Lubridate Package_DateTime

Mohamed Nachid

Using lubridate for dates and times

1 Introduction:

At first glance, dates and times seem simple. You use them all the time in your regular life, and they don't seem to cause much confusion. However, the more you learn about dates and times, the more complicated they seem to get. To warm up, try these three seemingly simple questions:

1.Does every year have 365 days? 2.Does every day have 24 hours? 3.Does every minute have 60 seconds?

Not Every year has 365 days, so there exists a rule for determining if a year is a leap year? (It has three parts.) Also many countries use daylight savings time (DST), so that some days have 23 hours, and others have 25. Note that some minutes have 61 seconds because every now and then leap seconds are added because the Earth's rotation is gradually slowing down.

In raw datasets, dates and times can be included in datasets as different types: strings, separate date-time components, or from an existing date/time object.

In this Tutorial will try to show you how to work with dates and times in R. We'll use tools from the lubridate package that can help us to deal with issues presented by dates and times in our dataset.

1.1 Prerequisites:

We'll be focusing on the lubridate package, which has ability to deal with dates and times in R,make sure in your mind that lubridate is not part of core tidyverse because you only need it when you're working with dates/times so go ahead and load that as well. We will also need nycflights13 for practice data. (If you haven't installed that yet, type install.packages("nycflights13") before you load it.)

Classes for dates and times:

R doesn't have a native class for storing times, It only has a class for dates and date-times. If you want know time classes, look into the hms package.

```
if (!require(tidyverse)) install.packages("tidyverse")
if (!require(lubridate)) install.packages("lubridate")
if (!require(nycflights13)) install.packages("nycflights13")
if (!require(hms)) install.packages("hms")
```

2 Creating date/times

to refer that data is Time, There are three types of date/time data:

- A **date**. *Tibbles print this as* .

- A **time** within a day. *Tibbles print this as* .
- A **date-time** *Tibbles print this as* .

To get the current date use today():

```
today()
```

```
## [1] "2022-02-12"
```

or date-time you can use now():

```
now()
```

```
## [1] "2022-02-12 18:34:26 CET"
```

Otherwise, there are three ways you're likely to create a date/time:

```
From a string.
From individual date-time components.
From an existing date/time object.
```

2.1 From strings

the functions in the "lubridate" package make it confusingly easy to transform a *string* class vector into a *date* class vector. To do this, all you have to do is indicate which elements are filled in the character string (and in what order). This is the name of the function that performs this function. For example, I want R to understand the elements that make up the character string "April 11, 2022": I will therefore tell it that the elements are the day (d like day) the month (m like month) and the year (y as year):

```
today()
```

```
## [1] "2022-02-12"
```

```
class('2022-02-11')
```

```
## [1] "character"
```

```
dmy("11 February 2022")
```

```
## [1] "2022-02-11"
```

```
day <- dmy("11 February 2022")
```

```
class(day)
```

```
## [1] "Date"
```

Switching between date-time and date objects

From other types

Use as_datetime() and as_date():

we may want to switch between a date-time and a date. That's the job of as_datetime() and as_date():

```
as_datetime(today())
```

```
## [1] "2022-02-12 UTC"
```

```
as_date(now())
```

```
## [1] "2022-02-12"
```

Sometimes we'll get date/times as numeric offsets from the "Unix Epoch", 1970-01-01. If the offset is in **seconds**, use **as_datetime()**; if it's in **days**, use **as_date()**.

```
as_datetime(86400) #86400s= 1day
```

```
## [1] "1970-01-02 UTC"
```

```
as_date(365) # 365days= 1 yr
```

```
## [1] "1971-01-01"
```

```
mdy("February 11th, 2022")
```

```
## [1] "2022-02-11"
```

```
ymd("2022-02-11")
```

```
## [1] "2022-02-11"
```

```
dmy("11-Feb-2022")
```

```
## [1] "2022-02-11"
```

```
ymd("20220211")
```

```
## [1] "2022-02-11"
```

These functions also take unquoted numbers. This is the most concise way to create a single date/time object, as you might need when filtering date/time data. ymd() is short and unambiguous:

```
ymd("20220211")
```

```
## [1] "2022-02-11"
```

```r
mdy("02112022")
```

```
## [1] "2022-02-11"
```

```r
dmy("11022022")
```

```
## [1] "2022-02-11"
```

To create a date-time, add h (hour), m (minute), or s (second) to your parsing function:

```r
ymd_hm('2022-Feb-11 18:00')
```

```
## [1] "2022-02-11 18:00:00 UTC"
```

```r
ymd_hms('2022-Feb-11 18:00:11')
```

```
## [1] "2022-02-11 18:00:11 UTC"
```

```r
hms(56, 34, 17)
```

```
## 17:34:56
```

```r
hm("17 55")
```

```
## [1] "17H 55M 0S"
```

```r
hm("17, 55")
```

```
## [1] "17H 55M 0S"
```

Adding a time zone also creates a date-time object:

```r
ymd(20220211, tz = "UTC")
```

```
## [1] "2022-02-11 UTC"
```

Retrieve date elements

Now that we know how to make R understand how to interpret a character string as a date-time, we will be able to perform some simple operations.

To start, we can try to isolate one of the elements of the date-time (just the year, or just the month, or just the hour, etc.).

