Package ‘lubridate’

September 13, 2016

Maintainer Vitalie Spinu <spinuvit@gmail.com>
License GPL-2
Title Make Dealing with Dates a Little Easier
LazyData true
Type Package
Description Functions to work with date-times and time-spans: fast and user-friendly parsing of date-time data, extraction and updating of components of a date-time (years, months, days, hours, minutes, and seconds), algebraic manipulation on date-time and time-span objects. The 'lubridate' package has a consistent and memorable syntax that makes working with dates easy and fun.
Enhances chron, timeDate, zoo, xts, its, tis, timeSeries, fts, tseries
Version 1.6.0
Depends methods, R (>= 3.0.0)
Imports stringr
Suggests testthat, knitr, covr
BugReports https://github.com/hadley/lubridate/issues
VignetteBuilder knitr
Collate 'Dates.r' 'POSIXt.r' 'util.r' 'timespans.r' 'intervals.r'
'diftimes.r' 'durations.r' 'periods.r' 'accessors-date.R'
'accessors-day.r' 'accessors-dst.r' 'accessors-hour.r'
'accessors-minute.r' 'accessors-month.r' 'accessors-quarter.r'
'accessors-second.r' 'accessors-tz.r' 'accessors-week.r'
'accessors-year.r' 'am-pm.r' 'time-zones.r' 'numeric.r'
'coercion.r' 'constants.r' 'data.r' 'decimal-dates.r'
'deprecated.r' 'guess.r' 'help.r' 'hidden.r' 'ininstants.r'
'leap-years.r' 'ops-addition.r' 'ops-+%r' 'ops-compare.r'
'ops-division.r' 'ops-integer-division.r' 'ops-modulo.r'
'ops-multiplication.r' 'ops-subtraction.r' 'parse.r' 'pretty.r'
'round.r' 'stamp.r' 'update.r'
RoxygenNote 5.0.1
NeedsCompilation yes
Author Garrett Grolemund [aut], Vitalie Spinu [aut, cre], Hadley Wickham [aut], Ian Lyttle [ctb], Imanuel Constigan [ctb], Jason Law [ctb], Doug Mitarotonda [ctb], Joseph Larmarange [ctb], Jonathan Boiser [ctb], Chel Hee Lee [ctb]
Repository CRAN
Date/Publication 2016-09-13 13:11:52

R topics documented:
lubridate-package .................................................. 3
am ................................................................. 5
as.duration .......................................................... 6
as.interval ......................................................... 7
as.period .......................................................... 8
as_date .............................................................. 9
date ................................................................. 11
DateUpdate .......................................................... 12
date_decimal ........................................................ 13
day ................................................................. 13
days_in_month ...................................................... 15
decimal_date ....................................................... 15
Deprecated-lubridate ............................................... 16
dst ................................................................. 17
duration ............................................................. 17
Duration-class ...................................................... 19
fit_to_timeline .................................................... 19
force_tz ........................................................... 20
guess_formats .................................................... 21
hour ................................................................. 23
interval ............................................................. 24
Interval-class .................................................... 26
is.Date ............................................................ 27
is.difftime ......................................................... 28
is.instant ........................................................ 28
is.POSIXt .......................................................... 29
is.timespan ....................................................... 30
lakers ............................................................... 30
leap_year .......................................................... 31
make_datetime .................................................... 31
make_difftime .................................................... 32
Description

Lubridate provides tools that make it easier to parse and manipulate dates. These tools are grouped below by common purpose. More information about each function can be found in its help documentation.

Details

Parsing dates

Lubridate’s parsing functions read strings into R as POSIXct date-time objects. Users should choose the function whose name models the order in which the year (‘y’), month (‘m’) and day (‘d’) elements appear the string to be parsed: dmy, myd, ymd, ydm, dmy, ymd_hms). A very flexible and user friendly parser is provided by parse_date_time.
Lubridate can also parse partial dates from strings into Period-class objects with the functions hm, hms and ms.

Lubridate has an inbuilt very fast POSIX parser, ported from the fasttime package by Simon Urbanek. This functionality is as yet optional and could be activated with options(lubridate.fasttime = TRUE). Lubridate will automatically detect POSIX strings and use fast parser instead of the default strftime utility.

Manipulating dates

Lubridate distinguishes between moments in time (known as instants) and spans of time (known as time spans, see TimeSpan-class). Time spans are further separated into Duration-class, Period-class and Interval-class objects.

Instants

Instants are specific moments of time. Date, POSIXct, and POSIXlt are the three object classes Base R recognizes as instants. is.Date tests whether an object inherits from the Date class. is.POSIXt tests whether an object inherits from the POSIXlt or POSIXct classes. is.instant tests whether an object inherits from any of the three classes.

now returns the current system time as a POSIXct object. today returns the current system date. For convenience, 1970-01-01 00:00:00 is saved to origin. This is the instant from which POSIXct times are calculated. Try unclass(now()) to see the numeric structure that underlies POSIXct objects. Each POSIXct object is saved as the number of seconds it occurred after 1970-01-01 00:00:00.

Conceptually, instants are a combination of measurements on different units (i.e., years, months, days, etc.). The individual values for these units can be extracted from an instant and set with the accessor functions second, minute, hour, day, yday, mday, wday, week, month, year, tz, and dst. Note: the accessor functions are named after the singular form of an element. They shouldn’t be confused with the period helper functions that have the plural form of the units as a name (e.g., seconds).

Rounding dates

Instants can be rounded to a convenient unit using the functions ceiling_date, floor_date and round_date.

Time zones

Lubridate provides two helper functions for working with time zones. with_tz changes the time zone in which an instant is displayed. The clock time displayed for the instant changes, but the moment of time described remains the same. force_tz changes only the time zone element of an instant. The clock time displayed remains the same, but the resulting instant describes a new moment of time.

Timespans

A timespan is a length of time that may or may not be connected to a particular instant. For example, three months is a timespan. So is an hour and a half. Base R uses difftime class objects to record timespans. However, people are not always consistent in how they expect time to behave. Sometimes the passage of time is a monotone progression of instants that should be as mathematically reliable as the number line. On other occasions time must follow complex conventions and rules so that the clock times we see reflect what we expect to observe in terms of daylight, season, and congruence with the atomic clock. To better navigate the nuances of time, lubridate creates three additional timespan classes, each with its own specific and consistent behavior: Interval-class, Period-class and Duration-class.
is.difftime tests whether an object inherits from the difftime class. is.timespan tests whether an object inherits from any of the four timespan classes.

Durations

Durations measure the exact amount of time that occurs between two instants. This can create unexpected results in relation to clock times if a leap second, leap year, or change in daylight savings time (DST) occurs in the interval.

Functions for working with durations include is.duration, as.duration and duration. dseconds, dminutes, dhours, ddays, dweeks and dyeears convenient lengths.

Periods

Periods measure the change in clock time that occurs between two instants. Periods provide robust predictions of clock time in the presence of leap seconds, leap years, and changes in DST.

Functions for working with periods include is.period, as.period and period. seconds, minutes, hours, days, weeks, months and years quickly create periods of convenient lengths.

Intervals

Intervals are timespans that begin at a specific instant and end at a specific instant. Intervals retain complete information about a timespan. They provide the only reliable way to convert between periods and durations.

Functions for working with intervals include is.interval, as.interval, interval, int_shift, int_flip, int_aligns, int_overlaps, and %within%. Intervals can also be manipulated with intersect, union, and setdiff().

Miscellaneous

decimal_date converts an instant to a decimal of its year. leap_year tests whether an instant occurs during a leap year. pretty_dates provides a method of making pretty breaks for date-times lakers is a data set that contains information about the Los Angeles Lakers 2008-2009 basketball season.

References


Description

Does date time occur in the am or pm?

Usage

am(x)

pm(x)
as.duration

Arguments

x a date-time object

Value

TRUE or FALSE depending on whether x occurs in the am or pm

Examples

x <- ymd("2012-03-26")
  am(x)
  pm(x)

Description

as.duration changes Interval, Period and numeric class objects to Duration objects. Numeric objects are changed to Duration objects with the seconds unit equal to the numeric value.

Usage

as.duration(x, ...)

Arguments

x Object to be coerced to a duration

... Parameters passed to other methods. Currently unused.

Details

Durations are exact time measurements, whereas periods are relative time measurements. See Period-class. The length of a period depends on when it occurs. Hence, a one to one mapping does not exist between durations and periods. When used with a period object, as.duration provides an inexact estimate of the length of the period; each time unit is assigned its most common number of seconds. A period of one month is converted to 2628000 seconds (approximately 30.42 days). This ensures that 12 months will sum to 365 days, or one normal year. For an exact transformation, first transform the period to an interval with as.interval.

Value

A duration object

See Also

Duration-class, duration
as.interval

Examples

```r
span <- interval(ymd("2009-01-01"), ymd("2009-08-01")) # interval
as.duration(span)
# numeric
dur <- duration(hours = 10, minutes = 6)
as.numeric(dur, "hours")
# numeric
as.numeric(dur, "minutes")
```

as.interval(x, start, ...)

Arguments

- `x`: a duration, difftime, period, or numeric object that describes the length of the interval
- `start`: a POSIXt or Date object that describes when the interval begins
- `...`: additional arguments to pass to as.interval

Description

as.interval changes difftime, Duration, Period and numeric class objects to intervals that begin at the specified date-time. Numeric objects are first coerced to timespans equal to the numeric value in seconds.

Usage

as.interval(x, start, ...)

Details

as.interval can be used to create accurate transformations between Period objects, which measure time spans in variable length units, and Duration objects, which measure timespans as an exact number of seconds. A start date-time must be supplied to make the conversion. Lubridate uses this start date to look up how many seconds each variable length unit (e.g. month, year) lasted for during the time span described. See as.duration, as.period.

Value

an interval object

See Also

interval
Examples

diff <- make_difftime(days = 31) # difftime
as.interval(diff, ymd("2009-01-01"))
as.interval(diff, ymd("2009-02-01"))

dur <- duration(days = 31) # duration
as.interval(dur, ymd("2009-01-01"))
as.interval(dur, ymd("2009-02-01"))

per <- period(months = 1) # period
as.interval(per, ymd("2009-01-01"))
as.interval(per, ymd("2009-02-01"))
as.interval(3600, ymd("2009-01-01")) # numeric

as.period

Change an object to a period.

Description

as.period changes Interval, Duration, difftime and numeric class objects to Period class objects with the specified units.

Usage

as.period(x, unit, ...)

Arguments

x
an interval, difftime, or numeric object

unit
A character string that specifies which time units to build period in. unit is only implemented for the as.period.numeric method and the as.period.interval method. For as.period.interval, as.period will convert intervals to units no larger than the specified unit.

... additional arguments to pass to as.period

Details

Users must specify which time units to measure the period in. The exact length of each time unit in a period will depend on when it occurs. See Period-class and period. The choice of units is not trivial; units that are normally equal may differ in length depending on when the time period occurs. For example, when a leap second occurs one minute is longer than 60 seconds.

Because periods do not have a fixed length, they can not be accurately converted to and from Duration objects. Duration objects measure time spans in exact numbers of seconds, see Duration-class. Hence, a one to one mapping does not exist between durations and periods. When used with a Duration object, as.period provides an inexact estimate; the duration is broken into time units based
on the most common lengths of time units, in seconds. Because the length of months are particularly variable, a period with a months unit can not be coerced from a duration object. For an exact transformation, first transform the duration to an interval with `as.interval`.

Coercing an interval to a period may cause surprising behavior if you request periods with small units. A leap year is 366 days long, but one year long. Such an interval will convert to 366 days when unit is set to days and 1 year when unit is set to years. Adding 366 days to a date will often give a different result than adding one year. Daylight savings is the one exception where this does not apply. Interval lengths are calculated on the UTC timeline, which does not use daylight savings. Hence, periods converted with seconds or minutes will not reflect the actual variation in seconds and minutes that occurs due to daylight savings. These periods will show the "naive" change in seconds and minutes that is suggested by the differences in clock time. See the examples below.

