Practice R

Tutorial 04: Data manipulation

Edgar Treischl

5/7/23

Data manipulation with dplyr

Welcome to tutorial of the Practice R book (Treischl 2023). Practice R is a text book for the social sciences which provides several tutorials supporting students to learn R. Feel free to inspect the tutorials even if you are not familiar with the book, but keep in mind these tutorials are supposed to complement the Practice R book.

In Chapter 4 we used the `dplyr` package to manipulate data (Wickham et al. 2022). In addition, we will learn how to systemically manipulate categorical variables with the `forcats` package (Wickham 2022) in Chapter 5. Both packages help you to handle many common steps to manipulate data. This tutorial gives a `dplyr` recap and asks you to apply the introduced functions.

As the next output shows, we use the `gss2016` again to select variables, create a filter, generate new variables, and summarize the data. Ask R to provide a description of the data (`?data`) if you are not familiar with the `gss2016` data yet.

```r
# The setup of tutorial 4
library(dplyr)
library(PracticeR)
head(gss2016)[1:9]
```

```
#> # A tibble: 6 x 9
#>    year    id ballot     age childs sibs  degree race  sex
#>   <dbl> <dbl> <labell> <dbl>  <dbl> <lab> <fct>  <fct> <fct>
#> 1  2016     1 1           47      3 2     Bache~ White Male
#> 2  2016     2 2           61      0 3     High ~ White Male
#> 3  2016     3 3           72      2 3     Bache~ White Male
#> 4  2016     4 1           43      4 3     High ~ White Fema~
#> 5  2016     5 3           55      2 2     Gradu~ White Fema~
#> # i 1 more row
```

Select

Especially in case of large and cluttered data, we use `select()` to specify which variables we work with. For example, pick only one variable such as school `degree` from the `gss2016` data.

```
# Select a variable
select(gss2016, degree)
```

```
#> # A tibble: 2,867 x 1
#>   degree
#>   <fct>
#> 1 Bachelor
#> 2 High School
#> 3 Bachelor
#> 4 High School
#> 5 Graduate
#> # i 2,862 more rows
```

`Select` comes with handy functions and applies the same logic as `base` R. For example, select several columns by providing a start (e.g., `id`) and endpoint (e.g., `degree`).

```
# Select all variables from x to y
select(gss2016, id:degree) |> head()
```

```
#> # A tibble: 6 x 6
#>      id ballot       age childs sibs       degree
#>   <dbl> <labelled> <dbl>  <dbl> <labelled> <fct>
#> 1     1 1             47      3 2          Bachelor
#> 2     2 2             61      0 3          High School
#> 3     3 3             72      2 3          Bachelor
#> 4     4 1             43      4 3          High School
#> 5     5 3             55      2 2          Graduate
#> # i 1 more row
```

Maybe we need all columns except the variables shown in the last output. Ask for the opposite and insert parentheses and a minus signs to turn the selection around.

```
# Turn around the selection
select(gss2016, -(id:degree)) |> head()
```

```
#> # A tibble: 6 x 27
#>    year race  sex    region     income16 relig marital padeg
#>   <dbl> <fct> <fct>  <fct>      <fct>    <fct> <fct>   <fct>
#> 1  2016 White Male   New Engla~ $170000~ None  Married Grad~
#> 2  2016 White Male   New Engla~ $50000 ~ None  Never ~ Lt H~
#> 3  2016 White Male   New Engla~ $75000 ~ Cath~ Married High~
#> 4  2016 White Female New Engla~ $170000~ Cath~ Married <NA>
#> 5  2016 White Female New Engla~ $170000~ None  Married Bach~
#> # i 1 more row
#> # i 19 more variables: madeg <fct>, partyid <fct>,
#> #   polviews <fct>, happy <fct>, partners <fct>,
#> #   grass <fct>, zodiac <fct>, pres12 <labelled>,
#> #   wtssall <dbl>, income_rc <fct>, agegrp <fct>,
#> #   ageq <fct>, siblings <fct>, kids <fct>, religion <fct>,
#> #   bigregion <fct>, partners_rc <fct>, obama <dbl>, ...
```

