Programando con dplyr

Fuente: vignettes/programming.Rmd*

1. Introducción

Muchos verbos de dplyr usan evaluación ordenada (tidy evaluation) en algún sentido. La evaluación ordenada es un tipo especial de evaluación no estándar (NSE[1]) usada a lo largo del tidy-verso. Hay dos formas básicas que se encuentran en dplyr:

- `arrange()`, `count()`, `filter()`, `group_by()`, `mutate()` y `summarise()` usan **enmascaramiento de datos** (data masking) por lo que se puede usar variables de conjuntos de datos como si fueran variables de entorno (por ejemplo se escribiría `my_variable` en vez de `df$my_variable`).

- `across()`, `relocate()`, `rename()`, `select()` y `pull()` usan **selección ordenada** (tidy selection) por lo que se pueden escoger variables fácilmente en base a su posición, nombre o tipo (por ejemplo, `starts_with("x")` o `is.numeric`).

Para determinar cuando el argumento de una función usa enmascaramiento de datos o selección ordenada, se debe revisar la documentación: en la lista de argumentos, se verá `<data-masking>` o `<tidy-select>`.

El enmascaramiento de datos y la selección ordenada hacen que la exploración interactiva de datos sea rápida y fluida, pero también agregan algunos nuevos retos cuando pretendemos usarlos indirectamente como por ejemplo en un lazo for o en una función. Esta viñeta muestra como superar esos desafíos. En primero lugar, cubriremos las nociones básicas de enmascaramiento de datos y selección ordenada, hablaremos de como usarlos de manera indirecta y luego mostraremos una serie de consejos para resolver problemas comunes.

Esta viñeta pondrá a tu disposición el mínimo conocimiento necesario para que te conviertas en un programador de evaluación ordenara eficaz. Si tu deseo es conocer más sobre la teoría subyacente, o de manera precisa sus diferencias con la NSE, recomendamos leer los capítulos de Metaprogramming in *Advanced R*.

2. Enmascaramiento de datos

El enmascaramiento de datos agiliza la manipulación de datos puesto que requiere teclear menos. En muchas (pero no en todas) las funciones de R base se necesita referirnos a las variables con $, conduciendo a un código que repite muchas veces el nombre del conjunto de datos:

```
starwars[starwars$homeworld == "Naboo" & starwars$species == "Human", ]
```

El equivalente en dplyr de este código es más conciso puesto que el enmascaramiento de datos nos permite escribir `starwars` una sola vez:

```
starwars %>% filter(homeworld == "Naboo", species == "Human")
```

*Traducido de aquí.

[1] por sus siglas en inglés (Non-standard evaluation).

2.1. Variables de conjunto de datos y de entorno

La idea principal detrás del enmascaramiento de datos es que este difumina la línea entre los dos diferentes significados de la palabra "variable":

- **variable de entorno** (env-variable): son variables de "programación" que habitan en el entorno. Usualmente se crean con `<-`.
- **variable de conjunto de datos** (data-variable): son variables "estadísticas" que están en los conjuntos de datos. Usualmente provienen de archivos de datos (por ejemplo, `.csv`, `.xlsx`), o son creadas mediante la manipulación de variables existentes.

Para entender un poco mejor estas definiciones, veamos el siguiente código:

```
> df <- data.frame(x=runif(3), y=runif(3))
> df$x
> [1] 0.08075014 0.83433304 0.6007608
```

En el, se crea una variable de entorno `df`, que contiene dos variables de conjunto de datos, x y y. Luego, extrae la variable de conjunto de datos x desde a variable de entorno df usando `$`.

Yo considero que difuminar el significado de "variable" es una característica muy buena para la exploración interactiva de datos porque nos permite fererirnos a las variables de conjunto de datos como son, sin ningún prefijo. Además, esto parece ser bastante intuitivo puesto que mucho usuarios nuevos de R intentarán escribir $diamonds[x == 0 | y == 0,]$.

