# Working with Date & Time Variables

Dates and time variables often come up when you have data on the timing of particular events such as when a flight took off, or when a particular weather event happened, or what the price of a given stock was on a given day. R provides a nice suite of tools for working with this sort of data that we'll explore in this lecture. In particular, we'll work with the `lubridate` package. It is part of the tidyverse, but it needs to be loaded separately when you're going to use it.

Basically, there are 3 types of variables that refer to days and times:

1. A **date**, denoted in R as `<date>`.

2. A **time**, denoted in R as `<time>`

3. A **date-time**, denoted in R as `<dttm>` (or, confusingly, as POSICxt, but don't worry about that right now)

These are pretty self-explanatory, but as you'll see below, because dates and times can be entered in multiple different ways, there are some complications.

## Creating Dates and Times Using the `ymd()` Function

Part of the reason why dates and times are more complicated than you might expect is that you can write them in many different ways. For example, the following are all equally valid ways of writing the same date:

- January 31, 2019
- 31 January 2019
- 31 Jan 2019
- 01/31/19
- 01/31/2019

And there are even more possibilities: I could have 31 Jan 19, for example! The easiest way to create dates and times is with the `ymd` function in `lubridate`. This function expects a year `y`, a month `m`, and a day `d` and combines them into a date. It's actually quite flexible:

```
library(lubridate)

##
## Attaching package: 'lubridate'

## The following object is masked from 'package:base':
##
##     date
```

```r
ymd("2019-01-31")

## [1] "2019-01-31"

mdy("January 31, 2019")

## [1] "2019-01-31"

dmy("31-Jan-19")

## [1] "2019-01-31"

dmy("31-01-19")

## [1] "2019-01-31"
```

So notice that `ymd` (and its variants) can create dates from basically any format commonly used to supply dates. Note that if you put the day first (as is 31 January 2019), then the function becomes `dmy`, and so forth.

If you need a date-time, you can create it with the `ymd_hms` function (for hours, minutes, seconds):

```r
ymd_hms("2019-01-31 07:11:12")

## [1] "2019-01-31 07:11:12 UTC"

ymd_hm("2019-01-31 07:11")

## [1] "2019-01-31 07:11:00 UTC"
```

If you're interested, there's another function called `parse_date()` covered in Chapter 11 of the textbook that is more powerful but less user-friendly. I think `ymd()` is easier to use, so I'd stick to that for now, but you may disagree, so it's worth consulting the textbook.

### What Happens If Excel Produces Wacky Non-Dates?

Sometimes, you'll read in an Excel file that contains date variables that have values like `41103` instead of an actual date value (this is because of how Excel internally stores dates, see https://stackoverflow.com/questions/31647364/data-difference-in-as-posixct-with-excel). If that happens, and `ymd` does not work, you can try the `excel_numeric_to_date()` function inside the `janitor` package. This is a rare case, but one worth mentioning as it does arise from time to time.

### Test Yourself: Trump Approval Data

FiveThirtyEight provides a public repository of all of the polls tracking Donald Trump's approval rating over time. I've saved a version of this data on Canvas called `538_Trump_Approval.csv`. Download and read the data into R, and convert `modeldate` and `timestamp` to a date and a date-time, respectively. Both of these record the date the poll was conducted.

**Answer:** The R code appears below:

```
setwd("~/Dropbox/DATA101")
trump <- read_csv(file="Data/Raw/538_Trump_Approval.csv")

## Parsed with column specification:
## cols(
##   president = col_character(),
##   subgroup = col_character(),
##   modeldate = col_character(),
##   approve_estimate = col_double(),
##   approve_hi = col_double(),
##   approve_lo = col_double(),
##   disapprove_estimate = col_double(),
##   disapprove_hi = col_double(),
##   disapprove_lo = col_double(),
##   timestamp = col_character()
## )

trump <- trump %>% mutate(poll_date=mdy(modeldate),
                          poll_datetime=mdy_hm(timestamp))
select(trump,poll_date,poll_datetime,everything())

## # A tibble: 1,935 x 12
##    poll_date  poll_datetime       president subgroup modeldate
##    <date>     <dttm>              <chr>     <chr>    <chr>
##  1 2018-10-29 2018-10-29 10:02:00 Donald T… Voters   10/29/18
##  2 2018-10-29 2018-10-29 10:02:00 Donald T… Adults   10/29/18
##  3 2018-10-29 2018-10-29 10:01:00 Donald T… All pol… 10/29/18
##  4 2018-10-28 2018-10-28 08:27:00 Donald T… Adults   10/28/18
##  5 2018-10-28 2018-10-28 08:27:00 Donald T… Voters   10/28/18
##  6 2018-10-28 2018-10-28 08:26:00 Donald T… All pol… 10/28/18
##  7 2018-10-27 2018-10-27 14:38:00 Donald T… Voters   10/27/18
##  8 2018-10-27 2018-10-27 14:38:00 Donald T… Adults   10/27/18
##  9 2018-10-27 2018-10-27 14:37:00 Donald T… All pol… 10/27/18
## 10 2018-10-26 2018-10-26 21:21:00 Donald T… All pol… 10/26/18
## # … with 1,925 more rows, and 7 more variables: approve_estimate <dbl>,
## #   approve_hi <dbl>, approve_lo <dbl>, disapprove_estimate <dbl>,
## #   disapprove_hi <dbl>, disapprove_lo <dbl>, timestamp <chr>
```

Now our new variable `poll_date` is a `<date>` variable, while the original `modeldate` variable is a character (`<chr>`) variable. This means that we can use `poll_date` easily as a date variable. You should understand what the final call to `select()` is doing (if not, refer back to your notes on `dplyr` from earlier in the course).

Note that I called these variables `poll_date` and `poll_datetime` rather than (say) `date`. `Date` is a function in R, so I'll get errors when I try use a variable called `date`. Giving the variable a more distinctive name is helpful and avoids errors.

In the homework, you'll use this data to draw some inferences about Trump's approval over time.

## Creating Dates & Times Using `make_datetime()`

Sometimes, you'll find data where the date or date-time is entered as a string, and you'll use the the `ymd` function above. But sometimes you'll have that same information spread across several variables. For example, in our `flights` data from the `nycflights13` dataset, we have the year, month, day, hour and minute each flight left, but they are all in separate variables! We'll need to combine them to create date and time objects. We'll do this with the `make_date()` and `make_datetime()` functions:

```
library(nycflights13)
flights %>%
  select(year, month, day, hour, minute) %>%
  mutate(departure = make_datetime(year, month, day, hour, minute))

## # 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
```

Note that R now understands `departure` to be a date-time variable type.