Here again, it is through the name of the function used that we will specify which element interests us.

```r
t <- ymd_hms("2022.02.11 17h37min52s")
date(t)
```

```
## [1] "2022-02-11"
```

```r
hour(t)
```

```
## [1] 17
```

```r
minute(t)
```

```
## [1] 37
```

```r
second(t)
```

```
## [1] 52
```

Getting individual components of a date or time

You can pull out individual parts of the date and time with year(), month(), mday() (day of the month), yday() (day of the year), wday() (day of the week), hour(), minute(), and second() functions.

```r
t <- ymd_hms("2022.02.11 17h37min52s")
year(t)
```

```
## [1] 2022
```

```r
month(t)
```

```
## [1] 2
```

```r
mday(t)
```

```
## [1] 11
```

```r
yday(t)
```

```
## [1] 42
```

```r
wday(t)
```

```
## [1] 6
```

Setting *label=TRUE* for month() and wday() returns the abbreviated name of the month or day of the week. Try also setting *abbr=FALSE*.

```r
t <- ymd_hms("2022.02.11 17h37min52s")
month(t, label = T, abbr = F )
```

```
## [1] février
## 12 Levels: janvier < février < mars < avril < mai < juin < juillet < ... < décembre
```

```r
wday(t, label = T, abbr = F )
```

```
## [1] vendredi
## 7 Levels: dimanche < lundi < mardi < mercredi < jeudi < ... < samedi
```

Round

You can also round a date, up (ceiling_date()), down (floor_date()), or towards the nearest (round_date()):

Hour

```r
t <- ymd_hms("2022.02.11 17h37min52s")
ceiling_date(t, 'hour')
```

```
## [1] "2022-02-11 18:00:00 UTC"
```

```r
floor_date(t, 'hour')
```

```
## [1] "2022-02-11 17:00:00 UTC"
```

```r
round_date(t, 'hour')
```

```
## [1] "2022-02-11 18:00:00 UTC"
```

```r
t <- ymd_hms("2022.02.11 17h37min52s")
round_date(t, ".5s")
```

```
## [1] "2022-02-11 17:37:52 UTC"
```

```r
round_date(t, "sec")
```

```
## [1] "2022-02-11 17:37:52 UTC"
```

```r
round_date(t, "second")
```

```
## [1] "2022-02-11 17:37:52 UTC"
```

```r
round_date(t, "minute")
```

```
## [1] "2022-02-11 17:38:00 UTC"
```

```r
round_date(t, "5 mins")
```

```
## [1] "2022-02-11 17:40:00 UTC"
```

```r
round_date(t, "hour")
```

```
## [1] "2022-02-11 18:00:00 UTC"
```

```r
round_date(t, "2 hours")
```

```
## [1] "2022-02-11 18:00:00 UTC"
```

```r
round_date(t, "day")
```

```
## [1] "2022-02-12 UTC"
```

```r
round_date(t, "week")
```

```
## [1] "2022-02-13 UTC"
```

```r
round_date(t, "month")
```

```
## [1] "2022-02-01 UTC"
```

```r
round_date(t, "bimonth")
```

```
## [1] "2022-03-01 UTC"
```

```r
round_date(t, "quarter")
```

```
## [1] "2022-01-01 UTC"
```

```r
round_date(t, "halfyear")
```

```
## [1] "2022-01-01 UTC"
```

```r
round_date(t, "year")
```

```
## [1] "2022-01-01 UTC"
```

```r
t <- ymd_hms("2022.02.11 17h37min52s")
floor_date(t, ".5s")
```

```
## [1] "2022-02-11 17:37:52 UTC"
```

```r
floor_date(t, "sec")
```

```
## [1] "2022-02-11 17:37:52 UTC"
```

```r
floor_date(t, "second")
```

```
## [1] "2022-02-11 17:37:52 UTC"
```

```r
floor_date(t, "minute")
```

```
## [1] "2022-02-11 17:37:00 UTC"
```

```r
floor_date(t, "5 mins")
```

```
## [1] "2022-02-11 17:35:00 UTC"
```

```r
floor_date(t, "hour")
```

```
## [1] "2022-02-11 17:00:00 UTC"
```

```r
floor_date(t, "2 hours")
```

```
## [1] "2022-02-11 16:00:00 UTC"
```

```r
floor_date(t, "day")
```

```
## [1] "2022-02-11 UTC"
```

```r
floor_date(t, "week")
```

```
## [1] "2022-02-06 UTC"
```

```r
floor_date(t, "month")
```

```
## [1] "2022-02-01 UTC"
```

```r
floor_date(t, "bimonth")
```

```
## [1] "2022-01-01 UTC"
```

```r
floor_date(t, "quarter")
```

```
## [1] "2022-01-01 UTC"
```

```r
floor_date(t, "halfyear")
```

```
## [1] "2022-01-01 UTC"
```

```r
floor_date(t, "year")
```

```
## [1] "2022-01-01 UTC"
```

```r
t <- ymd_hms("2022.02.11 17h37min52s")
ceiling_date(t, ".5s")
```

```
## [1] "2022-02-11 17:37:52 UTC"
```

```r
ceiling_date(t, "sec")
```

```
## [1] "2022-02-11 17:37:52 UTC"
```

```r
ceiling_date(t, "second")
```

```
## [1] "2022-02-11 17:37:52 UTC"
```

```r
ceiling_date(t, "minute")
```

```
## [1] "2022-02-11 17:38:00 UTC"
```

```r
ceiling_date(t, "5 mins")
```

```
## [1] "2022-02-11 17:40:00 UTC"
```

```r
ceiling_date(t, "hour")
```

```
## [1] "2022-02-11 18:00:00 UTC"
```

```r
ceiling_date(t, "2 hours")
```

```
## [1] "2022-02-11 18:00:00 UTC"
```

```r
ceiling_date(t, "day")
```

```
## [1] "2022-02-12 UTC"
```

```r
ceiling_date(t, "week")
```

```
## [1] "2022-02-13 UTC"
```

```r
ceiling_date(t, "month")
```

```
## [1] "2022-03-01 UTC"
```

```r
ceiling_date(t, "bimonth")
```

```
## [1] "2022-03-01 UTC"
```

```r
ceiling_date(t, "quarter")
```

```
## [1] "2022-04-01 UTC"
```

```r
ceiling_date(t, "halfyear")
```

```
## [1] "2022-07-01 UTC"
```

```r
ceiling_date(t, "year")
```

```
## [1] "2023-01-01 UTC"
```

Periods or durations

```r
t1 <- dmy("17/07/2019")
t2 <- dmy("17/04/2022")
diff <- t2-t1
class(diff)
```

```
## [1] "difftime"
```

The diff object tells us about the "time difference" between t1 and t2. It is an object of class difftime (class which is not specifically related to the use of lubridate). This "time difference" can be addressed in various ways by lubridate. We can indeed consider this difference in terms of period or in terms of duration.

