HW2

OmerFarukYahsi

17.4 Time spans

We'll learn about how arithmetic with dates works, including subtraction, addition, and division. Along the way, you'll learn about two important classes that represent time spans:

- **Durations**, which represent an exact number of seconds.
  - **Precision:** Durations provide precise measurements of time in seconds.
  - **Use Case:** Suitable for scenarios where the exact time difference between two points in time needs to be calculated, such as in financial calculations (e.g., loan duration).
  - **Representation:** Represents an exact number of seconds and is not affected by variations in the number of days in months or years.
- **Periods**, which represent human units like weeks and months.
  - **Granularity:** Periods are more human-centric, providing a way to represent time in terms of weeks, months, and years.
  - **Use Case:** Suitable for scenarios where the focus is on calendar units and variations in the number of days (e.g., project planning).
  - **Representation:** Represents a length of time in terms of human-readable units, accommodating differences in the number of days in months and years.

How do you pick between duration, periods ? As always, pick the simplest data structure that solves your problem. If you only care about physical time, use a duration; if you need to add human times, use a period.

This chapter will focus on the **lubridate** package, which makes it easier to work with dates and times in R. As of the latest tidyverse release, lubridate is part of core tidyverse. We will also need nycflights13 for practice data.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.3     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.3     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.0
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becor
```

```
library(nycflights13)
library(lubridate) #for older tidyverse versions
```

17.4.1 Durations

In R, when you subtract two dates, you get a difftime object:

```
# How old is Hadley?
h_age <- today() - ymd("1979-10-14")

h_age
```

```
Time difference of 16109 days
```

A `difftime` class object records a time span of seconds, minutes, hours, days, or weeks. This ambiguity can make difftimes a little painful to work with, so lubridate provides an alternative which always uses seconds: the **duration**.

```
as.duration(h_age)
```

```
[1] "1391817600s (~44.1 years)"
```

Durations come with a bunch of convenient constructors:

```
  dseconds(15)
```

```
[1] "15s"
```

```
  dminutes(10)
```

```
[1] "600s (~10 minutes)"
```

```
  dhours(c(12, 24))
```

```
[1] "43200s (~12 hours)" "86400s (~1 days)"
```

```
  ddays(0:5)
```

```
[1] "0s"                "86400s (~1 days)"  "172800s (~2 days)"
[4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
```

```
  dweeks(3)
```

```
[1] "1814400s (~3 weeks)"
```

```
  dyears(1)
```

```
[1] "31557600s (~1 years)"
```

Durations always record the time span in seconds. Larger units are created by converting minutes, hours, days, weeks, and years to seconds: 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 7 days in a week. Larger time units are more problematic. A year uses the "average" number of days in a year, i.e. 365.25. There's no way to convert a month to a duration, because there's just too much variation.

You can add and multiply durations:

```
  2 * dyears(1)
```

```
[1] "63115200s (~2 years)"
```

```
dyears(1) + dweeks(12) + dhours(15)
```

```
[1] "38869200s (~1.23 years)"
```

You can add and subtract durations to and from days:

```
tomorrow <- today() + ddays(1)

tomorrow
```

```
[1] "2023-11-22"
```

```
last_year <- today() - dyears(1)

last_year
```

```
[1] "2022-11-20 18:00:00 UTC"
```

However, because durations represent an exact number of seconds, sometimes you might get an unexpected result:

```
one_am <- ymd_hms("2026-03-08 01:00:00", tz = "America/New_York")

one_am
```

```
[1] "2026-03-08 01:00:00 EST"
```

```
one_am + ddays(1)
```

```
[1] "2026-03-09 02:00:00 EDT"
```

Why is one day after 1am March 8, 2am March 9? If you look carefully at the date you might also notice that the time zones have changed. March 8 only has 23 hours because it's when DST starts, so if we add a full days worth of seconds we end up with a different time.

17.4.2 Periods

To solve this problem, lubridate provides **periods**. Periods are time spans but don't have a fixed length in seconds, instead they work with "human" times, like days and months. That allows them to work in a more intuitive way:

```
one_am
```

```
[1] "2026-03-08 01:00:00 EST"
```

```
one_am + days(1)
```

```
[1] "2026-03-09 01:00:00 EDT"
```

Like durations, periods can be created with a number of friendly constructor functions.

