E392: Problem Set 3

Data Management

Spring 2018

Answer Key

Answers are below in italic font. I graded the questions on ***Tibbles and data frames*** *and* ***Mutating joins****.*

Please work on the following questions and hand in your solutions in groups of at most 2 students. You are asked to answer all questions, but we will only select 2 questions randomly to grade.

Part 1: R questions

Question 1: Data import and tidying

Tibbles and data frames

1. How can you tell if an object is a tibble? (Hint: try printing `mtcars`, which is a regular data frame). *There are several ways to do this. For example, you can switch your environment window to the* ***`Grid`*** *view and you will see the type of each object. Base R frames will be listed as* ***`data.frame`****, tibbles as* ***`tbl_df`****. Alternatively, you can simply type the name of your data object in the console. If it is a tibble, the very first printed line will say so.*
2. Compare and contrast the following operations on a `data.frame` and an equivalent tibble. What is different? Why might the default data frame behaviors cause you frustration? *There are several things that can be frustrating about base R data frames. First, a base data frame will by default convert strings as factor variables which is rarely what we want to do these days. Tibbles don't do that. When you select a single variable from a base R data frame, you will not get a data frame, but a vector. If you extract several variables from a data frame, you will get a new data frame. That's inconsistent and can cause problems when you want to write general code. In contrast, when you extract parts of a tibble, you will always get a new tibble. Finally, note that a base R data frame allows for partial matching. If you want to select variable "x", but the data frame does not contain a variable called "x", base R will look for a partialy match and print you variable "xyz" instead. That's very confusing and a frequent source of bugs. Therefore, a tibble would give you a warning and not a variable at all in a partial matching case.*

```r
df <- data.frame(abc = 1, xyz = "a")
df$x
```

```
## [1] a
## Levels: a
```

```r
df[, "xyz"]
```

```
## [1] a
## Levels: a
```

```r
df[, c("abc", "xyz")]
```

```
##   abc xyz
## 1   1   a
```

```r
# Perform same operations on a tibble.
tibble_df <- tibble(
            abc = 1,
            xyz = "a"
)
tibble_df
```

```
## # A tibble: 1 x 2
##     abc xyz
##   <dbl> <chr>
## 1  1.00 a
```

```r
tibble_df$x
```

```
## Warning: Unknown or uninitialised column: 'x'.
```

```
## NULL
```

```r
tibble_df[, "xyz"]
```

```
## # A tibble: 1 x 1
##   xyz
##   <chr>
## 1 a
```

```r
tibble_df[, c("abc", "xyz")]
```

```
## # A tibble: 1 x 2
##     abc xyz
##   <dbl> <chr>
## 1  1.00 a
```

Data import

1. What function would you use to read a file where fields were separated with "|"? *This is a rather uncommon delimiter, so you would have to use the general read-function `read_delim()` and specify the delimiter as `|`: `read_delim(filename,delim = "|")`.*

2. What are the most important arguments to `read_fwf()`? *This function is useful for parsing data that comes in fixed-width-format, i.e. each field has exactly the same width. This is a useful format for large data sets since it is very quick to parse. The downside is that you have to specify how wide each field is. So by far the most important argument is to specify the field width or the column positions at which a new variable starts.*

3. Sometimes strings in a CSV file contain commas. To prevent them from causing problems they need to be surrounded by a quoting character, like `"` or `'`. By default, `read_csv()` assumes that the quoting character will be `"`. If you need to customize many input arguments, you can use `read_delim()` instead. What arguments do you need to specify to read the following text into a data frame? *Let's first analyse what the data contains. It has two rows separated by the `\n` (line break) command. Most likely, the first line contains the variable labels, so it just has one observation. Moreover the data contains two variables none of which are purely numeric, so we would have to keep both as strings. Values are separated by commas, so `read_csv()` would be appropriate, however we also have some single-quotes to indicate variable value "boundaries". In order to make sure these get imported correctly, we can use the general import command `read_delim()` and specify the `delim`-argument as a comma, and the `quote`-argument as `'`.*

```r
A <- "x,y\n1,'a,b'"
# Using the general import command.
read_delim(A, delim=",", quote ="'")
```

```
## # A tibble: 1 x 2
##       x y
##   <int> <chr>
## 1     1 a,b
```

```r
# read_csv() will also work here.
read_csv(A,quote="'")
```

```
## # A tibble: 1 x 2
##       x y
##   <int> <chr>
## 1     1 a,b
```

