Package 'lubridate'

November 3, 2017

Type Package

Version 1.7.1

Title Make Dealing with Dates a Little Easier

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.
Parts of the 'CCTZ' source code, released under the Apache 2.0 License,
are included in this package. See <https://github.com/google/cctz> for more
details.

Maintainer Vitalie Spinu <spinuvit@gmail.com>

License GPL-2

Depends methods, R (>= 3.0.0)

Imports stringr, Rcpp (>= 0.11),

LinkingTo Rcpp,

Suggests testthat, knitr, covr

Enhances chron, fts, timeSeries, timeDate, tis, tseries, xts, zoo

SystemRequirements A system with zoneinfo data (e.g.
/usr/share/zoneinfo) as well as a recent-enough C++11 compiler
(such as g++-4.8 or later). On Windows the zoneinfo included
with R is used.

VignetteBuilder knitr

LazyData true

Collate 'Dates.r' 'POSIXt.r' 'RcppExports.R' 'util.r' 'timespans.r'
'intervals.r' 'difftimes.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' 'hidden.r'
'instants.r' 'leap-years.r' 'ops-addition.r' 'ops-compare.r'
'ops-division.r' 'ops-integer-division.r' 'ops-m+.r'
'ops-modulo.r' 'ops-multiplication.r' 'ops-subtraction.r'
'package.r' 'parse.r' 'pretty.r' 'round.r' 'stamp.r' 'update.r'
'zzz.R'

RoxygenNote 6.0.1

URL http://lubridate.tidyverse.org,

https://github.com/tidyverse/lubridate

BugReports https://github.com/tidyverse/lubridate/issues

NeedsCompilation yes

Author Vitalie Spinu [aut, cre],
Garrett Grolemund [aut],
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 2017-11-03 12:26:50 UTC

R **topics documented:**

---

am   *Does date time occur in the am or pm?*

Description

Does date time occur in the am or pm?

Usage

```
am(x)

pm(x)
```

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)
```

---

as.duration   *Change an object to a duration*

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. 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](), [duration()]()

Examples

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

---

as.interval   *Change an object to an* 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, ...)
```

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

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 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](). 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](), [period()]()

Examples

```
span <- interval(ymd_hms("2009-01-01 00:00:00"), ymd_hms("2010-02-02 01:01:01")) #interval
as.period(span)
as.period(span, units = "day")
"397d 1H 1M 1S"
leap <- interval(ymd("2016-01-01"), ymd("2017-01-01"))
as.period(leap, unit = "days")
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")
as.period(dst, unit = "hours")
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

```
as_date(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 'character'
as_date(x, tz = NULL)

as_datetime(x, ...)

## 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 'character'
as_datetime(x, 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

```
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))
## need not suply origin
as_date(10)
```

---

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

```
date(x)

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

```
x <- ymd_hms("2012-03-26 23:12:13", tz = "America/New_York")
date(x)
as.Date(x) # by default as.Date assumes you want to know the date in UTC
as.Date(x, tz = "America/New_York")
date(x) <- as.Date("2000-01-02")
x
```

DateTimeUpdate   *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

```
## S3 method for class 'POSIXt'
update(object, ..., roll = FALSE,
  week_start = getOption("lubridate.week.start", 7), simple = NULL)
```

Arguments

object	a date-time object
...	named arguments: years, months, ydays, wdays, mdays, days, hours, minutes, seconds, tzs (time zone compnent)
roll	logical. If TRUE, and the resulting date-time lands on a non-existent civil time instant (DST, 29th February, etc.) roll the date till next valid point. When FALSE, the default, produce NA for non existing date-times.
week_start	week starting day (Default is 7, Sunday). Set lubridate.week.start option to control this.
simple	logical. Deprecated. Same as roll.

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

