R: Data Transformation 3

Sakol Suethanapornkul

28 November 2020

Recap

So far, we have discussed the following functions:

```r
library(tidyverse)
data(diamonds)
```

```r
select(diamonds, starts_with("c"))
relocate(diamonds, cut, .after = price)
filter(diamonds, price > 327 | x >= 3.9)
mutate(diamonds, total = x * y * z)
```

EXERCISE: Describe what each function does. How do they differ from one another?

Recap

mutate() allows us to create new columns in an existing tibble. It can be extremely handy:

```
mpg <- mutate(mpg, source = "EPA")
mpg <- mutate(mpg, cty_km = cty * 1.61,
            hwy_km = hwy * 1.61)
mpg <- mutate(mpg, size = if_else(cyl < 6, "small",
                                  "big") )
```

mutate() works well with row_number()

```
x <- c(1, 3, 8, 2, 4, 6, 9)
row_number(x)

mpg <- mutate(mpg, no = row_number() )
```

Part I: Pipe %>%

EXERCISE: Complete the following steps with the `mpg` data frame:

[1] Create a subset of `mpg` that consists of `manufacturer`, `model`, `year`, `cyl`, `cty`, and `class`;

[2] Move the `year` column to the front;

[3] Add a new column that converts mpg in `cty` to kpg; and

[4] Drop the original `cty` column.

Part I: Pipe %>%

One possible way to carry out these four steps is:

```
mpg_small <- select(mpg, starts_with("m"),
                         year,
                         starts_with("c")
                   )
mpg_small <- relocate(mpg_small, year)
mpg_small <- mutate(mpg_small, cty_k = cty * 1.61)
mpg_small <- select(mpg_small, !cty)
```

But note that we modify `mpg_small` several times. What could possibly go wrong with that?

Part I: Pipe %>%

You may choose to pursue another option, creating a new variable at each intermediate step:

```r
mpg_small <- select(mpg, starts_with("m"),
                         year,
                         starts_with("c")
                    )
mpg_small2 <- relocate(mpg_small, year)
mpg_small3 <- mutate(mpg_small2, cty_k = cty * 1.61)
mpg_small4 <- select(mpg_small3, !cty)
```

But this approach isn't ideal either. Why?

Part I: Pipe %>%

More often that not, we often have to wrangle an existing data frame in order to answer certain questions. What we have done in the previous two slides is fine but not perfect. . .

Part I: Pipe %>%

We'd like to combine these incremental steps into one big chunk of codes that shows the **transformations**, rather than *what is getting transformed*.

Moreover, we want to be able to show the pipeline of our code.

We are going to do that with %>% (meaning "and then") from the package `magrittr` which `tidyverse` has access to.

```r
library(magrittr)
```

Part I: Pipe %>%

We start with a data frame and pipe it into one function after another:

NOTE a shortcut for %>% is **command+shift+m**

```
mpg %>%
```

Part I: Pipe %>%

We start with a data frame and pipe it into one function after another:

NOTE a shortcut for %>% is **command+shift+m**

```
mpg %>%
  select(starts_with("m"), year, starts_with("c")) %>%
```

Part I: Pipe %>%

We start with a data frame and pipe it into one function after another:

NOTE a shortcut for %>% is **command+shift+m**

```r
mpg %>%
  select(starts_with("m"), year, starts_with("c")) %>%
  relocate(year) %>%
```

Part I: Pipe %>%

We start with a data frame and pipe it into one function after another:

NOTE a shortcut for %>% is **command+shift+m**

```
mpg %>%
  select(starts_with("m"), year, starts_with("c")) %>%
  relocate(year) %>%
  mutate(cty_k = cty * 1.61) %>%
```

Part I: Pipe %>%

We start with a data frame and pipe it into one function after another:

NOTE a shortcut for %>% is **command+shift+m**

```
mpg %>%
  select(starts_with("m"), year, starts_with("c")) %>%
  relocate(year) %>%
  mutate(cty_k = cty * 1.61) %>%
  select(!cty)
```

Part I: Pipe %>%

And if you want to save this into a new data frame, it is really easy:

```r
mpg_small <- mpg %>%
  select(starts_with("m"), year, starts_with("c")) %>%
  relocate(year) %>%
  mutate(cty_k = cty * 1.61) %>%
  select(!cty)
```

