_zoned_time, 99
from a naive-time, *12*
from a sys-time, *7*

get_day (clock-getters), 37
get_day.clock_iso_year_week_day (iso-year-week-day-getters), 104
get_day.clock_year_day (year-day-getters), 191
get_day.clock_year_month_day (year-month-day-getters), 200
get_day.clock_year_month_weekday (year-month-weekday-getters), 210
get_day.clock_year_quarter_day (year-quarter-day-getters), 221
get_day.Date (Date-getters), 56
get_day.POSIXt (posixt-getters), 137
get_hour (clock-getters), 37
get_hour.clock_iso_year_week_day (iso-year-week-day-getters), 104
get_hour.clock_year_day (year-day-getters), 191
get_hour.clock_year_month_day (year-month-day-getters), 200
get_hour.clock_year_month_weekday (year-month-weekday-getters), 210
get_hour.clock_year_quarter_day (year-quarter-day-getters), 221
get_hour.POSIXt (posixt-getters), 137
get_index (clock-getters), 37
get_index.clock_year_month_weekday (year-month-weekday-getters), 210
get_microsecond (clock-getters), 37
get_microsecond.clock_iso_year_week_day (iso-year-week-day-getters), 104
get_microsecond.clock_year_day (year-day-getters), 191
get_microsecond.clock_year_month_day (year-month-day-getters), 200
get_microsecond.clock_year_month_weekday (year-month-weekday-getters), 210
get_microsecond.clock_year_quarter_day (year-quarter-day-getters), 221
get_millisecond (clock-getters), 37
get_millisecond.clock_iso_year_week_day (iso-year-week-day-getters), 104
get_millisecond.clock_year_day (year-day-getters), 191
get_millisecond.clock_year_month_day (year-month-day-getters), 200
get_millisecond.clock_year_month_weekday (year-month-weekday-getters), 210
get_millisecond.clock_year_quarter_day (year-quarter-day-getters), 221
get_minute (clock-getters), 37
get_minute.clock_iso_year_week_day (iso-year-week-day-getters), 104
get_minute.clock_year_day (year-day-getters), 191
get_minute.clock_year_month_day (year-month-day-getters), 200
get_minute.clock_year_month_weekday (year-month-weekday-getters), 210
get_minute.clock_year_quarter_day (year-quarter-day-getters), 221
get_minute.POSIXt (posixt-getters), 137
get_month (clock-getters), 37
get_month.clock_year_month_day (year-month-day-getters), 200
get_month.clock_year_month_weekday (year-month-weekday-getters), 210
get_month.Date (Date-getters), 56
get_month.POSIXt (posixt-getters), 137
get_nanosecond (clock-getters), 37
get_nanosecond.clock_iso_year_week_day (iso-year-week-day-getters), 104
get_nanosecond.clock_year_day (year-day-getters), 191
get_nanosecond.clock_year_month_day (year-month-day-getters), 200

get_nanosecond.clock_year_month_weekday *(year-month-weekday-getters)*, 210
get_nanosecond.clock_year_quarter_day *(year-quarter-day-getters)*, 221
get_quarter *(clock-getters)*, 37
get_quarter.clock_year_quarter_day *(year-quarter-day-getters)*, 221
get_second *(clock-getters)*, 37
get_second.clock_iso_year_week_day *(iso-year-week-day-getters)*, 104
get_second.clock_year_day *(year-day-getters)*, 191
get_second.clock_year_month_day *(year-month-day-getters)*, 200
get_second.clock_year_month_weekday *(year-month-weekday-getters)*, 210
get_second.clock_year_quarter_day *(year-quarter-day-getters)*, 221
get_second.POSIXt *(posixt-getters)*, 137
get_week *(clock-getters)*, 37
get_week.clock_iso_year_week_day *(iso-year-week-day-getters)*, 104
get_year *(clock-getters)*, 37
get_year.clock_iso_year_week_day *(iso-year-week-day-getters)*, 104
get_year.clock_year_day *(year-day-getters)*, 191
get_year.clock_year_month_day *(year-month-day-getters)*, 200
get_year.clock_year_month_weekday *(year-month-weekday-getters)*, 210
get_year.clock_year_quarter_day *(year-quarter-day-getters)*, 221
get_year.Date *(Date-getters)*, 56
get_year.POSIXt *(posixt-getters)*, 137
getter generics, *56*, *104*, *137*, *191*, *201*, *211*, *221*

