Chapter 16 & 17 Executables & Exercises

16.3.1 Exercises

1. Explore the distribution of `rincome` (reported income). What makes the default bar chart hard to understand? How could you improve the plot?

- Problems with this chart: Labels overlap and are hard to read (especially if there are many categories).
- The levels are not ordered in a meaningful way (e.g., not by income range or
- frequency).
- Some responses are non-numeric, like “refused” or “Don’t know”, which add clutter.

Improvements by Reorder by frequency:

- gss_cat|>
- mutate(rincome = fct_infreq(rincome)) |>
- ggplot(aes(rincome)) +
- geom_bar() +
- coord_flip() +
- labs(title = “Distribution of Reported Income”, x = “Income”, y = “Count”)

2. What is the most common `relig` in this survey? What’s the most common `partyid`?

- gss_cat |>count(relig, sort = TRUE)
- #> Most common relig: “Protestant”

- gss_cat |> count(partyid, sort = TRUE)
- #> Most common partyid: “Independent”

3. Which `relig` does `denom` (denomination) apply to? How can you find out with a table? How can you find out with a visualization?

- Using a table:
- Gss_cat |>
- count(relig, denom) |>
- arrange(desc(n))

- Using a visualization:
- gss_cat |>

-   filter(!is.na(denom)) |>

- ggplot(aes(relig)) +
- geom_bar() +
- labs(title = "Religions with Denomination Data",
-     x = "Religion", y = "Count")

16.4.1 Exercises

1. There are some suspiciously high numbers in `tvhours`. Is the mean a good summary?

- When a variable like tvhours has suspiciously high values (outliers), the mean is not a good summary. The mean is sensitive to extreme values and can be pulled upward, giving a misleading picture of the typical amount of TV watched.
- Instead, use the median, which is more robust to outliers and gives a better sense of the "typical" value when the data is skewed or has unusual spikes.

2. For each factor in `gss_cat` identify whether the order of the levels is arbitrary or principled.

- Principled order: marital, rincome, partyid
- Arbitrary order: race, relig, denom
- Not factors: year, tvhours

3. Why did moving “Not applicable” to the front of the levels move it to the bottom of the plot?

- ggplot2 orders bars vertically from top to bottom based on factor levels.
- The first level appears at the bottom of a vertical bar chart when using coord_flip() or horizontal layout.

16.5.1 Exercises

1. How have the proportions of people identifying as Democrat, Republican, and Independent changed over time?

```
- gss_cat %>%
-   filter(partyid %in% c("Republican", "Independent", "Democrat")) %>%
-   count(year, partyid) %>%
-   group_by(year) %>%
-   mutate(prop = n / sum(n)) %>%
-   ggplot(aes(x = year, y = prop, color = partyid)) +
-   geom_line(size = 1.2) +
-   labs(
-     title = "Change in Party Identification Over Time",
-     x = "Year",
-     y = "Proportion",
-     color = "Party"
-   ) +
-   theme_minimal()
```

2. How could you collapse `rincome` into a small set of categories?

```
- gss_income_grouped <- gss_cat %>%
-   mutate(
-     rincome = fct_collapse(rincome,
-       "Not reported" = c("Refused", "Don't know", "No answer", "Not applicable"),
-       "Low"          = c("Lt $1000", "$1000 to 2999", "$3000 to 3999"),
-       "Middle"       = c("$4000 to 4999", "$5000 to 5999", "$6000 to 6999",
-                          "$7000 to 7999", "$8000 to 9999"),
-       "High"         = c("$10000 - 14999", "$15000 - 19999",
-                          "$20000 - 24999", "$25000 or more")
-     )
-   )

- # Check counts
- gss_income_grouped %>% count(rincome)
```

3. Notice there are 9 groups (excluding other) in the `fct_lump` example above. Why not 10? (Hint: type `?fct_lump`, and find the default for the argument `other_level` is "Other".)

- By default, fct_lump_n() keeps the top (n – 1) most frequent levels and lumps all remaining levels into a single "other" category. The default name for that catch-all is given by the other_level argument, which defaults to "Other".

17.2.5 Exercises

1. What happens if you parse a string that contains invalid dates?

- `ymd(c("2010-10-10", "bananas"))`
- ymd("2010-10-10") is a valid date string in year-month-day format, so it gets correctly parsed as 2010-10-10.
- ymd("bananas") is not a valid date format. As stated in Chapter 17, when ymd() encounters a string it cannot parse into a date, it returns NA and gives a warning, not an error.

