Package 'lubridate'

December 31, 2013

Imports plyr, stringr, memoise

Maintainer Garrett Grolemund <garrett@rstudio.com>

License GPL

Title Make dealing with dates a little easier

LazyData true

Type Package

Description Lubridate makes it easier to work with dates and times by providing functions to identify and parse date-time data, extract and modify components of a date-time (years, months, days, hours, minutes, and seconds), perform accurate math on date-times, handle time zones and Daylight Savings Time. Lubridate has a consistent, memorable syntax, that makes working with dates fun instead of frustrating.

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

Version 1.3.3

Depends methods, R (>= 3.0.0)

Suggests testthat, knitr

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

VignetteBuilder knitr

Collate 'time-zones.r' 'POSIXt.r' 'timespans.r' 'numeric.r' 'util.r' 'durations.r' 'periods.r' 'intervals.r' 'difftimes.r' 'Dates.r' 'coercion.r' 'ops-addition.r' 'ops-division.r' 'ops-integer-division.r' 'ops-modulo.r' 'ops-multiplication.r' 'ops-subtraction.r' 'accessors-day.r' 'accessors-dst.r' 'accessors-hour.r' 'accessors-minute.r' 'accessors-month.r' 'accessors-second.r' 'accessors-tz.r' 'accessors-week.r' 'accessors-year.r' 'am-pm.r' 'decimal-dates.r' 'epochs.r' 'help.r' 'instants.r' 'leap-years.r' 'parse.r' 'pretty.r' 'round.r' 'update.r' 'data.r' 'guess.r' 'stamp.r' 'ops-%m+%.r' 'accessors-quarter.r'

Author Garrett Grolemund [aut, cre],Hadley Wickham [aut],Vitalie Spinu [ctb],Imanuel Constigan [ctb],Chel Hee Lee [ctb],Richard Cotton [ctb],Ian Lyttle [ctb],Winston Chang [ctb]

NeedsCompilation yes

Repository CRAN

Date/Publication 2013-12-31 16:24:04

R topics documented:

add_epoch_to_date *Add epochs to dates*

Description

Method for adding epochs to dates. Epochs will be implemented in a later version of lubridate.

Usage

```
add_epoch_to_date(date, epoch)
```

Arguments

date	a date-time object to be used as the reference time from which future epochs will be counted
epoch	an epoch object that decribes the number and type of epochs to be used to define a new date-time

Value

the date-time that occurs that specified number of epochs after the original date-time

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

Description

Does date time occur in the am or pm?

Usage

```
am(x)
```

Arguments

x	a date-time object

Value

TRUE or FALSE depending on whethe 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

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`.

as.duration.period displays the message "estimate only: convert periods to intervals for accuracy" by default. You can turn this message off by setting the global `lubridate.verbose` option to FALSE with `options(lubridate.verbose = FALSE)`.

Value

A duration object

See Also

`Duration-class`, `new_duration`

Examples

```
span <- interval(ymd("2009-01-01"), ymd("2009-08-01")) #interval
# "2009-01-01 UTC--2009-08-01 UTC"
as.duration(span)
# 18316800s (~212 days)
as.duration(10) # numeric
# 10s
```

`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`, `new_interval`

Examples

```
diff <- new_difftime(days = 31) #difftime
as.interval(diff, ymd("2009-01-01"))
# 2009-01-01 UTC--2009-02-01 UTC
as.interval(diff, ymd("2009-02-01"))
# 2009-02-01 UTC--2009-03-04 UTC

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

```
# 2009-02-01 UTC--2009-03-04 UTC

per <- new_period(months = 1) #period
as.interval(per, ymd("2009-01-01"))
# 2009-01-01 UTC--2009-02-01 UTC
as.interval(per, ymd("2009-02-01"))
# 2009-02-01 UTC--2009-03-01 UTC

as.interval(3600, ymd("2009-01-01")) #numeric
# 2009-01-01 UTC--2009-01-01 01:00:00 UTC
```

`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 `new_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.

as.period.difftime and as.period.duration display the message "estimate only: convert difftimes (or duration) to intervals for accuracy" by default. You can turn this message off by setting the global `lubridate.verbose` option to FALSE with `options(lubridate.verbose = FALSE)`.

Value

a period object

See Also

Period-class, new_period

Examples

```
span <- new_interval(as.POSIXct("2009-01-01"), as.POSIXct("2010-02-02 01:01:01")) #interval
# 2009-01-01 CST--2010-02-02 01:01:01 CST
as.period(span)
# "1y 1m 1d 1H 1M 1S"
as.period(span, units = "day")
"397d 1H 1M 1S"
leap <- new_interval(ymd("2016-01-01"), ymd("2017-01-01"))
# 2016-01-01 UTC--2017-01-01 UTC
as.period(leap, unit = "days")
# "366d 0H 0M 0S"
as.period(leap, unit = "years")
# "1y 0m 0d 0H 0M 0S"
dst <- new_interval(ymd("2016-11-06", tz = "America/Chicago"),
ymd("2016-11-07", tz = "America/Chicago"))
# 2016-11-06 CDT--2016-11-07 CST
# as.period(dst, unit = "seconds")
# "86400S"
as.period(dst, unit = "hours")
# "24H 0M 0S"
```

---

ceiling_date *Round date-times up.*

Description

ceiling_date takes a date-time object and rounds it up to the nearest integer value of the specified time unit. Users can specify whether to round up to the nearest second, minute, hour, day, week, month, or year.

Usage