The `gss2016` data does not contain variables with a running number nor other systematic variable names. However, `dplyr` helps to select such variables without much effort. Consider toy data with several measurements and running numbers to illustrate how we can select such variables efficiently.

```r
# A new df to illustrate
df <- tibble(
  measurement_1 = 1:3,
  x1 = 1:3,
  measurement_2 = 1:3,
  x2 = 1:3,
  x3 = 1:3,
  other_variables = 1
)
```

Suppose we measured a variables several times and all start with an identical name (e.g., `measurement_`). Select all variables which start (or end) with a certain string. Thus, insert the `starts_with()` function and select all `measurement` variables.

```r
# Select variables that start with a string
select(df, starts_with("measurement"))
```

```
#> # A tibble: 3 x 2
#>   measurement_1 measurement_2
#>           <int>         <int>
#> 1             1             1
```

```
#> 2             2             2
#> 3             3             3
```

Or pick variables with the running number. The `num_range` functions needs the name (`x`) and the running number.

```
# Select based on a running number
select(df, num_range("x", 1:3))
```

```
#> # A tibble: 3 x 3
#>      x1    x2    x3
#>   <int> <int> <int>
#> 1     1     1     1
#> 2     2     2     2
#> 3     3     3     3
```

The package offers more helpers to select variables than I can possibly outline. For example, `contains()` checks if a variable includes a certain word; `matches()` let us specify search patterns (regular expression, see Chapter 10); and we can also include other functions to select variables. For example, the `is.numeric` function checks if an input is numeric and we can combine it with `where()` to select columns only *where* the content is numeric.

```
# Insert a function to select variables
gss2016 |> select(where(is.numeric))
```

```
#> # A tibble: 2,867 x 10
#>    year    id ballot   age childs sibs  pres12 wtssall obama
#>   <dbl> <dbl> <labe> <dbl>  <dbl> <lab> <labe>   <dbl> <dbl>
#> 1  2016     1 1         47      3 2     3        0.957     0
#> 2  2016     2 2         61      0 3     1        0.478     1
#> 3  2016     3 3         72      2 3     2        0.957     0
#> 4  2016     4 1         43      4 3     2        1.91      0
#> 5  2016     5 3         55      2 2     1        1.44      1
#> # i 2,862 more rows
#> # i 1 more variable: income <dbl>
```

Next, we filter data but since all R outputs are large due to the `gss2016` data, let us first create a smaller subset to reduce the size of the output and the length of this document.

```r
# Select a smaller subset for the rest of this tutorial
gss2016 <- select(PracticeR::gss2016, year:sex, income)
```

Filter

Use `filter()` to subset the data. The `dplyr` filters the data and returns a new data frame depending on the specified conditions. Use one or several relational or logical operators to select observations. For example, suppose you want to analyze persons who have a bachelor's `degree` only.

```r
# Apply a filter
gss2016 |>
  filter(degree == "Bachelor") |>
  head()
```

```
#> # A tibble: 6 x 10
#>    year    id ballot     age childs sibs  degree race  sex  
#>   <dbl> <dbl> <labell> <dbl>  <dbl> <lab> <fct>  <fct> <fct>
#> 1  2016     1 1           47      3 2     Bache~ White Male 
#> 2  2016     3 3           72      2 3     Bache~ White Male 
#> 3  2016    37 2           59      2 2     Bache~ White Male 
#> 4  2016    38 1           43      2 6     Bache~ White Fema~
#> 5  2016    39 3           58      0 1     Bache~ White Fema~
#> # i 1 more row
#> # i 1 more variable: income <dbl>
```

