30535 Skills Problem Set 5

Mia Jiang

5/24/2022

Front matter This submission is my work alone and complies with the 30535 integrity policy.

Add your initials to indicate your agreement: **M.J.**

Late coins used this pset: 0. Late coins left: 0.

1 R4DS Chapter 13 Joins continued

1.1

```
# Load data
flights_full <- nycflights13::flights
flights <- flights_full %>%
  head(100)
weather <- nycflights13::weather
airports <- nycflights13::airports
# Calculate the time
system.time(
  flights_weather <- left_join(flights, weather, by = "year")
  )
```

```
##    user  system elapsed
##    0.62    0.12    0.79
```

```
flights_weather <- left_join(flights, weather, by = "year")
# Calculate rows
flights %>%
  nrow()
```

```
## [1] 100
```

```
weather %>%
  nrow()
```

```
## [1] 26115
```

```
flights_weather %>%
  nrow()
```

```
## [1] 2611500
```

Answer
There are 2611500 rows in the merged data set, and we can refer to the *system* column above for the time the computer having used to join the datasets.

1.2

- If we use **left_join** to merge **flights** and **weather** based on **year**, since both the **flights** dataset and the **weather** dataset only include data for flights departed NYC in 2013, merging by **year** means each row in the **weather** dataset will find all rows in **flights** could be a correspondent row to merge. Therefore, each row in **flights** will be matched with all rows in **weather** separately, i.e., for each row in **flights** after **left_join** there will be 26115 rows with different weather information for the same flight.
- Then we will get $100 \times 26115 = 2611500$ rows in the merged dataset in total. And it will take around 0.1 sec to run the code

1.3

```
flights %>%
  filter(is.na(flights$tailnum) == TRUE) %>%
  summary()
```

```
##       year          month           day          dep_time   sched_dep_time
##  Min.   : NA   Min.   : NA   Min.   : NA   Min.   : NA   Min.   : NA
##  1st Qu.: NA   1st Qu.: NA   1st Qu.: NA   1st Qu.: NA   1st Qu.: NA
##  Median : NA   Median : NA   Median : NA   Median : NA   Median : NA
##  Mean   :NaN   Mean   :NaN   Mean   :NaN   Mean   :NaN   Mean   :NaN
##  3rd Qu.: NA   3rd Qu.: NA   3rd Qu.: NA   3rd Qu.: NA   3rd Qu.: NA
##  Max.   : NA   Max.   : NA   Max.   : NA   Max.   : NA   Max.   : NA
##    dep_delay      arr_time   sched_arr_time   arr_delay     carrier
##  Min.   : NA   Min.   : NA   Min.   : NA    Min.   : NA   Length:0
##  1st Qu.: NA   1st Qu.: NA   1st Qu.: NA    1st Qu.: NA   Class :character
##  Median : NA   Median : NA   Median : NA    Median : NA   Mode  :character
##  Mean   :NaN   Mean   :NaN   Mean   :NaN    Mean   :NaN
##  3rd Qu.: NA   3rd Qu.: NA   3rd Qu.: NA    3rd Qu.: NA
##  Max.   : NA   Max.   : NA   Max.   : NA    Max.   : NA
##      flight      tailnum             origin              dest
##  Min.   : NA   Length:0           Length:0           Length:0
##  1st Qu.: NA   Class :character   Class :character   Class :character
##  Median : NA   Mode  :character   Mode  :character   Mode  :character
##  Mean   :NaN
##  3rd Qu.: NA
##  Max.   : NA
##     air_time      distance        hour          minute      time_hour
##  Min.   : NA   Min.   : NA   Min.   : NA   Min.   : NA   Min.   :NA
##  1st Qu.: NA   1st Qu.: NA   1st Qu.: NA   1st Qu.: NA   1st Qu.:NA
##  Median : NA   Median : NA   Median : NA   Median : NA   Median :NA
##  Mean   :NaN   Mean   :NaN   Mean   :NaN   Mean   :NaN   Mean   :NA
##  3rd Qu.: NA   3rd Qu.: NA   3rd Qu.: NA   3rd Qu.: NA   3rd Qu.:NA
##  Max.   : NA   Max.   : NA   Max.   : NA   Max.   : NA   Max.   :NA
```