Parsing vectors

1. What happens if you try and set `decimal_mark` and `grouping_mark` to the same character? What happens to the default value of `grouping_mark` when you set `decimal_mark` to `,`? What happens to the default value of `decimal_mark` when you set the `grouping_mark` to `.`? *Let's explore some examples to figure this out. As expected, you get an error when you set decimal and grouping mark to the same value since it's not clear at all, what you want R to do. If you set one of them as the decimal mark (grouping mark), R will assume that the other symbol is your grouping mark (decimal mark) since these are by far the most common notational conventions.*

```
x <- "123,456"
parse_number(x, locale=locale(decimal_mark=",",grouping_mark=","))
```

```
## Error: `decimal_mark` and `grouping_mark` must be different
```

```
y <- "123.456,789"
parse_number(y, locale=locale(decimal_mark=","))
```

```
## [1] 123456.8
```

```
z <- "123.456,789"
parse_number(y,locale=locale(grouping_mark="."))
```

```
## [1] 123456.8
```

1. What are the most common encodings used in Europe? What are the most common encodings used in Asia? What are the most common encodings in your home country? Do some googling to find out. *Before UTF-8 was common, most European data was encodede in "Latin1" for Western European languages or "Latin2" for Eastern European languages. I know much less about Asian countries, but I would expect that most Japanese data used "Shift-JIS", Korean data used "EUC-KR" and Chinese data often used "GB" (for Guobiao).*

Spreading and gathering

1. Why are `gather()` and `spread()` not perfectly symmetrical? Carefully consider the following example. *The functions `spread()` and `gather()` are not perfectly symmetrical because column type information is not preserved between them. In the original table the column `year` was numeric, but after running `spread()` and `gather()` it is a character vector. This is because variable names are always converted to a character vector by `gather()`.*

```
stocks <- tibble(
  year   = c(2015, 2015, 2016, 2016),
  half  = c(   1,    2,     1,    2),
  return = c(1.88, 0.59, 0.92, 0.17)
```

```r
)
# Look at the data frame in two steps.
# When spreading the data.
stocks_spread <- spread(stocks,year, return)
stocks_gather <- gather(stocks_spread, "year", "return", `2015`:`2016`)
```

2. Both `spread()` and `gather()` have a `convert` argument. What does it do? *It will convert the newly generated variables to the most suitable types. For example, an old variable might have been saved as a string because of a delimiter, while the actual information is numeric.*

3. Why does this code fail? *This is pretty simple. Since 1999 and 2000 are nonstandard variable names (they start with a number and not a letter) you have to enclose them by ticks.*

```r
# This will work.
table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
```

```
## # A tibble: 6 x 3
##   country     year   cases
##   <chr>       <chr>  <int>
## 1 Afghanistan 1999     745
## 2 Brazil      1999   37737
## 3 China       1999  212258
## 4 Afghanistan 2000    2666
## 5 Brazil      2000   80488
## 6 China       2000  213766
```

4. Why does spreading this tibble fail? How could you add a new column to fix the problem? *Here we would like to have age and height as variables instead of creating additional rows for each person. The problem is that Philipp Woods has age information twice, most likely this is data collected over several years. So if we added a new column for this information, spreading should work.*

```r
people <- tribble(
  ~name,             ~key,    ~value,
  #-----------------|--------|------
  "Phillip Woods",   "age",       45,
  "Phillip Woods",   "height",   186,
  "Phillip Woods",   "age",       50,
  "Jessica Cordero", "age",       37,
  "Jessica Cordero", "height",   156
)

# This will fail.
#people_spread <- spread(people, key, value)
```

```r
# Create a new variable.
people2 <- tribble(
  ~name,             ~key,    ~value, ~year,
  #-----------------|--------|------
  "Phillip Woods",   "age",       45, 2010,
  "Phillip Woods",   "height",   186, 2010,
  "Phillip Woods",   "age",       50, 2015,
  "Jessica Cordero", "age",       37, 2010,
  "Jessica Cordero", "height",   156, 2010
)

# This will work.
people_spread2 <- spread(people2, key , value)
```