Can you adjust the code so that two conditions have to be fulfilled simultaneously. For example, keep only observations from adults (18 years and older) with a bachelor's degree.

```r
# Combine several conditions
gss2016 |>
  filter(degree == "Bachelor" & age > 17) |>
  head()
```

```
#> # A tibble: 6 x 10
#>    year    id ballot     age childs sibs  degree race  sex  
#>   <dbl> <dbl> <labell> <dbl>  <dbl> <lab> <fct>  <fct> <fct>
#> 1  2016     1 1           47      3 2     Bache~ White Male 
#> 2  2016     3 3           72      2 3     Bache~ White Male 
#> 3  2016    37 2           59      2 2     Bache~ White Male 
```

```
#> 4  2016    38 1           43      2 6     Bache~ White Fema~
#> 5  2016    39 3           58      0 1     Bache~ White Fema~
#> # i 1 more row
#> # i 1 more variable: income <dbl>
```

As outlined, keep your `base` R skills in mind when selecting or filtering data. For example, keep all degrees but exclude persons who have a `Bachelor`.

```
# All degrees, but not! Bachelors
gss2016 |>
  filter(degree != "Bachelor") |>
  head()
```

```
#> # A tibble: 6 x 10
#>    year    id ballot     age childs sibs  degree race  sex
#>   <dbl> <dbl> <labell> <dbl>  <dbl> <lab> <fct>  <fct> <fct>
#> 1  2016     2 2           61      0 3     High ~ White Male
#> 2  2016     4 1           43      4 3     High ~ White Fema~
#> 3  2016     5 3           55      2 2     Gradu~ White Fema~
#> 4  2016     6 2           53      2 2     Junio~ White Fema~
#> 5  2016     7 1           50      2 2     High ~ White Male
#> # i 1 more row
#> # i 1 more variable: income <dbl>
```

Use the `operators()` function from the `PracticeR` package when you have trouble to remember how `logical` and `relational` operators are implemented. The function inserts and runs examples via the console.

```
PracticeR::operators("logical")
#   Logical Operators
# > x <- TRUE
# > y <- FALSE
# > #Elementwise logical AND
# > x & y == TRUE
# [1] FALSE
# > #Elementwise logical OR
# > x | y == TRUE
# [1] TRUE
# > #Elementwise OR
# > xor(x, y)
# [1] TRUE
```

```
# > #Logical NOT
# > !x
# [1] FALSE
# > #In operator
# > 1:3 %in% rep(1:2)
# [1]  TRUE  TRUE FALSE
```

Mutate

In Chapter 4 I outline several ways to generate new variables based on observed ones. For example, raw data often contains a person's year of birth but not their age. With `mutate()` we can extend the data frame and estimate such a variable. Unfortunately, the `gss2016` has an `age` variable, but the variable does only reveal their age when the survey was conducted. To recap how `mutate()` works, recreate their birth year and a recent age variable, say for the year 2023.

```
# Create birth_year and a recent (year: 2023) age variable
gss2016 |>
  select(id, year, age) |>
  mutate(
    birth_year = year - age,
    age_2023 = 2023 - birth_year
  ) |>
  head()
```

```
#> # A tibble: 6 x 5
#>      id  year   age birth_year age_2023
#>   <dbl> <dbl> <dbl>      <dbl>    <dbl>
#> 1     1  2016    47       1969       54
#> 2     2  2016    61       1955       68
#> 3     3  2016    72       1944       79
#> 4     4  2016    43       1973       50
#> 5     5  2016    55       1961       62
#> # i 1 more row
```

Keep in mind that you can use relational and logical operators, as well other functions (e.g., log, rankings, etc.) to generate new variables. For example, generate a logical variable that indicates whether a person was an adult (older than 17) in the year 2016. The `if_else()` function helps you with this job.

```
# In theory: if_else(condition, true, false, missing = NULL)
gss2016 |>
  select(id, year, age) |>
  mutate(adult = if_else(age > 17, TRUE, FALSE)) |>
  head()
```

```
#> # A tibble: 6 x 4
#>      id  year   age adult
#>   <dbl> <dbl> <dbl> <lgl>
#> 1     1  2016    47 TRUE
#> 2     2  2016    61 TRUE
#> 3     3  2016    72 TRUE
#> 4     4  2016    43 TRUE
#> 5     5  2016    55 TRUE
#> # i 1 more row
```

