Extra material

Working with dates and times

Working with date-time data in R can be challenging due to the unintuitive and inconsistent commands across different date-time objects. Additionally, managing things like time zones, leap days, and daylight saving time can be tricky since R doesn't always handle these well. The `lubridate` and `hms` packages (loaded as part of `tideverse`) simplify date-time operations in R, making it easier to perform common tasks and enabling functionalities that R's base capabilities do not support. Unfortunately, we don't have enough time to cover all the details in this session. Instead, we will only give short introduction on how to work and manipulate date and time variables in R using the `lubridate` and `hms` packages. But if you want to learn more please have a look at the R for Data Science ebook.

First, what do we mean by Date/Time data? well, when we speak of Date/Time data we are mainly referring to three data types:

1. **Date** - a variable containing only the date when an observation was made (e.g. 2024-07-12). More formally, it is a day stored as the number of days since 1970-01-01
2. **Time** - a variable containing only the time when an observation was made (e.g. 18:15:00). Formally , the number of seconds since 00:00:00
3. **Date & Time** - combination of both the date and time (e.g. 2024-07-12 18:15:00). Formally, is a point on the timeline, stored as the number of seconds since 1970-01-01 00:00:00 UTC

There are several ways in which Date-time variables can be created. Here are some examples:

```r
# Example 1: string input with date Y/M/D format
ymd("2017-01-31")
```

```
[1] "2017-01-31"
```

```r
# Example 2: string input with date M/D/Y format
mdy("January 31st, 2017")
```

```
[1] "2017-01-31"
```

```r
# Example 3: string input with date D/M/Y format
dmy("31-Jan-2017")
```

```
[1] "2017-01-31"
```

```r
# Example 4: numeric input with date M/D/Y format
mdy(07082016)
```

```
[1] "2016-07-08"
```

```r
# Examples 5: string input with time H:M formate
hm("20:11")
```

```
[1] "20H 11M 0S"
```

```r
# Example 6:  string input with date-time D/M/Y H:M:S
ymd_hms("2017-01-31 20:11:59")
```

```
[1] "2017-01-31 20:11:59 UTC"
```

In this session, instead of creating data-time variables by ourselves, we will focus on already existing Date/Time Data. Let's look at some of the date-time variables in the `flights` data set, namely the scheduled departure dates and times:

Output

```
# A tibble: 336,776 x 5
   year month   day  hour minute
  <int> <int> <int> <dbl>  <dbl>
1  2013     1     1     5     15
2  2013     1     1     5     29
3  2013     1     1     5     40
```

```
 4  2013     1     1     5     45
 5  2013     1     1     6      0
 6  2013     1     1     5     58
 7  2013     1     1     6      0
 8  2013     1     1     6      0
 9  2013     1     1     6      0
10  2013     1     1     6      0
# i 336,766 more rows
```

R-Code

```
flights_dep <- flights %>%
  select(year, month, day, hour, minute)
flights_dep
```

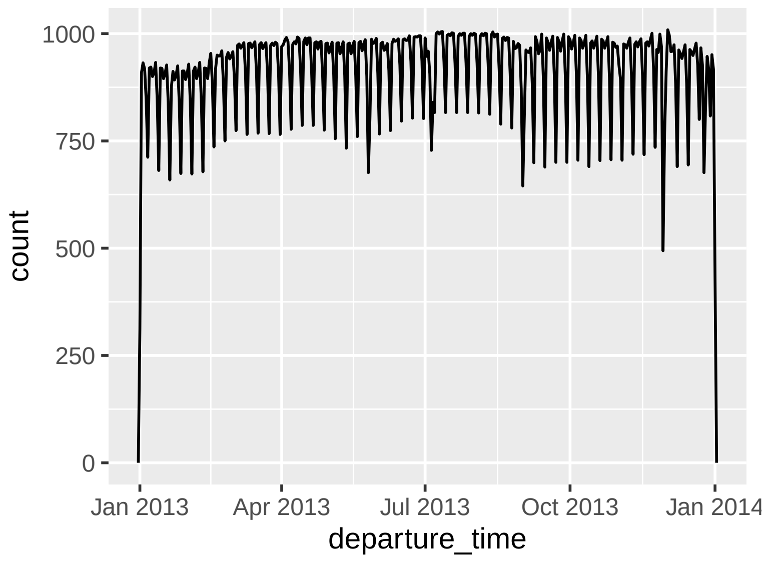
Instead of having separate date-time variables spread across different columns, we can use the `make_date()` or `make_datetime()`functions to create new date and date-time variables respectively:

```
flights_dep <- flights_dep %>%
    mutate(departure_time = make_datetime(year, month, day, hour, minute),
           departure_date = make_date(year, month, day))
flights_dep
```

```
# A tibble: 336,776 x 7
    year month   day  hour minute departure_time      departure_date
   <int> <int> <int> <dbl>  <dbl> <dttm>              <date>
 1  2013     1     1     5     15 2013-01-01 05:15:00 2013-01-01
 2  2013     1     1     5     29 2013-01-01 05:29:00 2013-01-01
 3  2013     1     1     5     40 2013-01-01 05:40:00 2013-01-01
 4  2013     1     1     5     45 2013-01-01 05:45:00 2013-01-01
 5  2013     1     1     6      0 2013-01-01 06:00:00 2013-01-01
 6  2013     1     1     5     58 2013-01-01 05:58:00 2013-01-01
 7  2013     1     1     6      0 2013-01-01 06:00:00 2013-01-01
 8  2013     1     1     6      0 2013-01-01 06:00:00 2013-01-01
 9  2013     1     1     6      0 2013-01-01 06:00:00 2013-01-01
10  2013     1     1     6      0 2013-01-01 06:00:00 2013-01-01
# i 336,766 more rows
```

