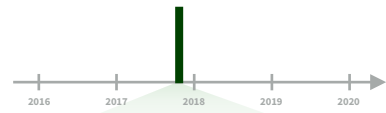


# Dates and times with lubridate : : CHEAT SHEET



## Date-times



2017-11-28 12:00:00

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)

1. Identify the order of the year (**y**), month (**m**), day (**d**), hour (**h**), minute (**m**) and second (**s**) elements in your data.
2. Use the function below whose name replicates the order. Each accepts a wide variety of input formats.

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")`

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

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

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

**today**(tzzone = "") 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("9/1/01", "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"
```

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"
```

### 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)`

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

**quarter(x, with\_year = FALSE)** 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)`

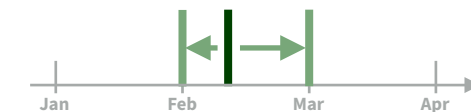
**leap\_year(x)** Is it a leap year?  
`leap_year(d)`

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

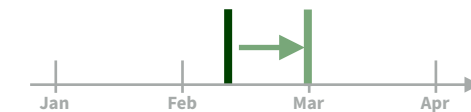
## Round Date-times



**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")`

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

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

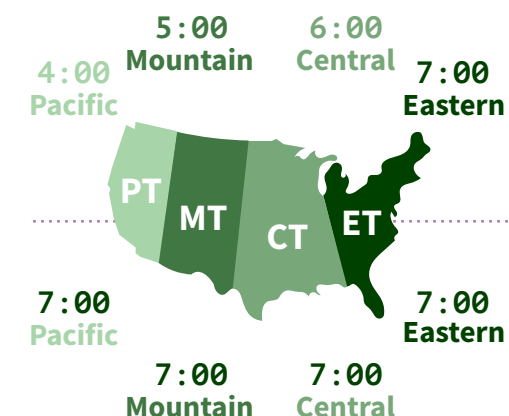
Tip: use a date with day > 12

## 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()`



**with\_tz(time, tzzone = "")** Get the **same date-time** in a new time zone (a new clock time).  
`with_tz(dt, "US/Pacific")`

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

# Math with Date-times

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

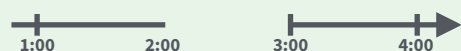


Math with date-times relies on the **timeline**, which behaves inconsistently. Consider how the timeline behaves during:

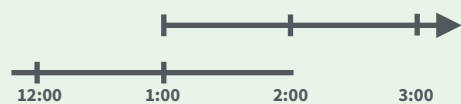
A normal day  
`nor <- ymd_hms("2018-01-01 01:30:00",tz="US/Eastern")`



The start of daylight savings (spring forward)  
`gap <- ymd_hms("2018-03-11 01:30:00",tz="US/Eastern")`



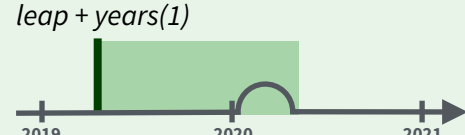
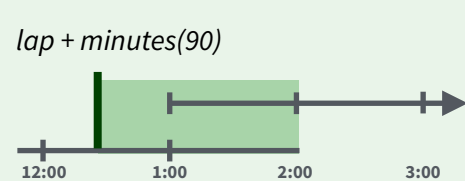
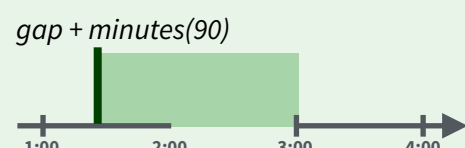
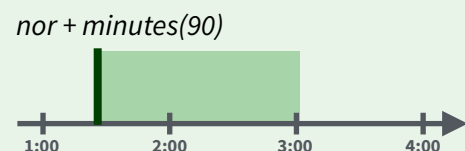
The end of daylight savings (fall back)  
`lap <- ymd_hms("2018-11-04 00:30:00",tz="US/Eastern")`



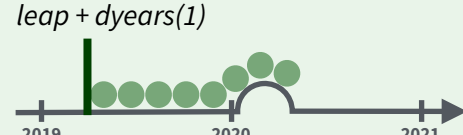
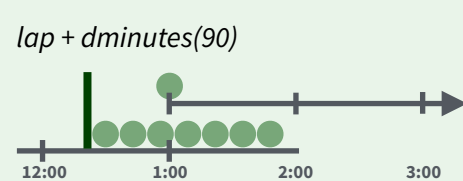
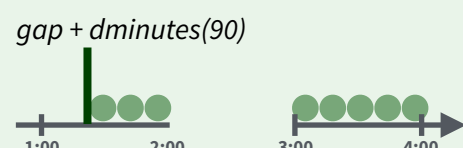
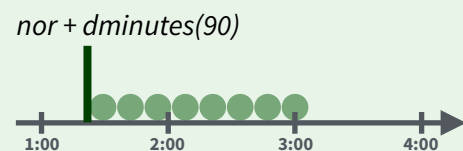
Leap years and leap seconds  
`leap <- ymd("2019-03-01")`



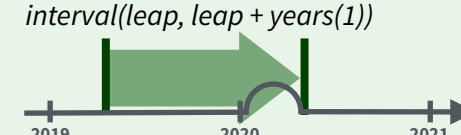
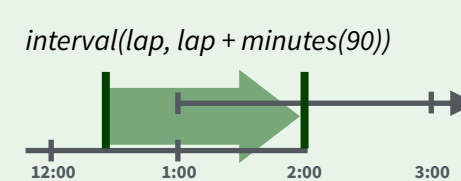
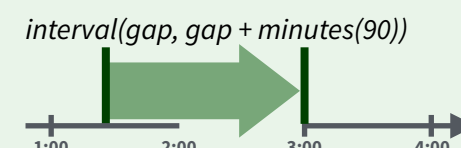
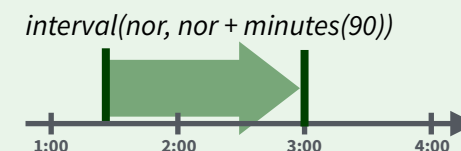
**Periods** track changes in clock times, which ignore time line irregularities.



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



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



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.

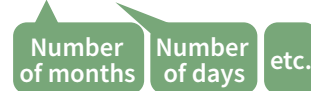
```
add_with_rollback(jan31, months(1),
roll_to_first = TRUE)
## "2018-03-01"
```

## 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**(i). `as.period(i)`

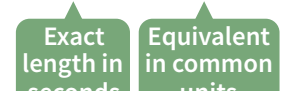
**period\_to\_seconds**(x) Convert a period to the "standard" number of seconds implied by the period. Also **seconds\_to\_period**(i).  
`period_to_seconds(p)`

## 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.
- 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**(i), **is.diffime**(i).  
`as.duration(i)`

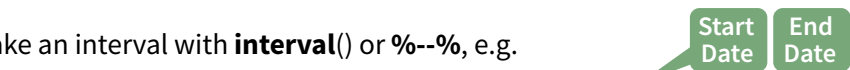
**make\_diffime**(x) Make diffime with the specified number of units.  
`make_diffime(99999)`

## INTERVALS

Divide an interval by a duration to determine its physical length, divide and 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) ## 2017-01-01 UTC--2017-11-28 UTC
j <- d %--% ymd("2017-12-31") ## 2017-11-28 UTC--2017-12-31 UTC
```



**%within%** 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**(i). `int_start(i) <- now(); int_start(i)`

**int\_aligns**(int1, int2) Do two intervals share a boundary? Also **int\_overlaps**(i, j). `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**(i). `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**(i). `as.interval(days(1), start = now())`

