# Chapter 3 and 4 Executables and Exercises

3.2.5 Exercises

1. In a single pipeline for each condition, find all flights that meet the condition:
    - Had an arrival delay of two or more hours

```
flights %>%
  filter(arr_delay >= 120)
```

    - Flew to Houston (IAH or HOU)

```
flights %>%
  filter(dest %in% c("IAH", "HOU"))
```

    - Were operated by United, American, or Delta

```
flights %>%
  filter(carrier %in% c("UA", "AA", "DL"))
```

    - Departed in summer (July, August, and September)

```
flights %>%
  filter(month %in% c(7, 8, 9))
```

    - Arrived more than two hours late but didn't leave late

```
flights %>%
  filter(arr_delay > 120, dep_delay <= 0)
```

    - Were delayed by at least an hour, but made up over 30 minutes in flight

```
flights %>%
  filter(dep_delay >= 60, (dep_delay - arr_delay) > 30)
```

2. Sort flights to find the flights with the longest departure delays. Find the flights that leave earliest in the morning.

```
flights %>%
  arrange(desc(dep_delay))
flights %>%
  arrange(dep_time) %>%
  slice_head(n = 1)
```

3. Sort flights to find the fastest flights. (Hint: Try including a math calculation inside of your function.)

- flights %>%
- mutate(speed = distance / (air_time / 60)) %>%
- arrange(desc(speed))
-

4. Was there a flight on every day of 2013?

- flights %>%
- distinct(year, month, day) %>%
- nrow()
-

5. Which flights traveled the farthest distance? Which traveled the least distance?

- Farthest:
- flights %>%
- arrange(desc(distance)) %>%
- slice_head(n = 1)
- Least distance:
- flights %>%
- arrange(distance) %>%
- slice_head(n = 1)
-

6. Does it matter what order you used filter() and arrange() if you're using both? Why/why not? Think about the results and how much work the functions would have to do.

- The final result will be the same, but ordering can affect performance. If you filter first and then arrange in a dplyr pipe, you reduce the number of rows to order. That means that arrange() has less work to do, which is quicker, especially with bigger data. Chapter 3 of R for Data Science emphasizes pipelines that are readable as well as efficient. Early filtering is a beneficial practice as it minimizes the computational expense for later steps.

## 3.3.5 Exercises

1. Compare dep_time, sched_dep_time, and dep_delay. How would you expect those three numbers to be related?

- **sched_dep_time** is the planned departure time.
- **dep_time** is when the flight actually left.
- **dep_delay** is the difference (in minutes) between the actual and scheduled departure times (i.e., dep_time minus sched_dep_time).

2. Brainstorm as many ways as possible to select dep_time, dep_delay, arr_time, and arr_delay from flights.

- These are ways to pick out just the **dep_time**, **dep_delay**, **arr_time**, and **arr_delay** columns from the flight data:
- Directly by name: flights %>%
- select(dep_time, dep_delay, arr_time, arr_delay)
-
- Using a character vector with all_of(): cols <- c("dep_time", "dep_delay", "arr_time", "arr_delay")
- flights %>%
- select(all_of(cols))
-
- Using one_of(): flights %>%
- select(one_of("dep_time", "dep_delay", "arr_time", "arr_delay"))
-
- Using matches() with a regular expression: flights %>%
- select(matches("^(dep|arr)_(time|delay)$"))

3. What happens if you specify the name of the same variable multiple times in a select()call?

- When you list the same variable multiple times in a **select()** call, dplyr will only include that variable once in the final output.

4. What does the any_of() function do? Why might it be helpful in conjunction with this vector? variables <- c("year", "month", "day", "dep_delay", "arr_delay")

- The **any_of()** function is a way to safely select columns using a character vector. For example, if you have:

  variables <- c("year", "month", "day", "dep_delay", "arr_delay")

  and then you write:

  flights %>%

select(any_of(variables))

it will look for those columns in the **flights** dataset and only select the ones that exist. If one of the names in your vector isn't found in the dataset, **any_of()** just skips it rather than throwing an error. This is especially useful when working with datasets that might have different column names or when you're writing code that should work even if some columns are missing.

5. Does the result of running the following code surprise you? How do the select helpers deal with upper and lower case by default? How can you change that default?

   flights |> select(contains("TIME"))

- This happens because, by default, the select helpers (like **contains()**) are case sensitive. Since the column names in **flights** are in lowercase (e.g., "dep_time"), and uppercases won't match them.

6. Rename air_time to air_time_min to indicate units of measurement and move it to the beginning of the data frame.

1. Use **rename()** to change air_time to air_time_min, which shows that the numbers represent minutes.
2. Then, use **select()** with **everything()** to move this renamed column to the front.