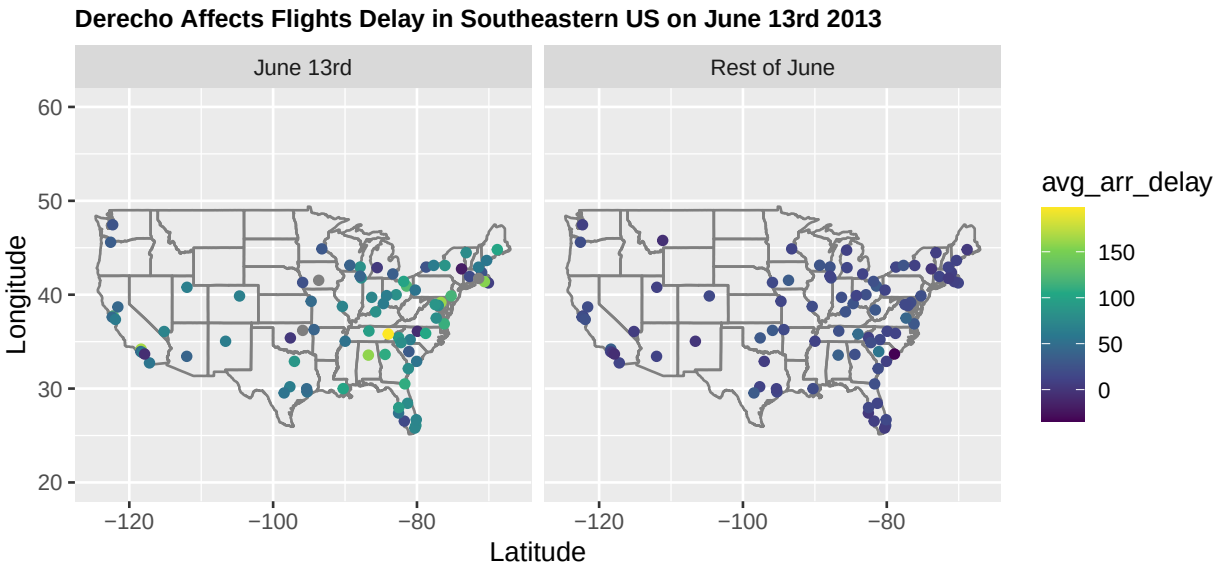
```
flights_full %>%
  filter(is.na(flights_full$tailnum) == TRUE) %>%
  summary()
```

```
##       year          month           day          dep_time    sched_dep_time
##  Min.   :2013   Min.   : 1.0   Min.   : 1.0   Min.   : NA    Min.   : 106
##  1st Qu.:2013   1st Qu.: 3.0   1st Qu.: 8.0   1st Qu.: NA    1st Qu.:1100
##  Median :2013   Median : 6.0   Median :12.0   Median : NA    Median :1600
##  Mean   :2013   Mean   : 5.8   Mean   :14.6   Mean   :NaN    Mean   :1476
##  3rd Qu.:2013   3rd Qu.: 8.0   3rd Qu.:22.0   3rd Qu.: NA    3rd Qu.:1850
##  Max.   :2013   Max.   :12.0   Max.   :31.0   Max.   : NA    Max.   :2229
##                                               NA's   :2512
##    dep_delay       arr_time    sched_arr_time   arr_delay      carrier
##  Min.   : NA    Min.   : NA    Min.   :   4   Min.   : NA    Length:2512
##  1st Qu.: NA    1st Qu.: NA    1st Qu.:1250   1st Qu.: NA    Class :character
##  Median : NA    Median : NA    Median :1754   Median : NA    Mode  :character
##  Mean   :NaN    Mean   :NaN    Mean   :1665   Mean   :NaN
##  3rd Qu.: NA    3rd Qu.: NA    3rd Qu.:2051   3rd Qu.: NA
##  Max.   : NA    Max.   : NA    Max.   :2359   Max.   : NA
##  NA's   :2512   NA's   :2512                  NA's   :2512
##      flight       tailnum             origin             dest
##  Min.   :   1   Length:2512        Length:2512        Length:2512
##  1st Qu.:1008   Class :character   Class :character   Class :character
##  Median :2169   Mode  :character   Mode  :character   Mode  :character
##  Mean   :2278
##  3rd Qu.:3459
##  Max.   :4484
##
##     air_time       distance         hour          minute
##  Min.   : NA    Min.   :  17   Min.   : 1.0   Min.   : 0
##  1st Qu.: NA    1st Qu.: 214   1st Qu.:11.0   1st Qu.: 0
##  Median : NA    Median : 544   Median :16.0   Median :20
##  Mean   :NaN    Mean   : 710   Mean   :14.5   Mean   :22
##  3rd Qu.: NA    3rd Qu.: 937   3rd Qu.:18.0   3rd Qu.:40
##  Max.   : NA    Max.   :4963   Max.   :22.0   Max.   :59
##  NA's   :2512
##    time_hour
##  Min.   :2013-01-02 15:00:00
##  1st Qu.:2013-03-06 08:00:00
##  Median :2013-06-07 10:00:00
##  Mean   :2013-06-08 20:58:28
##  3rd Qu.:2013-08-13 10:15:00
##  Max.   :2013-12-31 20:00:00
##
```

Answer
We can notice that all data with missing tailnum do not have any information about **dep_time** , **arr_time** and **arr_delay**. Therefore, it's probably the case that missing tailnum means the flight has being cancelled.

1.4

```r
# Question1 : What is the differences in flights delay on June 13rd compared to normal dates
# Question2 : Plot the comparison and cross-reference with Google
# Query
flights_full %>%
  filter(month == 6 & dest != "HNL" & dest != "ANC") %>%
  mutate(
    date_weather = ifelse(
      day == 13,
      "June 13rd",
      "Rest of June"
    )
  ) %>%
  group_by(dest, date_weather) %>%
  summarise(avg_arr_delay = mean(arr_delay, na.rm = TRUE)) %>%
  inner_join(airports, by = c("dest" = "faa")) %>%
  ggplot(aes(x = lon, y = lat, color = avg_arr_delay)) +
  borders("state") +
  geom_point() +
  ylim(20, 60) +
  labs(
    title = "Derecho Affects Flights Delay in Southeastern US on June 13rd 2013",
    y = "Longitude",
    x = "Latitude"
  ) +
  coord_quickmap() +
  scale_color_viridis_c() +
  facet_wrap(~date_weather,
    ncol = 2
  ) +
  theme(plot.title = element_text(face = "bold", size = 10))
```



Answer

Output as above

We can notice that on June 13rd there was a huge increase in arrival delay around Tennessee, North Carolina, Georgia and Alabama. Basically, the arrival delay in Southeastern US was hugely extent. Lots areas in the Midwest were also affected. By Searching on Google, we know that there was a **Derecho Series** occured between June 12nd and June 13rd in the US. The **June 13rd Derecho** occurred across the **Southern US** with major damage in **North Carolina, Tennessee, Georgia** and other Southeastern states. It aligns with the data results in the plot as we saw the largest delay occurred in Tennessee, and the broadly Southeast and the Midwest were also affected severely.