```
date <- ymd("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,
  week_start = getOption("lubridate.week.start", 7),
  locale = Sys.getlocale("LC_TIME"))

qday(x)
```

```
yday(x)

day(x) <- value

mday(x) <- value

qday(x) <- value

wday(x, week_start = getOption("lubridate.week.start", 7)) <- value

yday(x) <- value
```

Arguments

x	a POSIXct, POSIXlt, Date, chron, yearmon, yearqtr, zoo, zooreg, timeDate, xts, its, ti, jul, timeSeries, or fts 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.
week_start	day on which week starts following ISO conventions - 1 means Monday, 7 means Sunday (default). You can set lubridate.week.start option to control this parameter globally.
locale	locale to use for day names. Default to current locale.
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 or an ordered factor if label is TRUE.

See Also

[yday()](), [mday()]()

Examples

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

wday(ymd(080101))
```

```
wday(ymd(080101), label = TRUE, abbr = FALSE)
wday(ymd(080101), label = TRUE, abbr = TRUE)
wday(ymd(080101) + days(-2:4), label = TRUE, abbr = TRUE)

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
```

---

days_in_month *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

```
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

```
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

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

---

dst   *Get **d**aylight **s**avings **t**ime 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

x   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

```
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", ...)

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)

is.duration(x)
```

Arguments

num	the number or a character vector of time units. In string representation all unambiguous name units and abbreviations are supported; 'm' stands for month and 'M' for minutes, see examples. Fractional units are supported.
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   numeric value of the number of units to be contained in the duration.

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 date- times to create a user interface similar to object oriented programming.

Value

a duration object

See Also

as.duration() 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(mins = 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("day 2 sec") > "day 1sec"

## ELEMENTARY CONSTRUCTORS:

dseconds(1)
dminutes(3.5)

x <- ymd_hms("2009-08-03", tz="America/Chicago")
x + ddays(1) + dhours(6) + dminutes(30)
```

```
x + ddays(100) - dhours(8)

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

dweeks(1) - ddays(7)
c(1:3) * dhours(1)

# compare DST handling to durations
boundary <- ymd_hms("2009-03-08 01:59:59", tz="America/Chicago")
boundary + days(1) # period
boundary + ddays(1) # duration
is.duration(as.Date("2009-08-03")) # FALSE
is.duration(duration(days = 12.4)) # TRUE
```

Duration-class	*Duration class*

Description

Duration is an S4 class that extends the [Timespan] 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 numberline. They allow exact comparisons with other durations. See [Period] 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

```
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

```
## 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
## [1] "2012-11-04 02:00:00 CDT" Doesn't exist because clocks "fall back" to 1:00 CST
## [2] "2012-11-04 00:05:00 CST" Times are still CDT, not CST at this instant
## [3] "2010-03-14 02:00:00 CDT" DST gap
## [4] "2012-11-04 01:59:59 CDT" Does exist, but has deceptive internal structure

fit_to_timeline(tricky)
## Returns:
## [1] "2012-11-04 02:00:00 CST" instant paired with correct tz & DST combination
## [2] "2012-11-04 00:05:00 CDT" instant paired with correct tz & DST combination
## [3] NA - fake time changed to NA (compare to as.POSIXct(tricky))
## [4] "2012-11-04 01:59:59 CDT" -real instant, left as is

fit_to_timeline(tricky, simple = TRUE)
## Returns valid time-dates by extrapolating CDT and CST zones:
## [1] "2012-11-04 01:00:05 CST" "2012-11-04 01:05:00 CDT"
## [3] "2010-03-14 03:05:00 CDT" "2012-11-04 01:59:59 CDT"
```

```
## End(Not run)
```

---

force_tz   *Replace time zone to create new date-time*

Description

force_tz returns the date-time that has the same clock time as input time, but in the new time zone. force_tzs is the parallel version of force_tz in a sense that every element from time argument is matched with the corresponding time zone in tzones argument.

Usage

```
force_tz(time, tzone = "", roll = FALSE)

force_tzs(time, tzones, tzone_out = "UTC", roll = FALSE)
```

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.
roll	logical. If TRUE, and time falls into the DST-break, assume the next valid civil time, otherwise return NA. See examples.
tzones	character vector of timezones to be "enforced" on time time stamps. If time and tzones lengths differ, the smaller one is recycled in accordance with usual R conventions.
tzone_out	timezone of the returned date-time vector

Details

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.