7. Why doesn't the following work, and what does the error mean?

```
flights |> select(tailnum) |> arrange(arr_delay)
```

- This error occurs as a result of when users use **select(tailnum)**. Users only keep the tailnum column and drop everything else, including arr_delay. Then, when you try to arrange by arr_delay, it doesn't exist in your data anymore. The error is telling you that it can't find the arr_delay column to sort by.

### 3.5.7 Exercises

1. Which carrier has the worst average delays? Challenge: can you disentangle the effects of bad airports vs. bad carriers? Why/why not? (Hint: think about flights |> group_by(carrier, dest) |> summarize(n()))
- A carrier could look like it has the worst delays just because it flies to airports that have bad delays. It's hard to tell if the carrier or the airport is the real reason for the delays. for a couple of seconds
- A carrier might look worse simply because it flies to airports that often have delays. Grouping by both carrier and destination helps reveal this, but it's still hard to say if the delays are mainly the carrier's fault or the airport's.

2. Find the flights that are most delayed upon departure from each destination.
- flights %>% group_by(dest) %>%
-  slice_max(dep_delay, n = 1, with_ties = FALSE)

3. How do delays vary over the course of the day? Illustrate your answer with a plot.
- To see how delays change throughout the day, you can calculate the average delay for each hour and plot it. Typically, delays are smaller in the morning and get worse as the day goes on. A simple plot can show this trend, helping you visualize the pattern of delays over time.

4. What happens if you supply a negative n to slice_min() and friends?

- If you supply a negative number to slice_min() or similar functions (like slice_max()), it will essentially select the rows with the largest values instead, because a negative value is treated as if you're selecting the highest (or least negative) instead of the smallest (or most negative). For example, instead of picking the "smallest" values, it will pick the "largest" ones.

5. Explain what count() does in terms of the dplyr verbs you just learned. What does the sort argument to count() do?

- The **count()** function is like a shortcut for grouping data and then summarizing it to count the number of rows in each group. It's similar to doing:

- **group_by()** on one or more variables, and then
- **summarize(n = n())** to count the rows in each group.

The **sort** argument, when set to TRUE, orders the results so that the groups with the highest counts come first.

6. Suppose we have the following tiny data frame:

```
df <- tibble(
  x = 1:5,
  y = c("a", "b", "a", "a", "b"),
  z = c("K", "K", "L", "L", "K")
)
```

a). Write down what you think the output will look like, then check if you were correct, and describe what group_by() does.

- It doesn't change the data itself. Instead, it marks the data so that all rows with the same value of **y** are grouped together (one group for "a" and one for "b"). The output still shows all the rows and columns, but you'll see a note like # Groups: y [2] indicating there are 2 groups. This grouping is useful when you want to do calculations for each group separately later on.

b). Write down what you think the output will look like, then check if you were correct, and describe what arrange() does. Also, comment on how it's different from the group_by() in part (a).

- **What arrange() Does:**
- arrange() reorders the rows of a data frame based on one or more columns. It sorts the rows so you can see the data in a different order, such as alphabetically by y or numerically by x.
- **How It Differs from group_by():**
- group_by() doesn't change the order of rows at all—it just tags the rows into groups based on a column (or columns) for later operations.
- arrange() actively changes the order of the rows, while group_by() is more about organizing data for summaries or calculations without altering row order.

c). Write down what you think the output will look like, then check if you were correct, and describe what the pipeline does.

- I expect the output to be a tibble with two rows—one for each group in **y**—and two columns: **y** and **mean_x**. For the rows where **y** is "a", the mean of **x** should be (1 + 3 + 4)/3 = 2.67 (approximately). For the rows where **y** is "b", the mean should be (2 + 5)/2 = 3.5. So, the output might look like -

- y mean_x

  <chr> <dbl>

  1 a 2.67

  2 b 3.5

d). Write down what you think the output will look like, then check if you were correct, and describe what the pipeline does. Then, comment on what the message says.

- I expect the output to be a tibble with three rows and three columns. It should show:
- For y = "a" and z = "K", the mean of x is 1.
- For y = "a" and z = "L", the mean of x is (3 + 4)/2 = 3.5.
- For y = "b" and z = "K", the mean of x is (2 + 5)/2 = 3.5.
- The data is now only grouped by **y** instead of both **y** and **z**. It tells you that **summarize()** automatically drops the last grouping variable (in this case, **z**), so your output is only grouped by **y**.

e). Write down what you think the output will look like, then check if you were correct, and describe what the pipeline does. How is the output different from the one in part (d)?