1.5

```
flights %>% anti_join(airports, by = c("dest" = "faa"))
```

```
## # A tibble: 5 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      544            545        -1     1004           1022
## 2  2013     1     1      615            615         0     1039           1100
## 3  2013     1     1      628            630        -2     1137           1140
## 4  2013     1     1      701            700         1     1123           1154
## 5  2013     1     1      711            715        -4     1151           1206
## # ... with 11 more variables: 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>
```

```
airports %>% anti_join(flights, by = c("faa" = "dest"))
```

```
## # A tibble: 1,427 x 8
##    faa   name                            lat    lon   alt    tz dst   tzone
##    <chr> <chr>                         <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
##  1 04G   Lansdowne Airport              41.1  -80.6  1044    -5 A     America/~
##  2 06A   Moton Field Municipal Airport  32.5  -85.7   264    -6 A     America/~
##  3 06C   Schaumburg Regional            42.0  -88.1   801    -6 A     America/~
##  4 06N   Randall Airport                41.4  -74.4   523    -5 A     America/~
##  5 09J   Jekyll Island Airport          31.1  -81.4    11    -5 A     America/~
##  6 0A9   Elizabethton Municipal Airport 36.4  -82.2  1593    -5 A     America/~
##  7 0G6   Williams County Airport        41.5  -84.5   730    -5 A     America/~
##  8 0G7   Finger Lakes Regional Airport  42.9  -76.8   492    -5 A     America/~
##  9 0P2   Shoestring Aviation Airfield   39.8  -76.6  1000    -5 U     America/~
## 10 0S9   Jefferson County Intl          48.1 -123.    108    -8 A     America/~
## # ... with 1,417 more rows
```

Answer
The first expression will return data of flights whose airport destination was not in the FAA list of airport code. Since FAA is for aviation-related facilities inside the US, the tibble we get maybe international flights

- The second expression will return data of US airports' information for those destinations which were not the destination of any of the flight in the **flights** dataset. Since the **flights** data only include all flights departed NYC in 2013, the result we get probably is a tibble of US airports which did not have nonstop flight directly from NYC in 2013

2 R4DS Chapter 16: lubridate

2.1

```
# Wirte a fucntion to transform depature time format
clean_dttm <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100)
}
# Apply the function to the date set
flights_full <- flights_full %>%
  mutate(
    dep_time = clean_dttm(year, month, day, dep_time),
    sched_dep_time = clean_dttm(year, month, day, sched_dep_time),
    dep_delay_clean = dep_time - sched_dep_time,
    diff_dep_delay = dep_delay_clean - (dep_delay * 60)
  ) %>%
  select(diff_dep_delay, dep_delay_clean, dep_delay, dep_time, sched_dep_time, everything())
flights_clean_dep <- flights_full %>%
  filter(diff_dep_delay != 0)
# Difference between calculated departure delay and departure delay given
table(flights_clean_dep$diff_dep_delay)
```

```
##
## -86400
##   1207
```

```
# Difference between after taking into account of the correct date
flights_clean_dep <- flights_clean_dep %>%
  mutate(
    dep_time_clean = dep_time + days(1),
    dep_delay_clean = dep_time_clean - sched_dep_time,
    diff_dep_delay = dep_delay_clean - dep_delay
  )
table(flights_clean_dep$diff_dep_delay)
```

```
##
##    0
## 1207
```

Answer

If we use the *year*, *month*, *day*, *dep_delay* and *sched_dep_delay* columns to calculate the difference, since we cannot get both the *month* and *day* information for real departure time and scheduled departure time in the dataset, we will get 1207 rows with non-zero difference between the calculated departure delay and the number given in the dataset. However, after taking a closer look at those numbers, we notice that all those data follow the same pattern, i.e. the calculated number is 86400 secs smaller than the actual number. Since 86400 seconds equal the time of a whole day, it's likely that all those flights were posponeed to the next day and we neglected that for the first time. After correcting this, the data we get is the same as the actual number.

2.2

```
d1 <- "1213-Apr-03"
ymd(d1)
```

```
## [1] "1213-04-03"
```

```
d2 <- "06-Jun-2017"
dmy(d2)
```

```
## [1] "2017-06-06"
```

```
d3 <- "12/29/14" # Dec 29, 2014
mdy(d3)
```

```
## [1] "2014-12-29"
```

```
d4 <- "November 20, 1909"
mdy(d4)
```

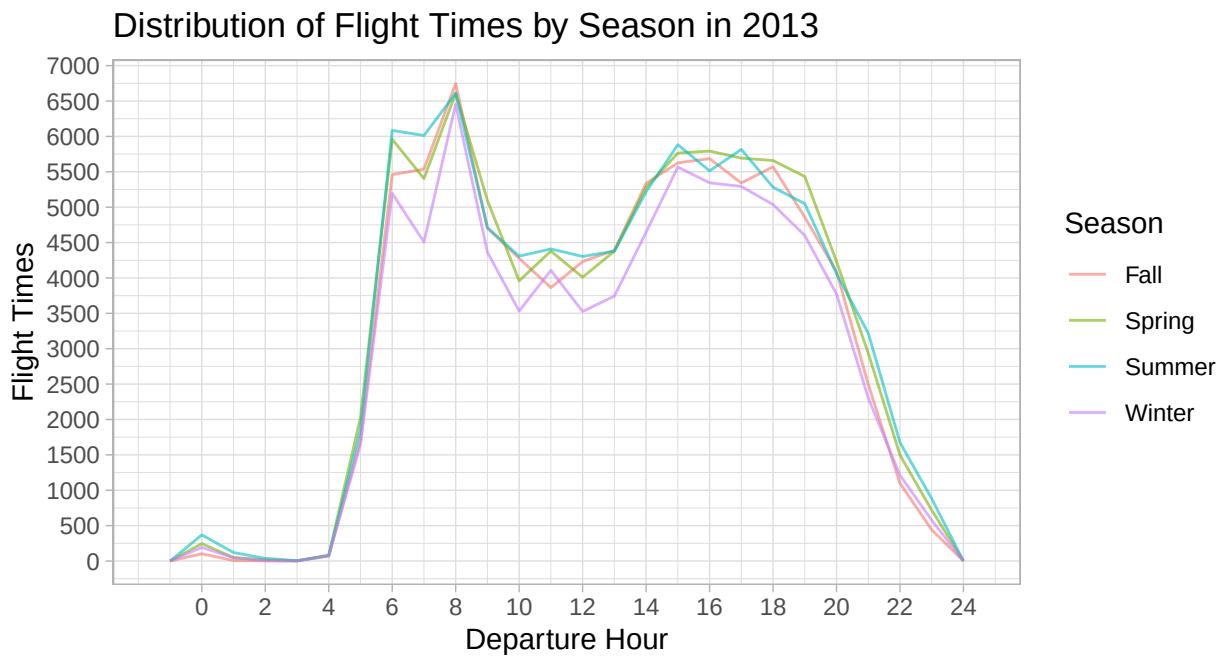
```
## [1] "1909-11-20"
```

```r
d5 <- c("January 2 (2016)", "January 2 (2018)")
mdy(d5)
```