It is specified as follows:

```r
as.duration(diff)
```

```
## [1] "86832000s (~2.75 years)"
```

```r
as.period(diff)
```

```
## [1] "1005d 0H 0M 0S"
```

Arithmetic calculations with periods or durations

In the previous part, we have already seen that it was possible to perform arithmetic operations on dates. Whatever the operation carried out, it is important to keep in mind the distinction between period and duration! The periods correspond to the xxx() functions (for example days() or months()) while the durations correspond to the dxxx() functions (for example ddays() or dyears())

```
t1+months(9)
```

```
## [1] "2020-04-17"
```

```
t1+ddays(268)
```

```
## [1] "2020-04-10"
```

```
ddays(268)/dweeks(1) # how week we have in 268 days
```

```
## [1] 38.28571
```

```
t2-dweeks(3)
```

```
## [1] "2022-03-27"
```

Note that these functions also allow you to create series at regular time intervals:

```
t1+months(1:9)
```

```
## [1] "2019-08-17" "2019-09-17" "2019-10-17" "2019-11-17" "2019-12-17"
## [6] "2020-01-17" "2020-02-17" "2020-03-17" "2020-04-17"
```

```
now()+minutes(seq(0,30,by=10))
```

```
## [1] "2022-02-12 18:34:27 CET" "2022-02-12 18:44:27 CET"
## [3] "2022-02-12 18:54:27 CET" "2022-02-12 19:04:27 CET"
```

Time intervals

Another way to look at analyzing a dataset that includes dates is to work on time intervals.

To transform two dates into a time interval with lubridate, we have two solutions (the interval() function, or the %–% operator. In both cases, we obtain the same result:

```
itv <- interval(t1,t2)
itv1 <- t1 %--% t2
itv
```

```
## [1] 2019-07-17 UTC--2022-04-17 UTC
```

```
itv1
```

```
## [1] 2019-07-17 UTC--2022-04-17 UTC
```

Having an interval allows you to perform certain operations, such as (for example) determining whether a given date (or "date-time") is part of the interval:

```
d1 <- dmy("04/12/2019")
d1 %within% itv
```

```
## [1] TRUE
```

One of the operations,the most useful in connection with these intervals, is thus to allow the occurrence of one or more events to be replaced in time intervals:

```
## [1] 2019-08-17 UTC--2019-09-17 UTC 2019-09-17 UTC--2019-10-17 UTC
## [3] 2019-10-17 UTC--2019-11-17 UTC 2019-11-17 UTC--2019-12-17 UTC
## [5] 2019-12-17 UTC--2020-01-17 UTC 2020-01-17 UTC--2020-02-17 UTC
## [7] 2020-02-17 UTC--2020-03-17 UTC 2020-03-17 UTC--2020-04-17 UTC
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
```

==========

We can use each accessor function to set the components of a date/time:

```
## [1] "2019-07-08 12:34:56 UTC"
```

```
## [1] "2022-07-08 12:34:56 UTC"
```

```
## [1] "2022-02-08 12:34:56 UTC"
```

```
## [1] "2022-02-08 13:34:56 UTC"
```

Alternatively, rather than modifying in place, you can create a new date-time with update(). This also allows you to set multiple values at once.

```
## [1] "2019-07-08 12:34:56 UTC"
```

```
## [1] "2022-02-11 02:34:56 UTC"
```

If values are too big, they will roll-over:

```
## [1] "2019-03-02"
```

```
## [1] "2019-02-17 16:00:00 UTC"
```

2.2 Applying the above codes

Instead of a single string, sometimes you'll have the individual components of the date-time spread across multiple columns. This is what we have in the flights data:

```
## # 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
## # ... with 336,766 more rows
```

Using mutate() and make_date() or make_datetime(), we can create date/time objects:

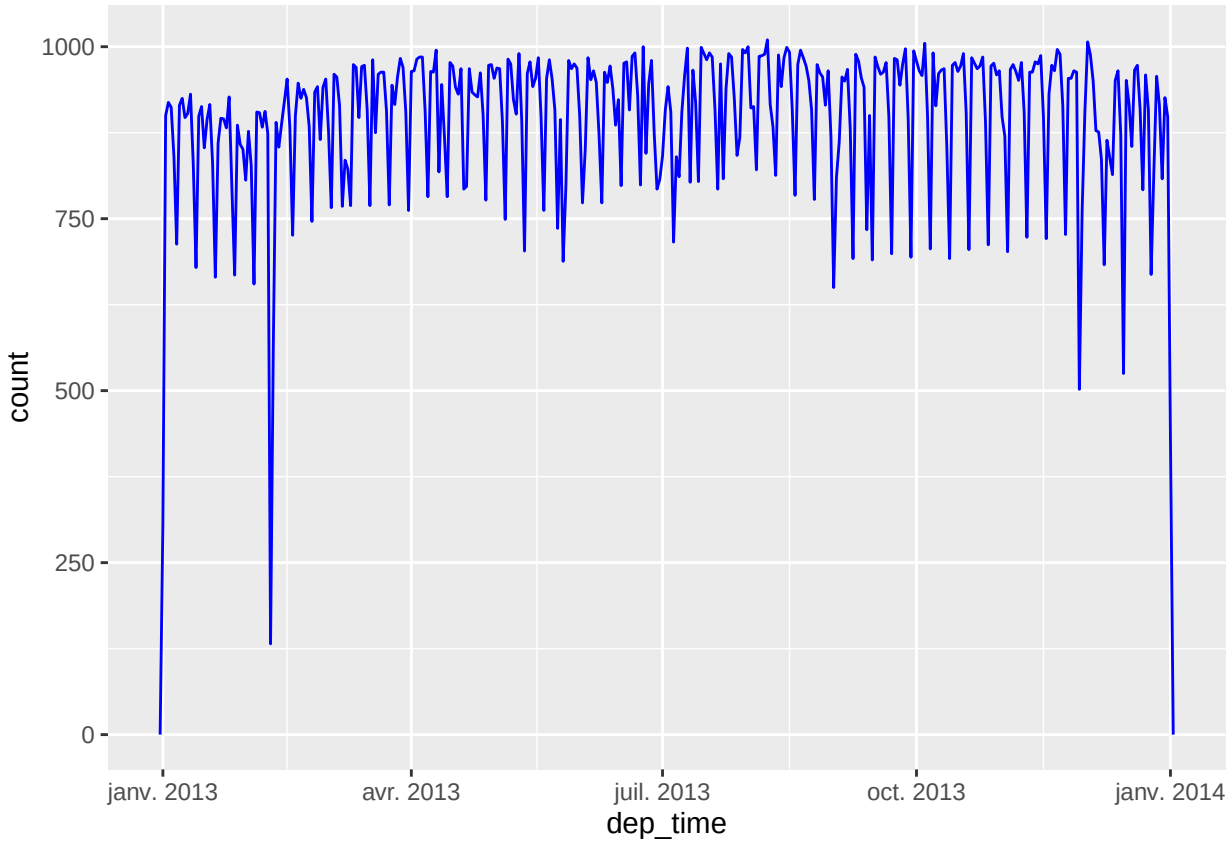
```
## # A tibble: 336,776 x 6
##     year month   day  hour minute departure
##    <int> <int> <int> <dbl>  <dbl> <dttm>
##  1  2013     1     1     5     15 2013-01-01 05:15:00
##  2  2013     1     1     5     29 2013-01-01 05:29:00
##  3  2013     1     1     5     40 2013-01-01 05:40:00
##  4  2013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

Let's do the same thing for each of the four time columns in -*flights*-. 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 Tutorial.

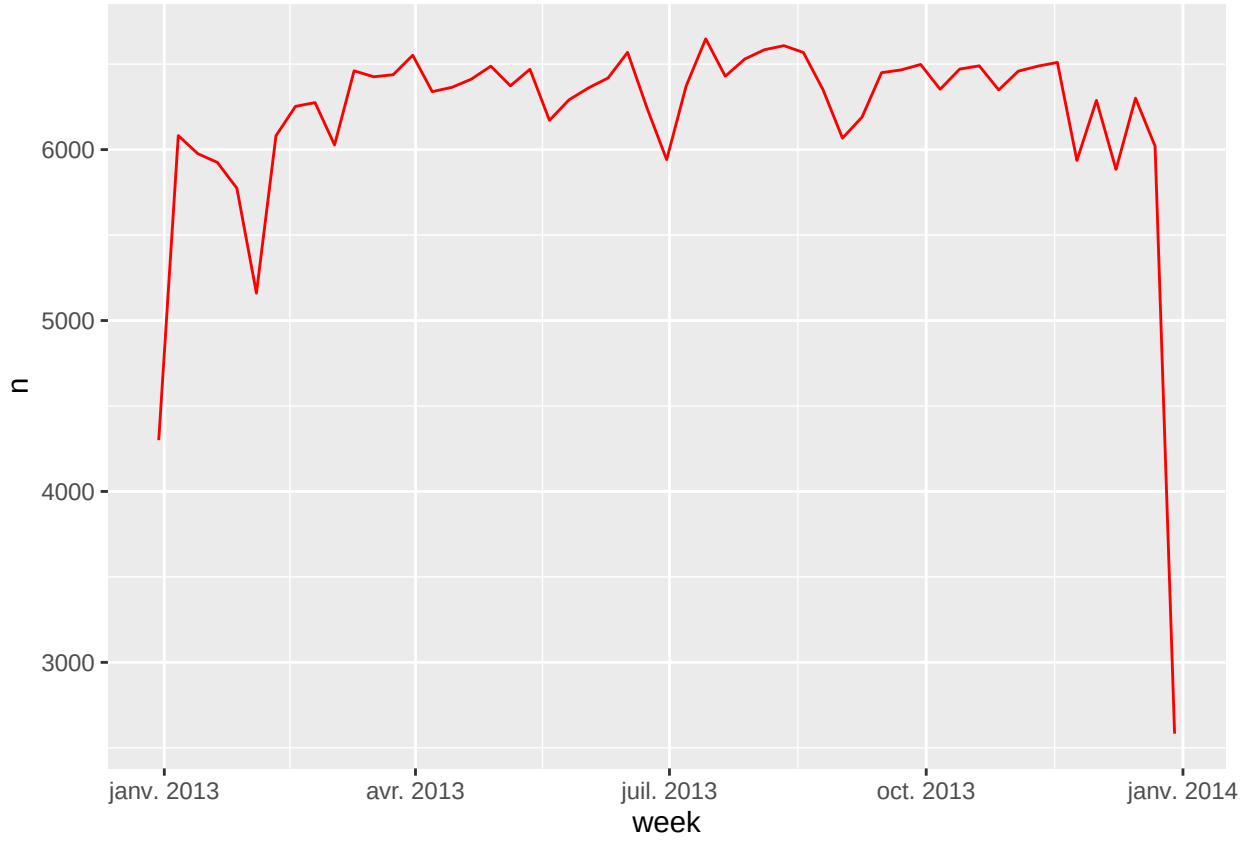
```