```
hours(c(12, 24))
```

```
[1] "12H 0M 0S" "24H 0M 0S"
```

```
days(7)
```

```
[1] "7d 0H 0M 0S"
```

```
months(1:6)
```

```
[1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S"
[5] "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
```

You can add and multiply periods:

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

```
[1] "60m 10d 0H 0M 0S"
```

```
days(50) + hours(25) + minutes(2)
```

```
[1] "50d 25H 2M 0S"
```

And of course, add them to dates. Compared to durations, periods are more likely to do what you expect:

```
# A leap year
ymd("2024-01-01") + dyears(1)
```

```
[1] "2024-12-31 06:00:00 UTC"
```

```
ymd("2024-01-01") + years(1)
```

```
[1] "2025-01-01"
```

```
# Daylight saving time
one_am + ddays(1)
```

```
[1] "2026-03-09 02:00:00 EDT"
```

```
one_am + days(1)
```

```
[1] "2026-03-09 01:00:00 EDT"
```

The times are represented in a slightly odd format, so we use modulus arithmetic to pull out the hour and minute components. Once we've created the date-time variables, we focus in on the variables we'll explore in the rest of the chapter.

```
make_datetime_100 <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100)
}

flights_dt <- flights |>
  filter(!is.na(dep_time), !is.na(arr_time)) |>
```

```
    mutate(
      dep_time = make_datetime_100(year, month, day, dep_time),
      arr_time = make_datetime_100(year, month, day, arr_time),
      sched_dep_time = make_datetime_100(year, month, day, sched_dep_time),
      sched_arr_time = make_datetime_100(year, month, day, sched_arr_time)
    ) |>
    select(origin, dest, ends_with("delay"), ends_with("time"))

  flights_dt
```

```
# A tibble: 328,063 x 9
   origin dest  dep_delay arr_delay dep_time            sched_dep_time
   <chr>  <chr>     <dbl>     <dbl> <dttm>              <dttm>
 1 EWR    IAH           2        11 2013-01-01 05:17:00 2013-01-01 05:15:00
 2 LGA    IAH           4        20 2013-01-01 05:33:00 2013-01-01 05:29:00
 3 JFK    MIA           2        33 2013-01-01 05:42:00 2013-01-01 05:40:00
 4 JFK    BQN          -1       -18 2013-01-01 05:44:00 2013-01-01 05:45:00
 5 LGA    ATL          -6       -25 2013-01-01 05:54:00 2013-01-01 06:00:00
 6 EWR    ORD          -4        12 2013-01-01 05:54:00 2013-01-01 05:58:00
 7 EWR    FLL          -5        19 2013-01-01 05:55:00 2013-01-01 06:00:00
 8 LGA    IAD          -3       -14 2013-01-01 05:57:00 2013-01-01 06:00:00
 9 JFK    MCO          -3        -8 2013-01-01 05:57:00 2013-01-01 06:00:00
10 LGA    ORD          -2         8 2013-01-01 05:58:00 2013-01-01 06:00:00
# i 328,053 more rows
# i 3 more variables: arr_time <dttm>, sched_arr_time <dttm>, air_time <dbl>
```

Let's use periods to fix an oddity related to our flight dates. Some planes appear to have arrived at their destination *before* they departed from New York City.

```
  flights_dt |>
    filter(arr_time < dep_time)
```

```
# A tibble: 10,633 x 9
   origin dest  dep_delay arr_delay dep_time            sched_dep_time
   <chr>  <chr>     <dbl>     <dbl> <dttm>              <dttm>
 1 EWR    BQN           9        -4 2013-01-01 19:29:00 2013-01-01 19:20:00
 2 JFK    DFW          59        NA 2013-01-01 19:39:00 2013-01-01 18:40:00
 3 EWR    TPA          -2         9 2013-01-01 20:58:00 2013-01-01 21:00:00
 4 EWR    SJU          -6       -12 2013-01-01 21:02:00 2013-01-01 21:08:00
 5 EWR    SFO          11       -14 2013-01-01 21:08:00 2013-01-01 20:57:00
```