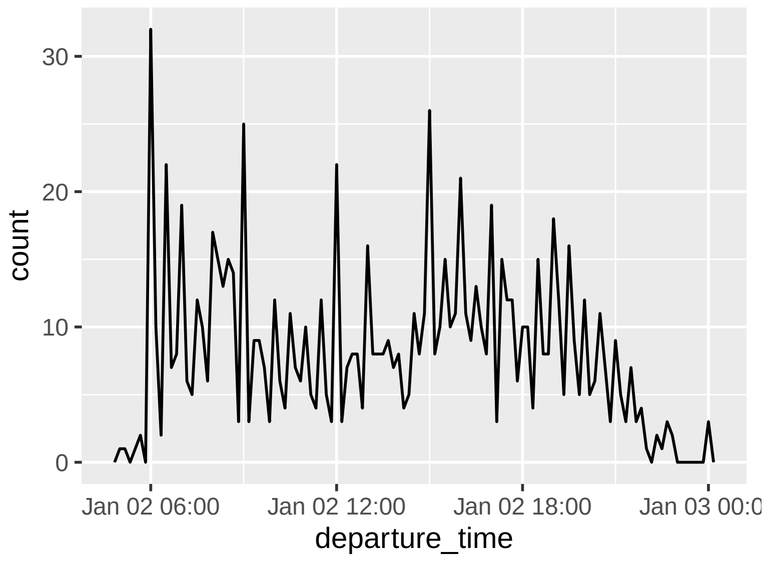
We then can visualize the distribution of the scheduled departure times across the year with ggplot by adding a `geom_freqpoly()` layer (which is similar to an histogram where the counts are displayed with lines instead of bars). Note that when you use date-times in a numeric context (like in a histogram), a binwidth of 1 is equivalent to 1 second, so a binwidth of 86400 is equivalent to one day.

```r
flights_dep %>%
  ggplot(aes(x = departure_time)) +
  geom_freqpoly(binwidth = 86400) # 86400 seconds = 1 day
```



Likewise, if were interested in the distribution of the scheduled departures for a given day:

```r
flights_dep %>%
  filter(departure_date == ymd(20130102)) %>%
  ggplot(aes(x = departure_time)) +
  geom_freqpoly(binwidth = 600) # 600 s = 10 minutes
```



In here, `binwidth = 600` means we are clumping all flights within each 10 minutes (600 s) together into one single data point in our frequency polygon.