- I expect the output to be similar to the previous one, but this time the data won't have any grouping information because we used `.groups = "drop"`. This tells R to remove the grouping after the summary is done.
- The difference from part (d) is that in part (d), the output kept some grouping (by **y** by default), but since we used **.groups = "drop"**, the result is no longer grouped at all.

f). Write down what you think the outputs will look like, then check if you were correct, and describe what each pipeline does. How are the outputs of the two pipelines different?

- **Pipeline 1:**

- **group_by(y, z):** It groups the data by both **y** and **z**.
- **summarize(mean_x = mean(x)):** It calculates the average of **x** for each group.
- **Default grouping behavior:** After summarizing, R drops the last grouping variable (**z**) and keeps the grouping by **y**. So the output still remembers the groups by **y**.

- **Pipeline 2:**
  - It does the same grouping and summarizing as Pipeline 1.
  - **.groups = "drop":** This tells R to remove all grouping from the final result, so the output is not grouped at all.

**How the Outputs Differ:**

Both pipelines show the same numbers and rows. The difference is that Pipeline 1's result is still grouped by **y** (it carries extra grouping info), while Pipeline 2's result is ungrouped.

## 4.6 Exercises

Restyle the following pipelines following the guidelines above.

```
flights |>
  filter(dest == "IAH") |>
  group_by(year, month, day) |>
  summarize(
    n = n(),
    delay = mean(arr_delay, na.rm = TRUE)
  ) |>
  filter(n > 10)
```

```
flights |>
  filter(
    carrier == "UA",
    dest %in% c("IAH", "HOU"),
    sched_dep_time > 900,
    sched_arr_time < 2000
```

```
-     ) |>
-     group_by(flight) |>
-     summarize(
-       delay = mean(arr_delay, na.rm = TRUE),
-       cancelled = sum(is.na(arr_delay)),
-       n = n()
-     ) |>
-     filter(n > 10)
```

## Executables

#Chapter3  Data transformation

#3.1.1 Prerequisites

library(nycflights13)

library(tidyverse)

#3.1.2 nycflights13

flights

glimpse(flights)

```
#3.1.3 dplyr basics

flights |>
  filter(dest == "IAH") |>
  group_by(year, month, day) |>
  summarize(
    arr_delay = mean(arr_delay, na.rm = TRUE)
  )



#3.2 Rows



# 3.2.1 filter()



flights |>
  filter(dep_delay > 120)



#Flights that deoparted on January 1
```

```
flights |>
  filter(month == 1 & day == 1)


# Flights that departed in January or February
flights |>
  filter(month == 1 | month == 2)


# A shorter way to select flights that departed in January or February
flights |>
  filter(month %in% c(1, 2))


#Assign a filtered dataframe to a variable
jan1 <- flights |>
  filter(month == 1 & day == 1)


# write "or" statements like you would in English
```

```
flights |>
  filter(month == 1 | 2)
```

#3.2.3 arrange()

#changes the order of the rows based on the value of the columns

```
flights |>
  arrange(year, month, day, dep_time)
```

```
flights |>
  arrange(desc(dep_delay))
```

#3.2.4 distinct()

#all the unique rows in a dataset, so technically, it primarily operates on the rows

```
# Remove duplicate rows, if any

flights |>

  distinct()



# Find all unique origin and destination pairs

flights |>

  distinct(origin, dest)



#Keep other columns when filtering for unique rows

flights |>

  distinct(origin, dest, .keep_all = TRUE)




#the number of occurences instead, you're better off swapping distinct() for count()

flights |>

  count(origin, dest, sort = TRUE)
```

#3.3 Columns

#3.3.1 mutate()

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60
```

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .before = 1
  )
```

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .after = day
  )

flights |>
  mutate(
    gain = dep_delay - arr_delay,
    hours = air_time / 60,
    gain_per_hour = gain / hours,
    .keep = "used"
  )
```

#3.3.2 select()

```
flights |>
  select(year, month, day)

flights |>
  select(year:day)

flights |>
  select(where(is.character))

flights |>
  select(tail_num = tailnum)
```

#3.3.3 rename()

```
flights |>
```

```
  rename(tail_num = tailnum)
```

#3.3.4 relocate()

```
flights |>
  relocate(time_hour, air_time)

flights |>
  relocate(year:dep_time, .after = time_hour)
flights |>
  relocate(starts_with("arr"), .before = dep_time)
```

#3.4 The pipe

```
flights |>
  filter(dest == "IAH") |>
```

```
  mutate(speed = distance / air_time * 60) |>
  select(year:day, dep_time, carrier, flight, speed) |>
  arrange(desc(speed))



arrange(
  select(
    mutate(
      filter(
        flights,
        dest == "IAH"
      ),
      speed = distance / air_time * 60
    ),
    year:day, dep_time, carrier, flight, speed
  ),
  desc(speed)
```