2. What does the `tzone` argument to `today()` do? Why is it important?

- The tzone argument in today() sets the time zone for the returned date. It's important because the date can differ depending on where you are in the world—what's April 16 in New York might already be April 17 in Nepal. Using tzone ensures you get the correct date for your desired location.

3. For each of the following date-times, show how you'd parse it using a readr column specification and a lubridate function.

```
d1 <- "January 1, 2010"
```

- Use mdy(d1)
- Because the format is month-day-year with a full month name.

```
d2 <- "2015-Mar-07"
```

- Use ymd(d2)
- This matches year-month-day with abbreviated month.

```
d3 <- "06-Jun-2017"
```

- Use dmy(d3)
- This follows day-month-year format with abbreviated month.

```
d4 <- c("August 19 (2015)", "July 1 (2015)")
```

- Use mdy(d4)
- Even though there's extra formatting, mdy() can still parse it correctly.

```
d5 <- "12/30/14" # Dec 30, 2014
```

- Use mdy(d5)
- This is in month-day-year format with a 2-digit year.

```
t1 <- "1705"
```

- Use hm(t1)
- It represents hours and minutes (17:05).

```
t2 <- "11:15:10.12 PM"
```

- Use parse_time(t2, "%I:%M:%OS %p")
- This uses 12-hour time with seconds and fractional seconds.

17.3.4 Exercises

1. How does the distribution of flight times within a day change over the course of the year?

- The distribution of flight times changes throughout the year due to factors like weather, daylight, and travel demand. For example, summer may have more early flights, while winter sees more delays and later departures. You can explore this by extracting hours and months using lubridate, then plotting flight times by month.

2. Compare `dep_time`, `sched_dep_time` and `dep_delay`. Are they consistent? Explain your findings.

- Yes, dep_time, sched_dep_time, and dep_delay are mostly consistent. dep_delay equals the difference between dep_time and sched_dep_time in minutes. Small differences can happen due to rounding or missing values from canceled flights.

3. Compare `air_time` with the duration between the departure and arrival. Explain your findings. (Hint: consider the location of the airport.)

- air_time records the actual time spent flying, while the duration between dep_time and arr_time includes time on the ground, like taxiing or waiting. If you calculate the difference between arr_time and dep_time, it's usually longer than air_time. Time zones and airport locations can also affect this if not handled properly—especially with long flights crossing zones.

4. How does the average delay time change over the course of a day? Should you use `dep_time` or `sched_dep_time`? Why?

- The average delay time typically increases as the day progresses—early morning flights have the least delays, while afternoon and evening flights tend to be more delayed due to compounding delays throughout the day.
- You should use sched_dep_time, not dep_time, because it reflects the planned schedule, allowing you to compare delay patterns based on when flights were supposed to depart—not when they actually did.

5. On what day of the week should you leave if you want to minimise the chance of a delay?

- library(nycflights13)

- library(dplyr)
- library(lubridate)
-
- flights %>%
-   mutate(weekday = wday(time_hour, label = TRUE)) %>% # extract weekday
-   group_by(weekday) %>%
-   summarise(avg_dep_delay = mean(dep_delay, na.rm = TRUE)) %>%
-   arrange(avg_dep_delay)

6. What makes the distribution of `diamonds$carat` and `flights$sched_dep_time`similar?

- Both diamonds$carat and flights$sched_dep_time have right-skewed distributions with common "preferred" values:

● In diamonds$carat, many diamonds are cut just below popular weights (like 0.99 instead of 1.00) due to pricing psychology.

● In flights$sched_dep_time, many flights are scheduled at round times like 600, 900, 1200, etc., because humans prefer neat, predictable departure times.

7. Confirm our hypothesis that the early departures of flights in minutes 20-30 and 50-60 are caused by scheduled flights that leave early. Hint: create a binary variable that tells you whether or not a flight was delayed.

- library(nycflights13)
- library(dplyr)
-
- flights %>%

- mutate(
- sched_min = sched_dep_time %% 100, # extract minutes from sched_dep_time
- delayed = dep_delay > 0 # TRUE if flight was delayed
-) %>%
- filter(sched_min %in% c(20:30, 50:60)) %>%
- group_by(sched_min) %>%
- summarise(
- total_flights = n(),
- prop_delayed = mean(delayed, na.rm = TRUE) # proportion of flights that were delayed
-)

17.4.4 Exercises

1. Explain `days(!overnight)` and `days(overnight)` to someone who has just started learning R. What is the key fact you need to know?

- The days() function creates a time duration based on the value you give it. If overnight is a logical value (TRUE or FALSE), then days(overnight) returns either 1 day (for TRUE) or 0 days (for FALSE). This is useful for adjusting times—like adding a day to an arrival time if a flight is overnight. The key fact is that days() creates a duration, not a date.