```
 6 LGA    FLL         -10        -2 2013-01-01 21:20:00 2013-01-01 21:30:00
 7 EWR    MCO          41        43 2013-01-01 21:21:00 2013-01-01 20:40:00
 8 JFK    LAX          -7       -24 2013-01-01 21:28:00 2013-01-01 21:35:00
 9 EWR    FLL          49        28 2013-01-01 21:34:00 2013-01-01 20:45:00
10 EWR    FLL          -9       -14 2013-01-01 21:36:00 2013-01-01 21:45:00
# i 10,623 more rows
# i 3 more variables: arr_time <dttm>, sched_arr_time <dttm>, air_time <dbl>
```

Lets show some of these physically impossible flights times:

```
filtered_flights <- flights_dt |>
  filter(arr_time < dep_time)

filtered_flights_inital <- select(filtered_flights, arr_time, dep_time)

filtered_flights_inital
```

```
# A tibble: 10,633 x 2
   arr_time            dep_time
   <dttm>              <dttm>
 1 2013-01-01 00:03:00 2013-01-01 19:29:00
 2 2013-01-01 00:29:00 2013-01-01 19:39:00
 3 2013-01-01 00:08:00 2013-01-01 20:58:00
 4 2013-01-01 01:46:00 2013-01-01 21:02:00
 5 2013-01-01 00:25:00 2013-01-01 21:08:00
 6 2013-01-01 00:16:00 2013-01-01 21:20:00
 7 2013-01-01 00:06:00 2013-01-01 21:21:00
 8 2013-01-01 00:26:00 2013-01-01 21:28:00
 9 2013-01-01 00:20:00 2013-01-01 21:34:00
10 2013-01-01 00:25:00 2013-01-01 21:36:00
# i 10,623 more rows
```

These are overnight flights. We used the same date information for both the departure and the arrival times, but these flights arrived on the following day. We can fix this by adding `days(1)` to the arrival time of each overnight flight.

```
flights_dt <- flights_dt |>
  mutate(
    overnight = arr_time < dep_time,
    arr_time = arr_time + days(overnight),
    sched_arr_time = sched_arr_time + days(overnight)
```

```
  )
```

Now all of our flights obey the laws of physics.

```
flights_dt |>
  filter(arr_time < dep_time)
```

```
# A tibble: 0 x 10
# i 10 variables: origin <chr>, dest <chr>, dep_delay <dbl>, arr_delay <dbl>,
#   dep_time <dttm>, sched_dep_time <dttm>, arr_time <dttm>,
#   sched_arr_time <dttm>, air_time <dbl>, overnight <lgl>
```

```
filtered_flights_corrected <- select(flights_dt, arr_time, dep_time)

filtered_flights_corrected
```

```
# A tibble: 328,063 x 2
   arr_time            dep_time
   <dttm>              <dttm>
 1 2013-01-01 08:30:00 2013-01-01 05:17:00
 2 2013-01-01 08:50:00 2013-01-01 05:33:00
 3 2013-01-01 09:23:00 2013-01-01 05:42:00
 4 2013-01-01 10:04:00 2013-01-01 05:44:00
 5 2013-01-01 08:12:00 2013-01-01 05:54:00
 6 2013-01-01 07:40:00 2013-01-01 05:54:00
 7 2013-01-01 09:13:00 2013-01-01 05:55:00
 8 2013-01-01 07:09:00 2013-01-01 05:57:00
 9 2013-01-01 08:38:00 2013-01-01 05:57:00
10 2013-01-01 07:53:00 2013-01-01 05:58:00
# i 328,053 more rows
```

Additional exercises

Calculating Elapsed Time between Two Specific Dates:

```
start_date <- ymd("1994-12-21")

end_date <- ymd("2023-11-21")
```

```r
elapsed_time <- end_date - start_date

elapsed_time #difftime class obejct
```

```
Time difference of 10562 days
```

```r
cat("Elapsed time:", as.duration(elapsed_time), "\n")
```

```
Elapsed time: 912556800
```

To enhance clarity in the output, we can represent the elapsed period in terms of days.