Now, notice that in the original `flights` data set, the hour and minute of the actual departure (`dep_time`) and arrival times (`arr_time`) are encoded together into a single integer. Let make a function that sets the actual times in a sensible format:

```r
make_datetime_flights <- function(year, month, day, time) {
  hour <- case_when(
      nchar(time)== 1 ~ time  %/%1,
      nchar(time)== 2 ~ time  %/%10,
      .default =  time  %/%100
    )

  min <- case_when(
      nchar(time)== 1 ~ time  %%1,
      nchar(time)== 2 ~ time  %%10,
      .default =  time  %%100
    )
  make_datetime(year, month, day, hour, min)
}
```

The new `make_datetime_flights()` function we just created separates the hour and minute of a given HM input and pass it on to `make_datetime` function. This is achieved by using a vectorized `case_when` argument based on the number of characters in the integer that uses the %/% or%% operator to find (or discards accordingly) the remainder of an integer division to obtain the hour and minute components (e.g. 951 %/% 100 and 951 %% 100 splits the entry 951 into 9 and 51 (9:15 am once converted to time-date data) while 15 %/% 10 and 15 %% 10 and gives 1 and 5 (equivalent to 1:05 am in date-time format) .

```r
flights_dt <- flights %>%
  filter(!is.na(dep_time), !is.na(arr_time)) %>%
  mutate(
    dep_time = make_datetime_flights(year, month, day, dep_time),
    arr_time = make_datetime_flights(year, month, day, arr_time),
    sched_dep_time = make_datetime_flights(year, month, day, sched_dep_time),
    sched_arr_time = make_datetime_flights(year, month, day, sched_arr_time)
  )

flights_dt %>%
  select(dep_time,arr_time,sched_dep_time,sched_arr_time) %>%
  slice(1:3)
```

```
# A tibble: 3 x 4
  dep_time            arr_time            sched_dep_time
  <dttm>              <dttm>              <dttm>
1 2013-01-01 05:17:00 2013-01-01 08:30:00 2013-01-01 05:15:00
2 2013-01-01 05:33:00 2013-01-01 08:50:00 2013-01-01 05:29:00
3 2013-01-01 05:42:00 2013-01-01 09:23:00 2013-01-01 05:40:00
# i 1 more variable: sched_arr_time <dttm>
```

Extracting individual date-time components

The lubridate package also provide us with different tools for extracting specific components from date-time objects (e.g. year, month, hours, minutes, etc). Suppose we are interested in finding out which day of the week each flight took place. The wday() functions allow us to extract the numeric entry of the day of the week, by including the argument label =TRUE, we can also print the name of the weekday as the output

```r
flights_dt %>%
   select(dep_time,arr_time,sched_dep_time,sched_arr_time) %>%
  mutate(weekday = wday(dep_time,label=TRUE)) %>%
  slice_sample(n=5)
```

```
# A tibble: 5 x 5
  dep_time            arr_time            sched_dep_time
  <dttm>              <dttm>              <dttm>
1 2013-09-14 07:21:00 2013-09-14 10:05:00 2013-09-14 07:29:00
2 2013-04-30 17:11:00 2013-04-30 20:01:00 2013-04-30 17:15:00
3 2013-11-24 15:26:00 2013-11-24 17:16:00 2013-11-24 15:30:00
4 2013-11-25 15:17:00 2013-11-25 17:53:00 2013-11-25 15:20:00
5 2013-07-20 16:59:00 2013-07-20 18:25:00 2013-07-20 16:59:00
# i 2 more variables: sched_arr_time <dttm>, weekday <ord>
```

Here are some more few examples of helper functions that allow you to extract individual date-time components:

```r
datetime <- ymd_hms("2026-07-08 12:34:56")

year(datetime)
```

```
[1] 2026
```

```r
month(datetime,label = TRUE)
```

```
[1] Jul
12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