### Value

- a period object

### See Also

- `Period-class.period`

### Examples

```r
span <- interval(as.POSIXct("2009-01-01"), as.POSIXct("2010-02-02 01:01:01")) # interval
as.period(span)
"397d 1H 1M 1S"

leap <- interval(ymd("2016-01-01"), ymd("2017-01-01"))
as.period(leap, unit = "days")

leap <- interval(ymd("2016-01-01"), ymd("2017-01-01"))
as.period(leap, unit = "years")

dst <- interval(ymd("2016-11-06", tz = "America/Chicago"),
ymd("2016-11-07", tz = "America/Chicago"))
# as.period(dst, unit = "seconds")

per <- period(hours = 10, minutes = 6)
as.numeric(per, "hours")
as.numeric(per, "minutes")
```

---

**as_date**

Convert an object to a date or date-time

### Description

Convert an object to a date or date-time
Usage

```r
as_date(x, ...)

as_datetime(x, ...)
```

## S4 method for signature 'POSIXt'
```
as_date(x, tz = NULL)
```

## S4 method for signature 'numeric'
```
as_date(x, origin = lubridate::origin)
```

## S4 method for signature 'POSIXt'
```
as_datetime(x, tz = "UTC")
```

## S4 method for signature 'numeric'
```
as_datetime(x, origin = lubridate::origin, tz = "UTC")
```

## S4 method for signature 'ANY'
```
as_datetime(x, tz = "UTC")
```

Arguments

- `x` a vector of `POSIXt`, numeric or character objects
- `...` further arguments to be passed to specific methods (see above).
- `tz` a time zone name (default: time zone of the POSIXt object `x`). See `OlsonNames`.
- `origin` a Date object, or something which can be coerced by `as.Date(origin, ...) to such an object (default: the Unix epoch of "1970-01-01"). Note that in this instance, `x` is assumed to reflect the number of days since `origin` at "UTC".

Value

a vector of `Date` objects corresponding to `x`.

Compare to base R

These are drop in replacements for `as.Date` and `as.POSIXct`, with a few tweaks to make them work more intuitively.

- `as_date` ignores the timezone attribute, resulting in a more intuitive conversion (see examples)
- Both functions provide a default origin argument for numeric vectors.
- `as_datetime` defaults to using UTC.

Examples

```r
dt_utc <- ymd_hms("2010-08-03 00:50:50")
dt_europe <- ymd_hms("2010-08-03 00:50:50", tz="Europe/London")
c(as_date(dt_utc), as.Date(dt_utc))
c(as_date(dt_europe), as.Date(dt_europe))
```
get date

## need not suply origin
as_date(18)

date

Get/set Date component of a date-time.

**Description**

Date-time must be a POSIXct, POSIXlt, Date, chron, yearmon, yearqtr, zoo, zooReg, timeDate, xts, its, ti, jul, timeSeries, and fts objects.

**Usage**

```r
date(x)
```

```r
date(x) <- value
```

**Arguments**

- `x` a date-time object
- `value` an object for which the `date()` function is defined

**Details**

date does not yet support years before 0 C.E. Also date is not defined for Period objects.

**Value**

the date of `x` as a Date

**Examples**

```r
x <- as.POSIXct("2012-03-26 23:12:13", tz = "Etc/GMT+8")
date(x)
as.Date(x) # by default as.Date assumes you want to know the date in UTC
as.Date(x, tz = "Etc/GMT+8")
date(x) <- as.Date("2000-01-02")
x
```
Changes the components of a date object

**Description**

update.Date and update.POSIXt return a date with the specified elements updated. Elements not specified will be left unaltered. update.Date and update.POSIXt do not add the specified values to the existing date, they substitute them for the appropriate parts of the existing date.

**Usage**

```r
## S3 method for class 'POSIXt'
update(object, ..., simple = FALSE)
```

**Arguments**

- `object`: a date-time object
- `...`: named arguments: years, months, ydays, wdays, mdays, days, hours, minutes, seconds, tzs (time zone component)
- `simple`: logical, passed to `fit_to_timeline`. If TRUE a simple fit to time line is performed and no NA are produced for invalid dates. Invalid dates are converted to meaningful dates by extrapolating the timezones.

**Value**

a date object with the requested elements updated. The object will retain its original class unless an element is updated which the original class does not support. In this case, the date returned will be a POSIXlt date object.

**Examples**

```r
date <- as.POSIXlt("2009-02-10")
update(date, year = 2010, month = 1, mday = 1)

update(date, year =2010, month = 13, mday = 1)

update(date, minute = 10, second = 3)
```
**date_decimal**

Converts a decimal to a date.

**Description**

Converts a decimal to a date.

**Usage**

```
date_decimal(decimal, tz = "UTC")
```

**Arguments**

- `decimal`: a numeric object
- `tz`: the time zone required

**Value**

a POSIXct object, whose year corresponds to the integer part of decimal. The months, days, hours, minutes and seconds elements are picked so the date-time will accurately represent the fraction of the year expressed by decimal.

**Examples**

```
date <- ymd("2009-02-10")
decimal <- decimal_date(date)  # 2009.11
date_decimal(decimal)  # "2009-02-10 UTC"
```

**day**

Get/set days component of a date-time.

**Description**

Get/set days component of a date-time.

**Usage**

```
day(x)
mday(x)
```

```
wday(x, label = FALSE, abbr = TRUE)
```

```
qday(x)
yday(x)
```
\begin{verbatim}
day(x) <- value
mday(x) <- value
qday(x) <- value
wday(x) <- value
yday(x) <- value

Arguments

x a POSIXct, POSIXlt, Date, chron, yearmon, yearqtr, zoo, zooreg, timeDate, xts,
its, ti, jul, timeSeries, or its object.

label logical. Only available for wday. TRUE will display the day of the week as an
ordered factor of character strings, such as "Sunday." FALSE will display the
day of the week as a number.

abbr logical. Only available for wday. FALSE will display the day of the week as
an ordered factor of character strings, such as "Sunday." TRUE will display an
abbreviated version of the label, such as "Sun". abbr is disregarded if label = FALSE.

value a numeric object

Details

day and day<- are aliases for mday and mday<- respectively.

Value

wday returns the day of the week as a decimal number (01-07, Sunday is 1) or an ordered factor
(Sunday is first).

See Also

yday, mday

Examples

x <- as.Date("2009-09-02")
wday(x) #4

wday(ymd(880101))
wday(ymd(880101), label = TRUE, abbr = FALSE)
# Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < Friday < Saturday
wday(ymd(880101), label = TRUE, abbr = TRUE)
# Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < Friday < Saturday
wday(ymd(880101) + days(-2:4), label = TRUE, abbr = TRUE)
# Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < Friday < Saturday
\end{verbatim}
days_in_month

```r
x <- as.Date("2009-09-02")
yday(x) # 245
mday(x) # 2
yday(x) <- 1  # "2009-01-01"
yday(x) <- 366 # "2010-01-01"
mday(x) > 3
```

Get the number of days in the month of a date-time.

Description

Date-time must be a POSIXct, POSIXlt, Date, chron, yearmon, yearqtr, zoo, zooreg, timeDate, xts, its, ti, jul, timeSeries, and fts objects.

Usage

```r
days_in_month(x)
```

Arguments

- `x`: a date-time object

Value

An integer of the number of days in the month component of the date-time object.

decimal_date

Converts a date to a decimal of its year.

Description

Converts a date to a decimal of its year.

Usage

```r
decimal_date(date)
```

Arguments

- `date`: a POSIXt or Date object

Value

- a numeric object where the date is expressed as a fraction of its year

Examples

```r
date <- ymd("2009-02-10")
decimal_date(date)  # 2009.11
```
Description

Deprecated function in lubridate package

Usage

new_period(...)  
new_interval(...)  
new_duration(...)  
new_difftime(...)  
eseconds(x = 1)  
eminutes(x = 1)  
ehours(x = 1)  
edays(x = 1)  
eweeks(x = 1)  
eyears(x = 1)  
emilliseconds(x = 1)  
emicroseconds(x = 1)  
enanoseconds(x = 1)  
epicoseconds(x = 1)  
here()  
olson_time_zones(order_by = c("name", "longitude"))

Arguments

... arguments to be passed to the functions (obscured to enforce the usage of new functions)  
x numeric value to be converted into duration  
order_by Return names alphabetically (the default) or from West to East.
**dst**

Get Daylight Savings Time indicator of a date-time.

**Description**

Date-time must be a POSIXct, POSIXlt, Date, chron, yearmon, yearqtr, zoo, zooreg, timeDate, xts, its, ti, jul, timeSeries, and fts objects.

**Usage**

dst(x)

**Arguments**

a date-time object

**Details**

A date-time’s daylight savings flag can not be set because it depends on the date-time’s year, month, day, and hour values.

**Value**

A logical. TRUE if DST is in force, FALSE if not, NA if unknown.

**Examples**

```r
x <- ymd("2012-03-26")
dst(x)
```

---

**duration**

Create a duration object.

**Description**

duration creates a duration object with the specified values. Entries for different units are cumulative. Durations display as the number of seconds in a time span. When this number is large, durations also display an estimate in larger units.; however, the underlying object is always recorded as a fixed number of seconds. For display and creation purposes, units are converted to seconds using their most common lengths in seconds. Minutes = 60 seconds, hours = 3600 seconds, days = 86400 seconds, weeks = 604800. Units larger than weeks are not used due to their variability.

**Usage**

duration(num = NULL, units = "seconds", ...)

```r
is.duration(x)
```
Arguments

- **num**: the number of time units to include in the duration. From v1.6.0, `num` can also be a character vector that specifies durations in a convenient shorthand format. All unambiguous name units and abbreviations are supported. See examples.
- **units**: a character string that specifies the type of units that `num` refers to. When `num` is character, this argument is ignored.
- **...**: a list of time units to be included in the duration and their amounts. Seconds, minutes, hours, days, and weeks are supported.
- **x**: an R object

Details

Durations record the exact number of seconds in a time span. They measure the exact passage of time but do not always align with measurements made in larger units of time such as hours, months and years. This is because the length of larger time units can be affected by conventions such as leap years and Daylight Savings Time. Base R provides a second class for measuring durations, the `difftime` class.

Duration objects can be easily created with the helper functions `dweeks`, `ddays`, `dminutes`, `dseconds`. These objects can be added to and subtracted to dates to create a user interface similar to object oriented programming.

Value

- a duration object

See Also

- `as.duration`

Examples

```
duration(day = -1)  # -86400s (~1 days)
duration(90, "seconds")
duration(1.5, "minutes")
duration(-1, "days")  # -86400s (~1 days)
duration(second = 90)
duration(minute = 1.5)
duration(min = 1.5)
duration(second = 3, minute = 1.5, hour = 2, day = 6, week = 1)
duration(hour = 1, minute = -60)
duration("2M 1sec")
duration("2hours 2minutes 1second")
duration("2d 2H 2M 2s")
duration("2days 2hours 2mins 2secs")
# Missing numerals default to 1. Repeated units are added up.
duration("day day")
# Comparison with characters is supported from v1.6.0.
```
**Duration-class**

```r
duration("day 2 sec") > "day 1sec"
is.duration(as.Date("2009-08-03")) # FALSE
is.duration(duration(days = 12.4)) # TRUE
```

---

**Description**

Duration is an S4 class that extends the `Timespan-class` class. Durations record the exact number of seconds in a time span. They measure the exact passage of time but do not always align with measurements made in larger units of time such as hours, months and years. This is because the exact length of larger time units can be affected by conventions such as leap years and Daylight Savings Time.

**Details**

Durations provide a method for measuring generalized timespans when we wish to treat time as a mathematical quantity that increases in a uniform, monotone manner along a continuous number-line. They allow exact comparisons with other durations. See `Period-class` for an alternative way to measure timespans that better preserves clock times.

Durations class objects have one slot: `.Data`, a numeric object equal to the number of seconds in the duration.

---

**fit_to_timeline**  
*Fit a POSIXlt date-time to the timeline*

---

**Description**

The POSIXlt format allows you to create instants that do not exist in real life due to daylight savings time and other conventions. `fit_to_timeline` matches POSIXlt date-times to a real times. If an instant does not exist, fit to timeline will replace it with an NA. If an instant does exist, but has been paired with an incorrect timezone/daylight savings time combination, `fit_to_timeline` returns the instant with the correct combination.