5. Tidy the simple tibble below. Do you need to spread or gather it? What are the variables? *This is information on a group of people listing their gender and whether their pregnant or not and how many individuals there are of each type. Since the gender variable is spread out over several columns, we need to gather the tibble.*

```r
preg <- tribble(
  ~pregnant, ~male, ~female,
  "yes",     NA,    10,
  "no",      20,    12
)
# Tidy the tibble.
gather(preg, sex, count, male, female)
```

```
## # A tibble: 4 x 3
##   pregnant sex    count
##   <chr>    <chr>  <dbl>
## 1 yes      male    NA
## 2 no       male    20.0
## 3 yes      female  10.0
## 4 no       female  12.0
```

Separating and uniting

1. What do the `extra` and `fill` arguments do in `separate()`? Experiment with the various options for the following two toy data sets. *These arguments become important when you split a variable, but you either have too many or too few parts. By default,* ***`separate()`*** *will issue a warning and drop too many or fill in* ***`NA`****s where there are too few pieces. You can customize this behavior.*

```r
# This is the default, when there are too many pieces.
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"))
```

```
## Warning: Too many values at 1 locations: 2
```

```
## # A tibble: 3 x 3
##     one   two three
## * <chr> <chr> <chr>
## 1 a     b     c
## 2 d     e     f
## 3 h     i     j
```

```r
# If you want to drop the extra components.
# No warning is issued.
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"), extra="drop")
```

```
## # A tibble: 3 x 3
##     one   two three
## * <chr> <chr> <chr>
## 1 a     b     c
## 2 d     e     f
## 3 h     i     j
```

```r
# If you want to keep all the information and munge
# all extra information into the last part.
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"), extra="merge")
```

```
## # A tibble: 3 x 3
##     one   two three
## * <chr> <chr> <chr>
## 1 a     b     c
## 2 d     e     f,g
## 3 h     i     j
```

```r
# This is the default when there are too few pieces.
tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"))
```

```
## Warning: Too few values at 1 locations: 2
```

```
## # A tibble: 3 x 3
##     one   two three
## * <chr> <chr> <chr>
## 1 a     b     c
## 2 d     e     <NA>
```

```
## 3 f     g     i
```

```r
# If you want to fill missing values on the right
# Same as default, but no warning is issued.
tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"), fill="right")
```

```
## # A tibble: 3 x 3
##   one   two   three
## * <chr> <chr> <chr>
## 1 a     b     c
## 2 d     e     <NA>
## 3 f     g     i
```

```r
# If instead you want to fill NAs from the left.
tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"), fill="left")
```

```
## # A tibble: 3 x 3
##   one   two   three
## * <chr> <chr> <chr>
## 1 a     b     c
## 2 <NA>  d     e
## 3 f     g     i
```

2. Both `unite()` and `separate()` have a `remove` argument. What does it do? Why would you set it to `FALSE`? *By default `remove` is set to TRUE and it will remove the original column(s) from the new data frame. It might be useful to keep the original column(s), for example if you want to check manually that your code executed correctly or if you think you'll need the data in its original form later on.*

Missing values

1. Compare and contrast the `fill` arguments to `spread()` and `complete()`. *For `spread()`, the fill argument explicitly sets the value to replace `NA`s. In `complete()`, the fill argument also sets a value to replace `NA`s but it is a named list, allowing for different values for different variables. Both functions replace both implicit and explicit missing values.*

2. What does the direction argument to `fill()` do? *It sets whether `NA`s should be replaced by the previous non-missing value or the next non-missing value.*

Question 2: Relational data and data types

The following questions use the `nycflights13` data discussed in class.