As R date-time vectors cannot hold elements with non-uniform time zones, force_tzs returns a vector with time zone tzone_out, UTC by default.

Value

a POSIXct object in the updated time zone

See Also

with_tz(), local_time()

Examples

```
x <- ymd_hms("2009-08-07 00:00:01", tz = "America/New_York")
force_tz(x, "UTC")
force_tz(x, "Europe/Amsterdam")

## DST skip:

y <- ymd_hms("2010-03-14 02:05:05 UTC")
force_tz(y, "America/New_York", roll=FALSE)
force_tz(y, "America/New_York", roll=TRUE)

## Heterogeneous time-zones:

x <- ymd_hms(c("2009-08-07 00:00:01", "2009-08-07 01:02:03"))
force_tzs(x, tzones = c("America/New_York", "Europe/Amsterdam"))
force_tzs(x, tzones = c("America/New_York", "Europe/Amsterdam"), tzone_out = "America/New_York")

x <- ymd_hms("2009-08-07 00:00:01")
force_tzs(x, tzones = c("America/New_York", "Europe/Amsterdam"))
```

---

guess_formats   *Guess possible date-times formats from a character vector*

Description

Guess possible date-times formats from a 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, weekdays 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, 1July2004 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, abreviated and full names
guess_formats(x, c("mdY", "BdY", "Bdy", "bdY", "bdy"), print_matches = TRUE)
guess_formats(x, c("dmy", "dbY", "dBy", "dBY"), print_matches = TRUE)


guess_formats(x, c("dBY HMS", "dbY HMS", "dmyHMS", "BdY H"), print_matches = TRUE)
```

```
guess_formats(x, c("ymd HMS"), 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, time-Date, 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 the 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](#) 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_end() and int_end<-() are accessors the end date of an interval. Note that changing the end date of an interval will change the length of the interval, since the start 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 = tz(start))

start %--% end

is.interval(x)

int_start(int)

int_start(int) <- value

int_end(int)

int_end(int) <- value

int_length(int)

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() – an Interval object

int_overlaps() – logical, TRUE if int1 and int2 overlap by at least one second. FALSE otherwise

int_aligns() – 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, `as.interval()`, `%within%`

Examples

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

date1 <- ymd_hms("2009-03-08 01:59:59")
date2 <- ymd_hms("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)
```

Interval-class   *Interval class*

Description

Interval is an S4 class that extends the [Timespan](#) 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](#) or [Duration](#) 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

```
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

```
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

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

```
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

Originally taken from www.basketballgeek.com/data/.

---

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
```

---

local_time   *Get local time from a date-time vector.*

Description

local_time retrieves day clock time in specified time zones. Computation is vectorized over both
dt and tz arguments, the shortest is recycled in accordance with standard R rules.

Usage

```
local_time(dt, tz = NULL, units = "secs")
```

Arguments

dt   a date-time object.

tz   a character vector of timezones for which to compute the local time.

units   passed directly to as.difftime().

Examples

```
x <- ymd_hms(c("2009-08-07 01:02:03", "2009-08-07 10:20:30"))
local_time(x, units = "secs")
local_time(x, units = "hours")
local_time(x, "Europe/Amsterdam")
local_time(x, "Europe/Amsterdam") == local_time(with_tz(x, "Europe/Amsterdam"))

x <- ymd_hms("2009-08-07 01:02:03")
local_time(x, c("America/New_York", "Europe/Amsterdam", "Asia/Shanghai"), unit = "hours")
```

---

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

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

---

make_difftime   *Create a difftime object.*

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

```
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

```
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, time-Date, xts, its, ti, jul, timeSeries, and fts objects.

Usage

```
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, time-Date, xts, its, ti, jul, timeSeries, and fts objects.