**Usage**

```r
fit_to_timeline(lt, class = "POSIXct", simple = FALSE)
```

**Arguments**

- `lt`  
a POSIXlt date-time object.
- `class`  
a character string that describes what type of object to return, POSIXlt or POSIXct. Defaults to POSIXct. This is an optimization to avoid needless conversions.
- `simple`  
if TRUE, lubridate makes no attempt to detect meaningless time-dates or to correct time zones. No NAs are produced and the most meaningful valid dates are returned instead. See examples.
Value

a POSIXct or POSIXlt object that contains no illusory date-times

Examples

```r
# Not run:
tricky <- structure(list(sec = c(5, 0, 0, -1),
                          min = c(0L, 5L, 5L, 0L),
                          hour = c(2L, 0L, 2L, 2L),
                          mday = c(4L, 4L, 14L, 4L),
                          mon = c(10L, 10L, 2L, 10L),
                          year = c(112L, 112L, 110L, 112L),
                          wday = c(0L, 0L, 0L, 0L),
                          yday = c(308L, 308L, 72L, 308L),
                          isdst = c(1L, 0L, 0L, 1L)),
                         .Names = c("sec", "min", "hour", "mday", "mon",
                                    "year", "wday", "yday", "isdst"),
                         class = c("POSIXlt", "POSIXt"),
                         tzone = c("America/Chicago", "CST", "CDT"))
tricky
# because clocks "fall back" to 1:00 CST

# CDT, not CST at this instant

# because clocks "spring forward" past this time
# for daylight savings

# has deceptive internal structure

fit_to_timeline(tricky)
[1] "2012-11-04 02:00:05 CST" "2012-11-04 00:05:00 CDT"

# with correct timezone & DST combination

# with correct timezone & DST combination

fit_to_timeline(tricky, simple = TRUE)
# Reduce to valid time-dates by extrapolating CDT and CST zones

# End(Not run)
```

Description

force_tz returns a the date-time that has the same clock time as x in the new time zone. Although the new date-time has the same clock time (e.g. the same values in the year, month, days, etc.
elements) it is a different moment of time than the input date-time. force_tz defaults to the Universal Coordinated time zone (UTC) when an unrecognized time zone is inputted. See Sys.timezone for more information on how R recognizes time zones.

Usage

force_tz(time, tzone = "")

Arguments

time a POSIXct, POSIXlt, Date, chron date-time object, or a data.frame object. When a data.frame all POSIXt elements of a data.frame are processed with force_tz and new data.frame is returned.

tzone a character string containing the time zone to convert to. R must recognize the name contained in the string as a time zone on your system.

Value

a POSIXct object in the updated time zone

See Also

with_tz

Examples

x <- as.POSIXct("2009-08-07 00:00:01", tz = "America/New_York")
force_tz(x, "GMT")

guess_formats

Guess formats from the supplied date-time character vector.

Description

Guess formats from the supplied date-time character vector.

Usage

guess_formats(x, orders, locale = Sys.getlocale("LC_TIME"),
preproc_wday = TRUE, print_matches = FALSE)

Arguments

x input vector of date-times
orders format orders to look for. See examples.
locale locale to use, default to the current locale
preproc_wday whether to preprocess week days names. Internal optimization used by ymd_hms family of functions. If true week days are substituted with this format explicitly.
print_matches for development purpose mainly. If TRUE prints a matrix of matched templates.
Value

a vector of matched formats

Examples

\[
x <- c('February 20th 1973',
  "February 14, 2004",
  "Sunday, May 1, 2000",
  "Sunday, May 1, 2000",
  "February 14, 04",
  'Feb 20th 73',
  "January 5 1999 at 7pm",
  "Jan 3 2010",
  "Jan 1, 1999",
  "Jan 3 10",
  "01 3 2010",
  '1 3 10',
  '1 13 89',
  "5/27/1979",
  "12/31/99",
  "DOB:12/11/00",
  "--------",
  'Thu, 1 July 2004 22:30:00',
  'Thu, 1st of July 2004 at 22:30:00',
  'Thu, 1July 2004 at 22:30:00',
  'Thu, 1July04 22:30:00',
  'Thu, 1July04 22:30:00',
  '21 Aug 2011, 11:15:34 pm',
  "--------",
  "1979-05-27 05:00:59",
  "1979-05-27",
  "--------",
  "3 jan 2000",
  "17 april 85",
  "27/5/1979",
  "20 01 89",
  "00/13/10",
  "--------",
  "14 12 00",
  "03:23:22 pm")
\]

guess_formats(x, "BdY")
guess_formats(x, "Bdy")
## m also matches b and B; y also matches Y
guess_formats(x, "mdY", print_matches = TRUE)

## T also matches IMSp order
guess_formats(x, "T", print_matches = TRUE)

## b and B are equivalent and match, both, abbreviated and full names
guess_formats(x, c("mdY", "BdY", "Bdy", "bdY", "bdy"), print_matches = TRUE)
hour

Get/set hours component of a date-time.

Description
Date-time must be a POSIXct, POSIXlt, Date, Period, chron, yearmon, yearqtr, zoo, zooreg, timeDate, xts, its, ti, jul, timeSeries, and fts objects.

Usage
hour(x)

hour(x) <- value

Arguments

x a date-time object
value numeric value to be assigned to hour component

Value
the hours element of x as a decimal number

Examples
x <- ymd("2012-03-26")
hour(x)
hour(x) <- 1
hour(x) <- 25
hour(x) > 2
interval  

Utilities for creation and manipulation of Interval objects.

Description

interval creates an Interval-class object with the specified start and end dates. If the start date occurs before the end date, the interval will be positive. Otherwise, it will be negative.

%--% creates an interval that covers the range spanned by two dates. It replaces the original behavior of lubridate, which created an interval by default whenever two date-times were subtracted.

int_start and int_start<- are accessors the start date of an interval. Note that changing the start date of an interval will change the length of the interval, since the end date will remain the same.

int_flip reverses the order of the start date and end date in an interval. The new interval takes place during the same timespan as the original interval, but has the opposite direction.

int_shift shifts the start and end dates of an interval up or down the timeline by a specified amount. Note that this may change the exact length of the interval if the interval is shifted by a Period object. Intervals shifted by a Duration or difftime object will retain their exact length in seconds.

int_overlaps tests if two intervals overlap.

int_standardize ensures all intervals in an interval object are positive. If an interval is not positive, flip it so that it retains its endpoints but becomes positive.

int_aligns tests if two intervals share an endpoint. The direction of each interval is ignored.

int_align tests whether the earliest or latest moments of each interval occur at the same time.

int_diff returns the intervals that occur between the elements of a vector of date-times. int_diff is similar to the POSIXt and Date methods of diff, but returns an Interval object instead of a difftime object.

Usage

interval(start, end, tzone = attr(start, "tzone"))

start %--% end

is.interval(x)

int_start(int)

int_start(int) <- value

int_end(int)

int_end(int) <- value

int_length(int)
interval

int_flip(int)
int_shift(int, by)
int_overlaps(int1, int2)
int_standardize(int)
int_aligns(int1, int2)
int_diff(times)

Arguments

start a POSIXt or Date date-time object
end a POSIXt or Date date-time object
tzone a recognized timezone to display the interval in
x an R object
int an interval object
value interval's start/end to be assigned to int
by A period or duration object to shift by (for int_shift)
int1 an Interval object (for int_overlaps, int_aligns)
int2 an Interval object (for int_overlaps, int_aligns)
times A vector of POSIXct, POSIXlt or Date class date-times (for int_diff)

Details

Intervals are time spans bound by two real date-times. Intervals can be accurately converted to either period or duration objects using as_period, as_duration. Since an interval is anchored to a fixed history of time, both the exact number of seconds that passed and the number of variable length time units that occurred during the interval can be calculated.

Value

interval - Interval object.
int_start and int_end return a POSIXct date object when used as an accessor. Nothing when used as a setter.
int_length - numeric length of the interval in seconds. A negative number connotes a negative interval.
int_flip - flipped interval object
int_shift - interval object
int_overlaps logical, TRUE if int1 and int2 overlap by at least one second. FALSE otherwise
int_align logical, TRUE if int1 and int2 begin or end on the same moment. FALSE otherwise
int_diff - interval object that contains the n-1 intervals between the n date-time in times
See Also

`Interval-class, as.interval, %within%`

Examples

```r
interval(ymd(20090201), ymd(20090101))

date1 <- as.POSIXct("2009-03-08 01:59:59")
date2 <- as.POSIXct("2000-02-29 12:00:00")
interval(date2, date1)
interval(date1, date2)
span <- interval(ymd(20090101), ymd(20090201))

is.interval(period(months= 1, days = 15)) # FALSE
is.interval(interval(ymd(20090801), ymd(20090809))) # TRUE
int <- interval(ymd("2001-01-01"), ymd("2002-01-01"))
int_start(int)
int_start(int) <- ymd("2001-06-01")
int

int <- interval(ymd("2001-01-01"), ymd("2002-01-01"))
int_end(int)
int_end(int) <- ymd("2002-06-01")
int
int <- interval(ymd("2001-01-01"), ymd("2002-01-01"))
int_length(int)
int <- interval(ymd("2001-01-01"), ymd("2002-01-01"))
int_flip(int)
int <- interval(ymd("2001-01-01"), ymd("2002-01-01"))
int_shift(int, duration(days = 11))
int_shift(int, duration(hours = -1))
int1 <- interval(ymd("2001-01-01"), ymd("2002-01-01"))
int2 <- interval(ymd("2001-06-01"), ymd("2002-06-01"))
int3 <- interval(ymd("2003-01-01"), ymd("2004-01-01"))

int_overlaps(int1, int2) # TRUE
int_overlaps(int1, int3) # FALSE
int <- interval(ymd("2002-01-01"), ymd("2001-01-01"))
int_standardize(int)
int1 <- interval(ymd("2001-01-01"), ymd("2002-01-01"))
int2 <- interval(ymd("2001-06-01"), ymd("2002-01-01"))
int3 <- interval(ymd("2003-01-01"), ymd("2004-01-01"))

int_aligns(int1, int2) # TRUE
int_aligns(int1, int3) # FALSE
dates <- now() + days(1:10)
int_diff(dates)
```
Description

Interval is an S4 class that extends the `TimeSpan-class` class. An Interval object records one or more spans of time. Intervals record these timespans as a sequence of seconds that begin at a specified date. Since intervals are anchored to a precise moment of time, they can accurately be converted to `Period-class` or `Duration-class` class objects. This is because we can observe the length in seconds of each period that begins on a specific date. Contrast this to a generalized period, which may not have a consistent length in seconds (e.g., the number of seconds in a year will change if it is a leap year).

Details

Intervals can be both negative and positive. Negative intervals progress backwards from the start date; positive intervals progress forwards.

Interval class objects have two slots: `.Data`, a numeric object equal to the number of seconds in the interval; and `start`, a POSIXct object that specifies the time when the interval starts.

---

**is.Date**

*Is x a Date object?*

**Description**

Is x a Date object?

**Usage**

```r
is.Date(x)
```

**Arguments**

- `x` an R object

**Value**

TRUE if x is a Date object, FALSE otherwise.

**See Also**

`is.instant`, `is.timespan`, `is.POSIXt`

**Examples**

```r
is.Date(as.Date("2009-08-03")) # TRUE
is.Date(difftime(now(), 5, now())) # FALSE
```
is.difftime  

Is x a difftime object?

Description

Is x a difftime object?

Usage

is.difftime(x)

Arguments

x  an R object

Value

TRUE if x is a difftime object, FALSE otherwise.

See Also

is.instant, is.timespan, is.interval, is.period.

Examples

is.difftime(as.Date("2009-08-03")) # FALSE
is.difftime(make_difftime(days = 12.4)) # TRUE

is.instant  

Is x a date-time object?

Description

An instant is a specific moment in time. Most common date-time objects (e.g. POSIXct, POSIXlt, and Date objects) are instants.

Usage

is.instant(x)

is.timepoint(x)

Arguments

x  an R object
is.POSIXt

Value

TRUE if x is a POSIXct, POSIXlt, or Date object, FALSE otherwise.

See Also

is.timespan, is.POSIXt, is.Date

Examples

is.instant(as.Date("2009-08-03")) # TRUE
is.timepoint(5) # FALSE

is.POSIXt

Is x a POSIXct or POSIXlt object?

Description

Is x a POSIXct or POSIXlt object?

Usage

is.POSIXt(x)

is.POSIXlt(x)

is.POSIXct(x)

Arguments

x an R object

Value

TRUE if x is a POSIXct or POSIXlt object, FALSE otherwise.

See Also

is.instant, is.timespan, is.Date

Examples

is.POSIXt(as.Date("2009-08-03")) # FALSE
is.POSIXt(as.POSIXct("2009-08-03")) # TRUE
is.timespan  
*Is x a length of time?*

**Description**

Is x a length of time?

**Usage**

`is.timespan(x)`

**Arguments**

`x`  
an R object

**Value**

TRUE if x is a period, interval, duration, or difftime object, FALSE otherwise.