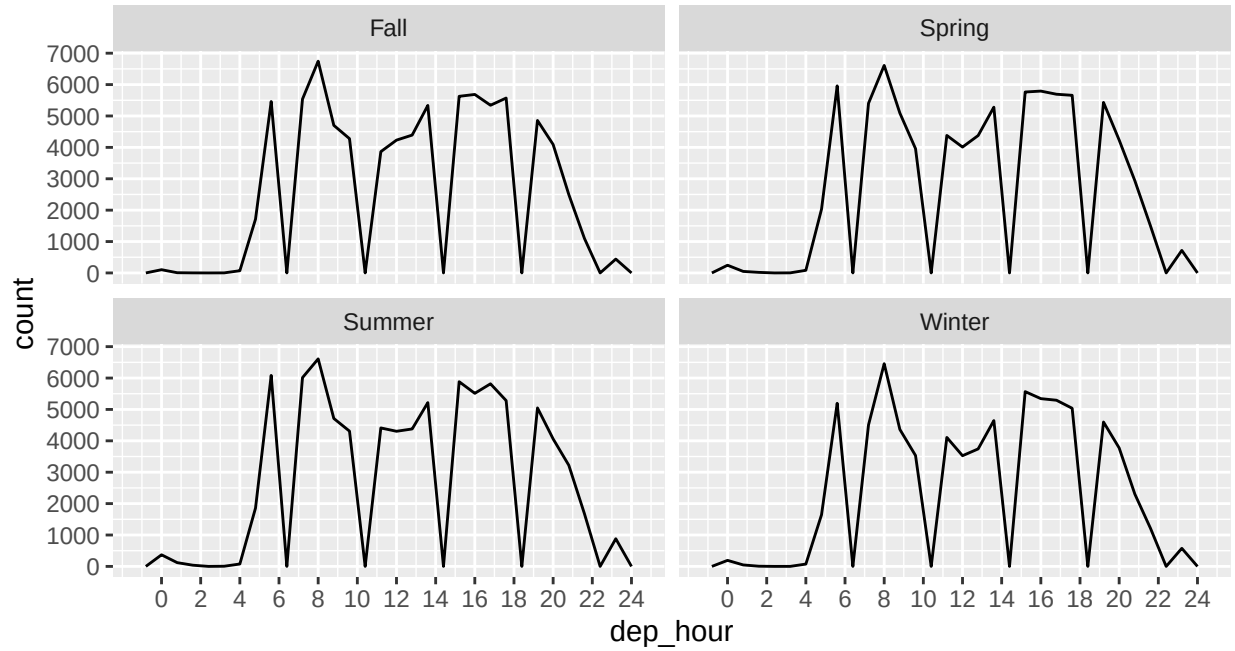
```
## [1] "2016-01-02" "2018-01-02"
```

2.3

```r
flights_clean_dep <- flights_clean_dep %>%
  select(tailnum, dep_time, dep_time_clean, sched_dep_time)
flights_plot <- flights_full %>%
  left_join(flights_clean_dep, by = c("tailnum", "sched_dep_time")) %>%
  mutate(
    dep_time = ifelse(
      is.na(dep_time_clean) == TRUE,
      dep_time.x,
      dep_time_clean
    ),
    dep_time = as_datetime(dep_time),
    season = ifelse(
      dep_time > as.Date("2013-12-22") | dep_time < as.Date("2013-03-20"),
      "Winter",
      ifelse(
        dep_time < as.Date("2013-06-21"),
        "Spring",
        ifelse(
          dep_time < as.Date("2013-09-23"),
          "Summer",
          "Fall"
        )
      )
    )
  ) %>%
  filter(!is.na(dep_time)) %>%
  mutate(dep_hour = hour(dep_time))
# Plot
flights_plot %>%
  ggplot(aes(dep_hour)) +
  geom_freqpoly(aes(color = as.factor(season)),
    binwidth = 1,
    alpha = 0.6
  ) +
  scale_x_continuous(breaks = seq(0, 24, 2)) +
  scale_y_continuous(breaks = seq(0, 7000, 500)) +
  labs(
    title = "Distribution of Flight Times by Season in 2013",
    x = "Departure Hour",
    y = "Flight Times",
    color = "Season"
  ) +
  theme_light()
```




```
# Facet season
flights_plot %>%
  ggplot(aes(dep_hour)) +
  geom_freqpoly(binwidth = .8) +
  scale_x_continuous(breaks = seq(0, 24, 2)) +
  scale_y_continuous(breaks = seq(0, 7000, 1000)) +
  facet_wrap(~season)
```



Answer
Based on the plot above, we can see that 8am had the most flights during the day among different seasons. More generally, morning time(6am to 8am) and afternoon time(2pm to 7pm) had the most flights compared to other times during the day. Noon time(10am-12am) has the lowest flights. The pattern applies to all four seasons.