2. Create a vector of dates giving the first day of every month in 2015. Create a vector of dates giving the first day of every month in the *current* year.

First day of every month in 2015:

- library(lubridate)

- first_days_2015 <- ymd("2015-01-01") + months(0:11)

First day of every month in the current year:

- current_year <- year(today())
- first_days_current <- ymd(paste0(current_year, "-01-01")) + months(0:11)

3. Write a function that given your birthday (as a date), returns how old you are in years.

```
-   library(lubridate)
-
-   calculate_age <- function(birthday) {
-     today_date <- today()
-     age <- year(today_date) - year(birthday)
-
-     # Adjust if birthday hasn't occurred yet this year
-     if (month(today_date) < month(birthday) ||
-         (month(today_date) == month(birthday) && day(today_date) < day(birthday))) {
-       age <- age - 1
-     }
-
-     return(age)
-   }
```

4. Why can't `(today() %--% (today() + years(1))) / months(1)` work?

- The expression (today() %--% (today() + years(1))) / months(1) doesn't work because months have different lengths—some are 28, 30, or 31 days—so R can't divide the interval evenly by months. As mentioned in the book, you can divide intervals by fixed durations (like days or seconds), but not by periods like months or years, which vary depending on the calendar.

Executables

#Chapter 16 & 17 Executables..........

# Chapter 16 Factors..............

# 16.1 Introduction.........

# 16.1.1 Prerequisites..........

library(tidyverse)

# 16.2 Factor basics...........

x1 <- c("Dec", "Apr", "Jan", "Mar")

#Using a string to record this variable has two problems:

#There are only twelve possible months, and there's nothing saving you from typos:
x2 <- c("Dec", "Apr", "Jam", "Mar")

#It doesn't sort in a useful way:

sort(x1)

```
month_levels <- c(
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
)

y1 <- factor(x1, levels = month_levels)
y1

sort(y1)

y2 <- factor(x2, levels = month_levels)
y2

y2 <- fct(x2, levels = month_levels)

factor(x1)

fct(x1)

levels(y2)

csv <- "
month,value
Jan,12
Feb,56
Mar,12"

df <- read_csv(csv, col_types = cols(month = col_factor(month_levels)))
df$month

# 16.3 General Social Survey............

gss_cat

gss_cat |>
  count(race)

# 16.4 Modifying factor order........

relig_summary <- gss_cat |>
```

```
  group_by(relig) |>
  summarize(
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )

ggplot(relig_summary, aes(x = tvhours, y = relig)) +
  geom_point()

ggplot(relig_summary, aes(x = tvhours, y = fct_reorder(relig, tvhours))) +
  geom_point()

relig_summary |>
  mutate(
    relig = fct_reorder(relig, tvhours)
  ) |>
  ggplot(aes(x = tvhours, y = relig)) +
  geom_point()

rincome_summary <- gss_cat |>
  group_by(rincome) |>
  summarize(
    age = mean(age, na.rm = TRUE),
    n = n()
  )

ggplot(rincome_summary, aes(x = age, y = fct_reorder(rincome, age))) +
  geom_point()

ggplot(rincome_summary, aes(x = age, y = fct_relevel(rincome, "Not applicable"))) +
  geom_point()

by_age <- gss_cat |>
  filter(!is.na(age)) |>
  count(age, marital) |>
  group_by(age) |>
  mutate(
    prop = n / sum(n)
  )
```

```
ggplot(by_age, aes(x = age, y = prop, color = marital)) +
  geom_line(linewidth = 1) +
  scale_color_brewer(palette = "Set1")

ggplot(by_age, aes(x = age, y = prop, color = fct_reorder2(marital, age, prop))) +
  geom_line(linewidth = 1) +
  scale_color_brewer(palette = "Set1") +
  labs(color = "marital")

gss_cat |>
  mutate(marital = marital |> fct_infreq() |> fct_rev()) |>
  ggplot(aes(x = marital)) +
  geom_bar()

# 16.5 Modifying factor levels...........

gss_cat |> count(partyid)

gss_cat |>
  mutate(
    partyid = fct_recode(partyid,
                      "Republican, strong"    = "Strong republican",
                      "Republican, weak"      = "Not str republican",
                      "Independent, near rep" = "Ind,near rep",
                      "Independent, near dem" = "Ind,near dem",
                      "Democrat, weak"        = "Not str democrat",
                      "Democrat, strong"      = "Strong democrat"
    )
  ) |>
  count(partyid)

gss_cat |>
  mutate(
    partyid = fct_recode(partyid,
                      "Republican, strong"    = "Strong republican",
                      "Republican, weak"      = "Not str republican",
                      "Independent, near rep" = "Ind,near rep",
                      "Independent, near dem" = "Ind,near dem",
                      "Democrat, weak"        = "Not str democrat",
                      "Democrat, strong"      = "Strong democrat",
                    "Other"              = "No answer",
                    "Other"              = "Don't know",
                    "Other"              = "Other party"
    )
  )

gss_cat |>
  mutate(
    partyid = fct_collapse(partyid,
                      "other" = c("No answer", "Don't know", "Other party"),
                      "rep" = c("Strong republican", "Not str republican"),
                      "ind" = c("Ind,near rep", "Independent", "Ind,near dem"),
                      "dem" = c("Not str democrat", "Strong democrat")
    )
  ) |>
  count(partyid)

gss_cat |>
  mutate(relig = fct_lump_lowfreq(relig)) |>
  count(relig)

gss_cat |>
  mutate(relig = fct_lump_n(relig, n = 10)) |>
  count(relig, sort = TRUE)

# 16.6 Ordered factors.........

ordered(c("a", "b", "c"))

# Chapter 17  Dates and times...............

# 17.1.1 Prerequisites.............

library(tidyverse)
library(nycflights13)

# 17.2 Creating date/times...........

today()
```

```
now()

# 17.2.1 During import...........

csv <- "
  date,datetime
  2022-01-02,2022-01-02 05:12
"
read_csv(csv)

csv <- "
  date
  01/02/15
"

read_csv(csv, col_types = cols(date = col_date("%m/%d/%y")))

read_csv(csv, col_types = cols(date = col_date("%d/%m/%y")))

read_csv(csv, col_types = cols(date = col_date("%y/%m/%d")))

# 17.2.2 From strings..............

ymd("2017-01-31")
#> [1] "2017-01-31"
mdy("January 31st, 2017")
#> [1] "2017-01-31"
dmy("31-Jan-2017")
#> [1] "2017-01-31"

ymd_hms("2017-01-31 20:11:59")
#> [1] "2017-01-31 20:11:59 UTC"
mdy_hm("01/31/2017 08:01")
#> [1] "2017-01-31 08:01:00 UTC"


ymd("2017-01-31", tz = "UTC")

# 17.2.3 From individual components..........
```