**See Also**

`is.instant`, `is.duration`, `is.difftime`, `is.period`, `is.interval`

**Examples**

```r
is.timespan(as.Date("2009-08-03")) # FALSE
is.timespan(duration(second = 1)) # TRUE
```

---

**lakers**  
*Lakers 2008-2009 basketball data set*

**Description**

This data set contains play by play statistics of each Los Angeles Lakers basketball game in the 2008-2009 season. Data includes the date, opponent, and type of each game (home or away). Each play is described by the time on the game clock when the play was made, the period in which the play was attempted, the type of play, the player and team who made the play, the result of the play, and the location on the court where each play was made.

**References**

leap_year

Is a year a leap year?

Description

If x is a recognized date-time object, leap_year will return whether x occurs during a leap year. If x is a number, leap_year returns whether it would be a leap year under the Gregorian calendar.

Usage

leap_year(date)

Arguments

date a date-time object or a year

Value

TRUE if x is a leap year, FALSE otherwise

Examples

x <- as.Date("2009-08-02")
leap_year(x) # FALSE
leap_year(2009) # FALSE
leap_year(2008) # TRUE
leap_year(1900) # FALSE
leap_year(2000) # TRUE

make_datetime

Efficient creation of date-times from numeric representations

Description

make_datetime is a very fast drop-in replacement for base::ISOdate and base::ISOdatetime. make_date produces objects of class Date.

Usage

make_datetime(year = 1970L, month = 1L, day = 1L, hour = 0L, min = 0L, sec = 0, tz = "UTC")

make_date(year = 1970L, month = 1L, day = 1L)
Arguments

- **year**: numeric year
- **month**: numeric month
- **day**: numeric day
- **hour**: numeric hour
- **min**: numeric minute
- **sec**: numeric second
- **tz**: time zone. Defaults to UTC.

Details

Input vectors are silently recycled. All inputs except `sec` are silently converted to integer vectors; `sec` can be either integer or double.

Examples

```r
make_datetime(year = 1999, month = 12, day = 22, sec = 10)
made_datetime(year = 1999, month = 12, day = 22, sec = c(10, 11))
```

Description

`make_difftime` creates a difftime object with the specified number of units. Entries for different units are cumulative. `difftime` displays durations in various units, but these units are estimates given for convenience. The underlying object is always recorded as a fixed number of seconds.

Usage

```r
make_difftime(num = NULL, units = "auto", ...)
```

Arguments

- **num**: Optional number of seconds
- **units**: a character vector that lists the type of units to use for the display of the return value (see examples). If `units` is "auto" (the default) the display units are computed automatically. This might create undesirable effects when converting `difftime` objects to numeric values in data processing.
- **...**: a list of time units to be included in the `difftime` and their amounts. Seconds, minutes, hours, days, and weeks are supported. Normally only one of `num` or `...` are present. If both are present, the `difftime` objects are concatenated.
Details

Conceptually, `difftime` objects are a type of duration. They measure the exact passage of time but do not always align with measurements made in larger units of time such as hours, months and years. This is because the length of larger time units can be affected by conventions such as leap years and Daylight Savings Time. `lubridate` provides a second class for measuring durations, the `Duration` class.

Value

A `difftime` object

See Also

duration, as.duration

Examples

```r
make_difftime(1)
make_difftime(60)
make_difftime(3600)
make_difftime(3600, units = "minute")
# Time difference of 60 mins
make_difftime(second = 90)
# Time difference of 1.5 mins
make_difftime(minute = 1.5)
# Time difference of 1.5 mins
make_difftime(second = 3, minute = 1.5, hour = 2, day = 6, week = 1)
# Time difference of 13.08441 days
make_difftime(hour = 1, minute = -60)
# Time difference of 0 secs
make_difftime(day = -1)
# Time difference of -1 days
make_difftime(120, day = -1, units = "minute")
# Time differences in mins
```

---

**minute**

*Get/set minutes component of a date-time.*

Description

Date-time must be a POSIXct, POSIXlt, Date, Period, chron, yearmon, yearqtr, zoo, zooreg, timeDate, xts, its, ti, jul, timeSeries, and fts objects.

Usage

```r
minute(x)
minute(x) <- value
```
Arguments

x a date-time object
value numeric value to be assigned

Value

the minutes element of x as a decimal number

Examples

x <- ymd("2012-03-26")
minute(x)
minute(x) <- 1
minute(x) <- 61
minute(x) > 2

month Get/set months component of a date-time.

Description

Date-time must be a POSIXct, POSIXlt, Date, Period, chron, yearmon, yearqtr, zoo, zooreg, timeSeries, and its objects.

Usage

month(x, label = FALSE, abbr = TRUE)
month(x) <- value

Arguments

x a date-time object
label logical. TRUE will display the month as a character string such as "January." FALSE will display the month as a number.
abbr logical. FALSE will display the month as a character string label, such as "January". TRUE will display an abbreviated version of the label, such as "Jan". abbr is disregarded if label = FALSE.
value a numeric object

Value

the months element of x as a number (1-12) or character string. 1 = January.
Examples

```r
x <- ymd("2012-03-26")
month(x)
month(x) <- 1
month(x) <- 13
month(x) > 3

month(ymd(08:01:01))
month(ymd(08:01:01), label = TRUE)
month(ymd(08:01:01), label = TRUE, abbr = FALSE)
month(ymd(08:01:01) + months(0:11), label = TRUE)
```

---

**ms**

Create a period with the specified hours, minutes, and seconds

**Description**

Transforms a character or numeric vector into a period object with the specified number of hours, minutes, and seconds. `hms()` recognizes all non-numeric characters except ` '-' ` as separators (` '-' ` is used for negative durations). After hours, minutes and seconds have been parsed, the remaining input is ignored.

**Usage**

```r
ms(..., quiet = FALSE, roll = FALSE)
hm(..., quiet = FALSE, roll = FALSE)
hms(..., quiet = FALSE, roll = FALSE)
```

**Arguments**

- `...` a character vector of hour minute second triples
- `quiet` logical. When TRUE function evaluates without displaying customary messages.
- `roll` logical. When TRUE smaller units are rolled over to higher units if they exceed the conventional limit. For example `hms("01:59:120", roll=TRUE)` produces period "2H 1M 0S".

**Value**

a vector of period objects

**See Also**

hm, ms
Examples

```r
ms(c("09:10", "09:02", "1:10"))
ms("7 6")
ms("6.5")
hm(c("09:10", "09:02", "1:10"))
hm("7 6")
hm("6.5")

x <- c("09:10:01", "09:10:02", "09:10:03")
hms(x)

hms("7 6 5", "3:23::2", "2 : 23 : 33", "Finished in 9 hours, 20 min and 4 seconds")
```

Description

The current time

Usage

```r
now(tzone = "")
```

Arguments

tzone a character vector specifying which time zone you would like the current time in. tzone defaults to your computer’s system timezone. You can retrieve the current time in the Universal Coordinated Time (UTC) with `now("UTC")`.

Value

the current date and time as a POSIXct object

See Also

here

Examples

```r
now()
now("GMT")
now("")
now() == now() # would be true if computer processed both at the same instant
now() < now() # TRUE
now() > now() # FALSE
```
Description
Origin is the date-time for 1970-01-01 UTC in POSIXct format. This date-time is the origin for the numbering system used by POSIXct, POSIXlt, chron, and Date classes.

Usage
origin

Format
An object of class POSIXct (inherits from POSIXt) of length 1.

Examples
origin

parse_date_time
Parse character and numeric date-time vectors with user friendly order formats.

Description
parse_date_time parses an input vector into POSIXct date-time object. It differs from strptime in two respects. First, it allows specification of the order in which the formats occur without the need to include separators and "%" prefix. Such a formatting argument is referred to as "order". Second, it allows the user to specify several format-orders to handle heterogeneous date-time character representations.
parse_date_time2 is a fast C parser of numeric orders.
fast_strptime is a fast C parser of numeric formats only that accepts explicit format arguments, just as strptime.

Usage
parse_date_time(x, orders, tz = "UTC", truncated = 0, quiet = FALSE, locale = Sys.getlocale("LC_TIME"), select_formats = .select_formats, exact = FALSE)

parse_date_time2(x, orders, tz = "UTC", exact = FALSE, lt = FALSE)

fast_strptime(x, format, tz = "UTC", lt = TRUE)
Arguments

- **x**: a character or numeric vector of dates
- **orders**: a character vector of date-time formats. Each order string is a series of formatting characters as listed in `strptime` but might not include the "\%" prefix, for example "ymd" will match all the possible dates in year, month, day order. Formatting orders might include arbitrary separators. These are discarded. See details for implemented formats.
- **tz**: a character string that specifies the time zone with which to parse the dates
- **truncated**: integer, number of formats that can be missing. The most common type of irregularity in date-time data is the truncation due to rounding or unavailability of the time stamp. If truncated parameter is non-zero `parse_date_time` also checks for truncated formats. For example, if the format order is "ymdHMS" and `truncated = 3`, `parse_date_time` will correctly parse incomplete dates like 2012-06-01 12:23, 2012-06-01 12 and 2012-06-01.
- **quiet**: logical. When TRUE progress messages are not printed, and "no formats found" error is suppressed and the function simply returns a vector of NAs. This mirrors the behavior of base R functions `strptime` and `as.POSIXct`. Default is FALSE.
- **locale**: locale to be used, see `locales`. On Linux systems you can use `system("locale -a")` to list all the installed locales.
- **select_formats**: a function to select actual formats for parsing from a set of formats which matched a training subset of `x`. It receives a named integer vector and returns a character vector of selected formats. Names of the input vector are formats (not orders) that matched the training set. Numeric values are the number of dates (in the training set) that matched the corresponding format. You should use this argument if the default selection method fails to select the formats in the right order. By default the formats with most formatting tokens (\%) are selected and %Y counts as 2.5 tokens (so that it has a priority over %y%m). See examples.
- **exact**: logical. If TRUE, `orders` parameter is interpreted as an exact `strptime` format and no training or guessing are performed.
- **lt**: logical. If TRUE returned object is of class POSIXlt, and POSIXct otherwise. For compatibility with base `strptime` function default is TRUE for `fast_strptime` and FALSE for `parse_date_time2`.
- **format**: a character string of formats. It should include all the separators and each format must be prefixed with argument of `strptime`.

Details

When several format-orders are specified `parse_date_time` sorts the supplied format-orders based on a training set and then applies them recursively on the input vector.

`parse_date_time`, and all derived functions, such as `ymd_hms`, `ymd` etc, will drop into `fast_strptime` instead of `strptime` whenever the guessed from the input data formats are all numeric.

The list below contains formats recognized by lubridate. For numeric formats leading 0s are optional. As compared to base `strptime`, some of the formats are new or have been extended for efficiency reasons. These formats are marked with "*". Fast parsers, `parse_date_time2` and `fast_strptime`, accept only formats marked with "!".
Abbreviated weekday name in the current locale. (Also matches full name)
A Full weekday name in the current locale. (Also matches abbreviated name).
You need not specify a and A formats explicitly. Wday is automatically handled if preproc_wday = TRUE
b! Abbreviated month name in the current locale (also matches full name). C parser understands
English months only.
B! Same as b.
d! Day of the month as decimal number (01–31 or 0–31)
H! Hours as decimal number (00–24 or 0–24).
I! Hours as decimal number (01–12 or 1–12).
j Day of year as decimal number (001–366 or 1–366).
q! Quarter (1-4). The quarter month is added to parsed month if m format is present.
mt! Month as decimal number (01–12 or 1–12). For parse_date_time. As lubridate extension, also matches abbreviated and full months names as b and B formats. C parser understands only English month names.
M! Minute as decimal number (00–59 or 0–59).
p! AM/PM indicator in the locale. Normally used in conjunction with I and not with H. But lubridate C parser accepts H format as long as hour is not greater than 12. C parser understands only English locale AM/PM indicator.
S! Second as decimal number (00–61 or 0–61), allowing for up to two leap-seconds (but POSIX-compliant implementations will ignore leap seconds).
0S Fractional second.
U Week of the year as decimal number (00–53 or 0-53) using Sunday as the first day 1 of the week (and typically with the first Sunday of the year as day 1 of week 1). The US convention.
w Weekday as decimal number (0–6, Sunday is 0).
w Week of the year as decimal number (00–53 or 0-53) using Monday as the first day of week (and typically with the first Monday of the year as day 1 of week 1). The UK convention.
y!* Year without century (00–99 or 0–99). In parse_date_time also matches year with century (Y format).
Y! Year with century.
z!* ISO8601 signed offset in hours and minutes from UTC. For example -0800, -08:00 or -08, all represent 8 hours behind UTC. This format also matches the Z (Zulu) UTC indicator. Because strftime doesn’t fully support ISO8601 this format is implemented as an union of 4 orders: Oo (Z), Oz (-0800), OO (-08:00) and Oo (-08). You can use these four orders as any other but it is rarely necessary. parse_date_time2 and fast_strptime support all of the timezone formats.
0m!* Matches numeric month and English alphabetic months (Both, long and abbreviated forms).
0p!* Matches AM/PM English indicator.
r!* Matches Ip and H orders.
R!* Matches HM and IMp orders.
T!* Matches IMSp, HMS, and HMOS orders.
Value

a vector of POSIXct date-time objects

Note

parse_date_time (and the derivatives ymd, ymd_hms etc) rely on a sparse guesser that takes at most
501 elements from the supplied character vector in order to identify appropriate formats from the
supplied orders. If you get the error All formats failed to parse and you are confident that
your vector contains valid dates, you should either set exact argument to TRUE or use functions
that don’t perform format guessing (fast_strptime, parse_date_time2 or strptime).