Relational data

1. Imagine you wanted to draw (approximately) the route each plane flies from its origin to its destination. What variables would you need? What tables would you need to combine? *You would need latitude and longitude of both the destination and the airport of origin. This information is contained in the* ***`airports`*** *table which we would have to merge twice, first for the origin airport and second for the destination airport.*

2. In the diagram on the lecture slides, I forgot to draw the relationship between `weather` and `airports`. What is the relationship and how should it appear in the diagram? *The variable* ***`faa`*** *in the* ***`airports`****-table contains the airport identifier so it should be matched to* ***`origin`*** *in the* ***`weather`****-table.*

3. `weather` only contains information for the origin (NYC) airports. If it contained weather records for all airports in the USA, what additional relation would it define with `flights`? *In order to get a matched data set with weather information for each flight's destination, we would merge* ***`year, month,day, hour, origin`*** *in* ***`weather`*** *to* ***`year,month,day,hour,dest`*** *in* ***`flights`****.*

4. We know that some days of the year are special, and fewer people than usual fly on them. How might you represent that data as a data frame? What would be the primary keys of that table? How would it connect to the existing tables? *In order to get a list of flights on special days, we could create an additional table containing a list of all special days as* ***`year,month,day`*** *combinations. We could use this for a filtering join based on the* ***`year,month,day`*** *key in the* ***`flights`*** *table.*

Keys

1. Add a surrogate key to `flights`.

```
flights %>%
    # Sorting is not necessary, but it might help
    # to have identifiers follow some logic.
  arrange(year, month, day, sched_dep_time, carrier, flight) %>%
  mutate(flight_id = row_number())
```

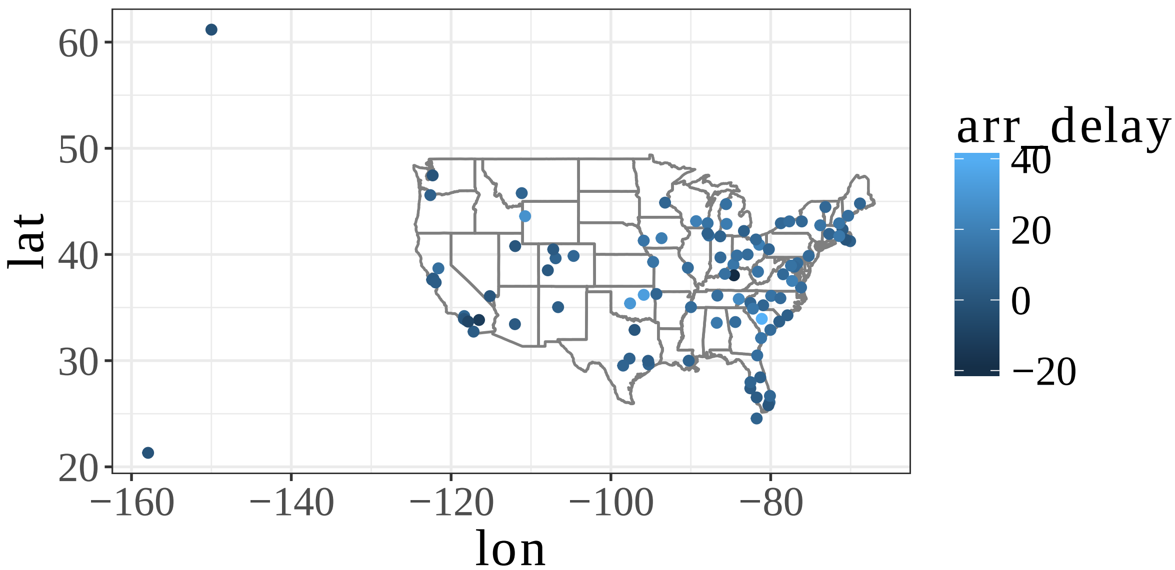
Mutating joins

1. Compute the average delay by destination, then join on the `airports` data frame so you can show the spatial distribution of delays. Here's an easy way to draw a map of the United States (be sure to install and load the required `maps`-package first!):

```
# Compute the average delay by destination airport.
delay_dest <- flights %>%
    group_by(dest) %>%
```

```r
    summarize(arr_delay = mean(arr_delay, na.rm = T))
# Join the delay data to the airport data.
airport_delays <- airports %>%
    inner_join(delay_dest, by=c("faa" = "dest"))

# Now let's draw the plot.
airport_delays %>%
  ggplot(aes(lon, lat)) +
    borders("state") +
    geom_point(aes(color=arr_delay)) +
    coord_quickmap()
```



