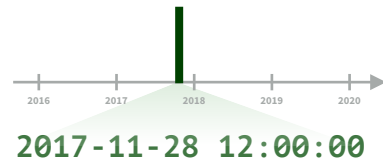


# Dates and times with lubridate : : CHEATSHEET



## Date-times



**2017-11-28 12:00:00**  
 A **date-time** is a point on the timeline, stored as the number of seconds since 1970-01-01 00:00:00 UTC

```
dt <- as_datetime(1511870400)
## "2017-11-28 12:00:00 UTC"
```

### PARSE DATE-TIMES (Convert strings or numbers to date-times)

- Identify the order of the year (**y**), month (**m**), day (**d**), hour (**h**), minute (**m**) and second (**s**) elements in your data.
- Use the function below whose name replicates the order. Each accepts a tz argument to set the time zone, e.g. ymd(x, tz = "UTC").

**2017-11-28T14:02:00** **ymd\_hms()**, **ymd\_hm()**, **ymd\_h()**.  
 ymd\_hms("2017-11-28T14:02:00")

**2017-22-12 10:00:00** **ydm\_hms()**, **ydm\_hm()**, **ydm\_h()**.  
 ydm\_hms("2017-22-12 10:00:00")

**11/28/2017 1:02:03** **mdy\_hms()**, **mdy\_hm()**, **mdy\_h()**.  
 mdy\_hms("11/28/2017 1:02:03")

**1 Jan 2017 23:59:59** **dmy\_hms()**, **dmy\_hm()**, **dmy\_h()**.  
 dmy\_hms("1 Jan 2017 23:59:59")

**20170131** **ymd()**, **ydm()**. ymd(20170131)

**July 4th, 2000** **mdy()**, **myd()**. mdy("July 4th, 2000")

**4th of July '99** **dmy()**, **dym()**. dmy("4th of July '99")

**2001: Q3** **yq()** Q for quarter. yq("2001: Q3")

**07-2020** **my()**, **ym()**. my("07-2020")

**2:01** **hms::hms()** Also lubridate::**hms()**, **hm()** and **ms()**, which return periods.\* hms::hms(seconds = 0, minutes = 1, hours = 2)

**2017.5** **date\_decimal(decimal, tz = "UTC")**  
 date\_decimal(2017.5)

**now(tzone = "")** Current time in tz (defaults to system tz). now()

**today(tzone = "")** Current date in a tz (defaults to system tz). today()

**fast\_strptime()** Faster strptime.  
 fast\_strptime("9/1/01", "%y/%m/%d")

**parse\_date\_time()** Easier strptime.  
 parse\_date\_time("09-01-01", "ymd")

**2017-11-28** A **date** is a day stored as the number of days since 1970-01-01

```
d <- as_date(17498)
## "2017-11-28"
```

### GET AND SET COMPONENTS

Use an accessor function to get a component.  
 Assign into an accessor function to change a component in place.

```
d ## "2017-11-28"
day(d) ## 28
day(d) <- 1
d ## "2017-11-01"
```

**2018-01-31 11:59:59** **date(x)** Date component. date(dt)

**2018-01-31 11:59:59** **year(x)** Year. year(dt)  
**isoyear(x)** The ISO 8601 year.  
**epiyear(x)** Epidemiological year.

**2018-01-31 11:59:59** **month(x, label, abbr)** Month.  
 month(dt)

**2018-01-31 11:59:59** **day(x)** Day of month. day(dt)  
**wday(x, label, abbr)** Day of week.  
**qday(x)** Day of quarter.

**2018-01-31 11:59:59** **hour(x)** Hour. hour(dt)

**2018-01-31 11:59:59** **minute(x)** Minutes. minute(dt)

**2018-01-31 11:59:59** **second(x)** Seconds. second(dt)

**2018-01-31 11:59:59 UTC** **tz(x)** Time zone. tz(dt)

**week(x)** Week of the year. week(dt)  
**isoweek()** ISO 8601 week.  
**epiweek()** Epidemiological week.

**quarter(x)** Quarter. quarter(dt)

**semester(x, with\_year = FALSE)** Semester. semester(dt)

**am(x)** Is it in the am? am(dt)  
**pm(x)** Is it in the pm? pm(dt)

**dst(x)** Is it daylight savings? dst(d)

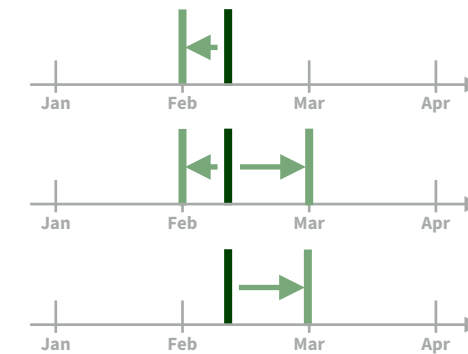
**leap\_year(x)** Is it a leap year?  
 leap\_year(d)

**update(object, ..., simple = FALSE)**  
 update(dt, mday = 2, hour = 1)

**12:00:00** An hms is a **time** stored as the number of seconds since 00:00:00

```
t <- hms::as_hms(85)
## "00:01:25"
```

## Round Date-times



**floor\_date(x, unit = "second")**  
 Round down to nearest unit.  
 floor\_date(dt, unit = "month")

**round\_date(x, unit = "second")**  
 Round to nearest unit.  
 round\_date(dt, unit = "month")

**ceiling\_date(x, unit = "second")**,  
 change\_on\_boundary = NULL)  
 Round up to nearest unit.  
 ceiling\_date(dt, unit = "month")

Valid units are second, minute, hour, day, week, month, bimonth, quarter, season, halfyear and year.