## # A tibble: 328,063 x 9
##    origin dest  dep_delay arr_delay dep_time            sched_dep_time
##    <chr>  <chr>     <dbl>     <dbl> <dttm>              <dttm>
##  1 EWR    IAH           2        11 2013-01-01 05:17:00 2013-01-01 05:15:00
##  2 LGA    IAH           4        20 2013-01-01 05:33:00 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
## # ... with 328,053 more rows, and 3 more variables: arr_time <dttm>,
## #   sched_arr_time <dttm>, air_time <dbl>
```

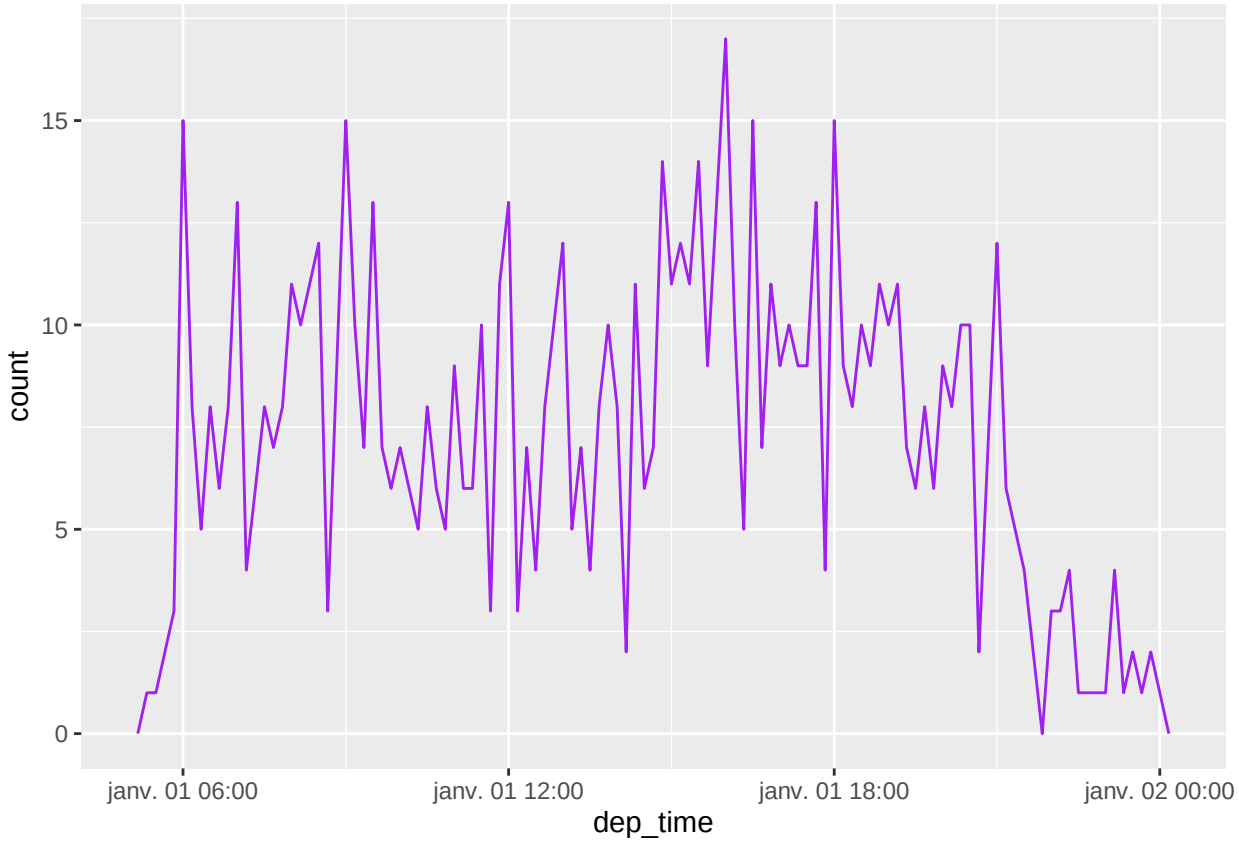
The visualisation of the distribution of departure times across the year:



The code below rounds the flight times to the nearest week!