2. Add the location of the origin *and* destination (i.e. the `lat` and `lon`) to `flights`. *We can proceed in two steps. Depending on what you want to do with that data it may be useful to rename some of the variables after the first merge.*

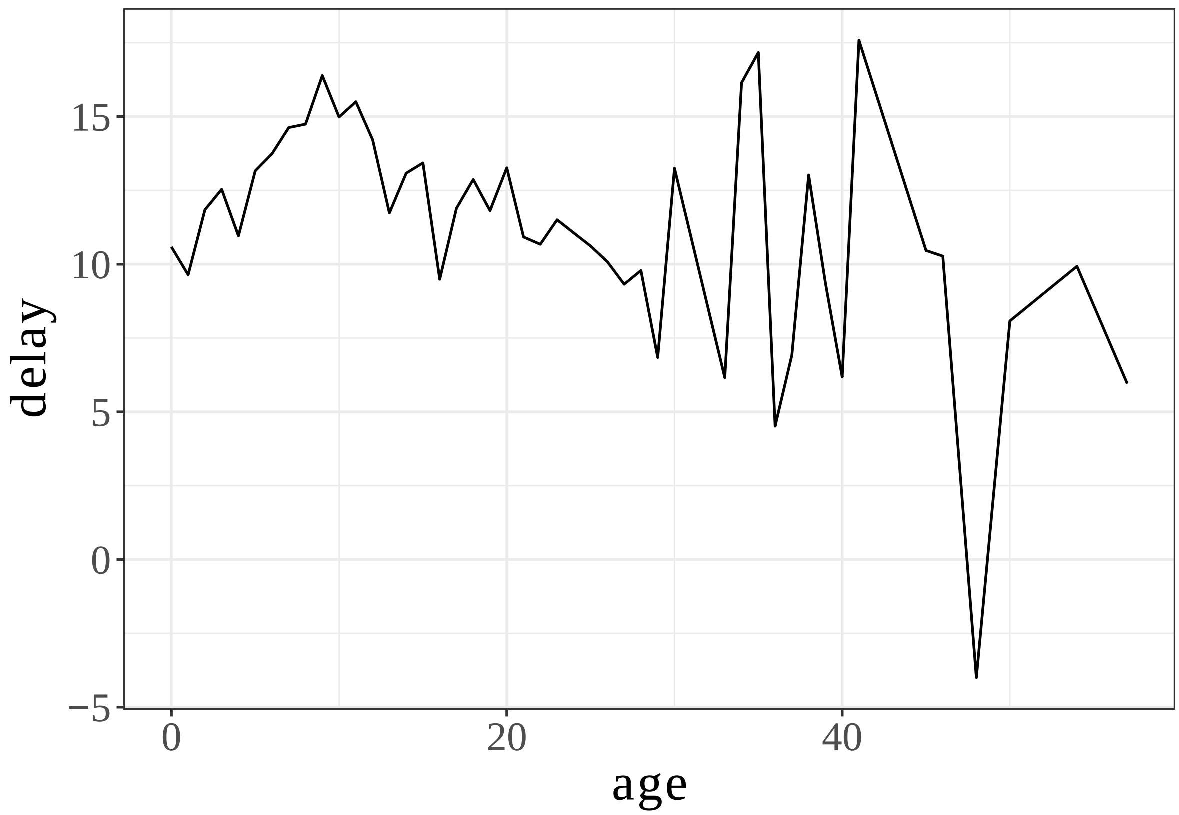
```r
flights_coordinates <- flights %>%
    left_join(airports, by = c("origin" = "faa")) %>%
    rename(lat_origin = lat, lon_origin = lon) %>%
    left_join(airports, by = c("dest" = "faa")) %>%
    rename(lat_dest = lat, lon_dest = lon)
```

3. Is there a relationship between the age of a plane and its delays? *Somewhat surprisingly, the age of a plane has nothing to do with its performance. Of course we might be missing other important factors here.*

```r
# Compute age of each plane.
plane_ages <- planes %>%
  mutate(age = 2013 - year) %>%
  select(tailnum, age)
```

```r
# Merge plane data to flight data.
flights %>%
  inner_join(plane_ages, by = "tailnum") %>%
  group_by(age) %>%
    # Let's remove cancelled flights here.
  filter(!is.na(dep_delay)) %>%
  summarise(delay = mean(dep_delay)) %>%
  ggplot(aes(x = age, y = delay)) +
  geom_line()
```