**rollback(dates, roll\_to\_first = FALSE, preserve\_hms = TRUE)** Roll back to last day of previous month. Also **rollforward()**. rollback(dt)

## Stamp Date-times

**stamp()** Derive a template from an example string and return a new function that will apply the template to date-times. Also **stamp\_date()** and **stamp\_time()**.

- Derive a template, create a function  
 sf <- stamp("Created Sunday, Jan 17, 1999 3:34")
- Apply the template to dates  
 sf(ymd("2010-04-05"))  
 ## [1] "Created Monday, Apr 05, 2010 00:00"

**Tip: use a date with day > 12**

## Time Zones

R recognizes ~600 time zones. Each encodes the time zone, Daylight Savings Time, and historical calendar variations for an area. R assigns one time zone per vector.

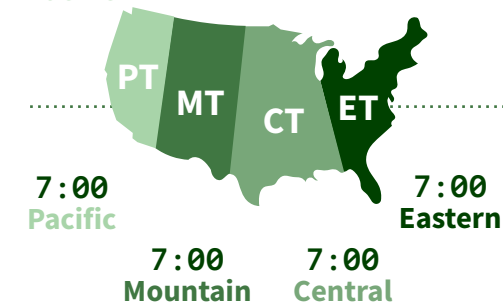
Use the **UTC** time zone to avoid Daylight Savings.

**OlsonNames()** Returns a list of valid time zone names. OlsonNames()

**Sys.timezone()** Gets current time zone.



**with\_tz(time, tzone = "")** Get the **same date-time** in a new time zone (a new clock time). Also **local\_time(dt, tz, units)**. with\_tz(dt, "US/Pacific")



**force\_tz(time, tzone = "")** Get the **same clock time** in a new time zone (a new date-time). Also **force\_tzs()**. force\_tz(dt, "US/Pacific")



# Math with Date-times

— Lubridate provides three classes of timespans to facilitate math with dates and date-times.



Math with date-times relies on the **timeline**, which behaves inconsistently. Consider how the timeline behaves during:

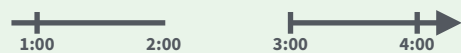
## A normal day

```
nor <- ymd_hms("2018-01-01 01:30:00",tz="US/Eastern")
```



## The start of daylight savings (spring forward)

```
gap <- ymd_hms("2018-03-11 01:30:00",tz="US/Eastern")
```



## The end of daylight savings (fall back)

```
lap <- ymd_hms("2018-11-04 00:30:00",tz="US/Eastern")
```



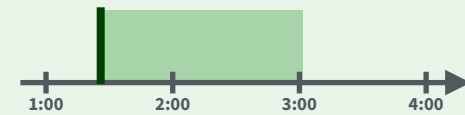
## Leap years and leap seconds

```
leap <- ymd("2019-03-01")
```

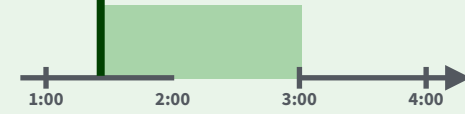


**Periods** track changes in clock times, which ignore time line irregularities.

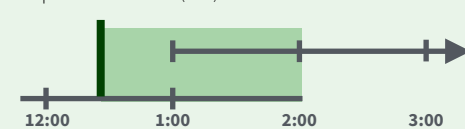
```
nor + minutes(90)
```



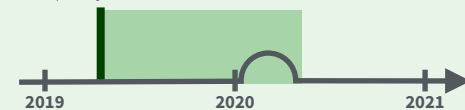
```
gap + minutes(90)
```



```
lap + minutes(90)
```

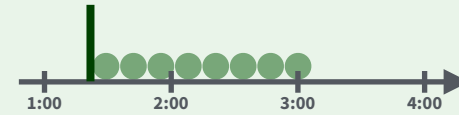


```
leap + years(1)
```



**Durations** track the passage of physical time, which deviates from clock time when irregularities occur.

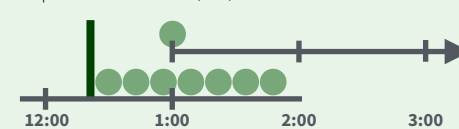
```
nor + dminutes(90)
```



```
gap + dminutes(90)
```



```
lap + dminutes(90)
```

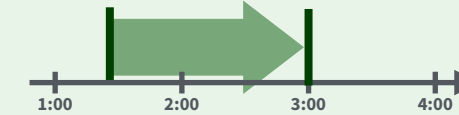


```
leap + dyears(1)
```



**Intervals** represent specific intervals of the timeline, bounded by start and end date-times.

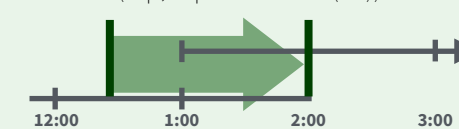
```
interval(nor, nor + minutes(90))
```



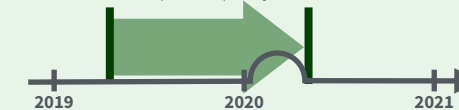
```
interval(gap, gap + minutes(90))
```



```
interval(lap, lap + minutes(90))
```



```
interval(leap, leap + years(1))
```



Not all years are 365 days due to **leap days**.