You can do this same calculation for the actual departure and arrival times, as well as the scheduled departure and arrival times. But let's look at those variables to see how they are formatted:

```
select(flights, c(dep_time, sched_dep_time, arr_time, sched_arr_time))

## # A tibble: 336,776 x 4
##    dep_time sched_dep_time arr_time sched_arr_time
##       <int>          <int>    <int>          <int>
##  1      517            515      830            819
##  2      533            529      850            830
##  3      542            540      923            850
##  4      544            545     1004           1022
##  5      554            600      812            837
##  6      554            558      740            728
##  7      555            600      913            854
##  8      557            600      709            723
##  9      557            600      838            846
## 10      558            600      753            745
## # … with 336,766 more rows
```

So we have times entered in strange formats. For example, in the first observation, the flight was scheduled to depart at 5:15 AM, and actually departed at 5:17 AM. But note how that gets formatted: 515 and 517, respectively. So to convert these to actual date-time variables, we'll need to reformat them so that R recognizes them as times. To do this, we'll use the modular arithmetic tools we learned in our `dplyr` library earlier in the term.

```r
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"))
```

Note that to do this, I've written my own function `make_datetime_100`, which uses our modular arithmetic tools to properly format the departure and arrival time for us. For example, suppose a flight left at 5:15. In our `dep_time` variable, that will be recorded as 515. So then if we say:

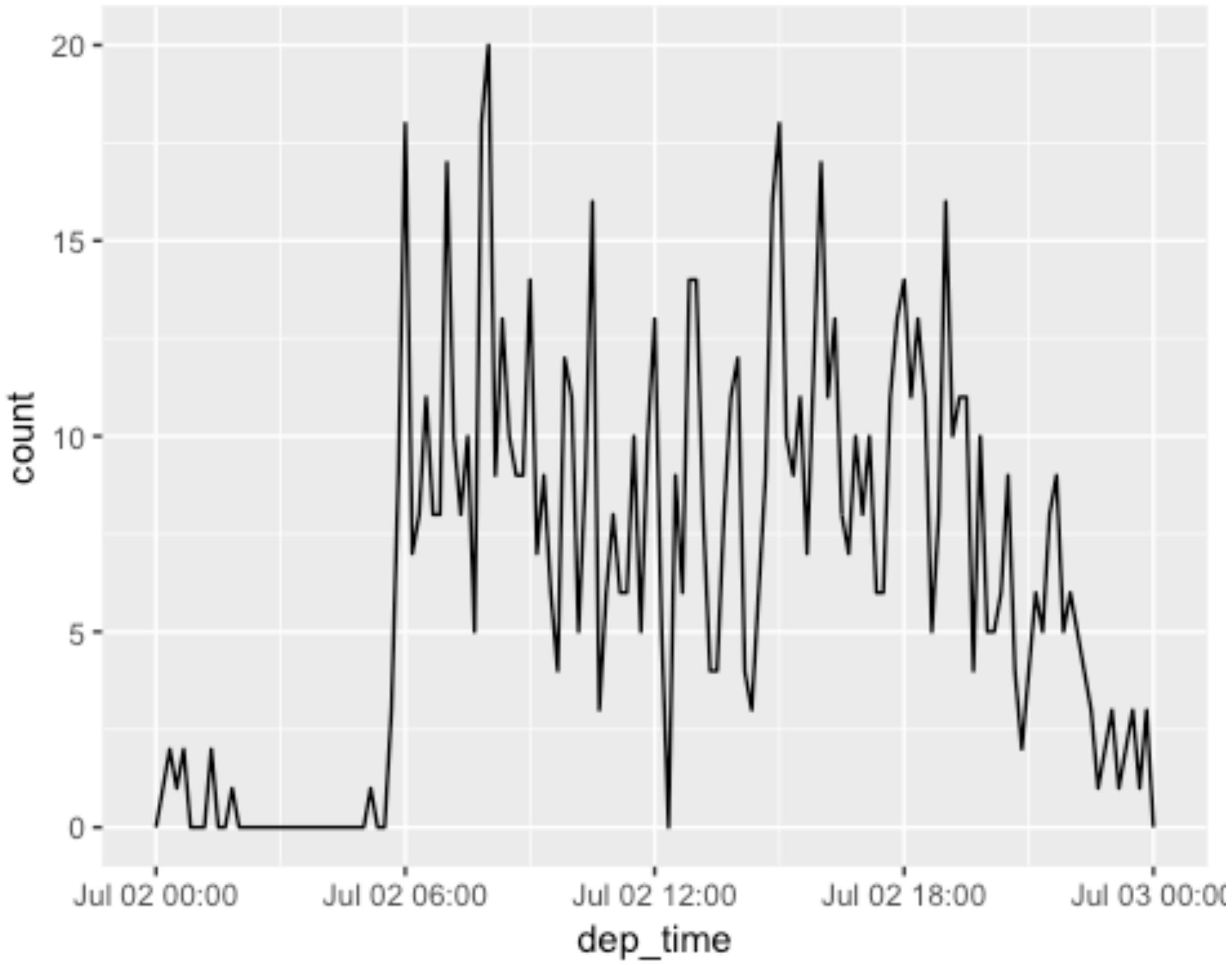
```r
515 %/% 100

## [1] 5

515 %% 100

## [1] 15
```

So the first line divides 515 by 100, and returns the quotient, which here is the hour. The second line divides 515 by 100 and gives the remainder, which here is the minutes. So this function just extracts the minutes and the hour from a slightly oddly formulated variable. Is this annoying? Absolutely. But this is also the reality of working with data! Doing these sorts of calculations make me appreciate nicely formatted data.

We haven't yet talked about what it means to write your own function in R, but we will in a future lecture. The idea of a function is that it's a way to automate something that you're going to do several times. Here, I needed to reformat 4 variables: scheduled departure time, actual departure time, scheduled arrival time, and actual arrival time. Each has a similar input format, and we want all of them in a similar output format. So by writing a function, I can simplify my workflow: otherwise, I'd have to write the same code 4 times. Writing the function just makes my life easier, and makes it less likely that I'll make a mistake (since I'm only writing the code once instead of four times).