```r
day(datetime)
```

```
[1] 8
```

```r
hour(datetime)
```

```
[1] 12
```

```r
minute(datetime)
```

```
[1] 34
```

Task

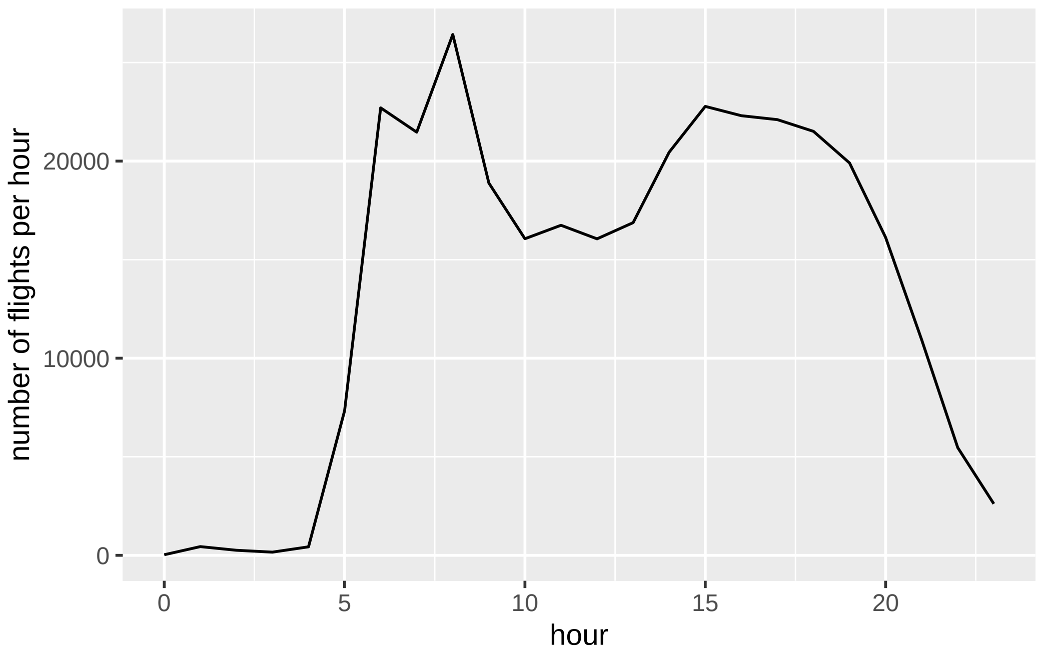
Can you make a plot showing how does the distribution of flight times within a day change over the course of the year? i.e., how many flights have taken off by each hour. Comment on the patterns

Take a hint

Within a day, we want to observe how the flight times differ. This means we should look at how flight times differ by the hour (i.e how many flights are taking off at every hour of the day). You can use the `hour()` function to extract the hours for every departure time and then count (using `summarize()`) how many flights have taken off by each hour. You can visualize the trend using `geom_line` in ggplot.

Click here to see the solution

```
flights_dt %>%
  mutate(hour = hour(dep_time)) %>%
  summarize(numflights_per_hour = n(),.by= hour)%>%
  ggplot(aes(x = hour, y = numflights_per_hour)) +
    geom_line() +
  labs(y="number of flights per hour",x = "hour")
```



We can see there is a peak of flights around 8am, a dip in flights from 10am-12pm, and then a drop off in number of flights past 7pm.

Task

Find out on what day of the week should you leave if you want to minimise the chance of a delay?
Take a hint
To find the days of the week that have the lowest average delay, first you need to assign a day to each observation using `wday()`. You can then use `summarize()` and group by the the day of the week to find the average delay time for each day of the week.
Click here to see the solution

```
flights_dt %>%
  mutate(wday = wday(sched_dep_time, label = TRUE)) %>%
  group_by(wday) %>%
  summarize ( avg_dep_delay_week = mean(dep_delay, na.rm = TRUE),
              avg_arr_delay_week = mean(arr_delay, na.rm = TRUE)) %>%
  slice_min(avg_dep_delay_week,n=1)
```

```
# A tibble: 1 x 3
  wday  avg_dep_delay_week avg_arr_delay_week
  <ord>              <dbl>              <dbl>
1 Sat                 7.62              -1.45
```

```
# Saturday has the lowest average delay at 7.61, and on average the flights even arrive ea
```

We can see there is a peak of flights around 8am, a dip in flights from 10am-12pm, and then a drop off in number of flights past 7pm.

Time intervals, durations and periods

Now that we have seen a few examples of R's date-time data structures, lets look into some of the time span classes.

- Duration: exact number of seconds.
- Periods: human units like weeks and months.
- Intervals: a time span defined by a start and end point.

Duration is simply defined by the exact amount of time between two time events. It does not consider what these two events are in terms of,e.g. calendar years or time zone (so things like leap years would be ignored), and the output is shown in seconds. For example, say we want to manually compute the departure delays in the `flights` data set (we will use the `flights_dt` data frame we created previously which has the `dep_time` and `sched_dep_times` in the correct date-time format).