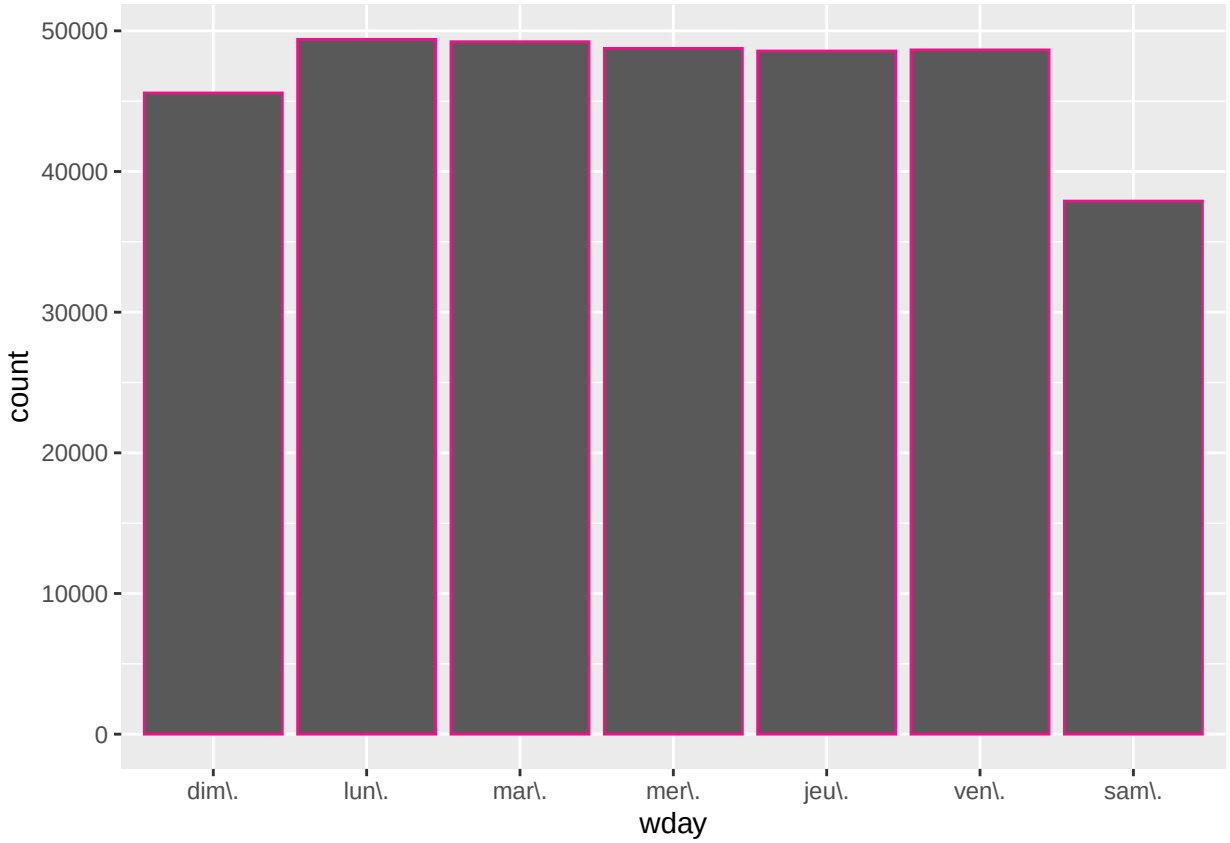


Or within a single day:

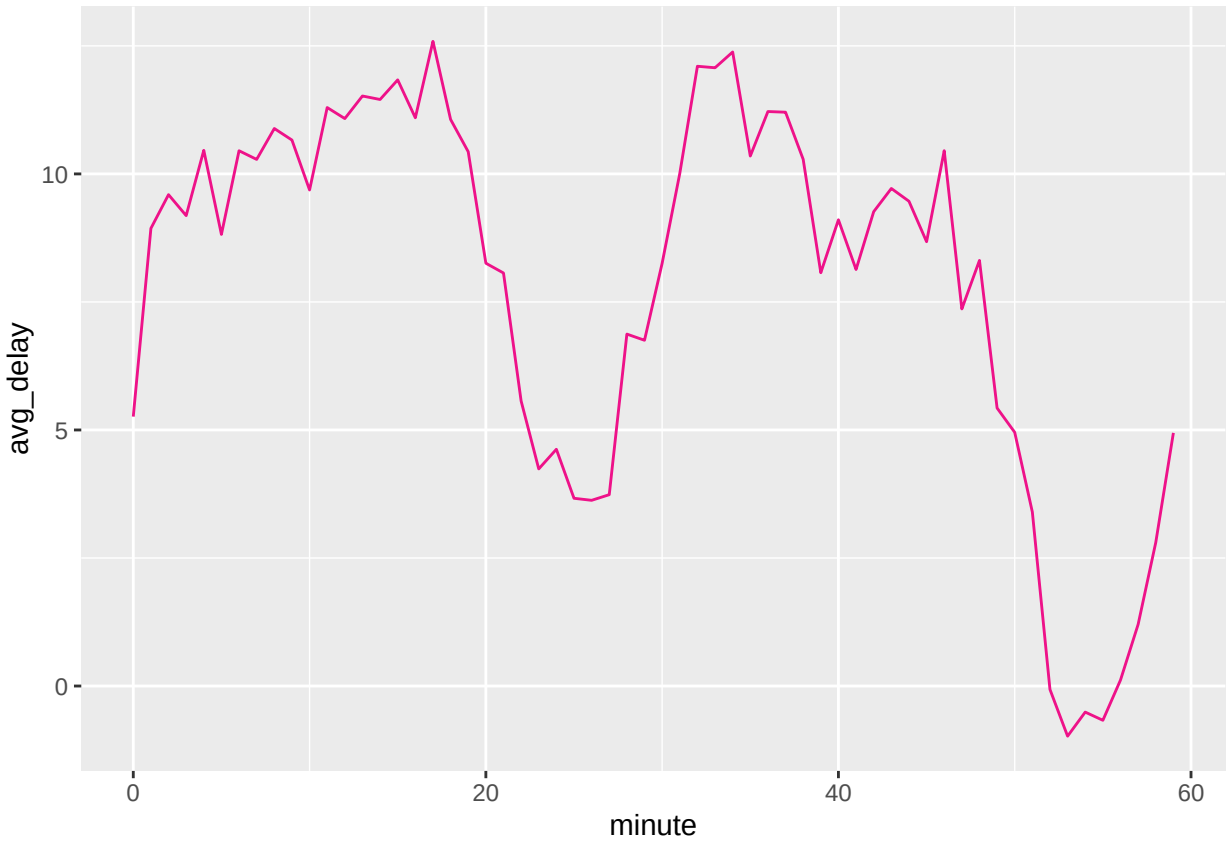


Note that when you use date-times in a numeric context (like in a histogram), 1 means 1 second, so a binwidth of 86400 means one day. For dates, 1 means 1 day.

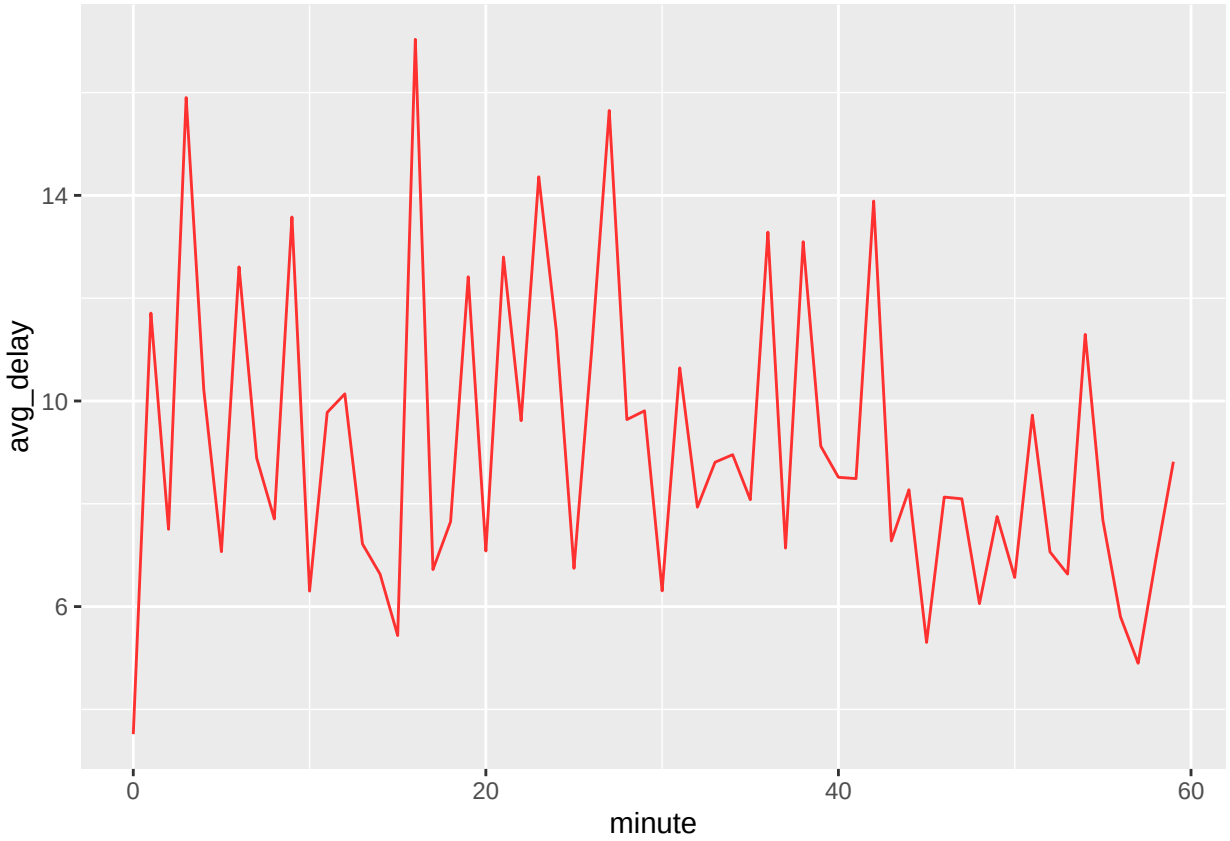
We can use wday() to see that more flights depart during the week than on the weekend:



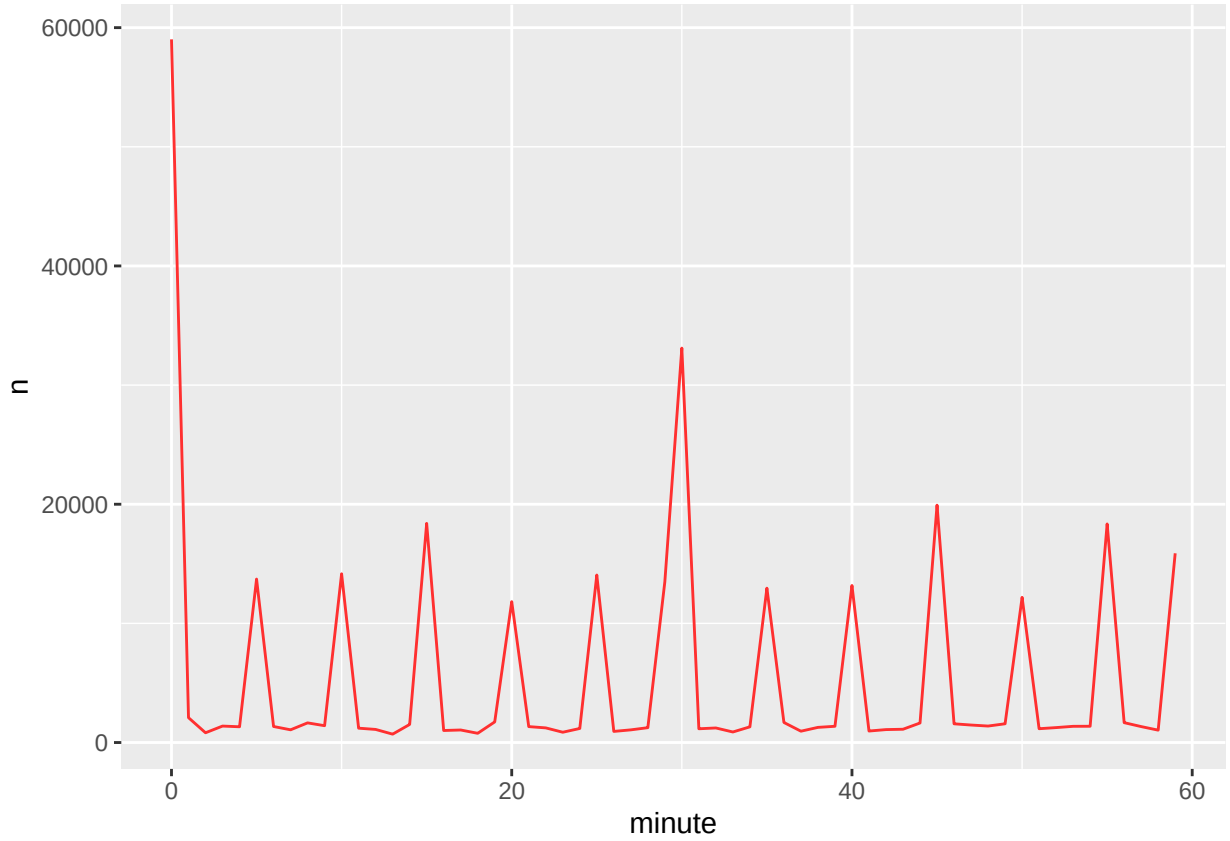
There's an interesting pattern if we look at the average departure delay by minute within the hour. It looks like flights leaving in minutes 20-30 and 50-60 have much lower delays than the rest of the hour!