For performance reasons, when timezone is not UTC, parse_date_time2 and fast_strptime per-
form no validity checks for daylight savings time. Thus, if your input string contains an invalid date
time which falls into DST gap and lt=TRUE you will get a POSIXlt object with a non-existen
time. If lt=FALSE your time instant will be adjusted to a valid time by adding an hour. See examples. If
you want to get NA for invalid date-times use fit_to_timeline explicitly.

See Also

strptime, ymd, ymd_hms

Examples

```r
## ** orders are much easier to write **
x <- c("09-01-01", "09-01-02", "09-01-03")
parse_date_time(x, "ymd")
parse_date_time(x, "y m d")
parse_date_time(x, "%Y%M%D")
# "2009-01-01 UTC" "2009-01-02 UTC" "2009-01-03 UTC"

## ** heterogenous date-times **
x <- c("09-01-01", "090102", "09-01 03", "09-01-03 12:02")
parse_date_time(x, c("ymd", "ymd H:M"))

## ** different ymd orders **
x <- c("2009-01-01", "02022010", "02-02-2010")
parse_date_time(x, c("dMy", "ymd"))
# "2009-01-01 UTC" "2010-02-02 UTC" "2010-02-02 UTC"

## ** truncated time-dates **
parse_date_time(x, "Ymd HMS", truncated = 3)

## ** specifying exact formats and avoiding training and guessing **
parse_date_time(x, c("%m-%d-%y", "%m%d%y", "%m-%d-%y %H:%M"), exact = TRUE)
parse_date_time(x, c("%m/%d/%Y %I:%M:%S", "%m/%d/%Y %H:%M"), exact = TRUE)

## ** quarters and partial dates **
parse_date_time(x, c("2016.2", "2016-04"), orders = "Yq")
parse_date_time(x, c("2016", "2016-04"), orders = c("Y", "Ym"))
```
Create a period object.

Description

period creates a period object with the specified values. period provides the behaviour of period in a way that is more suitable for automating within a function.

Usage

period(num = NULL, units = "second", ...)

is.period(x)
Arguments

num a numeric vector that lists the number of time units to be included in the period. From v1.6.0 num can also be a character vector that specifies durations in a convenient shorthand format. All unambiguous name units and abbreviations are supported. One letter "m" stands for months, "M" stands for minutes. See examples.

units a character vector that lists the type of units to be used. The units in units are matched to the values in num according to their order. When num is character, this argument is ignored.

... a list of time units to be included in the period and their amounts. Seconds, minutes, hours, days, weeks, months, and years are supported. Normally only one of num or ... are present. If both are present, the periods are concatenated.

x an R object

Details

Within a Period object, time units do not have a fixed length (except for seconds) until they are added to a date-time. The length of each time unit will depend on the date-time to which it is added. For example, a year that begins on 2009-01-01 will be 365 days long. A year that begins on 2012-01-01 will be 366 days long. When math is performed with a period object, each unit is applied separately. How the length of a period is distributed among its units is non-trivial. For example, when leap seconds occur 1 minute is longer than 60 seconds.

Periods track the change in the "clock time" between two date-times. They are measured in common time related units: years, months, days, hours, minutes, and seconds. Each unit except for seconds must be expressed in integer values.

Period objects can be easily created with the helper functions years, months, weeks, days, hours, minutes, and seconds. These objects can be added to and subtracted to date-times to create a user interface similar to object oriented programming.

Note: Arithmetic with periods can results in undefined behavior when non-existent dates are involved (such as February 29th). Please see Period-class for more details and %m+ and add_with_rollback for alternative operations.

Value

a period object

See Also

Period-class, quick_periods, %m+, add_with_rollback

Examples

period(c(90, 5), c("second", "minute"))
# "5M 90S"
period(-1, "days")
period(c(3, 1, 2, 13, 1), c("second", "minute", "hour", "day", "week"))
period(c(1, -60), c("hour", "minute"))
Description

Period is an S4 class that extends the `TimeSpan-class` class. Periods track the change in the "clock time" between two date-times. They are measured in common time related units: years, months, days, hours, minutes, and seconds. Each unit except for seconds must be expressed in integer values.

Details

The exact length of a period is not defined until the period is placed at a specific moment of time. This is because the precise length of one year, month, day, etc. can change depending on when it occurs due to daylight savings, leap years, and other conventions. A period can be associated with a specific moment in time by coercing it to an `Interval-class` object with `as.interval` or by adding it to a date-time with "+".

Periods provide a method for measuring generalized timespans when we wish to model clock times. Periods will attain intuitive results at this task even when leap years, leap seconds, gregorian days, daylight savings changes, and other events happen during the period. See `Duration-class` for an alternative way to measure timespans that allows precise comparisons between timespans.

The logic that guides arithmetic with periods can be unintuitive. Starting with version 1.3.0, lubridate enforces the reversible property of arithmetic (e.g. a date + period - period = date) by returning an NA if you create an implausible date by adding periods with months or years units to a date. For example, adding one month to January 31st, 2013 results in February 31st, 2013, which is not a real date. lubridate users have argued in the past that February 31st, 2013 should be rolled over to March 3rd, 2013 or rolled back to February 28, 2013. However, each of these corrections would destroy the reversibility of addition (Mar 3 - one month == Feb 3 != Jan 31, Feb 28 - one month == Jan 28 != Jan 31). If you would like to add and subtract months in a way that rolls the results back to the last day of a month (when appropriate) use the special operators, `%m+%`, `%m-%` or a bit more flexible `add_with_rollback`. 

```r
period(0, "second")
period (second = 90, minute = 5)
period(day = -1)
period(second = 3, minute = 1, hour = 2, day = 13, week = 1)
period(hour = 1, minute = -60)
period(second = 0)
period(c(1, -60), c("hour", "minute"), hour = c(1, 2), minute = c(3, 4))
period("2M 1sec")
period("2hours 2minutes 1second")
period("2d 2H 2M 2S")
# Missing numerals default to 1. Repeated units are added up.
duration("day day")
# Comparison with characters is supported from v1.6.0.
duration("day 2 sec") > "day 1sec"
is.period(as.Date("2009-08-03")) # FALSE
is.period(period(months= 1, days = 15)) # TRUE
```
Period class objects have six slots. 1) .Data, a numeric object. The apparent amount of seconds to add to the period. 2) minute, a numeric object. The apparent amount of minutes to add to the period. 3) hour, a numeric object. The apparent amount of hours to add to the period. 4) day, a numeric object. The apparent amount of days to add to the period. 5) month, a numeric object. The apparent amount of months to add to the period. 6) year, a numeric object. The apparent amount of years to add to the period.

---

**period_to_seconds**

Contrive a period to/from a given number of seconds.

---

**Description**

*period_to_seconds* approximately converts a period to seconds assuming there are 364.25 days in a calendar year and 365.25/12 days in a month. 
*seconds_to_period* create a period that has the maximum number of non-zero elements (days, hours, minutes, seconds). This computation is exact because it doesn’t involve years or months.

**Usage**

```
period_to_seconds(x)
seconds_to_period(x)
```

**Arguments**

- `x` A numeric object. The number of seconds to coerce into a period.

**Value**

A number (period) that roughly equates to the period (seconds) given.

---

**pretty_dates**

Computes attractive axis breaks for date-time data

---

**Description**

*pretty_dates* indentifies which unit of time the sub-intervals should be measured in to provide approximately n breaks. It then chooses a "pretty" length for the sub-intervals and sets start and endpoints that 1) span the entire range of the data, and 2) allow the breaks to occur on important date-times (i.e. on the hour, on the first of the month, etc.)

**Usage**

```
pretty_dates(x, n, ...)
```
quarter

Arguments

- **x**: a vector of POSIXct, POSIXlt, Date, or chron date-time objects
- **n**: integer value of the desired number of breaks
- ... additional arguments to pass to function

Value

a vector of date-times that can be used as axis tick marks or bin breaks

Examples

```r
x <- seq.Date(as.Date("2009-08-02"), by = "year", length.out = 2)
pretty_dates(x, 12)
```

---

**quarter**

Get the fiscal quarter and semester of a date-time.

Description

Quarters divide the year into fourths. Semesters divide the year into halves.

Usage

```r
quarter(x, with_year = FALSE)
semester(x, with_year = FALSE)
```

Arguments

- **x**: a date-time object of class POSIXct, POSIXlt, Date, chron, yearmon, yearqtr, zoo, zoo, timeDate, xts, its, ti, jul, timeSeries, fts or anything else that can be converted with as.POSIXlt
- **with_year**: logical indicating whether or not to include the quarter’s year.

Value

numeric

Examples

```r
x <- ymd(c("2012-03-26", "2012-05-04", "2012-09-23", "2012-12-31"))
quarter(x)
quarter(x, with_year = TRUE)
semester(x)
semester(x, with_year = TRUE)
```
quick_durations

**Description**

Quickly create Duration objects for easy date-time manipulation. The units of the duration created depend on the name of the function called. For Duration objects, units are equal to their most common lengths in seconds (i.e. minutes = 60 seconds, hours = 3600 seconds, days = 86400 seconds, weeks = 604800, years = 31536000).

**Usage**

- dseconds(x = 1)
- dminutes(x = 1)
- dhours(x = 1)
- ddays(x = 1)
- dweeks(x = 1)
- dyears(x = 1)
- dmilliseconds(x = 1)
- dmicroseconds(x = 1)
- dnanoseconds(x = 1)
- dpicoseconds(x = 1)

**Arguments**

- x 
  numeric value of the number of units to be contained in the duration.

**Details**

When paired with date-times, these functions allow date-times to be manipulated in a method similar to object oriented programming. Duration objects can be added to Date, POSIXt, and Interval objects.

Since version 1.4.0 the following functions are deprecated: eseconds, eminutes, ehours, edays, eweeks, eyears, emILLISECONDS, emicroseconds, enanoseconds, epicoseconds

**Value**

- a duration object
quick_periods

See Also
duration, days

Examples

dseconds(1)
dminutes(3.5)

x <- as.POSIXct("2009-08-03")
x + ddays(1) + dhours(6) + dminutes(30)
x + ddays(100) - dhours(8)

class(as.Date("2009-08-09") + ddays(1)) # retains Date class
class(as.Date("2009-08-09") + dhours(12))
# converts to POSIXt class to accommodate time units

dweeks(1) - ddays(7)
c(1:3) * dhours(1)
# compare DST handling to durations
boundary <- as.POSIXct("2009-03-08 01:59:59")
boundary + ddays(1) # period
boundary + ddays(1) # duration

quick_periods

Quickly create period objects.

Description

Quickly create Period objects for easy date-time manipulation. The units of the period created depend on the name of the function called. For Period objects, units do not have a fixed length until they are added to a specific date time, contrast this with duration. This makes periods useful for manipulations with clock times because units expand or contract in length to accommodate conventions such as leap years, leap seconds, and Daylight Savings Time.

Usage

seconds(x = 1)

minutes(x = 1)

hours(x = 1)

days(x = 1)

weeks(x = 1)
years(x = 1)
milliseconds(x = 1)
milliseconds(x = 1)
nanoseconds(x = 1)
picoseconds(x = 1)

## S3 method for class 'numeric'
months(x, abbreviate)

**Arguments**

- **x**: numeric value of the number of units to be contained in the period. With the exception of seconds(), x must be an integer.
- **abbreviate**: Ignored. For consistency with S3 generic in base namespace.