Usage

```
month(x, label = FALSE, abbr = TRUE, locale = Sys.getlocale("LC_TIME"))

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.
locale	for month, locale to use for month names. Default to current locale.
value	a numeric object

Value

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

Examples

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

month(ymd(080101))
month(ymd(080101), label = TRUE)
month(ymd(080101), label = TRUE, abbr = FALSE)
month(ymd(080101) + months(0:11), label = TRUE)
```

---

ms *Parse periods with* **h***our,* **m***inute, and* **s***econd components*

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

```
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 evalueates 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

```
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")
```

---

now   *The current time*

Description

The current time

Usage

```
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

```
now()
now("GMT")
now("")
now() == now() # would be TRUE if computer processed both at the same instant
now() < now() # TRUE
now() > now() # FALSE
```

---

origin   *1970-01-01 UTC*

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	*User friendly date-time parsing functions*

Description

parse_date_time() parses an input vector into POSIXct date-time object. It differs from base::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 formating argument is refered 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 base::strptime().

Usage

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

parse_date_time2(x, orders, tz = "UTC", exact = FALSE, lt = FALSE,
  cutoff_2000 = 68L)

fast_strptime(x, format, tz = "UTC", lt = TRUE, cutoff_2000 = 68L)
```

Arguments

x	a character or numeric vector of dates
orders	a character vector of date-time formats. Each order string is series of formatting characters as listed base::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 the 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. **NOTE:** The ymd family of functions are based on strptime() which currently fails to parse %Y-%m formats.
quiet	logical. When TRUE progress messages are not printed, and "no formats found" error is surpresed 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 formating tockens (%) are selected and %Y counts as 2.5 tockens (so that it has a priority over %y%m). Se examples.
exact	logical. If TRUE, the orders parameter is interpreted as an exact strptime() format and no training or guessing are performed (i.e. train, drop parameters are irrelevant).
train	logical, default TRUE. Whether to train formats on a subset of the input vector. The result of this is that supplied orders are sorted according to performance on this training set, which commonly results in increased performance. Please note that even when train is FALSE (and exact is FALSE) guessing of the actual formats is still performed on a pseudo-random subset of the original input vector. This might result in All formats failed to parse error. See notes below.
drop	logical, default FALSE. Whether to drop formats that didn't match on the training set. If FALSE, unmatched on the training set formats are tried as a last resort at the end of the parsing queue. Applies only when train=TRUE. Seating this parameter to TRUE might slightly speed up parsing in situations involving many formats. Prior to v1.7.0 this parameter was implicitly TRUE, which resulted in occasional surprising behavior when rare patterns where not present in the training set.
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.
cutoff_2000	integer. For y format, two-digit numbers smaller or equal to cutoff_2000 are parsed as 20th's century, 19th's otherwise. Available only for functions relying on lubridates internal parser.
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() selects (guesses) format-orders based on a training sub-set of the input strings. After guessing the formats are ordered accord-

ing to the performance on the training set and applied recursively on the entire input vector. You can disable training with `train=FALSE`.

`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 "*". The fast parsers `parse_date_time2()` and `fast_strptime()` accept only formats marked with "!".

a Abbreviated weekday name in the current locale. (Also matches full name)

A Full weekday name in the current locale. (Also matches abbreviated name).

    You don't need to specify a and A formats explicitly. Wday is automatically handled if `preproc_wday = TRUE`

b! Abbreviated or full month name in the current locale. The C parser currently understands only English month names.

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 the parsed month if m format is present.

m!* Month as decimal number (01–12 or 1–12). For `parse_date_time`. As a **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 the **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).

OS 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
    strptime doesn't fully support ISO8601 this format is implemented as an union of 4 orders: Ou
    (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.

Om!* Matches numeric month and English alphabetic months (Both, long and abbreviated forms).

Op!* 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()
perform 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 an 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

```
## ** 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"

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

## ** 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 **
x <- c("2011-12-31 12:59:59", "2010-01-01 12:11", "2010-01-01 12", "2010-01-01")
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(c('12/17/1996 04:00:00','4/18/1950 0130'),
                c('%m/%d/%Y %I:%M:%S','%m/%d/%Y %H%M'), exact = TRUE)

## ** quarters and partial dates **
parse_date_time(c("2016.2", "2016-04"), orders = "Yq")
parse_date_time(c("2016", "2016-04"), orders = c("Y", "Ym"))