Once you have date variables in R, plotting time trends becomes quite simple. For example, I can plot the time trend in departures on July 2nd, 2013:

```
flights_dt %>%
  filter(dep_time < ymd(20130703) & dep_time > ymd(20130702)) %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(binwidth = 600) # 600 s = 10 minutes
```



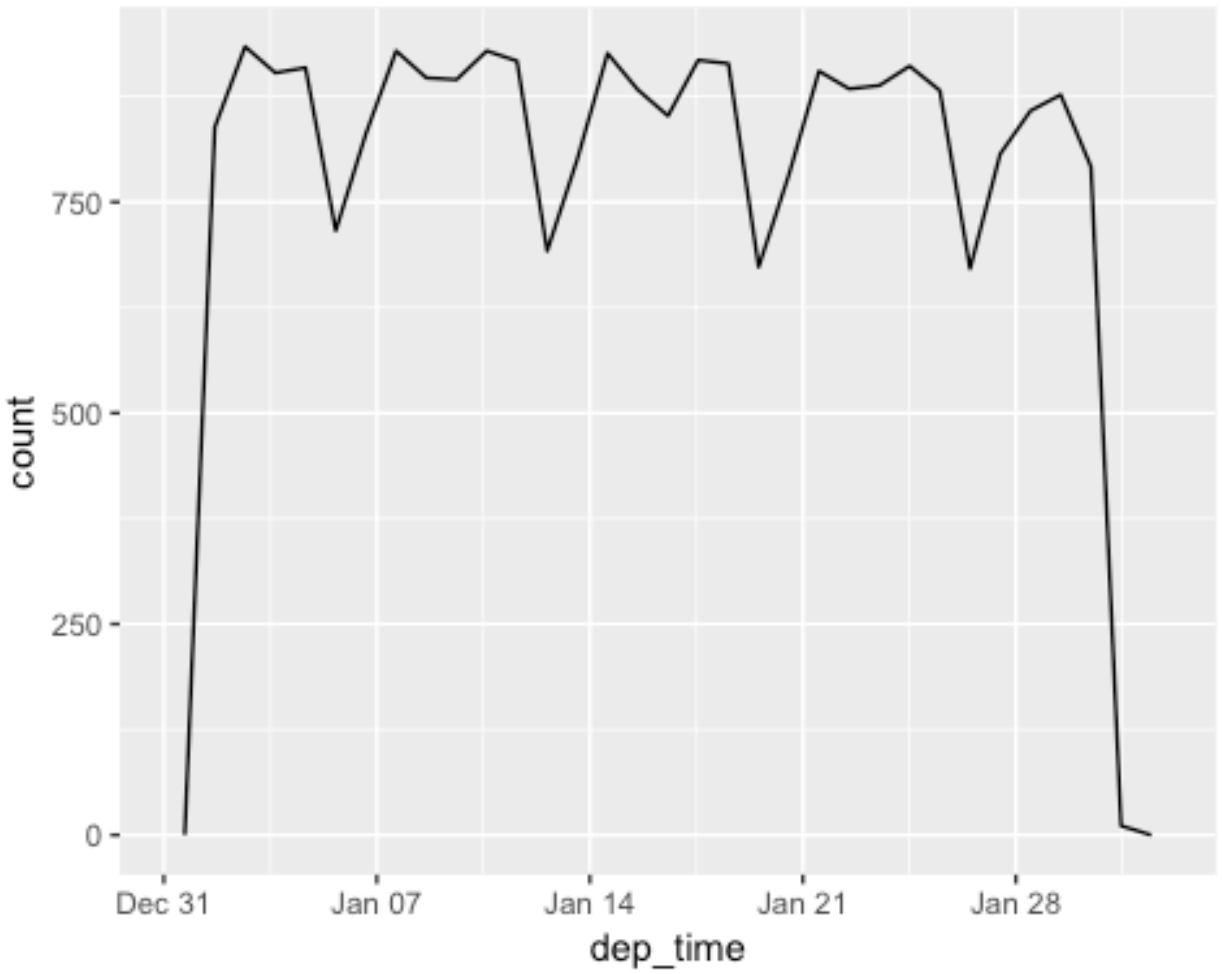
Remember that `geom_freqpoly()` is the analogue to a boxplot with lines (look back at our plotting lecture for more detail on this type of geom, or search R's help). This plot shows us (in 600 second, or 10 minute, bins) the number of flights that took off from one of our three airports on July 2nd, 2013. There are a few flights that leave shortly after midnight (delays from the night before), then it's quiet until around 6 AM, when the morning rush begins. Things slow down around mid-day a bit, pick up in the afternoon/evening, and then taper off toward midnight.

### Test Yourself: Flights in January

Show a plot of the number of flights per day in January.

**Answer**

```
flights_dt %>%
  filter(dep_time < ymd(20130131)) %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(bins=31)
```



Note that here, I wanted to change my `bins` argument to 31, so that R will plot the number of flights on each day. Notice that the number of flights is pretty consistent, averaging around 900 flights (give or take) per day. But there are drops most weeks. In the homework, you'll explore why that might be (think about whether some days have more flights than others).

## Splitting Components of Dates & Times

Above, we saw how to bring together information in several variables to create a date or date-time variable. But we can also split apart a date or date into its component pieces. The functions to do this are all intuitively named, as we can see:

```
dt <- ymd_hms("2019-06-18 05:15:58")
year(dt)

## [1] 2019

month(dt)

## [1] 6

mday(dt)

## [1] 18

yday(dt)

## [1] 169

wday(dt)

## [1] 3

hour(dt)

## [1] 5

minute(dt)

## [1] 15
```

So, for example, we can see that June 18th, 2019 was the 18th day of June (`mday`), but the 169th day of the year (`yday`). But to me, the coolest part of this function is the `wday()` function, which tells us which day of the week it is. If I want to see the day itself, rather than the number, I can use the `label=True` option, and adding `abbr = F` returns the full name, rather than the abbreviation:

```
wday(dt)

## [1] 3

wday(dt, label=T)

## [1] Tue
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat

wday(dt, label=T, abbr=F)

## [1] Tuesday
## 7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... <
Saturday
```

### Test Yourself: Days of the Week in the `flights` Data

Find out the day of the week of each flight in the `flights` dataset.