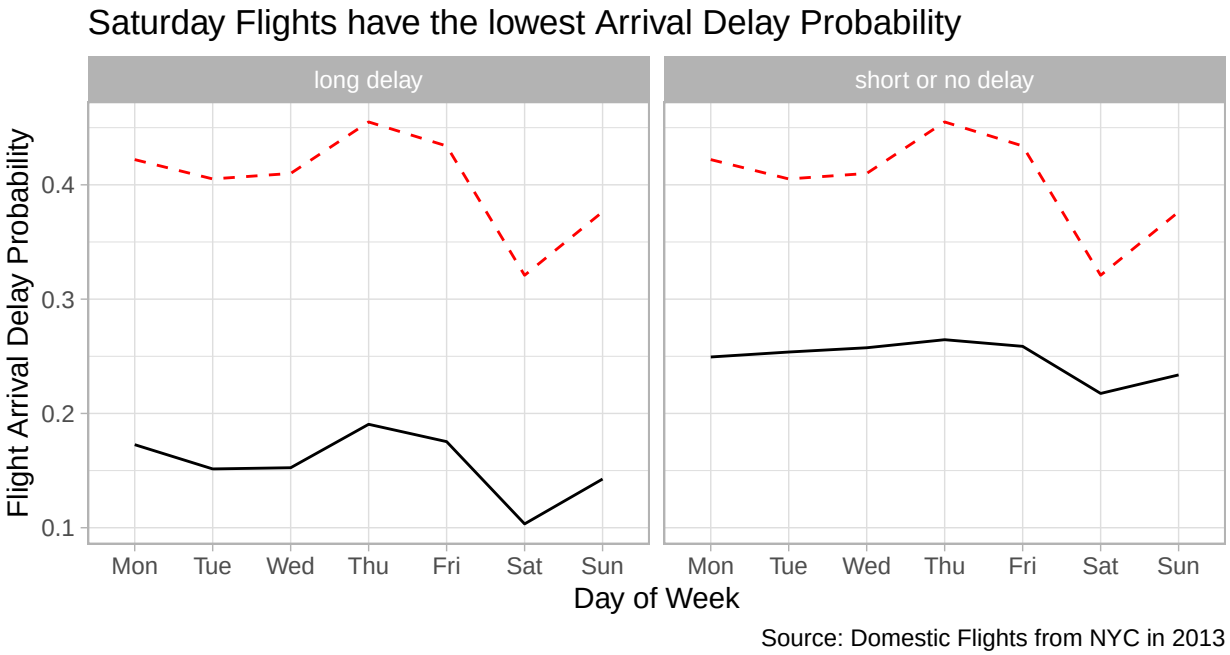
- To be more specific, although the pattern is quite similar between different seasons, summer has more flights throughout the day while winter has less flights. Also, during the winter, the number of flights starts to decline much earlier than all the other seasons, which aligns with the weather condition that day is shorter and the weather is usually worse at night. Also, we use Daylight Saving Time during winter so the 2nd highest peak occurs earlier.
- We also need to notice that there is a small increase around 12am during all four seasons. The number of flights then continue to decrease until 4 am. After 8pm at night, the number also continues to decrease no matter the season.

Reference https://www.calendardate.com/year2013.php https://statisticsglobe.com/convert-dates-seasons-r

2.4

```
# Query
flights_plot <-
  flights_plot %>%
  drop_na(arr_delay) %>%
  mutate(
    week_day = wday(dep_time, label = TRUE, week_start = 1),
    arr_delay_bi = ifelse(
      arr_delay > 0,
      1,
      0
      ),
    arr_delay_long = ifelse(
      arr_delay > 30,
      "long delay",
      "short or no delay"
      )
    ) %>%
  group_by(week_day) %>%
  mutate(
    count = n(),
    pct_arr_delay_total = sum(arr_delay_bi) / count
    ) %>%
  group_by(week_day, arr_delay_long) %>%
  mutate(
    pct_arr_delay = sum(arr_delay_bi) / count
  ) %>%
  distinct(week_day, arr_delay_long,
           .keep_all = TRUE) %>%
  select(week_day, arr_delay_long,   pct_arr_delay_total, pct_arr_delay) %>%
  arrange(week_day, arr_delay_long)
# Plot
flights_plot %>%
```

```
ggplot() +
geom_line(aes(x = week_day, y = pct_arr_delay,
              group = 1)) +
scale_y_continuous(breaks = seq(0, 0.5, 0.1)) +
labs(title = "Saturday Flights have the lowest Arrival Delay Probability",
    x = "Day of Week",
    y = "Flight Arrival Delay Probability",
    caption = "Source: Domestic Flights from NYC in 2013") +
facet_wrap(~ arr_delay_long) +
geom_line(aes(x = week_day, y = pct_arr_delay_total,
              group = 1),
          linetype = 2,
          color = "red") +
theme_light()
```



Answer

Within the dataset, Saturday flights have the lowest percentage of arrival delay during the whole week. The result does not change even though we only look at long delays

- *Definition of long delay*: Based on the delay rules of US Transportation Department, if a flight delays more than 30 mins, then the airline must notify passengers. Also based on the percentile distribution of the arrival delay in the dataset, we notice that the 75% percentile is 14 mins delay. Therefore, we consider flights with *30 mins and longer* delays as *long arrival delay flights*
- *Explanation of plots*: Based on the plot above, in both plots, Saturday has the lowest percentage of arrival delay flights. However, we also need to notice that the delay pattern changed a bit when we look at *long delay* flights and *all delay* flights. Saturday flights were much more likely to be punctuate compared to other days overall, while the difference is much smaller if we only consider flights with *long delay*. Generally, the differences of *long delay possibility* differs only a bit among different days in a week.