```
ceiling_date(x,
  unit = c("second", "minute", "hour", "day", "week", "month", "year"))
```

Arguments

x	a vector of date-time objects
unit	a character string specifying the time unit to be rounded to. Should be one of "second","minute","hour","day", "week", "month", or "year."

Value

x with the appropriate units rounded up

See Also

floor_date, round_date

Examples

```
x <- as.POSIXct("2009-08-03 12:01:59.23")
ceiling_date(x, "second")
# "2009-08-03 12:02:00 CDT"
ceiling_date(x, "minute")
# "2009-08-03 12:02:00 CDT"
ceiling_date(x, "hour")
# "2009-08-03 13:00:00 CDT"
ceiling_date(x, "day")
# "2009-08-04 CDT"
ceiling_date(x, "week")
# "2009-08-09 CDT"
ceiling_date(x, "month")
# "2009-09-01 CDT"
ceiling_date(x, "year")
# "2010-01-01 CST"
```

DateUpdate *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.

Arguments

object	a date-time object
years	a value to substitute for the date's year component
months	a value to substitute for the date's month component
ydays	a value to substitute for the date's yday component
wdays	a value to substitute for the date's wday component
mdays	a value to substitute for the date's mday component
days	a value to substitute for the date's mday component
hours	a value to substitute for the date's hour component
minutes	a value to substitute for the date's minute component
seconds	a value to substitute for the date's second component
tzs	a value to substitute for the date's tz component
...	...

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 <- as.POSIXlt("2009-02-10")
update(date, year = 2010, month = 1, mday = 1)
# "2010-01-01 CST"

update(date, year =2010, month = 13, mday = 1)
# "2011-01-01 CST"

update(date, minute = 10, second = 3)
# "2009-02-10 00:10:03 CST"
```

---

date_decimal *Converts a decimal to a date.*

Description

Converts a decimal to a date.

Usage

```
date_decimal(decimal, tz = NULL)
```

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"
```

---

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
```

dseconds *Quickly create exact time spans.*

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)
```

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.

Value

a duration object

See Also

duration, new_duration, days

Examples

```
dseconds(1)
# 1s
dminutes(3.5)
# 210s (~3.5 minutes)

x <- as.POSIXct("2009-08-03")
# "2009-08-03 CDT"
x + ddays(1) + dhours(6) + dminutes(30)
# "2009-08-04 06:30:00 CDT"
x + ddays(100) - dhours(8)
# "2009-11-10 15:00:00 CST"

class(as.Date("2009-08-09") + ddays(1)) # retains Date class
# "Date"
as.Date("2009-08-09") + dhours(12)
# "2009-08-09 12:00:00 UTC"
class(as.Date("2009-08-09") + dhours(12))
# "POSIXct" "POSIXt"
# converts to POSIXt class to accomodate time units

dweeks(1) - ddays(7)
# 0s
c(1:3) * dhours(1)
# 3600s (~1 hours)  7200s (~2 hours)  10800s (~3 hours)
#
# compare DST handling to durations
boundary <- as.POSIXct("2009-03-08 01:59:59")
# "2009-03-08 01:59:59 CST"
boundary + days(1) # period
# "2009-03-09 01:59:59 CDT" (clock time advances by a day)
boundary + ddays(1) # duration
# "2009-03-09 02:59:59 CDT" (clock time corresponding to 86400 seconds later)
```

---

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

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. duration provides the behavior of new_duration in a way that is more suitable for automating within a function.

Usage

```
duration(num = 0, units = "seconds")
```

Arguments

num	the number of time units to include in the duration
units	a character string that specifies the type of units that num refers to.

Details

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.

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

new_duration, as.duration

Examples

```
duration(90, "seconds")
# 90s
duration(1.5, "minutes")
# 90s
duration(-1, "days")
# -86400s (~-1 days)
```

Duration-class *Duration class*

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

```
fit_to_timeline(lt, class = "POSIXct")
```

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.

Value

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

Examples

```
## Not run:
tricky <- structure(list(sec = c(0, 0, 0, -1), min = c(0L, 5L, 0L, 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, 1L, 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" Doesn't exist
##because clocks "spring forward" past this time
## for daylight savings

## [4] "2012-11-04 01:59:59 CDT" Does exist, but
## has deceptive internal structure

fit_to_timeline(tricky)
```

```
## [1] "2012-11-04 02:00:00 CST" instant paired
## with correct timezone & DST combination

## [2] "2012-11-04 00:05:00 CDT" instant paired
## with correct timezone & 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

## End(Not run)
```

`floor_date` *Round date-times down.*

Description

floor_date takes a date-time object and rounds it down to the nearest integer value of the specified time unit. Users can specify whether to round down to the nearest second, minute, hour, day, week, month, or year.

Usage

```
floor_date(x,
  unit = c("second", "minute", "hour", "day", "week", "month", "year"))
```

Arguments

`x`	a vector of date-time objects
`unit`	a character string specifying the time unit to be rounded to. Should be one of "second","minute","hour","day", "week", "month", or "year."

Value

x with the appropriate units floored

See Also

`ceiling_date`, `round_date`

Examples

```
x <- as.POSIXct("2009-08-03 12:01:59.23")
floor_date(x, "second")
# "2009-08-03 12:01:59 CDT"
floor_date(x, "minute")
# "2009-08-03 12:01:00 CDT"
floor_date(x, "hour")
# "2009-08-03 12:00:00 CDT"
floor_date(x, "day")
```