invalid_any *(clock-invalid)*, 38
invalid_count *(clock-invalid)*, 38
invalid_detect *(clock-invalid)*, 38
invalid_remove *(clock-invalid)*, 38
invalid_resolve *(clock-invalid)*, 38
invalid_resolve(), *19*, *20*, *41*
is_duration, 112
is_iso_year_week_day, 113
is_naive_time, 113
is_sys_time, 114
is_weekday, 114
is_year_day, 115
is_year_month_day, 115
is_year_month_weekday, 116
is_year_quarter_day, 116
is_zoned_time, 117
iso-year-week-day, *26–28*, *31*, *32*, *34*, *37*, *40*
iso-year-week-day-arithmetic, 101
iso-year-week-day-boundary, 102
iso-year-week-day-count-between, 103
iso-year-week-day-getters, 104
iso-year-week-day-group, 106
iso-year-week-day-narrow, 107
iso-year-week-day-setters, 108
iso-year-week-day-widen, 109
iso_year_week_day, 110

naive-time, *25*
naive_time_info, 117
naive_time_parse, 119
naive_time_parse(), *71*, *164*, *168*, *243*

posixt-arithmetic, 124
posixt-boundary, 128
posixt-count-between, 131
posixt-formatting, 133
posixt-getters, 137
posixt-group, 138
posixt-rounding, 141
posixt-sequence, 144
posixt-setters, 149
posixt-shifting, 152

rounding generics, *58*, *141*

seq(), *154*, *155*, *157*, *158*, *160–162*
seq.clock_duration, 154
seq.clock_iso_year_week_day, 155
seq.clock_time_point, 157
seq.clock_year_day, 158
seq.clock_year_month_day, 160
seq.clock_year_month_weekday, 161
seq.clock_year_quarter_day, 162
set_day *(clock-setters)*, 40

set_day.clock_iso_year_week_day (iso-year-week-day-setters), 108
set_day.clock_year_day (year-day-setters), 194
set_day.clock_year_month_day (year-month-day-setters), 204
set_day.clock_year_month_weekday (year-month-weekday-setters), 214
set_day.clock_year_quarter_day (year-quarter-day-setters), 224
set_day.Date (Date-setters), 62
set_day.POSIXt (posixt-setters), 149
set_hour (clock-setters), 40
set_hour.clock_iso_year_week_day (iso-year-week-day-setters), 108
set_hour.clock_year_day (year-day-setters), 194
set_hour.clock_year_month_day (year-month-day-setters), 204
set_hour.clock_year_month_weekday (year-month-weekday-setters), 214
set_hour.clock_year_quarter_day (year-quarter-day-setters), 224
set_hour.POSIXt (posixt-setters), 149
set_index (clock-setters), 40
set_index.clock_year_month_weekday (year-month-weekday-setters), 214
set_microsecond (clock-setters), 40
set_microsecond.clock_iso_year_week_day (iso-year-week-day-setters), 108
set_microsecond.clock_year_day (year-day-setters), 194
set_microsecond.clock_year_month_day (year-month-day-setters), 204
set_microsecond.clock_year_month_weekday (year-month-weekday-setters), 214
set_microsecond.clock_year_quarter_day (year-quarter-day-setters), 224
set_millisecond (clock-setters), 40
set_millisecond.clock_iso_year_week_day (iso-year-week-day-setters), 108
set_millisecond.clock_year_day (year-day-setters), 194
set_millisecond.clock_year_month_day (year-month-day-setters), 204
set_millisecond.clock_year_month_weekday (year-month-weekday-setters), 214
set_millisecond.clock_year_quarter_day (year-quarter-day-setters), 224
set_minute (clock-setters), 40
set_minute.clock_iso_year_week_day (iso-year-week-day-setters), 108
set_minute.clock_year_day (year-day-setters), 194
set_minute.clock_year_month_day (year-month-day-setters), 204
set_minute.clock_year_month_weekday (year-month-weekday-setters), 214
set_minute.clock_year_quarter_day (year-quarter-day-setters), 224
set_minute.POSIXt (posixt-setters), 149
set_month (clock-setters), 40
set_month.clock_year_month_day (year-month-day-setters), 204
set_month.clock_year_month_weekday (year-month-weekday-setters), 214
set_month.Date (Date-setters), 62
set_month.POSIXt (posixt-setters), 149
set_nanosecond (clock-setters), 40
set_nanosecond.clock_iso_year_week_day (iso-year-week-day-setters), 108
set_nanosecond.clock_year_day (year-day-setters), 194
set_nanosecond.clock_year_month_day (year-month-day-setters), 204
set_nanosecond.clock_year_month_weekday (year-month-weekday-setters), 214
set_nanosecond.clock_year_quarter_day (year-quarter-day-setters), 224
set_quarter (clock-setters), 40
set_quarter.clock_year_quarter_day (year-quarter-day-setters), 224