**Answer**

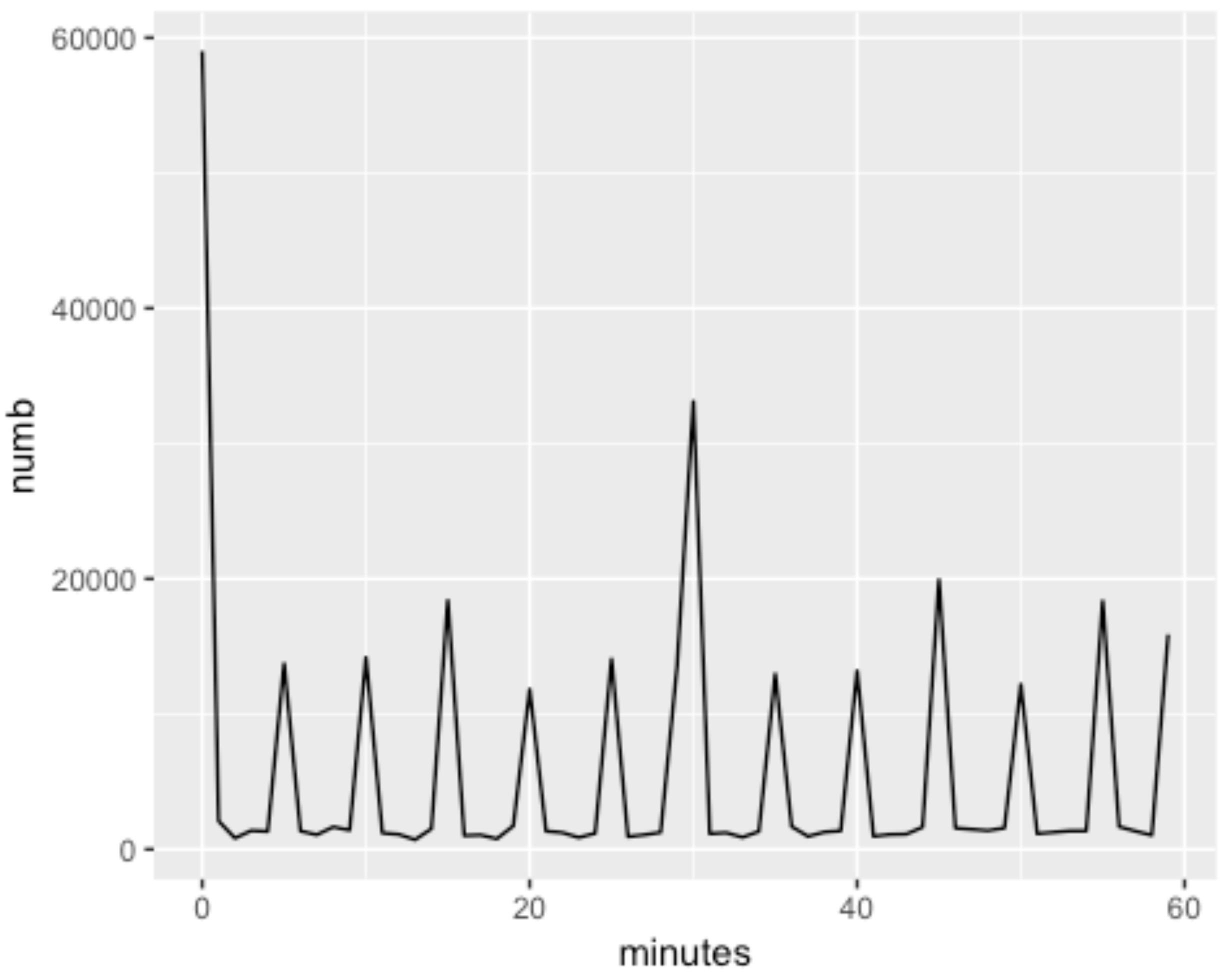
```
flights_dt %>%
  mutate(which_day = wday(dep_time, label=T)) %>%
  select(which_day,everything())
```

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

In the homework, you'll use this to see if some days of the week are more prone to delays than other days.

We can also use dates and date-times to examine interesting patterns in our temporal data. For example, we can see when (in the hour) flights are scheduled to depart:

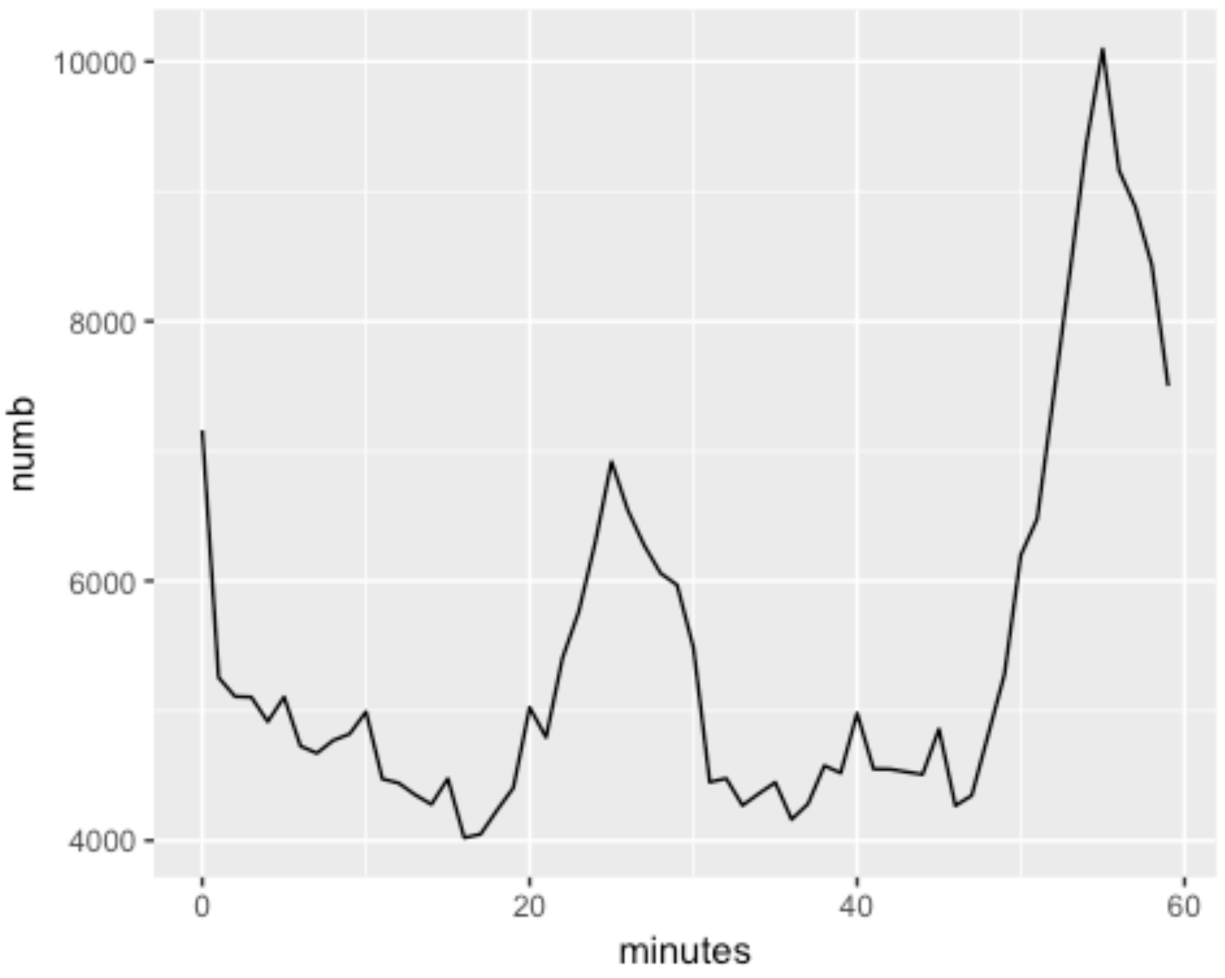
```
flights_dt %>%
  mutate(minutes = minute(sched_dep_time)) %>%
  group_by(minutes) %>%
  summarise(numb = n()) %>%
  ggplot(aes(minutes,numb)) +
    geom_line()
```