In terms of generating new variables, also keep the `case_when()` function in mind, which provides a very flexible approach. Suppose we need to identify parents with a academic background. Parents educational background has many levels or attributes in the `gss2016` data, which makes a first attempt harder to apply (and we learn more about factor variables in Chapter 5). For this reason I created a smaller toy data set and I started to prepare the code. Can you complete it? The variable `academic_parents` is supposed to identify persons with a high educational background (`education`) with one or more `kids`. All other conditions are set to `FALSE`.

```
# Data to illustrate
df <- data.frame(
  kids = c(0, 1, 3, 0, NA),
  educ = c("high", "low", "high", "low", NA)
)

# In theory: case_when(condition ~ value)
df |>
  mutate(academic_parents = case_when(
    kids >= 1 & educ == "high" ~ "TRUE",
    TRUE ~ "FALSE"
  ))
```

```
#>   kids educ academic_parents
#> 1    0 high            FALSE
#> 2    1  low            FALSE
```

```
#> 3     3 high              TRUE
#> 4     0  low             FALSE
#> 5    NA <NA>             FALSE
```

Summarize

The `summarize()` function collapses several columns into a single row. By the way, the `dplyr` package understands both, British (e.g., `summarise`) and American English (e.g. `summarize`) and it's up to you to decide which one you prefer.

Let's calculate the mean age of the survey participants. As outlined in Practice R, the variable has missing values which is why we need to drop them first. In Chapter 5 we will focus on this problem and we learn more about the consequences of such decisions. I already excluded missing values, can you `summarize()` the age?

```
# Exclude missing values but consider the consequences (see Chapter 5)
gss2016 <- gss2016 |>
  tidyr::drop_na(age, sex)

# Summarize age
gss2016 |> summarize(mean_age = mean(age))
```

```
#> # A tibble: 1 x 1
#>   mean_age
#>      <dbl>
#> 1     49.2
```

The `dplyr` package comes with several help functions to summarize data. For example, to count the number of observation per group (e.g., for `sex`), split the data by groups (`group_by`) and apply the `n()` function.

```
# County by (sex)
gss2016 |>
  group_by(sex) %>%
  summarize(count = n())
```

```
#> # A tibble: 2 x 2
#>   sex    count
#>   <fct>  <int>
#> 1 Male    1272
#> 2 Female  1585
```

Moreover, compare the groups by calculating the `median` age instead of the mean; add the standard deviation (`sd`); and count the number of distinct values (`n_distinct`) of the `degree` variable.

```
# Dplyr has more summary functions
gss2016 |>
  group_by(sex) |>
  summarise(
    median_age = median(age),
    sd_age = sd(age),
    distinct_degree = n_distinct(degree)
  )
```

```
#> # A tibble: 2 x 4
#>   sex    median_age sd_age distinct_degree
#>   <fct>       <dbl>  <dbl>           <int>
#> 1 Male           48   17.4               6
#> 2 Female         50   17.9               6
```

In the last examples we grouped the data and then collapsed it. The counterpart to `group` is `ungroup()` which we may add as a last step to disperse the data again. For example, we can estimate how old men or women are on average and add this information to the original data frame. Use `mutate()` instead of `summarise()` to see the logic behind `ungroup`.

```
# Mutate ungroups the data again
gss2016 |>
  select(id, sex, age) |>
  group_by(sex) |>
  mutate(count = round(mean(age), 2))
```

```
#> # A tibble: 2,857 x 4
#> # Groups:   sex [2]
#>      id sex      age count
#>   <dbl> <fct>  <dbl> <dbl>
#> 1     1 Male      47  48.3
#> 2     2 Male      61  48.3
#> 3     3 Male      72  48.3
#> 4     4 Female    43  49.8
#> 5     5 Female    55  49.8
#> # i 2,852 more rows
```

Arrange

Last but not least, keep the `arrange()` function in mind. It is easy to apply and I don't believe there is much to practice. However, it gives us the chance to repeat how `transmute()` and the `between()` function works.