```
# "2009-08-03 CDT"
floor_date(x, "week")
# "2009-08-02 CDT"
floor_date(x, "month")
# "2009-08-01 CDT"
floor_date(x, "year")
# "2009-01-01 CST"
```

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

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, or chron date-time object.
`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")
# "2009-08-07 00:00:01 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 (also checks en_US)
preproc_wday	whether to preprocess week days names. Internal optimization used by ymd_hms family of functions. If true week days are substituted with 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)
```

`here` *The current time in your local timezone*

Description

The current time in your local timezone

Usage

```
here()
```

Value

the current date and time as a POSIXct object

See Also

now

Examples

```
here()
```

---

hm *Create a period with the specified number of hours and minutes*

Description

Transforms a character or numeric vectors into a period object with the specified number of hours and minutes. Arbitrary non-numeric text can separate hours and minutes. After hours and minutes have been parsed, the remaining input is ignored.

Usage

```
hm(..., quiet = FALSE)
```

Arguments

...	character or numeric vectors of hour minute pairs
quiet	logical. When TRUE function evalueates without displaying customary messages.

Value

a vector of class Period

See Also

hms, ms

Examples

```
x <- c("09:10", "09:02", "1:10")
hm(x)
# [1] 9 hours and 10 minutes   9 hours and 2 minutes   1 hour and 10 minutes
hm("7 6")
# [1] 7 hours and 6 minutes
hm("6,5")
# [1] 6 hours and 5 minutes
```

hms *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 separators. After hours, minutes and seconds have been parsed, the remaining input is ingored.

Usage

```
hms(..., quiet = FALSE)
```

Arguments

...	a character vector of hour minute second triples
quiet	logical. When TRUE function evalueates without displaying customary messages.

Value

a vector of period objects

See Also

hm, ms

Examples

```
x <- c("09:10:01", "09:10:02", "09:10:03", "Collided at 9:20:04 pm")
hms(x)
# [1] 9 hours, 10 minutes and 1 second
# [2] 9 hours, 10 minutes and 2 seconds
# [3] 9 hours, 10 minutes and 3 seconds

hms("7 6 5", "3-23---2", "2 : 23 : 33")
## [1] 7 hours, 6 minutes and 5 seconds
## [2] 3 hours, 23 minutes and 2 seconds
## [3] 2 hours, 23 minutes and 33 seconds
```

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)
```

Arguments

x	a date-time object

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-class *Interval class*

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.

int_aligns *Test if two intervals share an endpoint*

Description

int_aligns tests for the case where two intervals begin or end at the same moment when arranged chronologically. The direction of each interval is ignored. int_align tests whether the earliest or latest moments of each interval occur at the same time.

Usage

```
int_aligns(int1, int2)
```

Arguments

int1	an Interval object
int2	an Interval object

Value

Logical. TRUE if int1 and int2 begin or end on the same moment. FALSE otherwise.

Examples

```
int1 <- new_interval(ymd("2001-01-01"), ymd("2002-01-01"))
# 2001-01-01 UTC--2002-01-01 UTC
int2 <- new_interval(ymd("2001-06-01"), ymd("2002-01-01"))
# 2001-06-01 UTC--2002-01-01 UTC
int3 <- new_interval(ymd("2003-01-01"), ymd("2004-01-01"))
# 2003-01-01 UTC--2004-01-01 UTC

int_aligns(int1, int2) # TRUE
int_aligns(int1, int3) # FALSE
```

int_diff *Extract the intervals within a vector of date-times*

Description

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

```
int_diff(times)
```

Arguments

times	A vector of POSIXct, POSIXlt or Date class date-times

Value

An interval object that contains the n-1 intervals between the n date-tiwe in times

Examples

```
dates <- now() + days(1:10)
int_diff(dates)
```

---

int_end *Access and change the end date of an interval*

Description

Note that changing the end date of an interval will change the length of the interval, since the start date will remain the same.

Usage

```
int_end(int)
```

Arguments

int	An interval object

Value

A POSIXct date object when used as an accessor. Nothing when used as a settor

See Also

int_start, int_shift, int_flip, int_length

Examples

```
int <- new_interval(ymd("2001-01-01"), ymd("2002-01-01"))
# 2001-01-01 UTC--2002-01-01 UTC
int_end(int)
# "2002-01-01 UTC"
int_end(int) <- ymd("2002-06-01")
int
# 2001-01-01 UTC--2002-06-01 UTC
```

int_flip *Flip the direction of an interval*

Description

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.

Usage

```
int_flip(int)
```

Arguments

int	An interval object

Value

An interval object

See Also

int_shift, int_start, int_end, int_length

Examples

```
int <- new_interval(ymd("2001-01-01"), ymd("2002-01-01"))
# 2001-01-01 UTC--2002-01-01 UTC
int_flip(int)
# 2002-01-01 UTC--2001-01-01 UTC
```

int_length *Get the length of an interval in seconds*

Description

Get the length of an interval in seconds

Usage

```
int_length(int)
```

Arguments

int	An interval object

Value

numeric The length of the interval in seconds. A negative number connotes a negative interval

See Also

int_start, int_shift, int_flip

Examples

```
int <- new_interval(ymd("2001-01-01"), ymd("2002-01-01"))
# 2001-01-01 UTC--2002-01-01 UTC
int_length(int)
# 31536000
```

---

int_overlaps *Test if two intervals overlap*

Description

Test if two intervals overlap

Usage