Perhaps not surprisingly, things are bundled into nice clusters: lots of flights on the hour and half-hour, with spikes every five minutes. This makes things easier to process: it seems more logical (and orderly) to leave at 5:15 PM rather than (say) 5:17 PM.

Of course, the real world is never so neat, so we can do the same calculation looking at the *actual* departure times:

```
flights_dt %>%
  mutate(minutes = minute(dep_time)) %>%
  group_by(minutes) %>%
  summarise(numb = n()) %>%
  ggplot(aes(minutes,numb)) +
  geom_line()
```



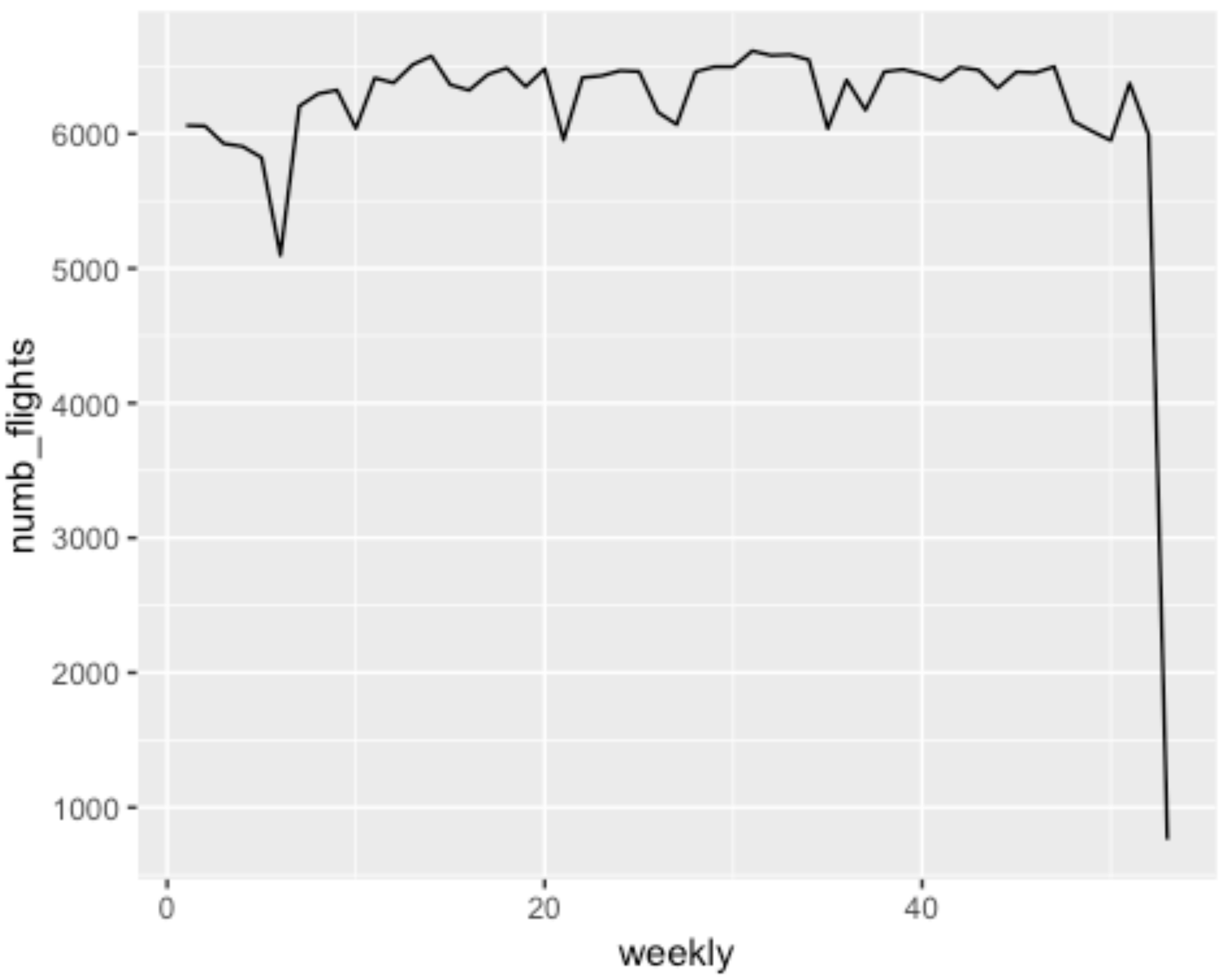
Yes, that looks much more like the chaotic mess I'd expect. The large spike shortly before the hour reflects airlines trying to get planes to leave on time (or early) for their hourly departure (remember that the modal flight leaves on the hour). For your 6:00 AM flight to leave "on time", they just have to depart from the gate before 6. So if they push back at (say) 5:56 AM, then they'll have an on-time departure, even if you then sit on the runway waiting to take off for 30 minutes.

### Test Yourself: How Many Flights Leave Each Week?

Draw a plot summarizing how many flights leave each week.