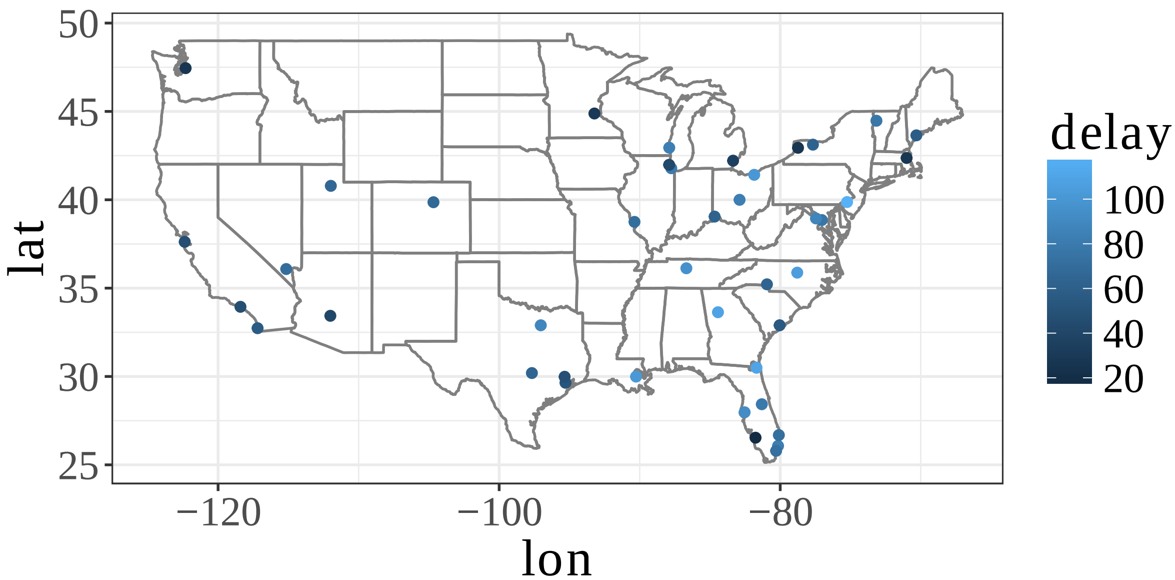
```
## Warning: Removed 1 rows containing missing values (geom_path).
```



4. What happened on June 13 2013? Display the spatial pattern of delays, and then use Google to cross-reference with the weather. *There were a couple of severe storms, mostly in the southeastern part of the US. We see that average delays on that day were higher in that region.*

```r
flights_0613 <- filter(flights, !is.na(dep_time), month == 6, day == 13)
flights_0613 %>%
  group_by(dest) %>%
  summarise(delay = mean(arr_delay,na.rm = TRUE), n = n()) %>%
    # Let's drop special locations with fewer than 5 flights on that day.
  filter(n > 5) %>%
  inner_join(airports, by = c("dest" = "faa")) %>%
```

```r
  ggplot(aes(lon, lat)) +
    borders("state") +
    geom_point(aes(colour = delay)) +
    coord_quickmap()
```



Filtering joins

1. Filter flights to only show flights with planes that have flown at least 100 flights.

```r
# Create a list of planes that satisfy our criteria.
planes_100 <- flights %>%
  group_by(tailnum) %>%
  count() %>%
  filter(n > 100)
# now use the planes_100 table to filter observations.
flights %>%
  semi_join(planes_100, by = "tailnum")
```

```
## # A tibble: 229,202 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515      2.00      830
##  2  2013     1     1      533            529      4.00      850
##  3  2013     1     1      544            545     -1.00     1004
##  4  2013     1     1      554            558     -4.00      740
##  5  2013     1     1      555            600     -5.00      913
##  6  2013     1     1      557            600     -3.00      709
##  7  2013     1     1      557            600     -3.00      838
##  8  2013     1     1      558            600     -2.00      849
```

```
## 9 2013 1 1 558 600 -2.00 853
## 10 2013 1 1 558 600 -2.00 923
## # ... with 229,192 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

2. Find the 48 hours (over the course of the whole year) that have the worst delays. Cross-reference it with the `weather` data. Can you see any patterns? *Total delay times were highest during some days of the summer travel season. Somewhat surprisingly, March 8 also ranks high on the list. After checking the historical weather data, we see that this was a day with heavy snowstorms in the NYC area.*