Part I: Pipe %>%

Given the following code:

```
mpg %>%
  select(starts_with("c"), year) %>%
  select(!cty)
```

this is what R does behind the scenes:

```
select(select(mpg, starts_with("c"), year), !cty )
```

Part I: Pipe %>%

EXERCISE:

[1] Describe what the following code does.

```
mpg_small <- mpg %>%
  select(starts_with("m"), where(is.numeric)) %>%
  mutate(hwy_dif = max(hwy) - hwy)
```

[2] What is wrong with this line of codes?

```
mpg_small <- mpg %>%
  select(starts_with("m")) %>%
  select(!cty)
```

Part I: Pipe %>%

[3] For this question, we will use Pew Research Center's American Trend Panel Wave survey. Copy and paste the code below to load the data set into R.

```r
library(haven)

#read spss file with read_sav()
#make snake cases with clean_names()

wtp59 <- read_sav("ATPW59.sav") %>%
  zap_label() %>%
  zap_labels() %>%
  janitor::clean_names()
```

Part I: Pipe %>%

Then complete the following steps with one block of codes:

1. Create a new data frame with just the questions from the W59 survey (which end with `_w59`);
2. Drop questions about Youtube (which begin with `yt`) from the data frame;
3. Move the last two columns on weight (`weight`) to third & fourth columns before `device_type_w59`; and
4. Drop two questions about a presidential candidate's quality (`demdealfght_w59` and `repdealfght_w59`).

Part I: Pipe %>%

[4] Create a new data frame from `wtp59` as follows:

1. Change values in `device_type_w59` column to categories [1 = laptop; 2 = smartphone; 3 = tablet; 4 = TV; 5 = NOTA];
2. Convert values in `lang_w59` column to categories [1 = English; 2 = Spanish];
3. Obtain the time each respondent spent completing the survey **Hint**: You'll need to subtract one column from the other;
4. Put these three newly created columns right after their original columns;
5. Select only respondents who completed the survey on their smartphones or tablets.

Part I: Pipe %>%

ANSWER: Let's look at [3]

```
wtp59_small <- wtp59 %>%
  select(ends_with("_w59")) %>%
  select(!starts_with("yt")) %>%
  relocate(starts_with("weight"),
           .before = device_type_w59) %>%
  select(!ends_with("dealfght"))
```

If this isn't yet obvious, we can string the same "verbs" several times.

Part I: Pipe %>%

ANSWER: Let’s look at [4]

Part II: summarize()

While mutate() creates new columns by adding “things” to each row of a tibble, summarize() collapses a column into a single row:

```
mpg %>%
  summarize(ave_cty = mean(cty),
            ave_hwy = mean(hwy)
            )
```

EXERCISE: What do we get in ave_cty and ave_hwy?

Part II: summarize()

summarize() operates on the whole column. If you'd like to get a summary by groups/conditions/units, you can pair summarize() with group_by():

```r
mpg %>%
  group_by(manufacturer) %>%
  summarize(ave_cty = mean(cty),
            ave_hwy = mean(hwy)
            )
```

You can group by multiple variables to roll up your data set.

Part II: `summarize()`

EXERCISE:

[1] find the average and standard deviation of `cty` by manufacturer and class;

[2] find the maximum value of `hwy` by year, manufacturer, and cycle;

[3] count the number of cars by manufacturer and class

Part II: summarize()

ANSWER:

```
mpg %>%
  group_by(manufacturer, class) %>%
  summarize(M = mean(cty), SD = sd(cty))

mpg %>%
  group_by(year, manufacturer, class) %>%
  summarize(Mx = max(hwy))

mpg %>%
  group_by(manufacturer, class) %>%
  summarize(num = n())
```

Part II: `summarize()`

You may have noticed `n()` in answer [3]. There are a number of useful count functions that work well with `summarize()`:

```
mpg %>%
  group_by(manufacturer, class) %>%
  summarize(num = n())

#or a shortcut
mpg %>%
  count(manufacturer, class)
```

Part II: `summarize()`

To count non-missing values, you can use `sum(!is.na(x))`. Conversely, use `sum(is.na(x))` to count missing values. For example,

```
mpg %>%
  group_by(manufacturer) %>%
  summarize(num = sum(!is.na(class)) )
```

How does this work?