## ** fast parsing **
## Not run:
  options(digits.secs = 3)
  ## random times between 1400 and 3000
  tt <- as.character(.POSIXct(runif(1000, -17987443200, 32503680000)))
  tt <- rep.int(tt, 1000)

  system.time(out <- as.POSIXct(tt, tz = "UTC"))
  system.time(out1 <- ymd_hms(tt)) # constant overhead on long vectors
  system.time(out2 <- parse_date_time2(tt, "YmdHMOS"))
  system.time(out3 <- fast_strptime(tt, "%Y-%m-%d %H:%M:%OS"))

  all.equal(out, out1)
  all.equal(out, out2)
  all.equal(out, out3)

## End(Not run)

## ** how to use `select_formats` argument **
## By default %Y has precedence:
parse_date_time(c("27-09-13", "27-09-2013"), "dmy")

## to give priority to %y format, define your own select_format function:

my_select <-   function(trained, drop=FALSE, ...){
   n_fmts <- nchar(gsub("[^%]", "", names(trained))) + grepl("%y", names(trained))*1.5
   names(trained[ which.max(n_fmts) ])
}

parse_date_time(c("27-09-13", "27-09-2013"), "dmy", select_formats = my_select)

## ** invalid times with "fast" parcing **
parse_date_time("2010-03-14 02:05:06",  "YmdHMS", tz = "America/New_York")
parse_date_time2("2010-03-14 02:05:06",  "YmdHMS", tz = "America/New_York")
parse_date_time2("2010-03-14 02:05:06",  "YmdHMS", tz = "America/New_York", lt = TRUE)
```

---

period   *Create a period object*

Description

period() creates a period object with the specified values.

Usage

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

is.period(x)

seconds(x = 1)

minutes(x = 1)

hours(x = 1)

days(x = 1)

weeks(x = 1)

years(x = 1)

milliseconds(x = 1)

microseconds(x = 1)

nanoseconds(x = 1)

picoseconds(x = 1)

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

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, "m" stands for months, "M" for minutes; see examples. Fractional units are supported but the fractional part is always converted to seconds.

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	Any R object for `is.periods` and a numeric value of the number of units for elementary constructors. With the exception of seconds(), x must be an integer.
abbreviate	Ignored. For consistency with S3 generic in base namespace.

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

Value

a period object

See Also

Period, period(), %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"))
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")
period("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("day 2 sec") > "day 1sec"

### ELEMENTARY CONSTRUCTORS

x <- ymd("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)
hours(1:3)

#sequencing
y <- ymd(090101) # "2009-01-01 CST"
y + months(0:11)

# compare DST handling to durations
boundary <- ymd_hms("2009-03-08 01:59:59", tz="America/Chicago")
boundary + days(1) # period
boundary + ddays(1) # duration
is.period(as.Date("2009-08-03")) # FALSE
is.period(period(months= 1, days = 15)) # TRUE
```

---

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 365.25 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, ...)
```

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

```
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 halfs.

Usage

```
quarter(x, with_year = FALSE, fiscal_start = 1)

semester(x, with_year = FALSE)
```

Arguments

x	a date-time object of class POSIXct, POSIXlt, Date, chron, yearmon, yearqtr, zoo, zooreg, 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.
fiscal_start	numeric indicating the starting month of a fiscal year

Value

numeric

Examples

```
x <- ymd(c("2012-03-26", "2012-05-04", "2012-09-23", "2012-12-31"))
quarter(x)
quarter(x, with_year = TRUE)
quarter(x, with_year = TRUE, fiscal_start = 11)
semester(x)
semester(x, with_year = TRUE)
```

---

rollback   *Roll back date to last day of previous month*

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 meaningfull specifications in English language are supported - secs, min, mins, 2 minutes, 3 years etc.

Rounding to fractional seconds is supported. Please note that rounding to fractions smaller than 1s can lead to large precision errors due to the floating point representation of the POSIXct objects. See examples.