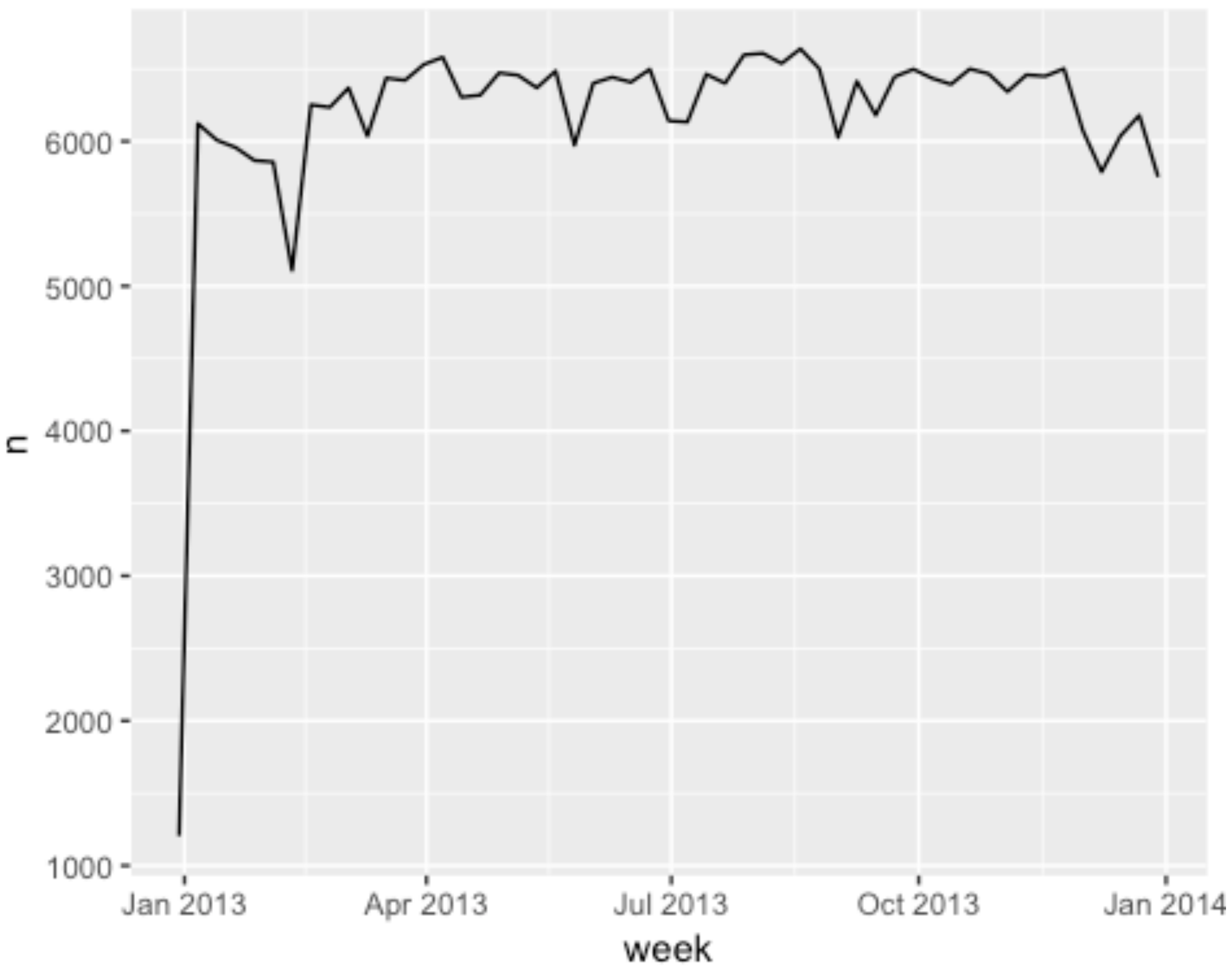
**Answer**:

```
flights_dt %>%
  mutate(weekly = week(dep_time)) %>%
  group_by(weekly) %>%
  summarise(numb_flights = n()) %>%
  ggplot(aes(weekly,numb_flights)) +
  geom_line()
```



Another way of tackling this problem would be to use R's date rounding functions: `round_date()`, `date_floor()` and `date_ceiling()`. Each takes a date and rounds them, rounds them down (floor), or rounds them up (ceiling). Let's re-create the plot above using the `round_date()` function:

```
flights_dt %>%
  count(week = round_date(dep_time, "week")) %>%
  ggplot(aes(week, n)) +
  geom_line()
```



Note that this pattern is different at the beginning and the end of the year. Why? This has to do with how `round_date()` works:

```
round_date(mdy("01/01/2013"), unit="week")

## [1] "2012-12-30"

round_date(mdy("01/02/2013"), unit="week")

## [1] "2012-12-30"

round_date(mdy("01/03/2013"), unit="week")

## [1] "2013-01-06"
```

So in our data, then, in the graph produced by `round_date()`, the first "week" has only 2 days (01/01/2013 - 01/02/2013). This is because it rounds to the nearest week, so Tuesday January 1st and Wednesday January 2nd get rounded to the week beginning Sunday, December 31st, but Thursday, January 3rd gets rounded to the week beginning Sunday, January 6th.

But if I group by week, the first week has (as we'd expect) 7 days: it forms a week as 01/01/2013 - 01/07/2013. Note that neither one is wrong, they're just very slightly different. One grouped by "calendar" weeks (running from Sunday to Saturday), the other groups by "logical" weeks (every 7 days). So here, because the year begins and ends mid-week, we get these small discrepancies at the beginning and end of the year.

### Test Yourself: Flights per half-hour

**Question:** Use the rounding functions to group flights into 30 minute bins.

**Answer:**

```
flights_dt %>%
  mutate(binned_dep = floor_date(dep_time,minutes(30))) %>%
  select(binned_dep,everything())

## # A tibble: 328,063 x 10
##    binned_dep          origin dest  dep_delay arr_delay dep_time
##    <dttm>              <chr>  <chr>     <dbl>     <dbl> <dttm>
##  1 2013-01-01 05:00:00 EWR    IAH           2        11 2013-01-01
05:17:00
##  2 2013-01-01 05:30:00 LGA    IAH           4        20 2013-01-01
05:33:00
##  3 2013-01-01 05:30:00 JFK    MIA           2        33 2013-01-01
05:42:00
##  4 2013-01-01 05:30:00 JFK    BQN          -1       -18 2013-01-01
05:44:00
##  5 2013-01-01 05:30:00 LGA    ATL          -6       -25 2013-01-01
05:54:00
##  6 2013-01-01 05:30:00 EWR    ORD          -4        12 2013-01-01
05:54:00
##  7 2013-01-01 05:30:00 EWR    FLL          -5        19 2013-01-01
05:55:00
##  8 2013-01-01 05:30:00 LGA    IAD          -3       -14 2013-01-01
05:57:00
##  9 2013-01-01 05:30:00 JFK    MCO          -3        -8 2013-01-01
05:57:00
## 10 2013-01-01 05:30:00 LGA    ORD          -2         8 2013-01-01
05:58:00
## # … with 328,053 more rows, and 4 more variables: sched_dep_time <dttm>,
## #   arr_time <dttm>, sched_arr_time <dttm>, air_time <dbl>
```

Here, I used `floor_date()`, but note that you could equivalently use the other rounding functions, just note that they'll give you slightly different answers.