```
flights_dt %>%
 mutate(
   dep_delay_manual =  dep_time - sched_dep_time) %>%
   select(dep_time,sched_dep_time,dep_delay_manual,dep_delay)  %>%
 slice(1:5)
```

```
# A tibble: 5 x 4
  dep_time            sched_dep_time      dep_delay_manual dep_delay
  <dttm>              <dttm>              <drtn>               <dbl>
1 2013-01-01 05:17:00 2013-01-01 05:15:00  120 secs                2
2 2013-01-01 05:33:00 2013-01-01 05:29:00  240 secs                4
3 2013-01-01 05:42:00 2013-01-01 05:40:00  120 secs                2
4 2013-01-01 05:44:00 2013-01-01 05:45:00  -60 secs               -1
5 2013-01-01 05:54:00 2013-01-01 06:00:00 -360 secs               -6
```

At first glance, we can see that the manually computed departure delays `dep_delay_manual` and the original delays `dep_delay` are not on the same format. By default, when you subtract two dates (e.g. `dep_time - sched_dep_time`), you get a `difftime` object which records a time span of seconds, minutes, hours, days, or weeks. This variability can make `difftime` objects difficult to work with. To address this, we can use convert a `difftime` object to a `duration` class using the `as.duration()` function. Additionally, the original delays `dep_delay`, which are measured in minutes but have no default date-time format, can also be transformed into a duration class using the `duration(units ="")` function.

```
flights_dt %>%
 mutate(
     dep_delay = duration(minute  = dep_delay),
     dep_delay_manual =  as.duration(dep_time - sched_dep_time))%>%
 select(dep_time,sched_dep_time,dep_delay_manual,dep_delay)  %>%
 slice(1:5)
```

```
# A tibble: 5 x 4
  dep_time            sched_dep_time      dep_delay_manual
  <dttm>              <dttm>              <Duration>
1 2013-01-01 05:17:00 2013-01-01 05:15:00 120s (~2 minutes)
2 2013-01-01 05:33:00 2013-01-01 05:29:00 240s (~4 minutes)
3 2013-01-01 05:42:00 2013-01-01 05:40:00 120s (~2 minutes)
4 2013-01-01 05:44:00 2013-01-01 05:45:00 -60s (~-1 minutes)
5 2013-01-01 05:54:00 2013-01-01 06:00:00 -360s (~-6 minutes)
# i 1 more variable: dep_delay <Duration>
```

`Durations` always record the time span in seconds. Instead, `periods` represent time spans without a fixed length in seconds; they work with "human" times, such as days and months. This allows them to operate in a more intuitive manner. periods can be created with different functions, here are some examples:

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

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

```
days(7)
```

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

```
months(1:3)
```

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

Lets see how the output changes when we use `periods` instead of `durations`:

```
flights_dt %>%
  mutate(
      dep_delay = period(minute  = dep_delay),
      dep_delay_manual =  as.period(dep_time - sched_dep_time))%>%
  select(dep_time,sched_dep_time,dep_delay_manual,dep_delay)  %>%
  slice(1:5)
```

```
# A tibble: 5 x 4
  dep_time            sched_dep_time      dep_delay_manual dep_delay
  <dttm>              <dttm>              <Period>         <Period>
1 2013-01-01 05:17:00 2013-01-01 05:15:00 2M 0S            2M 0S
2 2013-01-01 05:33:00 2013-01-01 05:29:00 4M 0S            4M 0S
3 2013-01-01 05:42:00 2013-01-01 05:40:00 2M 0S            2M 0S
4 2013-01-01 05:44:00 2013-01-01 05:45:00 -1M 0S           -1M 0S
5 2013-01-01 05:54:00 2013-01-01 06:00:00 -6M 0S           -6M 0S
```

The last type of time-span defined in `lubridate` are *intervals*. As with *durations*, intervals are expressed in physical time spans defined by a start and end points that are real date-times, i.e. intervals are *durations* defined by a calendar time. Lets suppose we are only given the scheduled departure times and the departure delay. We can create an interval time-span to compute the actual departure time as follows:

```r
flights_dt %>%
  select(sched_dep_time,dep_delay) %>%
  mutate(
      dep_delay_duration = duration(minute  = dep_delay),
      dep_delay_interval= as.interval(x = dep_delay_duration, start= sched_dep_time))%>%
  slice(1:5)
```

```
# A tibble: 5 x 4
  sched_dep_time      dep_delay dep_delay_duration
  <dttm>                  <dbl> <Duration>
1 2013-01-01 05:15:00         2 120s (~2 minutes)
2 2013-01-01 05:29:00         4 240s (~4 minutes)
3 2013-01-01 05:40:00         2 120s (~2 minutes)
4 2013-01-01 05:45:00        -1 -60s (~-1 minutes)
5 2013-01-01 06:00:00        -6 -360s (~-6 minutes)
# i 1 more variable: dep_delay_interval <Interval>
```