Desafortunadamente, este beneficio no es gratis. Cuando se empieza a programar con estas herramientas, se tiene que lidiar con la diferenciación. Esto será difícil puesto que nunca antes teníamos que pensar en ello, así que tomará un tiempo a que tu cerebro aprenda estos nuevos conceptos y categorías. Sin embargo, una vez que diferenciemos la idea de "variable" entre de entorno y de conjunto de datos, será bastante sencillo de usarlas.

2.2. Uso indirecto

El principal reto de programar con funciones que usan enmascaramiento de datos surge cuando introducimos cierta indirección, es decir, cuando queremos obtener la variable de conjunto de datos desde una variable de entorno en vez de escribir directamente el nombre de esta. Hay dos casos principales:

- Cuando la variable de conjunto de datos está en el argumento de una función (es decir, una variable de entorno que guarda una promesa[2]), se necesita escribir el argumento entre doble llaves, como $filter(df, \{\{\ var\ \}\})$.

  La siguiente función usa las doble llaves para crear una envoltura alrededor del $summarise()$ que calcula el valor mínimo y máximo de una variable, al igual que el número de observaciones que fueron agregadas:

```
var_summary <- function(data, var) {
data %>%
  summarise(n = n(), min = min({{ var }}), max = max({{ var }}))
}
mtcars %>%
  group_by(cyl) %>%
  var_summary(mpg)
```

- Cuando tenemos una variable de entorno que es un vector caracter, se necesita indexar al pronombre $.data$ con $[[\cdot]]$, como $summarise(df,\ mean = mean(.data[[var]]))$.

  El siguiente ejemplo usa $.data$ para contar el número de valores únicos en cada variable de $mtcars$:

[2] En R, los argumentos de las funciones son evaluados vagamente, esto significa que hasta que no se intenten usar, no tienen ningún valor asociado, solo una **promesa** que describe como calcular el valor. Se puede aprender más aquí.

```
for (var in names(mtcars)) {
  mtcars %% count(.data[[var]]) %% print()
}
```

Nótese que *.data* no es un data.frame; es un constructo especial, un pronombre, que nos permite acceder directamente a las variables del conjunto de datos con *.data$x*, o indirectamente con *.data[[var]]*. No espere que esto funcione en el ámbito de otras funciones.

3. Selección ordenada

El enmascaramiento de datos hace más fácil el cálculo de valores dentro de un conjunto de datos. En cambio, la selección ordenada es una herramienta complementaria que facilita trabajar con las columnas de un conjunto de datos.

3.1. El Lenguaje específico de dominio de tidyselect

El paquete tidyselect está debajo de todas las funciones que usan selección ordenada. Este provee un lenguaje específico de dominio (DSL por sus siglas en inglés[3]) que facilita seleccionar columnas por nombre, posición o tipo. Por ejemplo:

- *select(df, 1)* selecciona la primera columna *select(df, last_col())* selecciona la última columna.
- *select(df, c(a, b, c))* selecciona las columnas *a*, *b* y *c*.
- *select(df, starts_with("a"))* selecciona todas las columnas cuyos nombres empiezan con "a"; *select(df, ends_with("z"))* selecciona todas las columnas cuyos nombres terminan con "z".

Para más detalles dirigirse a ?dplyr_tidy_select.

3.2. Uso indirecto

Como con el enmascaramiento de datos, la selección ordenada simplifica una tarea común a costa de complicar una tarea no tan común. Cuando se quiere usar la selección ordenada indirectamente, con la especificación de las columnas guardada en una variable intermedia, necesitaremos aprender algunas nuevas herramientas. Nuevamente, existen dos formas de uso indirecto:

- Cuando tenemos la variable de conjunto de datos almacenada en una variable de entorno que es el argumento de una función, se usa la misma técnica que en el enmascaramiento de datos, escribimos el argumento entre doble llaves.

  La siguiente función agrega un data.frame calculando la media de todas las variables seleccionadas por el usuario

```
summarise_mean <- function(data, vars) {
  data %>% summarise(n = n(), across({{ vars }}, mean))
}
mtcars %>%
  group_by(cyl) %>%
  summarise_mean(where(is.numeric))
```

- Cuando se dispone de un vector caracter como variable de entorno, es necesario usar *all_of()* o *any_of()* dependiendo si se desea que la función falle (muestre un error?) si una variable no es encontrada.