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

flights |>
  select(year, month, day, hour, minute) |>
  mutate(departure = make_datetime(year, month, day, hour, minute))

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"))

flights_dt

flights_dt |>
  ggplot(aes(x = dep_time)) +
  geom_freqpoly(binwidth = 86400) # 86400 seconds = 1 day

flights_dt |>
  filter(dep_time < ymd(20130102)) |>
  ggplot(aes(x = dep_time)) +
  geom_freqpoly(binwidth = 600) # 600 s = 10 minutes

# 17.2.4 From other types..........

as_datetime(today())
#> [1] "2025-04-14 UTC"
as_date(now())
#> [1] "2025-04-14"
```

```
as_datetime(60 * 60 * 10)
#> [1] "1970-01-01 10:00:00 UTC"
as_date(365 * 10 + 2)
#> [1] "1980-01-01"


# 17.3 Date-time components........

# 17.3 Date-time components..........

datetime <- ymd_hms("2026-07-08 12:34:56")

year(datetime)
#> [1] 2026
month(datetime)
#> [1] 7
mday(datetime)
#> [1] 8

yday(datetime)
#> [1] 189
wday(datetime)
#> [1] 4

month(datetime, label = TRUE)

wday(datetime, label = TRUE, abbr = FALSE)

flights_dt |>
  mutate(wday = wday(dep_time, label = TRUE)) |>
  ggplot(aes(x = wday)) +
  geom_bar()

flights_dt |>
  mutate(minute = minute(dep_time)) |>
  group_by(minute) |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = TRUE),
    n = n()
  ) |>
  ggplot(aes(x = minute, y = avg_delay)) +
  geom_line()

sched_dep <- flights_dt |>
  mutate(minute = minute(sched_dep_time)) |>
  group_by(minute) |>
  summarize(
    avg_delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )

ggplot(sched_dep, aes(x = minute, y = avg_delay)) +
  geom_line()

# 17.3.2 Rounding...............

flights_dt |>
  count(week = floor_date(dep_time, "week")) |>
  ggplot(aes(x = week, y = n)) +
  geom_line() +
  geom_point()

flights_dt |>
  mutate(dep_hour = dep_time - floor_date(dep_time, "day")) |>
  ggplot(aes(x = dep_hour)) +
  geom_freqpoly(binwidth = 60 * 30)

flights_dt |>
  mutate(dep_hour = hms::as_hms(dep_time - floor_date(dep_time, "day"))) |>
  ggplot(aes(x = dep_hour)) +
  geom_freqpoly(binwidth = 60 * 30)