```
int_overlaps(int1, int2)
```

Arguments

int1	an Interval object
int2	an Interval object

Value

Logical. TRUE if int1 and int2 overlap by at least one second. FALSE otherwise.

Examples

```
int1 <- new_interval(ymd("2001-01-01"), ymd("2002-01-01"))
# 2001-01-01 UTC--2002-01-01 UTC
int2 <- new_interval(ymd("2001-06-01"), ymd("2002-06-01"))
# 2001-06-01 UTC--2002-06-01 UTC
int3 <- new_interval(ymd("2003-01-01"), ymd("2004-01-01"))
# 2003-01-01 UTC--2004-01-01 UTC

int_overlaps(int1, int2) # TRUE
int_overlaps(int1, int3) # FALSE
```

int_shift *Shift an interval along the timeline*

Description

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.

Usage

```
int_shift(int, by)
```

Arguments

int	An interval object
by	A period or duration object

Value

An interval object

See Also

int_flip, int_start, int_end, int_length

Examples

```
int <- new_interval(ymd("2001-01-01"), ymd("2002-01-01"))
# 2001-01-01 UTC--2002-01-01 UTC
int_shift(int, new_duration(days = 11))
# 2001-01-12 UTC--2002-01-12 UTC
int_shift(int, new_duration(hours = -1))
# 2000-12-31 23:00:00 UTC--2001-12-31 23:00:00 UTC
```

int_standardize *Ensures all intervals in an interval object are positive*

Description

If an interval is not positive, int_standardize flips it so that it retains its endpoints but becomes positive.

Usage

```
int_standardize(int)
```

Arguments

int	an Interval object

Examples

```
int <- new_interval(ymd("2002-01-01"), ymd("2001-01-01"))
# 2002-01-01 UTC--2001-01-01 UTC
int_standardize(int)
# 2001-01-01 UTC--2002-01-01 UTC
```

---

int_start *Access and change the start date of an interval*

Description

Note that changing the start date of an interval will change the length of the interval, since the end date will remain the same.

Usage

```
int_start(int)
```

Arguments

int	An interval object

Value

A POSIXct date object when used as an accessor. Nothing when used as a settor

See Also

int_end, int_shift, int_flip, int_length

Examples

```
int <- new_interval(ymd("2001-01-01"), ymd("2002-01-01"))
# 2001-01-01 UTC--2002-01-01 UTC
int_start(int)
# "2001-01-01 UTC"
int_start(int) <- ymd("2001-06-01")
int
# 2001-06-01 UTC--2002-01-01 UTC
```

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(new_difftime(days = 12.4)) # TRUE
```

`is.duration` *Is x a duration object?*

Description

Is x a duration object?

Usage

```
is.duration(x)
```

Arguments

x	an R object

Value

TRUE if x is a duration object, FALSE otherwise.

See Also

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

Examples

```
is.duration(as.Date("2009-08-03")) # FALSE
is.duration(new_duration(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)
```

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.interval *Is x an Interval object?*

Description

Is x an Interval object?

Usage

```
is.interval(x)
```

Arguments

x	an R object

Value

TRUE if x is an Interval object, FALSE otherwise.

See Also

is.instant, is.timespan, is.period, is.duration, Interval-class

Examples

```
is.interval(new_period(months= 1, days = 15)) # FALSE
is.interval(new_interval(ymd(20090801), ymd(20090809))) # TRUE
```

---

is.period *Is x a period object?*

Description

Is x a period object?

Usage

```
is.period(x)
```

Arguments

x	an R object

Value

TRUE if x is a period object, FALSE otherwise.

See Also

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

Examples

```
is.period(as.Date("2009-08-03")) # FALSE
is.period(new_period(months= 1, days = 15)) # TRUE
```

is.POSIXt *Is x a POSIXct or POSIXlt object?*

Description

Is x a POSIXct or POSIXlt object?

Usage

```
is.POSIXt(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(new_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

http://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
```

`lubridate` *Dates and times made easy with lubridate*

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`, `dym`, `mdy`, `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 `strptime` 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 andround_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 new_duration. dseconds, dminutes, dhours, ddays, dweeks, dyears and new_duration quickly create durations of 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 new_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, new_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

Garrett Grolemund, Hadley Wickham (2011). Dates and Times Made Easy with lubridate. Journal of Statistical Software, 40(3), 1-25. http://www.jstatsoft.org/v40/i03/.

---

lubridate_formats *Lubridate format orders used in* stamp

Description

Lubridate format orders used in stamp

Usage

```
lubridate_formats
```

Format

character vector of formats.

See Also

parse_date_time, ymd, ymd_hms

---

make_difftime *Makes a difftime object from a given number of seconds*

Description

Makes a difftime object from a given number of seconds

Usage

```
make_difftime(x)
```

Arguments

x	number value of seconds to be transformed into a difftime object

Value

a difftime object corresponding to x seconds

Examples

```
make_difftime(1)
make_difftime(60)
make_difftime(3600)
```

---

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

```
minute(x)
```

Arguments

x	a date-time object

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, timeDate, xts, its, ti, jul, timeSeries, and fts objects.