```r
# Convert difftime to period

elapsed_period <- as.period(elapsed_time)

# Extracting components of the period

days <- day(elapsed_period)

cat("Elapsed time:", days, "days\n")
```

```
Elapsed time: 10562 days
```

Adding and subtracting months from a date:

```r
future_date <- ymd("2023-05-17")

future_date_plus_3_months <- future_date + months(3)

future_date_plus_3_months
```

```
[1] "2023-08-17"
```

```r
future_date_minus_2_months <- future_date - months(2)

future_date_minus_2_months
```

```
[1] "2023-03-17"
```

Extracting Components of a Date and Formatting a Date:

```
sample_date <- ymd_hms("2023-03-15 12:30:45")

formatted_date <- format(sample_date, "%A, %B %d, %Y %I:%M %p")

formatted_date
```

```
[1] "Wednesday, March 15, 2023 12:30 PM"
```

%A: Represents the full weekday name (e.g., “Sunday”, “Monday”).

%B: Represents the full month name (e.g., “January”, “February”).

%d: Represents the day of the month as a zero-padded decimal number (01, 02, ..., 31).

%Y: Represents the year with century as a decimal number.

%I: Represents the hour (01, 02, ..., 12) using the 12-hour clock.

%M: Represents the minute as a zero-padded decimal number (00, 01, ..., 59).

%p: Represents either “AM” or “PM” in uppercase.

Lubridate supports weekly periods. Here, we create a sequence of weekly periods for the next three weeks. Adding weekly periods to vector of days:

```
date_vector <- c(today(), today() + days(5), today() - weeks(2))

three_weeks = weeks(3)

future_dates <- date_vector + three_weeks

future_dates
```

```
[1] "2023-12-12" "2023-12-17" "2023-11-28"
```

Filtering dates based on a period condition :

```
date_vector <- c(today(), today() + days(5), today() - weeks(2))
```

```r
filtered_dates <- date_vector[date_vector + days(3) < today()]

filtered_dates
```

```
[1] "2023-11-07"
```

If the **period** object explicitly specifies only the number of weeks (**weeks = 2**). When you print the **my_period** object, it represents 14 days, but it doesn't include any information about years, months, hours, minutes, or seconds because those components were not specified.

In Lubridate, when you create a **period** object and only specify one or more components (e.g., weeks), the other components are assumed to be zero. In following case, we only specified weeks, so the result shows 14 days and zeros for other components.

```r
# Create a period of 2 weeks
my_period <- period(weeks = 2)

# Display the period
print(my_period)
```

```
[1] "14d 0H 0M 0S"
```

```r
# Access components of the period
cat("Years:", year(my_period), "\n")
```

```
Years: 0
```

```r
cat("Months:", month(my_period), "\n")
```

```
Months: 0
```

```r
cat("Days:", day(my_period), "\n")
```

```
Days: 14
```

```r
cat("Hours:", hour(my_period), "\n")
```

```
Hours: 0
```

```r
cat("Minutes:", minute(my_period), "\n")
```

```
Minutes: 0
```

```r
cat("Seconds:", second(my_period), "\n")
```

```
Seconds: 0
```

To create a period with more components, you need to specify them explicitly. For example:

```r
# Create a period of 2 weeks, 3 days, 4 hours, 30 minutes, and 15 seconds
my_period <- period(weeks = 2, days = 3, hours = 4, minutes = 30, seconds = 15)

# Display the period
print(my_period)
```

```
[1] "17d 4H 30M 15S"
```

```r
# Access components of the period
cat("Years:", year(my_period), "\n")
```

```
Years: 0
```

```r
cat("Months:", month(my_period), "\n")
```

```
Months: 0
```

```r
cat("Days:", day(my_period), "\n")
```

```
Days: 17
```

```r
cat("Hours:", hour(my_period), "\n")
```

```
Hours: 4
```

```
cat("Minutes:", minute(my_period), "\n")
```

```
Minutes: 30
```

```
cat("Seconds:", second(my_period), "\n")
```

```
Seconds: 15
```

This will create a **period** object that includes the specified values for weeks, days, hours, minutes, and seconds.