# 17.3.3 Modifying components...........

(datetime <- ymd_hms("2026-07-08 12:34:56"))

year(datetime) <- 2030
datetime

month(datetime) <- 01
```

```
datetime

hour(datetime) <- hour(datetime) + 1
datetime

update(datetime, year = 2030, month = 2, mday = 2, hour = 2)
#> [1] "2030-02-02 02:34:56 UTC"

update(ymd("2023-02-01"), mday = 30)
#> [1] "2023-03-02"
update(ymd("2023-02-01"), hour = 400)
#> [1] "2023-02-17 16:00:00 UTC"

# 17.4 Time spans.........

# 17.4.1 Durations..........

# How old is Hadley?
h_age <- today() - ymd("1979-10-14")
h_age
#> Time difference of 16619 days

as.duration(h_age)

dseconds(15)
#> [1] "15s"
dminutes(10)
#> [1] "600s (~10 minutes)"
dhours(c(12, 24))
#> [1] "43200s (~12 hours)" "86400s (~1 days)"
ddays(0:5)
#> [1] "0s"                "86400s (~1 days)"  "172800s (~2 days)"
#> [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
dweeks(3)
#> [1] "1814400s (~3 weeks)"
dyears(1)
#> [1] "31557600s (~1 years)"

2 * dyears(1)
#> [1] "63115200s (~2 years)"
```

```
dyears(1) + dweeks(12) + dhours(15)
#> [1] "38869200s (~1.23 years)"

tomorrow <- today() + ddays(1)
last_year <- today() - dyears(1)

one_am <- ymd_hms("2026-03-08 01:00:00", tz = "America/New_York")

one_am
#> [1] "2026-03-08 01:00:00 EST"
one_am + ddays(1)
#> [1] "2026-03-09 02:00:00 EDT"

# 17.4.2 Periods.............

one_am
#> [1] "2026-03-08 01:00:00 EST"
one_am + days(1)
#> [1] "2026-03-09 01:00:00 EDT"

hours(c(12, 24))
#> [1] "12H 0M 0S" "24H 0M 0S"
days(7)
#> [1] "7d 0H 0M 0S"
months(1:6)
#> [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S"
#> [5] "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"

10 * (months(6) + days(1))
#> [1] "60m 10d 0H 0M 0S"
days(50) + hours(25) + minutes(2)
#> [1] "50d 25H 2M 0S"

# A leap year
ymd("2024-01-01") + dyears(1)
#> [1] "2024-12-31 06:00:00 UTC"
ymd("2024-01-01") + years(1)
#> [1] "2025-01-01"

# Daylight saving time
```

```
one_am + ddays(1)
#> [1] "2026-03-09 02:00:00 EDT"
one_am + days(1)
#> [1] "2026-03-09 01:00:00 EDT"

flights_dt |>
  filter(arr_time < dep_time)

flights_dt <- flights_dt |>
  mutate(
    overnight = arr_time < dep_time,
    arr_time = arr_time + days(overnight),
    sched_arr_time = sched_arr_time + days(overnight)
  )

flights_dt |>
  filter(arr_time < dep_time)

# 17.4.3 Intervals...............

years(1) / days(1)

y2023 <- ymd("2023-01-01") %--% ymd("2024-01-01")
y2024 <- ymd("2024-01-01") %--% ymd("2025-01-01")

y2023

y2024

y2023 / days(1)
#> [1] 365
y2024 / days(1)
#> [1] 366

# 17.5 Time zones..........

Sys.timezone()
#> [1] "UTC"

length(OlsonNames())
```

```
#> [1] 598
head(OlsonNames())

x1 <- ymd_hms("2024-06-01 12:00:00", tz = "America/New_York")
x1
#> [1] "2024-06-01 12:00:00 EDT"

x2 <- ymd_hms("2024-06-01 18:00:00", tz = "Europe/Copenhagen")
x2
#> [1] "2024-06-01 18:00:00 CEST"

x3 <- ymd_hms("2024-06-02 04:00:00", tz = "Pacific/Auckland")
x3
#> [1] "2024-06-02 04:00:00 NZST"

x1 - x2

x1 - x3

x4 <- c(x1, x2, x3)
x4

x4a <- with_tz(x4, tzone = "Australia/Lord_Howe")
x4a

x4a - x4

x4b <- force_tz(x4, tzone = "Australia/Lord_Howe")
x4b

x4b - x4
```