set_second *(clock-setters)*, 40
set_second.clock_iso_year_week_day *(iso-year-week-day-setters)*, 108
set_second.clock_year_day *(year-day-setters)*, 194
set_second.clock_year_month_day *(year-month-day-setters)*, 204
set_second.clock_year_month_weekday *(year-month-weekday-setters)*, 214
set_second.clock_year_quarter_day *(year-quarter-day-setters)*, 224
set_second.POSIXt *(posixt-setters)*, 149
set_week *(clock-setters)*, 40
set_week.clock_iso_year_week_day *(iso-year-week-day-setters)*, 108
set_year *(clock-setters)*, 40
set_year.clock_iso_year_week_day *(iso-year-week-day-setters)*, 108
set_year.clock_year_day *(year-day-setters)*, 194
set_year.clock_year_month_day *(year-month-day-setters)*, 204
set_year.clock_year_month_weekday *(year-month-weekday-setters)*, 214
set_year.clock_year_quarter_day *(year-quarter-day-setters)*, 224
set_year.Date *(Date-setters)*, 62
set_year.POSIXt *(posixt-setters)*, 149
setter generics, *62*, *108*, *149*, *194*, *204*, *214*, *224*
sys-parsing, 163
sys-time, *25*
sys_time_info, 169
sys_time_info(), *117*
sys_time_now, 171
sys_time_now(), *245*
sys_time_parse *(sys-parsing)*, 163
sys_time_parse(), *120*, *243*
sys_time_parse_RFC_3339 *(sys-parsing)*, 163

time-point, *34*
time-point-arithmetic, 172
time-point-rounding, 174
time_point_cast, 176
time_point_cast(), *33*
time_point_ceiling *(time-point-rounding)*, 174
time_point_count_between, 178
time_point_floor *(time-point-rounding)*, 174
time_point_floor(), *31*, *71*, *176*, *181*
time_point_precision, 180
time_point_round *(time-point-rounding)*, 174
time_point_shift, 181

vctrs::new_date(), *14*
vctrs::new_datetime(), *16*
vec_arith.clock_iso_year_week_day *(vec_arith.clock_year_day)*, 182
vec_arith.clock_naive_time *(vec_arith.clock_year_day)*, 182
vec_arith.clock_sys_time *(vec_arith.clock_year_day)*, 182
vec_arith.clock_weekday *(vec_arith.clock_year_day)*, 182
vec_arith.clock_year_day, 182
vec_arith.clock_year_month_day *(vec_arith.clock_year_day)*, 182
vec_arith.clock_year_month_weekday *(vec_arith.clock_year_day)*, 182
vec_arith.clock_year_quarter_day *(vec_arith.clock_year_day)*, 182

weekday, *34*, 183
weekday(), *46*, *64*, *152*, *181*, *184*
weekday-arithmetic, 184
weekday_code, 185
weekday_factor, 186

year-day, *26–28*, *31*, *32*, *34*, *37*, *40*
year-day-arithmetic, 187
year-day-boundary, 188
year-day-count-between, 190
year-day-getters, 191
year-day-group, 192
year-day-narrow, 193
year-day-setters, 194
year-day-widen, 196
year-month-day, *26–28*, *31*, *32*, *34*, *37*, *40*
year-month-day-arithmetic, 197
year-month-day-boundary, 198

year-month-day-count-between, 199
year-month-day-getters, 200
year-month-day-group, 202
year-month-day-narrow, 203
year-month-day-setters, 204
year-month-day-widen, 206
year-month-weekday, *26–28*, *31, 32*, *34*, *37*, *40*
year-month-weekday-arithmetic, 207
year-month-weekday-boundary, 208
year-month-weekday-count-between, 209
year-month-weekday-getters, 210
year-month-weekday-group, 212
year-month-weekday-narrow, 213
year-month-weekday-setters, 214
year-month-weekday-widen, 216
year-quarter-day, *26–28*, *31, 32*, *34*, *37*, *40*
year-quarter-day-arithmetic, 217
year-quarter-day-boundary, 219
year-quarter-day-count-between, 220
year-quarter-day-getters, 221
year-quarter-day-group, 222
year-quarter-day-narrow, 223
year-quarter-day-setters, 224
year-quarter-day-widen, 226
year_day, 227
year_day(), *29*
year_month_day, 229
year_month_day(), *29, 30*, *38*
year_month_day_parse, 230
year_month_day_parse(), *43*
year_month_weekday, 235
year_month_weekday(), *29, 30*
year_quarter_day, 237

zoned-parsing, 239
zoned-zone, 244
zoned_time_now, 245
zoned_time_parse_abbrev *(*zoned-parsing*)*, 239
zoned_time_parse_abbrev(), *120*, *168*
zoned_time_parse_complete *(*zoned-parsing*)*, 239
zoned_time_parse_complete(), *99*, *120*, *168*
zoned_time_precision, 246
zoned_time_set_zone *(*zoned-zone*)*, 244
zoned_time_zone *(*zoned-zone*)*, 244