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 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",
  week_start = getOption("lubridate.week.start", 7))

floor_date(x, unit = "seconds",
  week_start = getOption("lubridate.week.start", 7))

ceiling_date(x, unit = "seconds", change_on_boundary = NULL,
  week_start = getOption("lubridate.week.start", 7))
```

Arguments

x	a vector of date-time objects
unit	a character string specifying a time unit or a multiple of a unit to be rounded to. Valid base units are second, minute, hour, day, week, month, bimonth, quarter, season, halfyear and year. Arbitrary unique English abbreviations as in the [period()](#) constructor are allowed. Rounding to multiple of units (except weeks) is supported.
week_start	when unit is weeks specify the reference day; 7 being Sunday.
change_on_boundary	
	If NULL (the default) don't change instants on the boundary (ceiling_date(ymd_hms('2000-01-01 00: 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 possible. This is done by first rounding to an instant and then converting to the original class by usual R conventions.

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 **next** closest rounding unit boundary. For example, if the rounding unit is month then next closest boundary of 2000-01-01 is 2000-02-01 00:00:00.

   The motivation for this is that the "partial" 2000-01-01 is conceptually an interval (2000-01-01 00:00:00 – 2000-01-02 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.

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

3. If rounding unit is smaller than a day, return the instant from step 2 (POSIXct), otherwise convert to and return a Date object.

See Also

base::round()

Examples

```
## print fractional seconds
options(digits.secs=6)

x <- ymd_hms("2009-08-03 12:01:59.23")
round_date(x, ".5s")
round_date(x, "sec")
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 <- ymd_hms("2009-08-03 12:01:59.23")
floor_date(x, ".1s")
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, "season")
floor_date(x, "halfyear")
floor_date(x, "year")

x <- ymd_hms("2009-08-03 12:01:59.23")
ceiling_date(x, ".1 sec") # imprecise representation at 0.1 sec !!!
ceiling_date(x, "second")
ceiling_date(x, "minute")
ceiling_date(x, "5 mins")
ceiling_date(x, "hour")
ceiling_date(x, "day")
ceiling_date(x, "week")
ceiling_date(x, "month")
ceiling_date(x, "bimonth") == ceiling_date(x, "2 months")
ceiling_date(x, "quarter")
ceiling_date(x, "season")
ceiling_date(x, "halfyear")
```

```
ceiling_date(x, "year")

## As of R 3.4.2 POSIXct printing of fractional numbers is wrong
as.POSIXct("2009-08-03 12:01:59.3") ## -> "2009-08-03 12:01:59.2 CEST"
ceiling_date(x, ".1 sec") ## -> "2009-08-03 12:01:59.2 CEST"

x <- ymd("2000-01-01")
ceiling_date(x, "month")
ceiling_date(x, "month", change_on_boundary = TRUE)
```

---

second   *Get/set seconds component of a date-time*

Description

Date-time must be a POSIXct, POSIXlt, Date, Period, chron, yearmon, yearqtr, zoo, zooreg, time-Date, 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
```

stamp   *Format dates and times based on human-friendly templates*

Description

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

Usage

```
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 the 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 provide 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

See Also

guess_formats(), parse_date_time(), strptime()

Examples

```
D <- ymd("2010-04-05") - days(1:5)
stamp("March 1, 1999")(D)
sf <- stamp("Created on Sunday, Jan 1, 1999 3:34 pm")
sf(D)
stamp("Jan 01")(D)
stamp("Sunday, May 1, 2000", locale = "en_US")(D)
stamp("Sun Aug 5")(D) #=> "Sun Aug 04" "Sat Aug 04" "Fri Aug 04" "Thu Aug 04" "Wed Aug 03"
stamp("12/31/99")(D)              #=> "06/09/11"
stamp("Sunday, May 1, 2000 22:10", locale = "en_US")(D)
stamp("2013-01-01T06:00:00Z")(D)
stamp("2013-01-01T00:00:00-06")(D)
stamp("2013-01-01T00:00:00-08:00")(force_tz(D, "America/Chicago"))
```

---

timespan	*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.

- durations record the exact number of seconds in a time span. They measure the exact passage of time but do not always align with human measurements like hours, months and years.

- periods record the change in the clock time between two date-times. They are measured in human units: years, months, days, hours, minutes, and seconds.

- intervals are time spans bound by two real date-times. Intervals can be accurately converted to periods and durations.

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

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

date3 <- ymd_hms("2009-01-31 01:00:00")
date3 + c(0:11) * months(1)
```