**Details**

When paired with date-times, these functions allow date-times to be manipulated in a method similar to object-oriented programming. Period objects can be added to Date, POSIXct, and POSIXlt objects to calculate new date-times.

Note: Arithmetic with periods can result in undefined behavior when non-existent dates are involved (such as February 29th in non-leap years). Please see `Period-class` for more details and `%m+%` and `add_with_rollback` for alternative operations.

**Value**

a period object

**See Also**

`Period-class`, `period`, `ddays`, `%m+%`, `add_with_rollback`

**Examples**

```r
x <- as.POSIXct("2009-08-03")
x + days(1) + hours(6) + minutes(30)
x + days(100) - hours(8)

class(as.Date("2009-08-09") + days(1)) # retains Date class
as.Date("2009-08-09") + hours(12)
class(as.Date("2009-08-09") + hours(12))
# converts to POSIXt class to accomodate time units

years(1) - months(7)
c(1:3) * hours(1)
```
Description

rollback changes a date to the last day of the previous month or to the first day of the month. Optionally, the new date can retain the same hour, minute, and second information.

Usage

rollback(dates, roll_to_first = FALSE, preserve_hms = TRUE)

Arguments

dates A POSIXct, POSIXlt or Date class object.
roll_to_first Rollback to the first day of the month instead of the last day of the previous month
preserve_hms Retains the same hour, minute, and second information? If FALSE, the new date will be at 00:00:00.

Value

A date-time object of class POSIXlt, POSIXct or Date, whose day has been adjusted to the last day of the previous month, or to the first day of the month.

Examples

date <- ymd("2010-03-03")
rollback(date)

dates <- date + months(0:2)
rollback(dates)

date <- ymd_hms("2010-03-03 12:44:22")
rollback(date)
rollback(date, roll_to_first = TRUE)
rollback(date, preserve_hms = FALSE)
rollback(date, roll_to_first = TRUE, preserve_hms = FALSE)
round_date

Round, floor and ceiling methods for date-time objects.

Description

Rounding to the nearest unit or multiple of a unit are supported. All meaningful specifications in English language are supported - secs, min, mins, 2 minutes, 3 years etc.

round_date takes a date-time object and rounds it to the nearest value of the specified time unit. For rounding date-times which is exactly halfway between two consecutive units, the convention is to round up. Note that this is in line with the behavior of R's base round.POSIXt function but does not follow the convention of the base round function which "rounds to the even digit" per IEC 60559.

floor_date takes a date-time object and rounds it down to the nearest boundary of the specified time unit.

ceiling_date takes a date-time object and rounds it up to the nearest boundary of the specified time unit.

Usage

   round_date(x, unit = "second")
   floor_date(x, unit = "seconds")
   ceiling_date(x, unit = "seconds", change_on_boundary = NULL)

Arguments

x a vector of date-time objects
unit a character string specifying the time unit or a multiple of a unit to be rounded to. Valid base units are second, minute, hour, day, week, month, bimonth, quarter, halfyear, or year. Arbitrary unique English abbreviations as in period constructor are also supported. Rounding to multiple of units (except weeks) is supported from v1.6.0.
change_on_boundary If NULL (the default) don’t change instants on the boundary (ceiling_date(ymd_hms('2000-01-01 00:00:00'), "seconds") is 2000-01-01 00:00:00, but round up Date objects to the next boundary (ceiling_date(ymd("2000-01-01"), "month") is "2000-02-01"). When TRUE, instants on the boundary are rounded up to the next boundary. When FALSE, date-time on the boundary are never rounded up (this was the default for lubridate prior to v1.6.0. See section Rounding Up Date Objects below for more details.

Details

In lubridate rounding of a date-time objects tries to preserve the class of the input object whenever it is meaningful. This is done by first rounding to an instant and then converting to the original class by usual R conventions.
round_date

Rounding Up Date Objects

By default rounding up Date objects follows 3 steps:

1. Convert to an instant representing lower bound of the Date: 2000-01-01 -> 2000-01-01 00:00:00
2. Round up to the nearest closest rounding unit boundary. For example, if the rounding unit is month then next boundary for 2000-01-01 will be 2000-02-01 00:00:00.
   The motivation for this behavior is that 2000-01-01 is conceptually an interval (2000-01-01 00:00:00 -- 2000-01-00:00:00) and the day hasn’t started clocking yet at the exact boundary 00:00:00. Thus, it seems wrong to round up a day to its lower boundary.
3. If rounding unit is smaller than a day, return the instant from step 2 above (POSIXct), otherwise return the Date immediately following that instant.

The behavior on the boundary in the second step above can be changed by setting change_on_boundary to a non-NULL value.

See Also

round

Examples

```r
x <- as.POSIXct("2009-08-03 12:01:59.23")
round_date(x, "second")
round_date(x, "minute")
round_date(x, "5 mins")
round_date(x, "hour")
round_date(x, "2 hours")
round_date(x, "day")
round_date(x, "week")
round_date(x, "month")
round_date(x, "bimonth")
round_date(x, "quarter") == round_date(x, "3 months")
round_date(x, "halfyear")
round_date(x, "year")

x <- as.POSIXct("2009-08-03 12:01:59.23")
floor_date(x, "second")
floor_date(x, "minute")
floor_date(x, "hour")
floor_date(x, "day")
floor_date(x, "week")
floor_date(x, "month")
floor_date(x, "bimonth")
floor_date(x, "quarter")
floor_date(x, "halfyear")
floor_date(x, "year")

x <- as.POSIXct("2009-08-03 12:01:59.23")
ceiling_date(x, "second")
ceiling_date(x, "minute")
```
second

Get/set seconds component of a date-time.

Description

Date-time must be a POSIXct, POSIXlt, Date, Period, chron, yearmon, yearqtr, zoo, zooreg, timeDate, xts, its, ti, jul, timeSeries, and fts objects.

Usage

second(x)

second(x) <- value

Arguments

x a date-time object
value numeric value to be assigned

Value

the seconds element of x as a decimal number

Examples

x <- ymd("2012-03-26")
second(x)
second(x) <- 1
second(x) <- 61
second(x) > 2
Description

Stamps are just like format, but based on human-friendly templates like "Recorded at 10 am, September 2002" or "Meeting, Sunday May 1, 2000, at 10:20 pm".

Usage

```r
stamp(x, orders = lubridate_formats, locale = Sys.getlocale("LC_TIME"),
    quiet = FALSE)

stamp_date(x, locale = Sys.getlocale("LC_TIME"))

stamp_time(x, locale = Sys.getlocale("LC_TIME"))
```

Arguments

- **x**: a character vector of templates.
- **orders**: orders are sequences of formatting characters which might be used for disambiguation. For example "ymd hms", "aym" etc. See `guess_formats` for a list of available formats.
- **locale**: locale in which x is encoded. On linux like systems use `locale -a` in terminal to list available locales.
- **quiet**: whether to output informative messages.

Details

`stamp` is a stamping function date-time templates mainly, though it correctly handles all date and time formats as long as they are unambiguous. `stamp_date` and `stamp_time` are the specialized stamps for dates and times (MHS). These function might be useful when the input template is unambiguous and matches both a time and a date format.

Lubridate tries it’s best to figure our the formats, but often a given format can be interpreted in several ways. One way to deal with the situation is to provide unambiguous formats like 22/05/81 instead of 10/05/81 if you want d/m/y format. Another option is to use a more specialized `stamp_date` and `stamp_time`. The core function `stamp` give priority to longer date-time formats.

Another option is to proved a vector of several values as x parameter. Then lubridate will choose the format which fits x the best. Note that longer formats are preferred. If you have "22:23:00 PM" then "HMSp" format will be given priority to shorter "HMS" order which also fits the supplied string.

Finally, you can give desired format order directly as orders argument.

Value

a function to be applied on a vector of dates
Description of time span classes in lubridate.

Description

A time span can be measured in three ways: as a duration, an interval, or a period.

Details

Durations record the exact number of seconds in a time span. They measure the exact passage of time but do not always align with measurements made in larger units of time such as hours, months and years. This is because the exact length of larger time units can be affected by conventions such as leap years and Daylight Savings Time. Base R measures durations with the `difftime` class. lubridate provides an additional class, the duration class, to facilitate working with durations.

durations display as the number of seconds that occur during a time span. If the number is large, a duration object will also display the length in a more convenient unit, but these measurements are only estimates given for convenience. The underlying object is always recorded as a fixed number of seconds. For display and creation purposes, units are converted to seconds using their most common lengths in seconds. Minutes = 60 seconds, hours = 3600 seconds, days = 86400 seconds. Units larger than days are not used due to their variability.

duration objects can be easily created with the helper functions `dweeks`, `ddays`, `dhours`, `dminutes` and `dseconds`. These objects can be added to and subtracted from date-times to create a user interface similar to object oriented programming. Duration objects can be added to Date, POSIXct, and POSIXlt objects to return a new date-time.

Periods record the change in the clock time between two date-times. They are measured in common time related units: years, months, days, hours, minutes, and seconds. Each unit except for seconds must be expressed in integer values. With the exception of seconds, none of these units have a fixed length. Leap years, leap seconds, and Daylight Savings Time can expand or contract a unit of time depending on when it occurs. For this reason, periods do not have a fixed length until they are

See Also

`guess_formats`, `parse_date_time`, `strptime`
paired with a start date. Periods can be used to track changes in clock time. Because periods have a variable length, they must be paired with a start date as an interval (`as.interval`) before they can be accurately converted to and from durations.

Period objects can be easily created with the helper functions `years`, `months`, `weeks`, `days`, `minutes`, `seconds`. These objects can be added to and subtracted to date-times to create a user interface similar to object oriented programming. Period objects can be added to Date, POSIXct, and POSIXlt objects to return a new date-time.

Intervals are time spans bound by two real date-times. Intervals can be accurately converted to periods and durations. Since an interval is anchored to a fixed moment of time, the exact length of all units of time during the interval can be calculated. To accurately convert between periods and durations, a period or duration should first be converted to an interval with `as.interval`. An interval displays as the start and end points of the time span it represents.

See Also

`duration` for creating duration objects and `period` for creating period objects, and `interval` for creating interval objects.

Examples

duration(3690, "seconds")
period(3690, "seconds")
period(second = 30, minute = 1, hour = 1)
interval(ymd_hms("2009-08-09 13:01:30"), ymd_hms("2009-08-09 12:00:00"))

date <- as.POSIXct("2009-03-08 01:59:59") # DST boundary
date + days(1)
date + ddays(1)

date2 <- as.POSIXct("2000-02-29 12:00:00")
date2 + years(1)
# self corrects to next real day

date3 <- as.POSIXct("2009-01-31 01:00:00")
date3 + c(0:11) * months(1)

span <- date2 %--% date  # creates interval

date <- as.POSIXct("2009-01-01 00:00:00")
date + years(1)
date - days(3) + hours(6)
date + 3 * seconds(10)

months(6) + days(1)
time_length

Description

Timespan is an S4 class with no slots. It is extended by the `Interval-class`, `Period-class`, and `Duration-class` classes.

Description

Compute the exact length of a time span.

Usage

time_length(x, unit = "second")

## S4 method for signature 'Interval'
time_length(x, unit = "second")

Arguments

x a duration, period, difftime or interval
unit a character string that specifies with time units to use

Details

When x is an `Interval-class` object and unit are years or months, `timespan_length` takes into account the fact that all months and years don’t have the same number of days.

When x is a `Duration-class`, `Period-class` or `difftime` object, length in months or years is based on their most common lengths in seconds (see `timespan`).

Value

the length of the interval in the specified unit. A negative number connotes a negative interval or duration

See Also

timespan

Examples

int <- interval(ymd("1980-01-01"), ymd("2014-09-18"))
time_length(int, "week")

# Exact age
time_length(int, "year")
```r
# Age at last anniversary
t trunc(time_length(int, "year"))

# Example of difference between intervals and durations
int <- interval(ymd("1900-01-01"), ymd("1999-12-31"))
time_length(int, "year")
time_length(as.duration(int), "year")
```

---

**today**  
*The current date*

**Description**  
The current date

**Usage**  
today(tzone = "")

**Arguments**  
tzone  
A character vector specifying which time zone you would like to find the current date of. tzone defaults to the system time zone set on your computer.