Part II: summarize()

Recall:

```
nums <- c(1, 5, NA, 7, 9, NA, 12, NA)
!is.na(nums)
```

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

When you embed this comparison operation inside `sum()`, you essentially sum up how many `TRUE` there are in a vector. Logical `TRUE` is converted to 1.

```
sum(!is.na(nums))
```

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

Part II: summarize()

You can apply this to various situations:

```
mpg %>%
  group_by(manufacturer) %>%
  summarize(cty_over15 = sum(cty > 15) )

mpg %>%
  group_by(manufacturer) %>%
  summarize(m_cty    = mean(cty),
            m_cty_15 = mean(cty > 15) )
```

There are many other useful summary functions: `sd()`, `median()`, `n_distinct()`.

Part II: summarize()

summarize() is now even better as it can unpack multiple values from some functions like range() and quantile():

```r
nums <- 1:20
range(nums)
```

```
## [1]  1 20
```

Part II: summarize()

This codes work rather effortlessly:

```
mpg %>% group_by(manufacturer) %>%
  summarize(value = c("min", "max"),
            rng = range(cty)
            )
```

Instead of:

```
mpg %>% group_by(manufacturer) %>%
  summarize(low  = range(cty)[[1]],
            high  = range(cty)[[2]]
            )
```

Part II: `mutate()` and `summarize()`

It's useful to perform the same operation on multiple columns. But we may need a lot of typing to find averages of different flower species:

```
data(iris)

iris %>%
  group_by(Species) %>%
  summarize(M1 = mean(Sepal.Length),
            M2 = mean(Sepal.Width),
            M3 = mean(Petal.Length),
            M4 = mean(Petal.Width)
            )
```

Part II: mutate() and summarize()

across() is designed to address this redundancy. It uses the same syntax as select():

```
iris %>%
  group_by(Species) %>%
  summarize(across(starts_with("Sepal"), mean
                   )
            )

iris %>%
  group_by(Species) %>%
  summarize(across(Sepal.Length:Sepal.Width, mean
                   )
            )
```

Part II: `mutate()` and `summarize()`

Recall that we can select columns by their type, with `where(is.numeric)`

```r
iris %>%
  group_by(Species) %>%
  summarize(across(where(is.numeric), mean
                   )
            )
```

As you may have already noticed, inside `across()` we need two things.

```r
across(.cols = , .fns = )
```

Part II: mutate() and summarize()

Inside .fns = you can use a formula-like command:

```
iris %>%
  group_by(Species) %>%
  summarize(across(.cols = where(is.numeric),
                   .fns  = ~ mean(.x, na.rm = TRUE)
                   )
            )
```

Part II: `mutate()` and `summarize()`

This means you can write out what you want to compute inside `.fns`. For example, we want to `mutate` new columns where we center values:

```r
iris %>%
  group_by(Species) %>%
  mutate(across(.cols = where(is.numeric),
                .fns  = ~ .x - mean(.x)
                )
            )
```

Wrap-up

We have seen key verbs in `dplyr`:

```r
select(dat, col1, col2)
relocate(dat, col2, .before = col1)

filter(dat, col1 >= 10)
arrange(dat, desc(col1))

mutate(dat, col3 = col1 * col2)
summarize(dat, col3 = mean(col1))
```

Wrap-up

We have learned to string these verbs together:

```r
dat %>%
  select(col1:col5) %>%
  filter(!is.na(col5) & col4 > 10) %>%
  mutate(col6 = col2 ^ col3) %>%
  group_by(col1) %>%
  summarize(m  = mean(col6),
            sd = sd(col6))
```

Wrap-up

There are of course plenty other “verbs” we do not have time to cover:

```r
mpg %>% slice(n = 1)
mpg %>% group_by(manufacturer) %>% slice(n = 1)
mpg %>% slice_head(n = 6)
```

```r
mpg %>%
  rename(company = manufacturer) #new_name = old_name
```

```r
mpg %>% distinct(cyl)
mpg %>% group_by(manufacturer) %>% distinct(cyl)
```

Show & Tell

It's your turn to show me what you can do!