Usage

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

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

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))
# 1
month(ymd(080101), label = TRUE)
# "January"
month(ymd(080101), label = TRUE, abbr = TRUE)
# "Jan"
month(ymd(080101) + months(0:11), label = TRUE, abbr = TRUE)
# "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

ms *Create a period with the specified number of minutes and seconds*

Description

Transforms character or numeric vectors into a period object with the specified number of minutes and seconds. ms() Arbitrary text can separate minutes and seconds. Fractional separator is assumed to be ".". After minutes and seconds have been parsed, all numeric input is ignored.

Usage

```
ms(..., quiet = FALSE)
```

Arguments

...	character or numeric vectors of minute second pairs
quiet	logical. When TRUE function evalueates without displaying customary messages.

Value

a vector of class `Period`

See Also

`hms`, `hm`

Examples

```
x <- c("09:10", "09:02", "1:10")
ms(x)
# [1] 9 minutes and 10 seconds   9 minutes and 2 seconds   1 minute and 10 seconds
ms("7 6")
# [1] 7 minutes and 6 seconds
ms("6,5")
# 6 minutes and 5 seconds
```

new_difftime *Create a difftime object.*

Description

new_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. 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

```
new_difftime(...)
```

Arguments

...	a list of time units to be included in the difftime and their amounts. Seconds, minutes, hours, days, and weeks are supported.

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

```
new_difftime(second = 90)
# Time difference of 1.5 mins
new_difftime(minute = 1.5)
# Time difference of 1.5 mins
new_difftime(second = 3, minute = 1.5, hour = 2, day = 6, week = 1)
# Time difference of 13.08441 days
new_difftime(hour = 1, minute = -60)
# Time difference of 0 secs
new_difftime(day = -1)
# Time difference of -1 days
```

---

new_duration *Create a duration object.*

Description

new_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

```
new_duration(num = 0, ...)
```

Arguments

num	the number of seconds to be included in the duration (if not listing time units).
...	a list of time units to be included in the duration and their amounts. Seconds, minutes, hours, days, and weeks are supported.

Details

new_duration is meant to be used interactively on the command line. See duration, for a version that is better suited to automating within a function.

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

duration, as.duration

Examples

```
new_duration(second = 90)
# 90s
new_duration(minute = 1.5)
# 90s
new_duration(second = 3, minute = 1.5, hour = 2, day = 6, week = 1)
# 1130493s (~13.08 days)
new_duration(hour = 1, minute = -60)
# 0s
new_duration(day = -1)
# -86400s (~-1 days)
```

new_epoch *Create an epoch object*

Description

new_epoch creates an epoch object with the specified values. Implementation for epochs will be implemented in future versions of lubridate.

Usage

```
new_epoch(weekday, number)
```

Arguments

weekday	name of weekday to use as epoch
number	number of epochs to include

Value

an epoch object

See Also

saturdays

new_interval *Create an interval object.*

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.

Usage

```
new_interval(start, end, tzone = attr(start, "tzone"))
```

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

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.

%--% 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.

Value

an Interval object

See Also

Interval-class, as.interval

Examples

```
new_interval(ymd(20090201), ymd(20090101))
# 2009-02-01 UTC--2009-01-01 UTC

date1 <- as.POSIXct("2009-03-08 01:59:59")
date2 <- as.POSIXct("2000-02-29 12:00:00")
new_interval(date2, date1)
# 2000-02-29 12:00:00 CST--2009-03-08 01:59:59 CST
new_interval(date1, date2)
# 2009-03-08 01:59:59 CST--2000-02-29 12:00:00 CST

span <- new_interval(ymd(20090101), ymd(20090201))
# 2009-01-01 UTC--2009-02-01 UTC
```

---

new_period *Create a period object.*

Description

new_period creates a period object with the specified values. 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.

Usage

```
new_period(...)
```

Arguments

...	a list of time units to be included in the period and their amounts. Seconds, minutes, hours, days, weeks, months, and years are supported.

Details

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`, `minutes`, `seconds`. These objects can be added to and subtracted to date-times to create a user interface similar to object oriented programming.

new_period is meant to be used interactively on the command line. See `period`, for a version that is better suited to automating within a function.

Value

a period object

See Also

`period`, `as.period`

Examples

```
new_period (second = 90, minute = 5)
# "5M 90S"
new_period(day = -1)
# "-1d 0H 0M 0S"
new_period(second = 3, minute = 1, hour = 2, day = 13, week = 1)
# "20d 2H 1M 3S"
new_period(hour = 1, minute = -60)
# "1H -60M 0S"
new_period(second = 0)
# "0S"
```

---

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
```

---

olson_time_zones *Names of available time zones*

Description

The names of all available Olson-style time zones.

Usage

```
olson_time_zones(order_by = c("name", "longitude"))
```

Arguments

order_by	Return names alphabetically (the default) or from West to East.

Value

A character vector of time zone names.

Note

Olson-style names are the most readable and portable way of specifying time zones. This function gets names from the file shipped with R, stored in the file 'zone.tab'. ?Sys.timezone has more information.

See Also

Sys.timezone

Examples

```
## Not run:
olson_time_zones()
olson_time_zones("longitude")

## End(Not run)
```

---

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

POSIXt[1:1], format: "1970-01-01"

Examples

```
origin
# "1970-01-01 GMT"
```

---

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 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 strptime.

Usage

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

parse_date_time2(x, orders, tz = "UTC")

fast_strptime(x, format, tz = "UTC")
```

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 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. **NOTE:** 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.
format	a character string of formats. It should include all the separators and each format must be prefixed with 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 hence all the derived functions, such as ymd_hms, ymd etc, will drop into fast_strptime instead of strptime whenever the trained from input data formats are all numeric.