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

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

months(6) + days(1)
```

---

time_length   *Compute the exact length of a time span*

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 object and unit are years or months, time_length() takes into account the fact that all months and years don't have the same number of days.

When x is a Duration, Period 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")

# Age at last anniversary
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.

Usage

```
tz(x)

tz(x) <- value
```

Arguments

x   a date-time object of class a POSIXct, POSIXlt, Date, chron, yearmon, yearqtr, zoo, zooreg, 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 base::timezones() or base::DateTimeClasses.

Value

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

Examples

```
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)
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 reoccurring leap week.

epiweek() is the US CDC version of epidemiological week. It follows same rules as isoweek() but starts on Sunday. In other parts of the world the convention is to start epidemiological weeks on Monday, which is the same as isoweek.

Usage

```
week(x)

week(x) <- value

isoweek(x)

epiweek(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

<http://en.wikipedia.org/wiki/ISO_week_date> <http://www.cmmcp.org/epiweek.htm>

See Also

[isoyear()](#)

Examples

```
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 <- ymd_hms("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, time-Date, xts, its, ti, jul, timeSeries, and fts objects.

isoyear() returns years according to the ISO 8601 week calendar.

epiyear() returns years according to the epidemilogical week calendars.

Usage

```
year(x)

year(x) <- value

isoyear(x)

epiyear(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

References

http://en.wikipedia.org/wiki/ISO_week_date http://www.cmmcp.org/epiweek.htm

Examples

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

---

ymd   *Parse dates with **y**ear, **m**onth, and **d**ay components*

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

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

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

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

myd(..., 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 evalueates without displaying customary messages.

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 the `truncated` parameter is non-zero, the ymd functions also check for truncated formats. For example `ymd()` with `truncated = 2` will also parse incomplete dates like `2012-06` and `2012`.

NOTE: The ymd family of functions are based on `parse_date_time()` and thus directly drop to the 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.

Examples

```
x <- c("09-01-01", "09-01-02", "09-01-03")
ymd(x)
x <- c("2009-01-01", "2009-01-02", "2009-01-03")
ymd(x)
ymd(090101, 90102)
now() > ymd(20090101)
## TRUE
dmy(010210)
mdy(010210)

## 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", "200901 !!! 07")
ymd(x)

## 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:
## Not run: ymd("201002-01", "201002-1", "20102-1")
dmy("0312-2010", "312-2010")
```

```
## End(Not run)
```

---

ymd_hms
*Parse date-times with **y**ear, **m**onth, and **d**ay, **h**our, **m**inute, and **s**econd components.*

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

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

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

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

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

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

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

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

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

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

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

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

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

Arguments

`...`	a character vector of dates in year, month, day, hour, minute, second format
`quiet`	logical. When TRUE function evalueates 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 the `truncated` parameter is non-zero, the `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: The 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)`.

Value

a vector of POSIXct date-time objects

See Also

- ymd(), hms()
- parse_date_time() for the 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)


## ** heterogenuous formats **
x <- c(20100101120101, "2009-01-02 12-01-02", "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: 20090107 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",
"2013-01-24 19:39:07.880", "2013-01-24 19:39:07.880-06:00",
"2013-01-24 19:39:07.880-06", "2013-01-24 19:39:07.880Z"))

## ** 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 **
x <- c("2011-12-31 12:59:59", "2010-01-01 12:11", "2010-01-01 12", "2010-01-01")
ymd_hms(x, truncated = 3)
x <- c("2011-12-31 12:59", "2010-01-01 12", "2010-01-01")
ymd_hm(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:
## Not run:
ymd_hm("20100201 07-01", "20100201 07-1", "20100201 7-01")
## End(Not run)
```

%m+%	*Add and subtract months to a date without exceeding the last day of the new month*

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

```
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

```
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