```
)



flights1 <- filter(flights, dest == "IAH")

flights2 <- mutate(flights1, speed = distance / air_time * 60)

flights3 <- select(flights2, year:day, dep_time, carrier, flight, speed)

arrange(flights3, desc(speed))



#3.5 Groups



flights |>

  group_by(month)



#3.5.2 summarize()



flights |>

  group_by(month) |>
```

```
  summarize(
    avg_delay = mean(dep_delay)
  )


flights |>
  group_by(month) |>
  summarize(
    avg_delay = mean(dep_delay)
  )


flights |>
  group_by(month) |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = TRUE),
    n = n()
  )
```

#3.5.3 The slice_ functions

```
flights |>
  group_by(dest) |>
  slice_max(arr_delay, n = 1) |>
  relocate(dest)
```

#3.5.4 Grouping by multiple variables

```
daily <- flights |>
  group_by(year, month, day)
daily
```

```
daily_flights <- daily |>
  summarize(n = n())
```

```
daily_flights <- daily |>
  summarize(
    n = n(),
    .groups = "drop_last"
  )
```

#3.5.5 Ungrouping

```
daily |>
  ungroup()
```

```
daily |>
  ungroup() |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = TRUE),
```

```
    flights = n()
  )



#3.5.6 .by



flights |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE),
    n = n(),
    .by = month
  )



flights |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE),
    n = n(),
```

```
    .by = c(origin, dest)
  )
```

#3.6 Case study: aggregates and sample size

```
batters <- Lahman::Batting |>
  group_by(playerID) |>
  summarize(
    performance = sum(H, na.rm = TRUE) / sum(AB, na.rm = TRUE),
    n = sum(AB, na.rm = TRUE)
  )

batters

batters |>
  filter(n > 100) |>
```

```
  ggplot(aes(x = n, y = performance)) +
  geom_point(alpha = 1 / 10) +
  geom_smooth(se = FALSE)


batters |>
  arrange(desc(performance))


#Chapter 4  Workflow: code style


#4.1 Names


library(tidyverse)
library(nycflights13)


# Strive for:
short_flights <- flights |> filter(air_time < 60)
```

```
# Avoid:

SHORTFLIGHTS <- flights |> filter(air_time < 60)
```

#4.2 Spaces

```
# Strive for

z <- (a + b)^2 / d
```

```
# Avoid

z<-( a + b ) ^ 2/d
```

```
# Strive for

mean(x, na.rm = TRUE)
```

```
# Avoid
```

```
mean (x ,na.rm=TRUE)



flights |>
  mutate(
    speed      = distance / air_time,
    dep_hour   = dep_time %/% 100,
    dep_minute = dep_time %%  100
  )



#4.3 Pipes



# Strive for
flights |>
  filter(!is.na(arr_delay), !is.na(tailnum)) |>
  count(dest)
```

```
# Avoid

flights|>filter(!is.na(arr_delay), !is.na(tailnum))|>count(dest)



# Strive for

flights |>

  group_by(tailnum) |>

  summarize(

    delay = mean(arr_delay, na.rm = TRUE),

    n = n()

  )



# Avoid

flights |>

  group_by(

    tailnum

  ) |>
```

```
  summarize(delay = mean(arr_delay, na.rm = TRUE), n = n())



# Strive for

flights |>

  group_by(tailnum) |>

  summarize(

    delay = mean(arr_delay, na.rm = TRUE),

    n = n()

  )



# Avoid

flights|>

  group_by(tailnum) |>

  summarize(

    delay = mean(arr_delay, na.rm = TRUE),

    n = n()
```

```
  )



# Avoid

flights|>

  group_by(tailnum) |>

  summarize(

    delay = mean(arr_delay, na.rm = TRUE),

    n = n()

  )



# This fits compactly on one line

df |> mutate(y = x + 1)



# While this takes up 4x as many lines, it's easily extended to

# more variables and more steps in the future

df |>
```

```
  mutate(
    y = x + 1
  )
```

#4.4 ggplot2

```
flights |>
  group_by(month) |>
  summarize(
    delay = mean(arr_delay, na.rm = TRUE)
  ) |>
  ggplot(aes(x = month, y = delay)) +
  geom_point() +
  geom_line()
```

```
flights |>
```

```
  group_by(dest) |>
  summarize(
    distance = mean(distance),
    speed = mean(distance / air_time, na.rm = TRUE)
  ) |>
  ggplot(aes(x = distance, y = speed)) +
  geom_smooth(
    method = "loess",
    span = 0.5,
    se = FALSE,
    color = "white",
    linewidth = 4
  ) +
  geom_point()
```