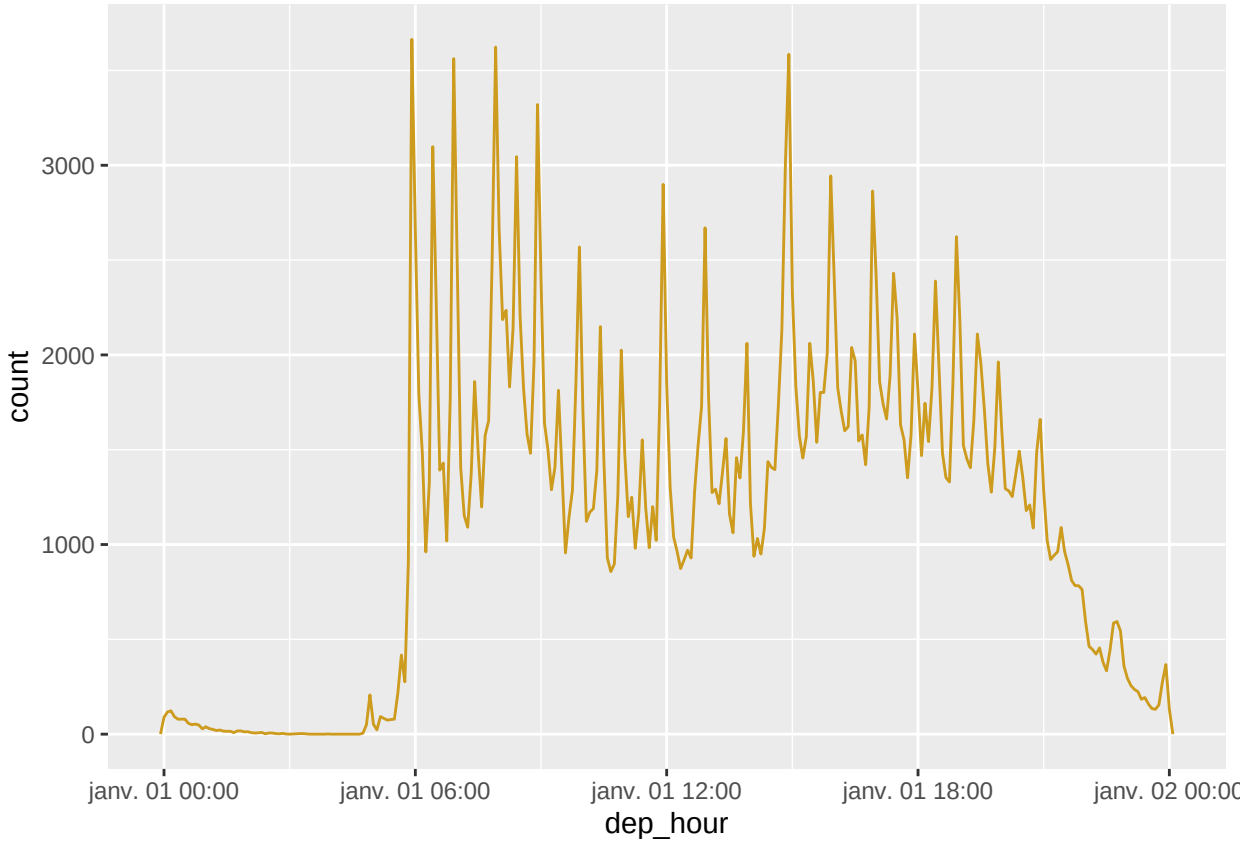
Interestingly, if we look at the scheduled departure time we don't see such a strong pattern:



So why do we see that pattern with the actual departure times? Well, like much data collected by humans, there's a strong bias towards flights leaving at "nice" departure times. Always be alert for this sort of pattern whenever you work with data that involves human judgement!



We can use update() to show the distribution of flights across the course of the day for every day of the year:



We can use periods to fix an oddity in the flight dataset: some flights appear to arrive at their destination before they departed from New York City. This is because they are overnight flights. We can address the issue by adding one day to all of the overnight flights.

```
## # 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
## # ... with 10,623 more rows, and 3 more variables: arr_time <dttm>,
## #   sched_arr_time <dttm>, air_time <dbl>
```

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.

Now all of our flights obey the laws of physics.

```
## # A tibble: 0 x 10
```

```
## # ... with 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>
```