Not all minutes are 60 seconds due to **leap seconds**.

It is possible to create an imaginary date by adding **months**, e.g. February 31st

```
jan31 <- ymd(20180131)
jan31 + months(1)
## NA
```

**%m+%** and **%m-%** will roll imaginary dates to the last day of the previous month.

```
jan31 %m+% months(1)
## "2018-02-28"
```

**add\_with\_rollback(e1, e2, roll\_to\_first = TRUE)** will roll imaginary dates to the first day of the new month.

```
add_with_rollback(jan31, months(1),
roll_to_first = TRUE)
## "2018-03-01"
```

## PERIODS

Add or subtract periods to model events that happen at specific clock times, like the NYSE opening bell.

Make a period with the name of a time unit **pluralized**, e.g.

```
p <- months(3) + days(12)
```

```
p
"3m 12d 0H 0M 0S"
```



- years(x = 1)** x years.
- months(x)** x months.
- weeks(x = 1)** x weeks.
- days(x = 1)** x days.
- hours(x = 1)** x hours.
- minutes(x = 1)** x minutes.
- seconds(x = 1)** x seconds.
- milliseconds(x = 1)** x milliseconds.
- microseconds(x = 1)** x microseconds.
- nanoseconds(x = 1)** x nanoseconds.
- picoseconds(x = 1)** x picoseconds.

**period(num = NULL, units = "second", ...)**  
An automation friendly period constructor.  
period(5, unit = "years")

**as.period(x, unit)** Coerce a timespan to a period, optionally in the specified units.  
Also **is.period()**. as.period(p)

**period\_to\_seconds(x)** Convert a period to the "standard" number of seconds implied by the period. Also **seconds\_to\_period()**.  
period\_to\_seconds(p)

## DURATIONS

Add or subtract durations to model physical processes, like battery life. Durations are stored as seconds, the only time unit with a consistent length. **Diffimes** are a class of durations found in base R.

Make a duration with the name of a period prefixed with a **d**, e.g.

```
dd <- ddays(14)
```

```
dd
"1209600s (~2 weeks)"
```



- dyears(x = 1)** 31536000x seconds.
- dmonths(x = 1)** 2629800x seconds.
- dweeks(x = 1)** 604800x seconds.
- ddays(x = 1)** 86400x seconds.
- dhours(x = 1)** 3600x seconds.
- dminutes(x = 1)** 60x seconds.
- dseconds(x = 1)** x seconds.
- dmilliseconds(x = 1)** x × 10<sup>-3</sup> seconds.
- dmicroseconds(x = 1)** x × 10<sup>-6</sup> seconds.
- dnanoseconds(x = 1)** x × 10<sup>-9</sup> seconds.
- dpicoseconds(x = 1)** x × 10<sup>-12</sup> seconds.

**duration(num = NULL, units = "second", ...)**  
An automation friendly duration constructor. duration(5, unit = "years")

**as.duration(x, ...)** Coerce a timespan to a duration. Also **is.duration()**, **is.diffime()**.  
as.duration(i)

**make\_diffime(x)** Make diffime with the specified number of units.  
make\_diffime(99999)

## INTERVALS

Divide an interval by a duration to determine its physical length, divide an interval by a period to determine its implied length in clock time.

Make an interval with **interval()** or **%--%**, e.g.

```
i <- interval(ymd("2017-01-01"), d)
j <- d %--% ymd("2017-12-31")
## 2017-01-01 UTC--2017-11-28 UTC
## 2017-11-28 UTC--2017-12-31 UTC
```



**a %within% b** Does interval or date-time *a* fall within interval *b*? now() %within% i



**int\_start(int)** Access/set the start date-time of an interval. Also **int\_end()**. int\_start(i) <- now(); int\_start(i)



**int\_aligns(int1, int2)** Do two intervals share a boundary? Also **int\_overlaps()**. int\_aligns(i, j)



**int\_diff(times)** Make the intervals that occur between the date-times in a vector.  
v <- c(dt, dt + 100, dt + 1000); int\_diff(v)



**int\_flip(int)** Reverse the direction of an interval. Also **int\_standardize()**. int\_flip(i)



**int\_length(int)** Length in seconds. int\_length(i)



**int\_shift(int, by)** Shifts an interval up or down the timeline by a timespan. int\_shift(i, days(-1))



**as.interval(x, start, ...)** Coerce a timespan to an interval with the start date-time. Also **is.interval()**. as.interval(days(1), start = now())