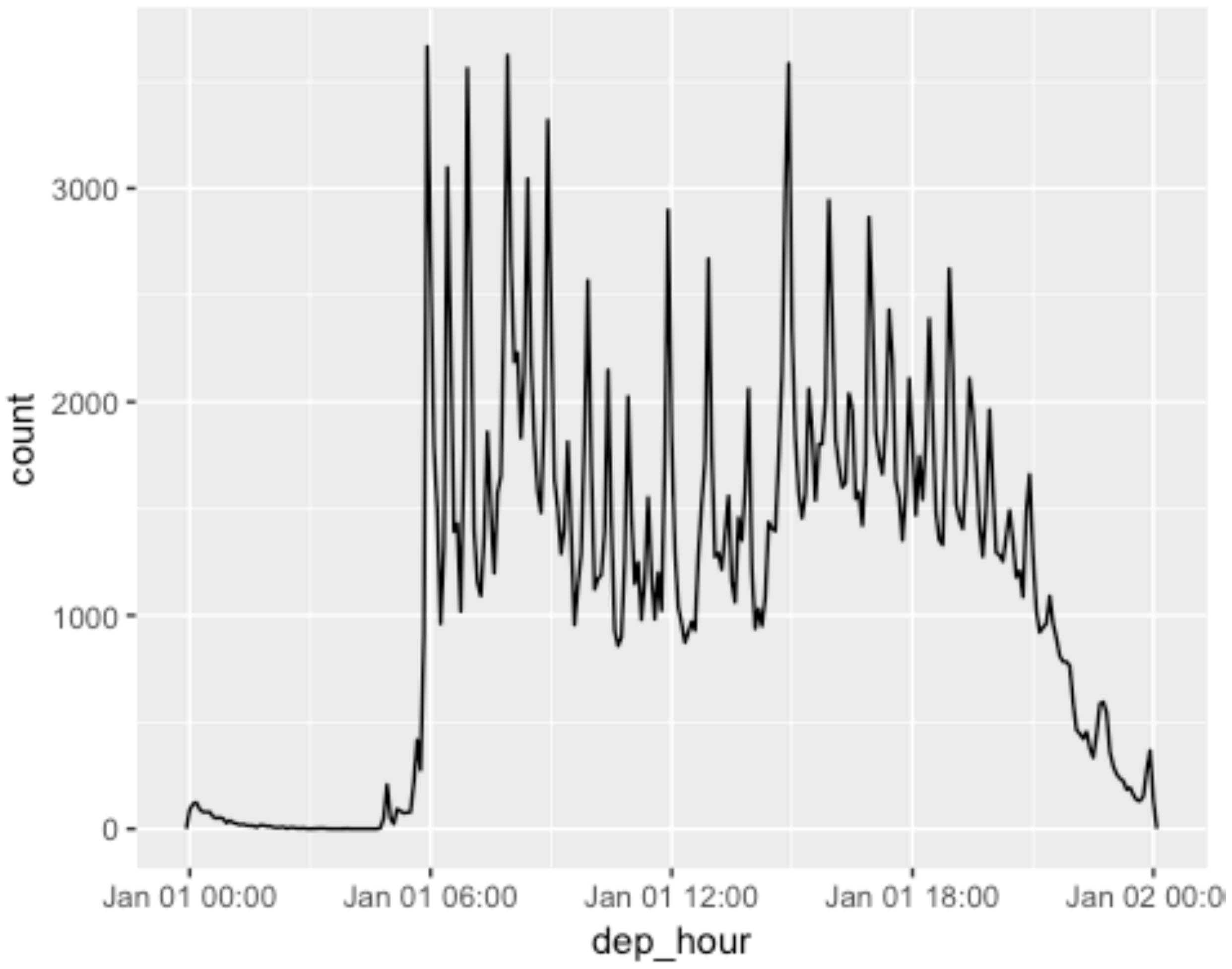
## Using `update()` to Find Patterns in Your Data

You can also use the `update()` function to find patterns in your data. Update is a function that allows you to update a part of a date or date-time variable. Recall that we earlier had a variable dt that was assigned the value January 29, 2019, 5:15:58 AM. But suppose instead that I wanted dt to represent August 27, 2019. I could use update to change it as follows:

```
dt <- ymd_hms("2019-01-29 05:15:58")
update(dt, month = 08, day=27)

## [1] "2019-08-27 05:15:58 UTC"
```

If this was all this could do, it wouldn't be very useful. But you can use update to set the larger components of a date variable and then look for patterns across the smaller components. For example, suppose I wanted to look at the pattern of departure times throughout the day. One way of doing this would be to use update() to set every day to January 1st, and then plot the pattern:

```
flights_dt %>%
  mutate(dep_hour = update(dep_time, yday = 1)) %>%
  ggplot(aes(dep_hour)) +
    geom_freqpoly(binwidth = 300)
```



So the key here is the `yday=1` syntax, which tells R that we should treat each day as January 1st (the first day of the year). That makes it simpler to just plot the departure times over the course of the day. You could (of course!) do the same thing using the `dplyr` tools, but this is just a bit easier. You'll frequently find that R has many ways to accomplish the same goal. It's good to have multiple tools in your belt, as different problems are easier to solve using different sets of tools.

## Durations and Periods

Finally, when we have two dates or date-time variables, you can find the difference between them. For example, how old was former President George H. W. Bush when he died on 30 November 2018? President Bush was born on 12 June 1924, so we can calculate it as follows:

```
nov30 <- dmy("30 November 2018")
ghwb <- dmy("12 June 1924")
gwb_age <- nov30 - ghwb
gwb_age

## Time difference of 34504 days

as.duration(gwb_age)

## [1] "2981145600s (~94.47 years)"
```

So when you create a difference between two times, R automatically expresses it as the number of days elapsed, but that's not very informative. So we can use the `as.duration()` function to express this in years, which is much more helpful.