**Value**  
the current date as a Date object

**Examples**  
today()
today("GMT")
today() == today("GMT") # not always true
today() < as.Date("2999-01-01") # TRUE (so far)

---

**tz**  
*Get/set time zone component of a date-time.*

**Description**  
Time zones are stored as character strings in an attribute of date-time objects. tz returns a date’s time zone attribute. When used as a settor, it changes the time zone attribute. R does not come with a predefined list zone names, but relies on the user’s OS to interpret time zone names. As a result, some names will be recognized on some computers but not others. Most computers, however, will recognize names in the timezone data base originally compiled by Arthur Olson. These names normally take the form "Country/City." A convenient listing of these timezones can be found at [http://en.wikipedia.org/wiki/List_of_tz_database_time_zones](http://en.wikipedia.org/wiki/List_of_tz_database_time_zones).
Usage

tz(x)

tz(x) <- value

Arguments

x a date-time object of class a POSIXct, POSIXlt, Date, chron, yearmon, yearqtr, zoo, zoo, timeDate, xts, its, ti, jul, timeSeries, fts or anything else that can be coerced to POSIXlt with as.POSIXlt

value timezone value to be assigned to x’s tzone attribute

Details

Setting tz does not update a date-time to display the same moment as measured at a different time zone. See with_tz. Setting a new time zone creates a new date-time. The numerical value of the hours element stays the same, only the time zone attribute is replaced. This creates a new date-time that occurs an integer value of hours before or after the original date-time.

If x is of a class that displays all date-times in the GMT timezone, such as chron, then R will update the number in the hours element to display the new date-time in the GMT timezone.

For a description of the time zone attribute, see timezones or DateTimeClasses.

Value

the first element of x’s tzone attribute vector as a character string. If no tzone attribute exists, tz returns "GMT".

Examples

```r
x <- ymd("2012-03-26")
tz(x)
tz(x) <- "GMT"
x
## Not run:
tz(x) <- "America/New_York"
x
tz(x) <- "America/Chicago"
x
tz(x) <- "America/Los_Angeles"
x
tz(x) <- "Pacific/Honolulu"
x
tz(x) <- "Pacific/Auckland"
x
tz(x) <- "Europe/London"
x
tz(x) <- "Europe/Berlin"
x
## End(Not run)
```
week

```r
Sys.setenv(TZ = "GMT")
now()
tz(now())
Sys.unsetenv("TZ")
```

---

**week**

*Get/set weeks component of a date-time.*

**Description**

week returns the number of complete seven day periods that have occurred between the date and January 1st, plus one.

isoweek returns the week as it would appear in the ISO 8601 system, which uses a reoccuring leap week.

**Usage**

```r
week(x)

week(x) <- value

isoweek(x)
```

**Arguments**

- **x**
  - a date-time object. Must be a POSIXct, POSIXlt, Date, chron, yearmon, yearqtr, zoo, zooReg, timeDate, xts, its, ti, jul, timeSeries, or fts object.

- **value**
  - a numeric object

**Value**

the weeks element of x as an integer number

**References**


**See Also**

isoyear

**Examples**

```r
x <- ymd("2012-03-26")
week(x)
week(x) <- 1
week(x) <- 54
week(x) > 3
```
with_tz

Get date-time in a different time zone

Description

with_tz returns a date-time as it would appear in a different time zone. The actual moment of time measured does not change, just the time zone it is measured in. with_tz defaults to the Universal Coordinated time zone (UTC) when an unrecognized time zone is inputted. See Sys.timezone for more information on how R recognizes time zones.

Usage

with_tz(time, tzone = '')

Arguments

time a POSIXct, POSIXlt, Date, chron date-time object or a data.frame object. When a data.frame all POSIXt elements of a data.frame are processed with with_tz and new data.frame is returned.
tzone a character string containing the time zone to convert to. R must recognize the name contained in the string as a time zone on your system.

Value

a POSIXct object in the updated time zone

See Also

force_tz

Examples

x <- as.POSIXct("2009-08-07 00:00:01", tz = "America/New_York")
with_tz(x, "GMT")

year

Get/set years component of a date-time.

Description

Date-time must be a POSIXct, POSIXlt, Date, Period, chron, yearmon, yearqtr, zoo, zooreg, timeDate, xts, its, ti, jul, timeSeries, and its objects.
**ymd**

**Usage**

```r
year(x)
year(x) <- value
isoyear(x)
```

**Arguments**

- `x` a date-time object
- `value` a numeric object

**Details**

year does not yet support years before 0 C.E.

**Value**

the years element of `x` as a decimal number

**Examples**

```r
x <- ymd("2012-03-26")
year(x)
year(x) <- 2001
year(x) > 1995
```

---

Parse dates according to the order in that year, month, and day elements appear in the input vector.

---

**Description**

Transforms dates stored in character and numeric vectors to Date or POSIXct objects (see tz argument). These functions recognize arbitrary non-digit separators as well as no separator. As long as the order of formats is correct, these functions will parse dates correctly even when the input vectors contain differently formatted dates. See examples.

**Usage**

```r
ymd(..., quiet = FALSE, tz = NULL, locale = Sys.getlocale("LC_TIME"), truncated = 0)
```

```r
ydm(..., quiet = FALSE, tz = NULL, locale = Sys.getlocale("LC_TIME"), truncated = 0)
```

```r
mdy(..., quiet = FALSE, tz = NULL, locale = Sys.getlocale("LC_TIME"),
```
truncated = 0)

ymd(..., quiet = FALSE, tz = NULL, locale = Sys.getlocale("LC_TIME"),
     truncated = 0)

dmy(..., quiet = FALSE, tz = NULL, locale = Sys.getlocale("LC_TIME"),
     truncated = 0)

dym(..., quiet = FALSE, tz = NULL, locale = Sys.getlocale("LC_TIME"),
     truncated = 0)

yq(..., quiet = FALSE, tz = NULL, locale = Sys.getlocale("LC_TIME"))

Arguments

... a character or numeric vector of suspected dates

quiet logical. When TRUE function evaluates without displaying customary mes-
sages.

tz Time zone indicator. If NULL (default) a Date object is returned. Otherwise a
     POSIXct with time zone attribute set to tz.

locale locale to be used, see locales. On linux systems you can use system("locale -a")
     to list all the installed locales.

truncated integer. Number of formats that can be truncated.

Details

If truncated parameter is non-zero ymd functions also check for truncated formats. For example
ymd with truncated = 2 will also parse incomplete dates like 2012-06 and 2012.
NOTE: ymd family of functions are based on 'parse_date_time' and thus directly drop to internal C
parser for numeric months, but use R’s ‘strptime’ for alphabetic months. This implies that some of
the ‘strptime’’s limitations are inherited by lubridate’s parser. For example truncated formats (like
%Y-%b) will not be parsed. Numeric truncated formats (like %Y-%m) are handled correctly by
lubridate’s C parser.

As of version 1.3.0, lubridate’s parse functions no longer return a message that displays which
format they used to parse their input. You can change this by setting the lubridate.verbose
option to TRUE with options(lubridate.verbose = TRUE).

Value

a vector of class POSIXct if tz argument is non-NULL or Date if tz is NULL (default)

See Also

parse_date_time for an even more flexible low level mechanism.
**ymd_hms**

Parse dates that have hours, minutes, or seconds elements.

**Description**

Transform dates stored as character or numeric vectors to POSIXct objects. ymd_hms family of functions recognize all non-alphanumeric separators (with the exception of "." if frac = TRUE) and correctly handle heterogeneous date-time representations. For more flexibility in treatment of heterogeneous formats, see low level parser `parse_date_time`.

**Usage**

```r
ymd_hms(..., quiet = FALSE, tz = "UTC", locale = Sys.getlocale("LC_TIME"), truncated = 0)
```

```r
ymd_hm(..., quiet = FALSE, tz = "UTC", locale = Sys.getlocale("LC_TIME"), truncated = 0)
```

```r
ymd_h(..., quiet = FALSE, tz = "UTC", locale = Sys.getlocale("LC_TIME"), truncated = 0)
```

```r
dmy_hms(..., quiet = FALSE, tz = "UTC", locale = Sys.getlocale("LC_TIME"), truncated = 0)
```

**Examples**

```r
x <- c("09-01-01", "09-01-02", "09-01-03")
ymd(x)
```

```r
x <- c("2009-01-01", "2009-01-02", "2009-01-03")
ymd(x)
```

```r
now() > ymd(20090101)
## TRUE
dmy(010210)
dmy(010210)
```

```r
## heterogenuous formats in a single vector:
x <- c(20090101, "2009-01-02", "2009 01 03", "2009-1-4", "2009-1, 5", "Created on 2009 1 6", "20090111 07")
ymd(x)
```

```r
## What lubridate might not handle:
```

```r
## Extremely weird cases when one of the separators is "" and some of the
## formats are not in double digits might not be parsed correctly:
## Not run: ymd("201002-01", "201002-1", "20102-1")
dmy("0312-2010", "312-2010")
## End(Not run)
```
Arguments

- **...** a character vector of dates in year, month, day, hour, minute, second format
- **quiet** logical. When TRUE function evaluates without displaying customary messages.
- **tz** a character string that specifies which time zone to parse the date with. The string must be a time zone that is recognized by the user's OS.
- **locale** locale to be used, see `locales`. On linux systems you can use `system("locale -a")` to list all the installed locales.
- **truncated** integer, indicating how many formats can be missing. See details.

Details

`ymd_hms()` functions automatically assigns the Universal Coordinated Time Zone (UTC) to the parsed date. This time zone can be changed with `force_tz`.

The most common type of irregularity in date-time data is the truncation due to rounding or unavailability of the time stamp. If truncated parameter is non-zero `ymd_hms` functions also check for truncated formats. For example `ymd_hms` with `truncated = 3` will also parse incomplete dates like `2012-06-01 12:23`, `2012-06-01 12` and `2012-06-01`. NOTE: `ymd` family of functions are based on `strptime` which currently fails to parse `%y-%m` formats.

As of version 1.3.0, lubridate's parse functions no longer return a message that displays which format they used to parse their input. You can change this by setting the `lubridate.verbose` option to true with `options(lubridate.verbose = TRUE)`.
ymd_hms

Value

a vector of POSIXct date-time objects

See Also

ymd, hms, parse_date_time for underlying mechanism.

Examples

x <- c("2010-04-14-04-35-59", "2010-04-01-12-00-00")
ymd_hms(x)
x <- c("2011-12-31 12:59:59", "2010-01-01 12:00:00")
ymd_hms(x)

## ** heterogenous formats **
x <- c("200012012012", "2009.01.03 12:01:03",
"2009-1-4 12-1-4",
"2009-1, 5 12:1, 5",
"200901-08 1201-08",
"2009 arbitrary 1 non-decimal 6 chars 12 in between 1 !!! 6",
"OR collapsed formats: 20001207 120107 (as long as prefixed with zeros)",
"Automatic wday, Thu, detection, 10-01-10 10:01:10 and p format: AM",
"Created on 10-01-11 at 10:01:11 PM")
ymd_hms(x)

## ** fractional seconds **
op <- options(digits.secs=3)
dmy_hms("20/2/06 11:16:16.683")
options(op)

## ** different formats for ISO8601 timezone offset **
ymd_hms(c("2013-01-24 19:39:07.880-0600",

## ** internationalization **
## Not run:
x_RO <- "Ma 2012 august 14 11:28:30 "
ymd_hms(x_RO, locale = "ro_RO.utf8")

## End(Not run)

## ** truncated time-dates **
ymd_hms(x, truncated = 3)
x <- c("2011-12-31 12:59", "2010-01-01 12", "2010-01-01")
ymd_hms(x, truncated = 2)

## ** What lubridate might not handle **
## Extremely weird cases when one of the separators is "" and some of the
## formats are not in double digits might not be parsed correctly:
Description

Adding months frustrates basic arithmetic because consecutive months have different lengths. With other elements, it is helpful for arithmetic to perform automatic roll over. For example, 12:00:00 + 61 seconds becomes 12:01:01. However, people often prefer that this behavior NOT occur with months. For example, we sometimes want January 31 + 1 month = February 28 and not March 3. \%m+% performs this type of arithmetic. Date \%m+% months(n) always returns a date in the nth month after Date. If the new date would usually spill over into the n + 1th month, \%m+% will return the last day of the nth month (rollback. Date \%m-% months(n) always returns a date in the nth month before Date.

add_with_rollback provides additional functionality to \%m+% and \%m-. It allows rollback to first day of the month instead of the last day of the previous month and controls whether HMS component of the end date is preserved or not.