Here are all the formats recognized by lubridate. For numeric formats leading 0s are optional. As compared to strptime, some of the formats have been extended for efficiency reasons. They are marked with "*". Formats accepted by parse_date_time2 and fast_strptime are 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 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.)

B Full month name in the current locale. (Also matches abbreviated name.)

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 0–12).

j Day of year as decimal number (001–366 or 1–366).

m*! Month as decimal number (01–12 or 1–12). For parse_date_time, also matches abbreviated and full months names as b and B formats.

M! Minute as decimal number (00–59 or 0–59).

p AM/PM indicator in the locale. Used in conjunction with I and **not** with H. An empty string in some locales.

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, lubridate represents this format internally as an union of 4 different orders: Ou (Z), Oz (-0800), OO (-08:00) and Oo (-08). You can use this formats as any other but it is rarely necessary. parse_date_time2 and fast_strptime support all of the timezone formats.

r* Matches Ip and H orders.

R* Matches HM andIMp orders.

T* Matches IMSp, HMS, and HMOS orders.

Value

a vector of POSIXct date-time objects

See Also

strptime, ymd, ymd_hms

Examples

```
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 formats **
x <- c("09-01-01", "090102", "09-01 03", "09-01-03 12:02")
parse_date_time(x, c("%y%m%d", "%y%m%d %H%M"))

## different ymd orders:
x <- c("2009-01-01", "02022010", "02-02-2010")
parse_date_time(x, c("%d%m%Y", "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, "%Y%m%d %H%M%S", truncated = 3)
parse_date_time(x, "ymd_hms", truncated = 3)
## [1] "2011-12-31 12:59:59 UTC" "2010-01-01 12:11:00 UTC"
## [3] "2010-01-01 12:00:00 UTC" "2010-01-01 00:00:00 UTC"

## ** fast parsing **
## Not run:
  options(digits.secs = 3)
  ## random times between 1400 and 3000
  tt <- as.character(.POSIXct(runif(1e6, -17987443200, 32503680000)))
  system.time(out <- as.POSIXct(tt, tz = "UTC"))
  system.time(out1 <- ymd_hms(tt)) ## format learning overhead
  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 **
## By default %Y has precedence:
parse_date_time(c("27-09-13", "27-09-2013"), "dmy")
## [1] "13-09-27 UTC"   "2013-09-27 UTC"

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

```
my_select <-   function(trained){
   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)
## '[1] "2013-09-27 UTC" "2013-09-27 UTC"
```

`period` *Create a period object.*

Description

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

Usage

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

Arguments

`num`	a numeric vector that lists the number of time units to be included in the period
`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.

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`, `minutes`, `seconds`. These objects can be added to and subtracted to date-times to create a user interface similar to object oriented programming.

Value

a period object

See Also

new_period, as.period

Examples

```
period(c(90, 5), c("second", "minute"))
#  "5M 90S"
period(-1, "days")
# "-1d 0H 0M 0S"
period(c(3, 1, 2, 13, 1), c("second", "minute", "hour", "day", "week"))
# "20d 2H 1M 3S"
period(c(1, -60), c("hour", "minute"))
# "1H -60M 0S"
period(0, "second")
# "0S"
```

Period-class *Period class*

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+% and %m-%.

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` *Convert a period to the number of seconds it appears to represent*

Description

Convert a period to the number of seconds it appears to represent

Usage

```
period_to_seconds(x)
```

Arguments

`x`	A period object

`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)
# "2009-08-02" "2010-08-02"
pretty_dates(x, 12)
## [1] "2009-08-01 GMT" "2009-09-01 GMT" "2009-10-01 GMT" "2009-11-01 GMT"
## [5] "2009-12-01 GMT" "2010-01-01 GMT" "2010-02-01 GMT" "2010-03-01 GMT"
## [9] "2010-04-01 GMT" "2010-05-01 GMT" "2010-06-01 GMT" "2010-07-01 GMT"
## [13] "2010-08-01 GMT" "2010-09-01 GMT"
```

---

quarter *Get the fiscal quarter of a date-time.*

Description

Fiscal quarters are a way of dividing the year into fourths. The first quarter (Q1) comprises January, February and March; the second quarter (Q2) comprises April, May, June; the third quarter (Q3) comprises July, August, September; the fourth quarter (Q4) October, November, December.

Usage

```
quarter(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 they quarter's year.

Value

numeric the fiscal quarter that the date-time occurs in

Examples

```
x <- ymd(c("2012-03-26", "2012-05-04", "2012-09-23", "2012-12-31"))
quarter(x)
# 1 2 3 4
quarter(x, with_year = TRUE)
# 2012.1 2012.2 2012.3 2012.4
```

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

Description

rollback changes a date to the last day of the previous month. The new date retains the same hour, minute, and second information.

Usage

```
rollback(dates)
```

Arguments

dates	A POSIXct, POSIXlt or Date class object.

Value

A date-time object of class POSIXlt, POSIXct or Date, whose day has been adjusted to the last day of the previous month. date <- ymd("2010-03-03") # "2010-03-03 UTC" rollback(date) # "2010-02-28 UTC"

dates <- date + months(0:2) "2010-03-03 UTC" "2010-04-03 UTC" "2010-05-03 UTC" rollback(dates) "2010-02-28 UTC" "2010-03-31 UTC" "2010-04-30 UTC"

round_date *Rounding for date-times.*

Description

round_date takes a date-time object and rounds it to the nearest integer value of the specified time unit. Users can specify whether to round to the nearest second, minute, hour, day, week, month, or year.

Usage