Consider the steps to build a restricted age sample to examine adults only. Use `mutate` to create a logical variable (`age_filter`) that indicates if a person is between 18 and 65. Furthermore, explore the difference between `mutate()` and `transmute()` if you can't remember it.

```
# Create a restricted analysis sample
# between: x >= left & x <= right
gss2016 |>
  transmute(age,
    age_filter = between(age, 18, 65)
  )
```

```
#> # A tibble: 2,857 x 2
#>     age age_filter
#>   <dbl> <lgl>
#> 1    47 TRUE
#> 2    61 TRUE
#> 3    72 FALSE
#> 4    43 TRUE
#> 5    55 TRUE
#> # i 2,852 more rows
```

Next, we need a `filter()` to restrict the sample, but how can we know that code worked? We can inspect the entire data frame with `View`, but we can also use `arrange()` to inspect if the filter was correctly applied. Sort in ascending and descending (`desc`) order.

```
# Filter and arrange the data
gss2016 |>
  transmute(age,
    age_filter = between(age, 18, 65)
  ) |>
  filter(age_filter == "TRUE") |>
  arrange(desc(age)) |>
  head()
```

```
#> # A tibble: 6 x 2
#>     age age_filter
```

```
#>   <dbl> <lgl>
#> 1    65 TRUE
#> 2    65 TRUE
#> 3    65 TRUE
#> 4    65 TRUE
#> 5    65 TRUE
#> # i 1 more row
```

The `dplyr` package offers many functions to manipulate data and this tutorial only summarizes the main functions. Consider the cheat sheet and the package website for more information.

```r
# The dplyr website
PracticeR::show_link("dplyr", browse = FALSE)
#> [1] "https://dplyr.tidyverse.org/"
```

Keep in mind that data preparation steps may appear simple, but only as long as we are not supposed to prepare data on our own. In the latter case we will often need several attempts to come up with a solution that works. Thus, be patient with yourself when your first attempts will not work. Most of the time we all need more than one shot to come up with a workable solution. In addition, we will use the package one more time to combine data in Chapter 5 and other `dplyr` functions will appear through the Practice R book. Thus, there will be plenty of opportunities to apply and develop your `dplyr` skills.

There are often different approaches that lead to the same result. As the artwork by Jake Clark illustrates and the Practice R info box about *data manipulation approaches* underlines, the `subset()` function from `base` R does essentially the same as `dplyr::filter`. Base R provides the most stable solution, while `dplyr` is more verbose and often easier to learn. Don't perceive them as two different dialects that forces us to stick to one approach. Instead, embrace them both because you will come across different approaches if you use Google to solve a problem. Fortunately, many roads lead to Rome.

Summary

Keep the main `dplyr` functions in mind, among them:

- Keep rows that match a condition (`filter`)
- Order rows using column values (`arrange`)
- Keep or drop columns using their names and types (`select`)
- Create, modify, and delete columns (`mutate, transmute`)
- Summarize each group down to one row (`summarize`)
- Change column order (`relocate`)
- Vectorized if-else (`if_else`)

- A general vectorized if-else (`case_when`)
- Apply a function (or functions) across multiple columns (`across`)
- Select all variables or the last variable (e.g., `everything`)

And the following `base` functions:

- The names of an object (`names`)
- Sub-setting vectors, matrices and data frames (`subset`)
- Apply a function over a list or vector (`lapply, sapply`)
- Read R code from a file, a connection or expressions (`source`)

References

Treischl, Edgar J. 2023. *Practice R: An Interactive Textbook.* De Gruyter Oldenbourg.

Wickham, Hadley. 2022. *forcats: Tools for Working with Categorical Variables (Factors).* https://CRAN.R-project.org/package=forcats.

Wickham, Hadley, Romain François, Lionel Henry, and Kirill Müller. 2022. *dplyr: A Grammar of Data Manipulation.* https://CRAN.R-project.org/package=dplyr.