Usage

e1 \%m+% e2

add_with_rollback(e1, e2, roll_to_first = FALSE, preserve_hms = TRUE)

Arguments

e1 A period or a date-time object of class POSIXlt, POSIXct or Date.
e2 A period or a date-time object of class POSIXlt, POSIXct or Date. Note that one of e1 and e2 must be a period and the other a date-time object.
roll_to_first rollback to the first day of the month instead of the last day of the previous month (passed to rollback)
preserve_hms retains the same hour, minute, and second information? If FALSE, the new date will be at 00:00:00 (passed to rollback)

Details

\%m+% and \%m-% handle periods with components less than a month by first adding/substracting months and then performing usual arithmetics with smaller units.

\%m+% and \%m-% should be used with caution as they are not one-to-one operations and results for either will be sensitive to the order of operations.
Value

A date-time object of class POSIXlt, POSIXct or Date

Examples

```r
jan <- ymd_hms("2010-01-31 03:04:05")
jan + months(1:3) # Feb 31 and April 31 returned as NA
# NA "2010-03-31 03:04:05 UTC" NA
jan +%m% months(1:3) # No rollover

leap <- ymd("2012-02-29")
"2012-02-29 UTC"
leap +%m% years(1)
leap +%m% years(-1)
leap +%m-% years(1)
```

%within%

Tests whether a date or interval falls within an interval

Description

If a is an interval, both its start and end dates must fall within b to return TRUE.

Usage

a %within% b

Arguments

a An interval or date-time object
b An interval

Value

A logical

Examples

```r
int <- interval(ymd("2001-01-01"), ymd("2002-01-01"))
int2 <- interval(ymd("2001-06-01"), ymd("2002-01-01"))

ymd("2001-05-03") %within% int # TRUE
int2 %within% int # TRUE
ymd("1999-01-01") %within% int # FALSE
```
Index

*Topic POSIXt
  ymd_hms, 63
*Topic chron
  am, 5
  as.duratio, 6
  as.interval, 7
  as.period, 8
  date, 11
  date_decimal, 13
  DateUpdate, 12
  day, 13
  decimal_date, 15
  dst, 17
  duration, 17
  force_tz, 20
  hour, 23
  is.Date, 27
  is.difftime, 28
  is.instant, 28
  is.POSIXt, 29
  is.timespan, 30
  leap_year, 31
  make_difftime, 32
  minute, 33
  month, 34
  now, 36
  origin, 37
  parse_date_time, 37
  period, 41
  pretty_dates, 44
  quick_durations, 46
  quick_periods, 47
  round_date, 50
  second, 52
  time_length, 56
  timespan, 54
  today, 57
  tz, 57
  week, 59
  with_tz, 60
  year, 60
  ymd, 61
*Topic classes
  as.duration, 6
  as.interval, 7
  as.period, 8
  duration, 17
  make_difftime, 32
  period, 41
  timespan, 54
*Topic data
  lakers, 30
  origin, 37
*Topic dplot
  pretty_dates, 44
*Topic logic
  is.Date, 27
  is.difftime, 28
  is.instant, 28
  is.POSIXt, 29
  is.timespan, 30
  leap_year, 31
*Topic manip
  as.duration, 6
  as.interval, 7
  as.period, 8
  date, 11
  date_decimal, 13
  DateUpdate, 12
  day, 13
  decimal_date, 15
  force_tz, 20
  hour, 23
  minute, 33
  month, 34
  quick_durations, 46
  quick_periods, 47
  round_date, 50
second, 52
tz, 57
week, 59
with_tz, 60
year, 60

*Topic **math**
time_length, 56

*Topic **methods**
as.duration, 6
as.interval, 7
as.period, 8
date, 11
date_decimal, 13
day, 13
decimal_date, 15
dst, 17
hour, 23
minute, 33
month, 34
second, 52
time_length, 56
tz, 57
year, 60

*Topic **parse**
ymd_hms, 63

*Topic **period**
ms, 35
time_length, 56

*Topic **utilities**
date, 11
day, 13
dst, 17
hour, 23
minute, 33
month, 34
now, 36
pretty_dates, 44
second, 52
today, 57
tz, 57
week, 59
year, 60

*, TimeSpan, TimeSpan-method
(TimeSpan-class), 55
%-%(interval), 24
%/%, TimeSpan, TimeSpan-method
(TimeSpan-class), 55
%/%, difftime, TimeSpan-method

 amat, 5
 as.duration, 5, 6, 7, 18, 25, 33
 as.duration, character-method
 (as.duration), 6
 as.duration, difftime-method
 (as.duration), 6
 as.duration, Duration-method
 (as.duration), 6
 as.duration, Interval-method
 (as.duration), 6
 as.duration, logical-method
 (as.duration), 6
 as.duration, numeric-method
 (as.duration), 6
 as.duration, Period-method
 (as.duration), 6
 as.interval, 5, 6, 7, 9, 26, 43, 55
 as.interval, difftime-method
 (as.interval), 7
 as.interval, Duration-method
 (as.interval), 7
 as.interval, Interval-method
 (as.interval), 7

add_with_rollback, 42, 43, 48
add_with_rollback (%m+%), 66
am, 5
as.duration, 5, 6, 7, 18, 25, 33
as.duration, character-method
(as.duration), 6
as.duration, difftime-method
(as.duration), 6
as.duration, Duration-method
(as.duration), 6
as.duration, Interval-method
(as.duration), 6
as.duration, logical-method
(as.duration), 6
as.duration, numeric-method
(as.duration), 6
as.duration, Period-method
(as.duration), 6
as.interval, 5, 6, 7, 9, 26, 43, 55
as.interval, difftime-method
(as.interval), 7
as.interval, Duration-method
(as.interval), 7
as.interval, Interval-method
(as.interval), 7
as.interval, logical-method
  (as.interval), 7
as.interval, numeric-method
  (as.interval), 7
as.interval, Period-method
  (as.interval), 7
as.interval, POSIXt-method
  (as.interval), 7
as.period, 5, 7, 8, 25
as.period, character-method (as.period), 8
as.period, difftime-method (as.period), 8
as.period, Duration-method (as.period), 8
as.period, Interval-method (as.period), 8
as.period, logical-method (as.period), 8
as.period, numeric-method (as.period), 8
as.period, Period-method (as.period), 8
as_date, 9
as_date, numeric-method (as_date), 9
as_date, POSIXt-method (as_date), 9
as_date_time (as_date), 9
as_date_time, ANY-method (as_date), 9
as_date_time, numeric-method (as_date), 9
as_date_time, POSIXt-method (as_date), 9
ceiling_date, 4
ceiling_date (round_date), 50

days_in_month, 15
days (quick_periods), 47
days (quick_durations), 46
day, 4, 13
day< (day), 13
days, 5, 42, 47, 55
days, (quick_periods), 47
days_in_month, 15
ddays, 5, 18, 48, 54
ddays (quick_durations), 46
diff, 24
diff, 56
d.microseconds (quick_durations), 46
dmilliseconds (quick_durations), 46
dminutes (quick_durations), 46
dminutes, 5, 18, 54
dmy, 3
dmy (ymd), 61
dmy_h (ymd_hms), 63
dmy_hms (ymd_hms), 63
dnanoseconds (quick_durations), 46
dpicoseconds (quick_durations), 46
dseconds, 5, 18, 54
dseconds (quick_durations), 46
dst, 4, 17
duration, 5, 6, 17, 33, 47, 55
Duration-class, 19

dweeks (quick_durations), 46
dweeks, 5, 18, 54
dyeas, 5
dyeas (quick_durations), 46
dym, 3
dym (ymd), 61
edays (Deprecated-lubridate), 16
ehours (Deprecated-lubridate), 16
emilliseconds (Deprecated-lubridate), 16
emilliseconds (Deprecated-lubridate), 16
eminutes (Deprecated-lubridate), 16
enanoseconds (Deprecated-lubridate), 16
epicoseconds (Deprecated-lubridate), 16
esseconds (Deprecated-lubridate), 16
eweeks (Deprecated-lubridate), 16
eyears (Deprecated-lubridate), 16

fast_strptime (parse_date_time), 37
fit_to_timeline, 19, 40
floor_date, 4
floor_date (round_date), 50
force_tz, 4, 20, 60, 64
format, 53
guess_formats, 21, 53, 54

here, 36
here (Deprecated-lubridate), 16
hm, 4, 35
hm (ms), 35
hms, 4, 65
hms (ms), 35
hour, 4, 23
hour< (hour), 23
INDEX

hours, 5, 42
hours (quick_periods), 47
instant (is.instant), 28
instants, 4
instants (is.instant), 28
int_aligns, 5
int_aligns (interval), 24
int_diff (interval), 24
int_end (interval), 24
int_end (interval), 24
int_end<- (interval), 24
int_flip, 5
int_flip (interval), 24
int_length (interval), 24
int_overlaps, 5
int_overlaps (interval), 24
int_shift, 5
int_shift (interval), 24
int_standardize (interval), 24
int_start (interval), 24
int_start<- (interval), 24
interval, 5, 7, 24, 55
Interval-class, 26
is.Date, 4, 27, 29
is_difftime, 5, 28, 30
is.duration, 5, 30
is.duration (duration), 17
is.instant, 4, 27, 28, 29, 30
is.interval, 5, 28, 30
is.interval (interval), 24
is.period, 5, 28, 30
is.period (period), 41
is.POSIXct (is.POSIXt), 29
is.POSIXlt (is.POSIXt), 29
is.POSIXt, 4, 27, 29, 29
is.timepoint (is.instant), 28
is.timespan, 5, 27–29, 30
isoweek (week), 59
isoyear, 59
isoyear (year), 60
lakers, 5, 30
leap_year, 5, 31
locales, 38, 62, 64
lubridate (lubridate-package), 3
lubridate-package, 3
make_date (make_datetime), 31
make_datetime, 31
make_difftime, 32
mday, 4, 14
mday (day), 13
mday<- (day), 13
mdy, 3
mdy (ymd), 61
mdy_h (ymd_hms), 63
mdy_hms (ymd_hms), 63
milliseconds (quick_periods), 47
milliseconds (quick_periods), 47
minute, 4, 33
minute<- (minute), 33
minutes, 5, 42, 55
minutes (quick_periods), 47
month, 4, 34
month<- (month), 34
months, 5, 42, 55
months.numeric (quick_periods), 47
ms, 4, 35, 35
myd, 3
myd (ymd), 61
nanoseconds (quick_periods), 47
new_difftime (Deprecated-lubridate), 16
new_duration (Deprecated-lubridate), 16
new_interval (Deprecated-lubridate), 16
new_period (Deprecated-lubridate), 16
now, 4, 36
olson_time_zones
  (Deprecated-lubridate), 16
OlsonNames, 10
origin, 4, 37
parse_date_time, 3, 37, 54, 62, 63, 65
parse_date_time2 (parse_date_time), 37
period, 5, 8, 9, 41, 41, 48, 50, 55
Period-class, 43
period_to_seconds, 44
picoseconds (quick_periods), 47
pm (am), 5
POSIXct, 66
POSIXlt, 66
POSIXt, 10
pretty_dates, 5, 44
qday (day), 13
qday<-(day), 13
quarter, 45
quick_durations, 46
quick_periods, 42, 47
rollback, 49, 66
round, 50, 51
round.POSIXt, 50
round_date, 4, 50
second, 4, 52
second<-(second), 52
seconds, 4, 5, 42, 55
seconds (quick_periods), 47
seconds_to_period (period_to_seconds), 44
semester (quarter), 45
stamp, 53
stamp_date (stamp), 53
stamp_time (stamp), 53
strptime, 4, 37, 38, 54
Sys.timezone, 21, 60
time_length, 56
time_length, Interval-method
  (time_length), 56
timespan, 54, 56
Timespan-class, 55
timespans (timespan), 54
timezones, 58
today, 4, 57
tz, 4, 57
tz<-(tz), 57
update.POSIXt (DateUpdate), 12
wday, 4
wday (day), 13
wday<-(day), 13
week, 4, 59
week<-(week), 59
weeks, 5, 42, 55
weeks (quick_periods), 47
with_tz, 4, 21, 58, 60
yday, 4, 14
yday (day), 13
yday<-(day), 13
ydm, 3
ydm (ymd), 61
ydm_h (ymd_hms), 63
ydm_hm (ymd_hms), 63
ydm_hms (ymd_hms), 63
year, 4, 60
year<-(year), 60
years, 5, 42, 55
years (quick_periods), 47
ymd, 3, 40, 61, 65
ymd_h (ymd_hms), 63
ymd_hm (ymd_hms), 63
ymd_hms, 3, 40, 63
yq (ymd), 61
yq (ymd), 61
ymd_hms, 3, 40, 63
yq (ymd), 61
ymd_hms, 3, 40, 63
yq (ymd), 61