2.5

```
# Every 7th day of each month in 2012
ymd("2012-01-07") + months(0:11)
```

```
##  [1] "2012-01-07" "2012-02-07" "2012-03-07" "2012-04-07" "2012-05-07"
##  [6] "2012-06-07" "2012-07-07" "2012-08-07" "2012-09-07" "2012-10-07"
## [11] "2012-11-07" "2012-12-07"
```

```
# Every 5th day of each month in 2020
ymd("2020-01-05") + months(0:11)
```

```
##  [1] "2020-01-05" "2020-02-05" "2020-03-05" "2020-04-05" "2020-05-05"
##  [6] "2020-06-05" "2020-07-05" "2020-08-05" "2020-09-05" "2020-10-05"
## [11] "2020-11-05" "2020-12-05"
```

3 R4DS 19: Functions

3.1

```
# Function
str_parse <- function(x, collapse = ", ") {
  str_c(x, collapse = collapse)
}
# Test
test <- letters[1:2]
str_parse(test)
```

```
## [1] "a, b"
```

```
test <- letters[1]
str_parse(test)
```

```
## [1] "a"
```

Answer
The function works for a vector of length 1 or length 2. It will always return a vector, but the number of elements in the vector depends on that of the tested vector

Reference https://r-lang.com/letters-in-r/

3.2

```
age <- function(birthday) {
  (birthday%--% today()) %/% years(1)
  }
age(ymd("1999-07-01"))
```

```
## [1] 22
```

Answer
It works as I get the correct answer of my age.

3.3

```
# Variance Function
variance <- function(x, na.rm = TRUE) {
  xbar <- mean(x, na.rm = TRUE)
  sum((x - xbar) ^ 2) / (length(x) - 1)
  }
# Test
x <- seq(1, 6, 2)
variance(x)
```

```
## [1] 4
```

```
var(x)
```

```
## [1] 4
```

```
library(moments)
# Skewness Function
skew <- function(x, na.rm = FALSE) {
  xbar <- mean(x, na.rm = FALSE)
  var <- variance(x, na.rm = FALSE)
  (sum((x - xbar)^3) / (length(x) - 2)) / var^(3 / 2)
}
# Test
x <- c(3, 6, 20, 120, 500)
skew(x)
```

```
## [1] 1.6
```

```
skewness(x)
```

```
## [1] 1.3
```

```
str(diamonds)
```

```
## tibble [53,940 x 10] (S3: tbl_df/tbl/data.frame)
##  $ carat  : num [1:53940] 0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth  : num [1:53940] 61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num [1:53940] 55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int [1:53940] 326 326 327 334 335 336 336 337 337 338 ...
##  $ x      : num [1:53940] 3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num [1:53940] 3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z      : num [1:53940] 2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

```
diamonds %>%
  drop_na() %>%
  summarise_if(
    is.numeric,
    list(variance = variance, mean = mean)
  ) %>%
  pivot_longer(
    cols = everything(),
    names_to = c("diamond_dimension", "index"),
    names_pattern = "(.*)_(.*)",
    values_to = "count"
  ) %>%
  pivot_wider(
    names_from = index,
    values_from = count
  )
```

```
## # A tibble: 7 x 3
##   diamond_dimension     variance     mean
##   <chr>                    <dbl>    <dbl>
## 1 carat                    0.225    0.798
## 2 depth                    2.05    61.7
## 3 table                    4.99    57.5
## 4 price             15915629.    3933.
## 5 x                        1.26     5.73
## 6 y                        1.30     5.73
## 7 z                        0.498    3.54
```

```
var_test <- as.data.frame(sapply(diamonds, var))
```

```
## Error in FUN(X[[i]], ...): Calling var(x) on a factor x is defunct.
##   Use something like 'all(duplicated(x)[-1L])' to test for a constant vector.
```

```
mean_test <- as.data.frame(sapply(diamonds, mean))
cbind(var_test, mean_test) %>%
  na.omit()
```

```
## Error in cbind(var_test, mean_test): object 'var_test' not found
```

Answer
The answer we get with *summarise_if()* is the same as the result with *sapply*, therefore it passed the test

3.4

```
# First Function
check_prefix <- function(string, prefix) {
  str_sub(string, 1, nchar(prefix)) == prefix
}
# Test
check_prefix(c("arrtime", "arrdelay", "depdelay"), "arr")
```

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

```
# Second Function
drop_last <- function(x) {
  if (length(x) <= 1) {
    return(NULL)
  }
  x[-length(x)]
}
# Test
drop_last(1:3)
```

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

```
drop_last(1)
```

```
## NULL
```

Answer
The *first* function is used to check whether the string starts with a specific prefix, therefore, we name it as *check_prefix*. The *second* function is used to drop the last value of a vector if its length is greater than 1. If its length is 1 or 0, we will get *NULL* instead. Therefore, we name it as *drop_last*

3.5

```
greetings <- function(time = lubridate::now()) {
  hour <- lubridate::hour(time)
  ifelse(hour < 12,
    "Good morning",
    ifelse(hour < 18,
      "Good afternoon",
      "Good evening"
    )
  )
}
# Test
greetings(ymd_hm("2022-01-01 08:01"))
```

```
## [1] "Good morning"
```

```
greetings(ymd_hm("2022-01-01 12:01"))
```

```
## [1] "Good afternoon"
```

```
greetings(ymd_hm("2022-01-01 18:01"))
```

```
## [1] "Good evening"
```

Answer
We set time before 12pm as *morning*, after 6pm as *evening*, and the rest(12pm to 6pm) as *afternoon*. The test results showed that the function works.

4 R4DS Chapter 20: Vectors

4.1