```
# Let's look at accumulated arrival delays for a given day.
total_delay_data <- flights %>%
  group_by(year, month, day) %>%
  summarise(del_day_total = sum(arr_delay, na.rm = TRUE)) %>%
  mutate(delay_48 = del_day_total + lag(del_day_total)) %>%
  arrange(desc(delay_48))
```

3. You might expect that there's an implicit relationship between plane and airline, because each plane is flown by a single airline. Confirm or reject this hypothesis using the tools you've learned above. *In general, the relationship need not be unique. For example, planes can be sold or airlines may merge. However, it's not clear that these planes would show up in our data. So let's just create a list of planes that have been used by more than one airline for a flight from NYC in 2013. And we see that there are 18 planes that were used by two airlines.*

```
planes_transfer <- flights %>%
  group_by(tailnum, carrier) %>%
  count() %>% ungroup() %>%
  select(tailnum) %>%
  group_by(tailnum) %>%
    count() %>%
  filter(n>1)
```

Strings

1. In code that doesn't use the `stringr`-package, you'll often see `paste()` and `paste0()`. What's the difference between the two functions? What `stringr`-function are they equivalent to? How do the functions differ in their handling of `NA`? *`paste()` and `paste0()` concatenate strings. While the former separates components by spaces, the latter does not add spaces. So the default behavior of `str_c()` is closer to `paste0()`. `strc_c()` contains `NA` as in any other numeric R function, i.e. `NA` is contagious and everything that it's combined with will also be `NA`. In contrast, `paste()` will convert `NA` to a string called `NA` and will then treat it as just another string.*

2. In your own words, describe the difference between the `sep` and `collapse` arguments to `str_c()`. *The* ***`sep`*** *argument takes a fixed string that is inserted between the single strings that are joined. If you want to concatenate several elements of a vector into one, the* ***`collapse`*** *argument allows you to specify how within the single output string, the vector comonents are separated.*

```
a <- "abc"
b <- "def"
c <- "ghi"
# Illustration of the sep argument.
d_sep <- str_c(a,b,c,sep="X")
# Illustration of the collapse argument.
# Letters is just a vector of all letters of the alphabet.
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
str_c(letters,collapse=",")
```

```
## [1] "a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z"
```

```
str_c(letters,collapse="")
```

```
## [1] "abcdefghijklmnopqrstuvwxyz"
```

3. What does `str_wrap()` do? When might you want to use it? *This function wraps a string into a special format so that it fits for example to a particular width. This is useful if you want your strings to fit to a particular screen, such as the console of a remote computer that often has a fixed width of 80 characters.*

```
a <- "This is a very long string that might easily exceed the page width."
# Add line break characters after every 5 characters.
(a_wrap <- str_wrap(a, width=5))
```

```
## [1] "This\nis a\nvery\nlong\nstring\nthat\nmight\neasily\nexceed\nthe\npage\nwid
```

```
# Let's check how the wrapped string looks like when printed.
writeLines(a_wrap)
```

```
## This
## is a
## very
## long
## string
## that
## might
## easily
## exceed
```

```
## the
## page
## width.
```

4. What does `str_trim()` do? What's the opposite of `str_trim()`? *It trims white space from a string. You can use `str_pad()` to add whitespace to the string.*

Part 2: Your project

Question 3

Continue working on your project and report your progress either on acquiring the data or evaluating the data's quality.

By now you may have discovered that your idea is a dead end. If that is the case explain why. Think about whether you can slightly modify your research question to suit your data better. If that is not possible, repeat last week's exercise for one of your other ideas.

If based on your reading of the data documentation, you are happy with you data, get started exploring your data more formally using the tools we discussed in class, i.e. data visualization, transformation and exploratory data analysis. In at most one page, summarize your most important findings.