Durations are always measured in seconds, but you can construct durations by referencing a set of helper functions:

```
dminutes(13)

## [1] "780s (~13 minutes)"

dhours(22)

## [1] "79200s (~22 hours)"

ddays(7)

## [1] "604800s (~1 weeks)"

dweeks(23)

## [1] "13910400s (~23 weeks)"

dyears(2)

## [1] "63072000s (~2 years)"
```

So those functions allow you to easily compute durations. For example, we can calculate how old former President Bush would have been on 25 November 2016, two years and 5 days before his death:

```
as.duration(gwb_age - dyears(2) - ddays(5))

## [1] "2917641600s (~92.45 years)"
```

Note, however, that there is no `dmonth` function. Why? Because not all months have the same number of days, so we can't compute them in the same way. To solve this, R has a set of functions called periods that operate more like "human" recorded time. These functions are very similar to durations, but they behave more naturally:

```
nov30 + months(1)

## [1] "2018-12-30"

## this fixes leap years as well
ymd("2012-01-01") + dyears(1)

## [1] "2012-12-31"

ymd("2012-01-01") + years(1)

## [1] "2013-01-01"
```

These `period()` variables can be helpful at cleaning up date-time inconsistencies. For example, in our flights data, there are some flights that seem to arrive *before* they departed:

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

Ah, we can see what happened: these are overnight flights. For example, the first flight from Newark to Puerto Rico leaves on January 1st at 7:29 PM, and then lands at 12:03 AM on January 2nd. But here, it looks like it landed at 12:03 AM on January 1st (before it departed) because we assume the flight took off and landed on the same day in our data (which was the case for the vast majority of flights). We can fix this using the `days()` function:

```
flights_dt <- flights_dt %>%
  mutate(
    overnight = arr_time < dep_time,
    arr_time = arr_time + days(overnight * 1),
    sched_arr_time = sched_arr_time + days(overnight * 1)
  )
flights_dt

## # A tibble: 328,063 x 10
##    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 4 more variables: arr_time <dttm>,
## #   sched_arr_time <dttm>, air_time <dbl>, overnight <lgl>
```

So here I added 1 day to the arrival of each overnight flight to have it land on the correct day. Now if I check my original logical syntax:

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

## # 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>
```

I find that all of the flight arrival/departure times make sense.

## Appendix: All code used to make this document

```
knitr::opts_chunk$set(echo = TRUE)
knitr::opts_knit$set(root.dir = '~/Dropbox/DATA101')
library(tidyverse)
library(lubridate)

ymd("2019-01-31")
mdy("January 31, 2019")
dmy("31-Jan-19")
dmy("31-01-19")
ymd_hms("2019-01-31 07:11:12")
ymd_hm("2019-01-31 07:11")
setwd("~/Dropbox/DATA101")
trump <- read_csv(file="Data/Raw/538_Trump_Approval.csv")
trump <- trump %>% mutate(poll_date=mdy(modeldate),
                          poll_datetime=mdy_hm(timestamp))
select(trump,poll_date,poll_datetime,everything())
library(nycflights13)
flights %>%
```

```r
  select(year, month, day, hour, minute) %>%
  mutate(departure = make_datetime(year, month, day, hour, minute))
select(flights, c(dep_time, sched_dep_time, arr_time, sched_arr_time))
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"))
515 %/% 100
515 %% 100
flights_dt %>%
  filter(dep_time < ymd(20130703) & dep_time > ymd(20130702)) %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(binwidth = 600) # 600 s = 10 minutes
flights_dt %>%
  filter(dep_time < ymd(20130131)) %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(bins=31)
dt <- ymd_hms("2019-06-18 05:15:58")
year(dt)
month(dt)
mday(dt)
yday(dt)
wday(dt)
hour(dt)
minute(dt)
wday(dt)
wday(dt, label=T)
wday(dt, label=T, abbr=F)
flights_dt %>%
  mutate(which_day = wday(dep_time, label=T)) %>%
  select(which_day,everything())
flights_dt %>%
  mutate(minutes = minute(sched_dep_time)) %>%
  group_by(minutes) %>%
  summarise(numb = n()) %>%
  ggplot(aes(minutes,numb)) +
    geom_line()

flights_dt %>%
  mutate(minutes = minute(dep_time)) %>%
  group_by(minutes) %>%
  summarise(numb = n()) %>%
  ggplot(aes(minutes,numb)) +
  geom_line()
flights_dt %>%
  mutate(weekly = week(dep_time)) %>%
  group_by(weekly) %>%
  summarise(numb_flights = n()) %>%
  ggplot(aes(weekly,numb_flights)) +
  geom_line()
flights_dt %>%
  count(week = round_date(dep_time, "week")) %>%
  ggplot(aes(week, n)) +
  geom_line()
round_date(mdy("01/01/2013"), unit="week")
round_date(mdy("01/02/2013"), unit="week")
round_date(mdy("01/03/2013"), unit="week")
flights_dt %>%
  mutate(binned_dep = floor_date(dep_time,minutes(30))) %>%
  select(binned_dep,everything())
dt <- ymd_hms("2019-01-29 05:15:58")
update(dt, month = 08, day=27)
flights_dt %>%
  mutate(dep_hour = update(dep_time, yday = 1)) %>%
  ggplot(aes(dep_hour)) +
    geom_freqpoly(binwidth = 300)
nov30 <- dmy("30 November 2018")
ghwb <- dmy("12 June 1924")
gwb_age <- nov30 - ghwb
gwb_age
as.duration(gwb_age)
dminutes(13)
dhours(22)
ddays(7)
dweeks(23)
dyears(2)
as.duration(gwb_age - dyears(2) - ddays(5))
nov30 + months(1)
## this fixes leap years as well
ymd("2012-01-01") + dyears(1)
ymd("2012-01-01") + years(1)
flights_dt %>%
  filter(arr_time < dep_time)
flights_dt <- flights_dt %>%
  mutate(
    overnight = arr_time < dep_time,
    arr_time = arr_time + days(overnight * 1),
    sched_arr_time = sched_arr_time + days(overnight * 1)
  )
flights_dt
```

```r
flights_dt %>%
  filter(arr_time < dep_time)
```