```
# Create a double
x <- rnorm(10, 2, sd = 1)
typeof(x)
```

```
## [1] "double"
```

```
# Test
x
```

```
##  [1] 0.82 2.20 0.41 1.69 2.72 2.91 4.80 0.93 2.63 0.53
```

```
ceiling(x)
```

```
##  [1] 1 3 1 2 3 3 5 1 3 1
```

```
floor(x)
```

```
##  [1] 0 2 0 1 2 2 4 0 2 0
```

```
trunc(x)
```

```
##  [1] 0 2 0 1 2 2 4 0 2 0
```

```
round(x)
```

```
##  [1] 1 2 0 2 3 3 5 1 3 1
```

```
round(x, digits = 1)
```

```
##  [1] 0.8 2.2 0.4 1.7 2.7 2.9 4.8 0.9 2.6 0.5
```

Answer
There are 4 functions which allow us to convert a *double* into an *integer*, i.e. **ceiling()**, **floor()**, **trunc()**, **round()**

- **ceiling**: It will transform a single numeric argument x to its smallest integer which is no less than x. The returning result would be a numeric vector. That is, it will convert each element in the argument into **its nearest integer which is larger than itself**. For example, in the double we create, both **2.277** and **2.922** are transformed to **3** since they are **both greater than 2 while smaller than 3**. Usually, with this method, numbers with **the same values before decimal places** will be changed to **the same value**.

- **floor()**: Its method is quite similar to **ceiling**. However, the key difference is that instead of transforming each element into its nearest integer which is larger than itself, **floor** will change the element to **its nearest integer which is no greater than itself**. numbers with the same values before decimal places will also be changed to the same value. For example, in the double we create, both **0.268** and **0.683** are transformed to **0** since they are **both greater than 0 while smaller than 1**.
- **trunc()**: It will return a numeric vector with each single numeric argument x being truncated to an integer toward 0. In the double we create, using **floor** and **trunc** will create the same result since they are both round the double to its nearest integer which is close to 0.
- **round()**: It will round the values to the specified number of decimal places. The default decimal digits are 0 but we can change it by *digits* =. As for the rounding method, it follows the IEEE Standard, which is **round to nearest, ties to even**. It means for those with decimal smaller than 0.5 it will be rounded down and the rest half will be rounded up. For those with .5 as decimal, it will be rounded to the nearest even number

Reference R help function

4.2

```
last_value <- function(x) {
  ifelse(
    length(x),
    x[[length(x)]],
    x
  )
}
x <- seq(1, 5, 1)
last_value(x)
```

```
## [1] 5
```

```
even_order <- function(x) {
  if (length(x)) {
    x[seq_along(x) %% 2 == 0]
  } else {
    x
  }
}
even_order(1)
```

```
## numeric(0)
```

```
even_order("a")
```

```
## character(0)
```

```
even_order(x)
```

```
## [1] 2 4
```

Answer

- The function works to extract the last value. We need to use *[[* instead of *[* to get the last value
- The function to get the values in even positions also passed the test. Also, we notice that if there is no value at even order in the vector, it will returns with *object_type(0)*

4.3

```
knitr::include_graphics('Diagram_ps5.png')
```

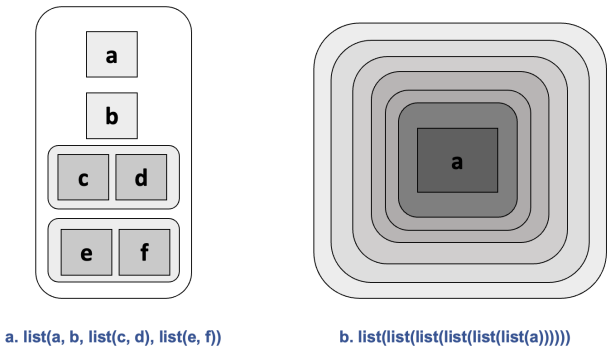


Figure 1: Nested Set Diagram

Answer
The diagram above showed the nested set. In the diagram, lists have rounded corners while atomic vectors have square corners.

4.4

```
x <- 1:5
y <- letters[1]
max_length <- max(c(length(x), length(y)))
max_length
```

```
## [1] 5
```

```
tibble(
  "x" = c(x, rep(NA, max_length - length(x))),
  "y" = y
  )
```

```
## # A tibble: 5 x 2
##       x y
##   <int> <chr>
## 1     1 a
```

```
## 2     2 a
## 3     3 a
## 4     4 a
## 5     5 a
```

```
tibble(
  "x" = c(x, rep(NA, max_length - length(x))),
  "y" = c(y, rep(NA, max_length - length(y)))
  )
```

```
## # A tibble: 5 x 2
##       x y
##   <int> <chr>
## 1     1 a
## 2     2 <NA>
## 3     3 <NA>
## 4     4 <NA>
## 5     5 <NA>
```

Answer
We can not create a tibble with columns in different length. Examples above showed that we can create a tibble with different length. However, for those columns which have less values than the maximum number of values in a single column in the tibble, R fill out those blanks with *NA*. In fact, *tibbles* require every element of a data frame be vectors with th e same length, so technically it's not possible and that's' why we need NA or R will automatically fill out the missing rows with the same value as the existing ones

5 R4DS Chapter 21: Iterations

5.1

```
# Mean of every column in mtcars
mean_mtcar <- vector("double", length(mtcars))
names(mean_mtcar) <- names(mtcars)

for (i in names(mtcars)) {
  mean_mtcar[i] <- mean(mtcars[[i]])
  }
mean_mtcar
```

```
##    mpg    cyl   disp     hp   drat     wt   qsec     vs     am   gear   carb
##  20.09   6.19 230.72 146.69   3.60   3.22  17.85   0.44   0.41   3.69   2.81
```

```
# Number of unique values in each column of mpg
mpg_unique <- vector("double", length(mpg))
names(mpg_unique) <- names(mpg)

for (i in names(mpg)) {
  mpg_unique[i] <- n_distinct(mpg[[i]])
  }
mpg_unique
```

```
## manufacturer        model        displ         year          cyl        trans
##           15           38           35            2            4           10
##          drv          cty          hwy           fl        class
##            3           21           27            5            7
```

```
# Generate 10 random points distributed poissons for each of lamda = 1, 3, 10, 30 and 100
n <- 10
lamda <- c(1, 3, 10, 30, 100)
poissons <- vector("list", length(lamda))

for (i in seq_along(poissons)) {
  poissons[[i]] <- rpois(n, lambda = lamda[i])
  }
poissons
```