```
round_date(x,
  unit = c("second", "minute", "hour", "day", "week", "month", "year"))
```

Arguments

x	a vector of date-time objects
unit	a character string specifying the time unit to be rounded to. Should be one of "second","minute","hour","day", "week", "month", or "year."

Value

x with the appropriate units rounded

See Also

floor_date, ceiling_date

Examples

```
x <- as.POSIXct("2009-08-03 12:01:59.23")
round_date(x, "second")
# "2009-08-03 12:01:59 CDT"
round_date(x, "minute")
# "2009-08-03 12:02:00 CDT"
round_date(x, "hour")
# "2009-08-03 12:00:00 CDT"
round_date(x, "day")
# "2009-08-04 CDT"
round_date(x, "week")
# "2009-08-02 CDT"
round_date(x, "month")
# "2009-08-01 CDT"
round_date(x, "year")
# "2010-01-01 CST"
```

---

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)
```

Arguments

x	a date-time object

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
```

---

seconds *Quickly create relative timespans.*

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 new_duration. This makes periods useful for manipulations with clock times because units expand or contract in length to accomodate conventions such as leap years, leap seconds, and Daylight Savings Time.

Usage

```
seconds(x = 1)
```

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.

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.

Value

a period object

See Also

Period-class, new_period, ddays

Examples

```
x <- as.POSIXct("2009-08-03")
# "2009-08-03 CDT"
x + days(1) + hours(6) + minutes(30)
# "2009-08-04 06:30:00 CDT"
x + days(100) - hours(8)
# "2009-11-10 15:00:00 CST"

class(as.Date("2009-08-09") + days(1)) # retains Date class
# "Date"
as.Date("2009-08-09") + hours(12)
# "2009-08-09 12:00:00 UTC"
class(as.Date("2009-08-09") + hours(12))
# "POSIXt"  "POSIXct"
# converts to POSIXt class to accomodate time units

years(1) - months(7)
# "1y -7m 0d 0H 0M 0S"
c(1:3) * hours(1)
# "1H 0M 0S" "2H 0M 0S" "3H 0M 0S"
hours(1:3)
# "1H 0M 0S" "2H 0M 0S" "3H 0M 0S"

#sequencing
y <- ymd(090101) # "2009-01-01 CST"
y + months(0:11)
# [1] "2009-01-01 CST" "2009-02-01 CST" "2009-03-01 CST" "2009-04-01 CDT"
# [5] "2009-05-01 CDT" "2009-06-01 CDT" "2009-07-01 CDT" "2009-08-01 CDT"
# [9] "2009-09-01 CDT" "2009-10-01 CDT" "2009-11-01 CDT" "2009-12-01 CST"

# compare DST handling to durations
boundary <- as.POSIXct("2009-03-08 01:59:59")
# "2009-03-08 01:59:59 CST"
boundary + days(1) # period
# "2009-03-09 01:59:59 CDT" (clock time advances by a day)
boundary + edays(1) # duration
# "2009-03-09 02:59:59 CDT" (clock time corresponding to 86400
# seconds later)
```

`seconds_to_period` *Contrive a period from a given number of seconds*

Description

seconds_to_period uses estimates of time elements (in seconds) to create the period that has the maximum number of large elements(years > months > days > hours > minutes > seconds) and roughly equates to a given number of seconds. Note that the actual number of seconds in a period depends on when the period occurs. Since there is no one-to-one relationship between the periods that seconds_to_period makes and the number of seconds given as input, these periods should be treated as rough estimates only.

Usage

```
seconds_to_period(x)
```

Arguments

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

Value

A period that roughly equates to the number of seconds given.

---

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 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

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")(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")(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"))
```

---

sundays *Quickly create common epoch objects*

Description

Creates an epoch object that uses the functions name as the reference epoch event. Implementation for epochs will be implemented in future versions of lubridate.

Usage

```
sundays(x = 1)
```

Arguments

x	number of epochs to be included

`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.

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 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

`new_duration` for creating duration objects and `new_period` for creating period objects, and `new_interval` for creating interval objects

Examples

```
duration(3690, "seconds")
# 3690s (~1.02 hours)
period(3690, "seconds")
# "3690S"
new_period(second = 30, minute = 1, hour = 1)
# "1H 1M 30S"
interval(ymd_hms("2009-08-09 13:01:30"), ymd_hms("2009-08-09 12:00:00"))
# 2009-08-09 12:00:00 -- 2009-08-09 13:01:30

date <- as.POSIXct("2009-03-08 01:59:59") # DST boundary
# "2009-03-08 01:59:59 CST"
date + days(1)
# "2009-03-09 01:59:59 CDT" periods preserve clock time
date + edays(1)
# "2009-03-09 02:59:59 CDT" durations preserve exact passage of time

date2 <- as.POSIXct("2000-02-29 12:00:00")
date2 + years(1)
# "2001-03-01 12:00:00 CST"
# self corrects to next real day

date3 <- as.POSIXct("2009-01-31 01:00:00")
date3 + c(0:11) * months(1)
# [1] "2009-01-31 01:00:00 CST" "2009-03-03 01:00:00 CST"
# [3] "2009-03-31 01:00:00 CDT" "2009-05-01 01:00:00 CDT"
# [5] "2009-05-31 01:00:00 CDT" "2009-07-01 01:00:00 CDT"
# [7] "2009-07-31 01:00:00 CDT" "2009-08-31 01:00:00 CDT"
# [9] "2009-10-01 01:00:00 CDT" "2009-10-31 01:00:00 CDT"
# [11] "2009-12-01 01:00:00 CST" "2009-12-31 01:00:00 CST"

span <- date2 %--% date  #creates interval
# "2000-02-29 12:00:00 CST--2009-03-08 01:59:59 CST"

date <- as.POSIXct("2009-01-01 00:00:00")
# "2009-01-01 GMT"
date + years(1)
# "2010-01-01 GMT"
date - days(3) + hours(6)
# "2008-12-29 06:00:00 GMT"
date + 3 * seconds(10)
# "2009-01-01 00:00:30 GMT"

months(6) + days(1)
# "6m 1d 0H 0M 0S"
```