```
## [[1]]
##  [1] 1 0 1 0 2 0 1 3 2 0
##
## [[2]]
##  [1] 5 7 3 2 4 5 6 3 2 4
##
## [[3]]
##  [1]  9 11  9 10  7 15 10  7  9 11
##
## [[4]]
##  [1] 20 40 33 30 32 32 31 35 36 29
##
## [[5]]
##  [1]  92  95  97  97 107  93  95  85 106  91
```

5.2

```
files <- dir("data/", pattern = "\\.csv$", full.names = TRUE)
file_list <- vector("list", length(files))

for (i in seq_along(files)) {
  file_list[[i]] <- read_csv(files[[i]])
}

files_df <- bind_rows(file_list)
files_df
```

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

Note: It returns *character(0)* since there is no such files in our computer yet
Reference: https://dplyr.tidyverse.org/reference/bind.html

5.3

```r
# I will replicate the mean of each numeric column for dataset iris
show_mean <- function(df, digits = 2) {
  maxstr <- max(str_length(names(df)))
  for (nm in names(df)) {
    if (is.numeric(df[[nm]])) {
      cat(str_c(str_pad(str_c(nm, ":"), maxstr + 1L, side = "right"),
                format(mean(df[[nm]]), digits = digits, nsmall = digits),
                sep = " "),
          "\n")
    }
  }
}
# Test
show_mean(iris)
```

```
## Sepal.Length: 5.84
## Sepal.Width:  3.06
## Petal.Length: 3.76
## Petal.Width:  1.20
```

```r
as.tibble(sapply(iris, mean))
```

```
## # A tibble: 5 x 1
##   value
##   <dbl>
## 1  5.84
## 2  3.06
## 3  3.76
## 4  1.20
## 5 NA
```

Answer
We get the same answers for each column means of *iris* as the given one in the question

5.4

```r
# Mean of every column in mtcars
map_dbl(mtcars, mean)
```

```
##    mpg    cyl   disp     hp   drat     wt   qsec     vs     am   gear   carb
##  20.09   6.19 230.72 146.69   3.60   3.22  17.85   0.44   0.41   3.69   2.81
```

```r
# Number of unique values in each column of mpg
map_int(mpg, ~ length(unique(.)))
```

```
## manufacturer        model        displ         year          cyl        trans
##           15           38           35            2            4           10
##          drv          cty          hwy           fl        class
##            3           21           27            5            7
```

```r
# Generate 10 random points distributed poissons for each of lamda = 1, 3, 10, 30 and 100
map_dfc(c(1, 3, 10, 30, 100), rpois, n = 10)
```

```
## # A tibble: 10 x 5
##     ...1  ...2  ...3  ...4  ...5
##    <int> <int> <int> <int> <int>
##  1     2     5    10    32   115
##  2     1     4     9    39   106
##  3     0     3    17    42    87
##  4     0     1    11    34   110
##  5     3     2    11    29    98
##  6     2     3    11    33   119
##  7     1     4     8    42   121
##  8     1     1     7    26    95
##  9     1     2     8    41   103
## 10     0     3     7    39    89
```

5.5

```r
map_dfc(files, ~ read_csv(.))
```

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

5.6

```r
five_squares <- (1:5)^2
map(list(five_squares), rnorm)
```

```
## [[1]]
## [1] -1.24 -0.84 -0.61 -0.83  0.46
```

```r
map(five_squares, rnorm)
```

```
## [[1]]
## [1] 0.98
##
## [[2]]
## [1]  1.52  1.07 -1.15 -0.18
##
## [[3]]
## [1]  0.41 -0.66  0.48  0.60  0.95 -0.91  0.02 -1.87 -1.07
##
## [[4]]
##  [1] -1.194  1.693  0.458 -0.499 -0.806  0.477 -0.563  0.914  2.150 -1.587
## [11]  1.567 -0.038  0.844 -0.309  0.030 -1.764
##
## [[5]]
##  [1] -1.215  0.238 -0.336 -1.194  0.531 -0.611 -1.253  0.081  0.227 -1.683
## [11] -0.090  0.637  0.614 -2.149  0.846  1.599  0.750 -0.782 -0.290  0.398
## [21] -2.141 -0.877  1.410  0.606 -0.297
```

```
map(five_squares, rnorm, n = 5)
```

```
## [[1]]
## [1] 2.104 1.185 2.292 0.069 1.388
##
## [[2]]
## [1] 4.2 4.8 4.5 4.1 3.4
##
## [[3]]
## [1]  9.2 11.0  9.1  9.8  7.9
##
## [[4]]
## [1] 15 15 16 15 16
##
## [[5]]
## [1] 26 25 27 23 23
```

Answer
We will always get a list as output with *map()*:

- a. The map() function applies *rnorm* to each element of the transformed *five_squares* and returning a list of the same length and same number of numeric vectors, i.e. 1 list with 5 numeric numbers. *rnorm* will run for 5 times since there are 5 elements in the list but each time the list will be treated as a whole. And *rnorm* will use standard normalized distribution as default since there is no expression to define the mean and sd. Although *five_squares* is not a list, by using the function *list()* it has been transformed into a list with length 1 and therefore *rnorm* will be applied based on the list unit. *list(five_squares)* here will give the same result as *list(c(1, 4, 9, 16, 100))*
- b. The map() function applies *rnorm* to each element within the vector. That is, it will pass the value of 1, 4, 9, 16, 25 to the first argument **n** of *rnorm* during each run and take standard normal distribution samples randomly. It will also return a list, with each element at the length of the value of each element in *five_squares*.
- c. The map() function also runs *rnorm* for each element within the vector. However, 1, 4, 9, 16, 25 serve as the **mean** for *rnorm* instead and we will get 5 vectors each with 5 numbers. It differs from *b* because now the first argument of *rnorm* has already been stated(n = 5) so the inputs from *five_squares* pass to the 2nd argument *mean* instead