`Timespan-class` *Timespan class*

Description

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

---

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)
```

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

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

```
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")
```

---

wday *Get/set days 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

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

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 a character string label 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 a character string label such as "Sunday." TRUE will display an abbreviated version of the label, such as "Sun". abbr is disregarded if label = FALSE.

Value

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

See Also

yday, mday

Examples

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

wday(ymd(080101))
# 3
wday(ymd(080101), label = TRUE)
# "Tuesday"
wday(ymd(080101), label = TRUE, abbr = TRUE)
# "Tues"
wday(ymd(080101) + days(-2:4), label = TRUE, abbr = TRUE)
# "Sun"   "Mon"   "Tues"  "Wed"   "Thurs" "Fri"   "Sat"
```

week *Get/set weeks 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. Weeks is the number of complete seven day periods that have occured 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

```
week(x)
```

Arguments

x	a date-time object

Value

the weeks element of x as an integer number

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, or chron date-time object.
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")
# "2009-08-07 04:00:01 GMT"
```

---

yday *Get/set days 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

```
yday(x)
```

Arguments

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

Value

yday returns the day of the year as a decimal number (01-366). mday returns the day of the month as a decimal number (01-31).

See Also

wday

Examples

```
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
```

`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 fts objects.

Usage

```
year(x)
```

Arguments

`x`	a date-time object

Details

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

Value

the years element of x as a decimal number

Examples

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

ymd *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 POSIXct objects. 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 = "UTC",
  locale = Sys.getlocale("LC_TIME"), truncated = 0)
```

Arguments

`...`	a character or numeric vector of suspected dates
`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. Number of formats that can be truncated.

Details

`ymd` family of functions automatically assign the Universal Coordinated Time Zone (UTC) to the parsed dates. This time zone can be changed with `force_tz`.

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 `strptime` which currently correctly parses "%y" format, but 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 class POSIXct

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)
## "2009-01-01 UTC" "2009-01-02 UTC" "2009-01-03 UTC"
x <- c("2009-01-01", "2009-01-02", "2009-01-03")
ymd(x)
## "2009-01-01 UTC" "2009-01-02 UTC" "2009-01-03 UTC"
ymd(090101, 90102)
## "2009-01-01 UTC" "2009-01-02 UTC"
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 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

```
ymd_hms(..., 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 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).

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)
# [1] "2010-04-14 04:35:59 UTC" "2010-04-01 12:00:00 UTC"
x <- c("2011-12-31 12:59:59", "2010-01-01 12:00:00")
ymd_hms(x)
# [1] "2011-12-31 12:59:59 UTC" "2010-01-01 12:00:00 UTC"


## ** 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")
## "2006-02-20 11:16:16.683 UTC"
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)
## [1] "2011-12-31 12:59:59 UTC" "2010-01-01 12:11:00 UTC"
## [3] "2010-01-01 12:00:00 UTC" "2010-01-01 00:00:00 UTC"
x <- c("2011-12-31 12:59", "2010-01-01 12", "2010-01-01")
ymd_hm(x, truncated = 2)
## [1] "2011-12-31 12:59:00 UTC" "2010-01-01 12:00:00 UTC"
## [3] "2010-01-01 00:00:00 UTC"
## ** 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)
## "2010-02-01 07:01:00 UTC" "2010-02-01 07:01:00 UTC"   NA
```

`%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.

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, month. Date nth month before Date.

Usage

```
e1 %m+% e2
```

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.

Details

These must be added separately with traditional arithmetic. they are not a 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")
# "2010-01-31 03:04:05 UTC"
jan + months(1:3) # Feb 31 and April 31 prompt "rollover"
# "2010-03-03 03:04:05 UTC" "2010-03-31 03:04:05 UTC" "2010-05-01 03:04:05 UTC"
jan %m+% months(1:3) # No rollover
# "2010-02-28 03:04:05 UTC" "2010-03-31 03:04:05 UTC" "2010-04-30 03:04:05 UTC"

leap <- ymd("2012-02-29")
"2012-02-29 UTC"
leap %m+% years(1)
# "2013-02-28 UTC"
leap %m+% years(-1)
leap %m-% years(1)
# "2011-02-28 UTC"
```

%within% *Tests whether a date or interval falls within an interval*

Description

otherwise. 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 <- new_interval(ymd("2001-01-01"), ymd("2002-01-01"))
# 2001-01-01 UTC--2002-01-01 UTC
int2 <- new_interval(ymd("2001-06-01"), ymd("2002-01-01"))
# 2001-06-01 UTC--2002-01-01 UTC

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

Index