  El siguiente código usa *all_of()* para seleccionar todas las variables encontradas en un vector caracter luego, *!all_of* para seleccionar todas las variables que no están en un vector caracter:

[3] Domain specific language.

```r
    vars <- c("mpg", "vs")
    mtcars %>% select(all_of(vars))
    mtcars %>% select(!all_of(vars))
```

4. ¿ Y cómo...?

Los siguientes ejemplos resuelven un puñado de problemas comunes. Se muestra el mínimo de código necesario para captar las ideas básicas; la mayoría de los problemas reales requerirán más código o combinar diferentes técnicas.

4.1. Conjunto de datos proporcionado por el usuario

Si revisas la documentación, verás que *.data* nunca usa enmascaramiento de datos o selección ordenada. Eso significa que no es necesario hacer algo especial en tu función

```r
mutate_y <- function(data) {
  mutate(data, y = a + x)
}
```

4.2. Eliminando R CMD check NOTE s

Si estás escribiendo un paquete y tienes una función que usa variables de un conjunto de datos:

```r
my_summary_function <- function(data) {
  data %>%
    filter(x > 0) %>%
    group_by(grp) %>%
    summarise(y = mean(y), n = n())
}
```

Recibirás una R CMD check NOTE :

```
N checking R code for possible problems
  my_summary_function: no visible binding for global variable "x", "grp", "y"
  Undefined global functions or variables:
  x grp y
}
```

Puedes eliminar esta nota usando *.data$var* importando *.data* de su origen en el paquete rlang (el paquete subyacente que implementa evaluación ordenada):

```r
#' @importFrom rlang .data
my_summary_function <- function(data) {
  data %>%
    filter(.data$x > 0) %>%
    group_by(.data$grp) %>%
    summarise(y = mean(.data$y), n = n())
}
```

4.3. Una o más expresiones proporcionadas por el usuario

Si se desea que el usuario proporcione un expresión que pase a un argumento que utilice enmascaramiento de datos o selección ordenada, coloque el argumento entre doble llaves:

```r
my_summarise <- function(data, group_var) {
  data %>%
    group_by({{ group_var }}) %>%
    summarise(mean = mean(mass))
}
```

Esto se generaliza de una manera sencilla si se quiere usar una expresión proporcionada por el usuario en múltiples lugares:

```r
my_summarise2 <- function(data, expr) {
  data %>%
    summarise(mean = mean({{ expr }}),
              sum = sum({{ expr }}),
              n = n())
}
```

Si se desea que el usuario proporcione múltiples expresiones, encierre entre doble llaves cada una de ellas:

```r
my_summarise3 <- function(data, mean_var, sd_var) {
  data %>%
    summarise(mean = mean({{ mean_var }}),
              sd = sd({{ sd_var }}))
}
```

Si se desea utilizar el nombre de las variables en el resultado, se puede usar sintaxis de glue en conjunto con `:=`:

```r
my_summarise4 <- function(data, expr) {
  data %>%
  summarise("mean_{{expr}}" := mean({{ expr }}),
            "sum_{{expr}}" := sum({{ expr }}),
            "n_{{expr}}" := n())
}

my_summarise5 <- function(data, mean_var, sd_var) {
  data %>%
    summarise("mean_{{mean_var}}" := mean({{ mean_var }}),
              "sd_{{sd_var}}" := sd({{ sd_var }}))
}
```

4.4. Cualquier número de expresiones proporcionadas por el usuario

Si se quiere tomar un número arbitrario de expresiones proporcionadas por el usuario, usamos `...`. Esto suele ser útil cuando se le quiere dar control total de una parte específica de la programación en tubo al usuario, como un *group_by()* o un *mutate()*.

```r
my_summarise <- function(.data, ...) {
  .data %>%
    group_by(...) %>%
    summarise(mass = mean(mass, na.rm = TRUE),
              height = mean(height, na.rm = TRUE))
}

starwars %>% my_summarise(homeworld)
> # A tibble: 49 x 3
>   homeworld    mass height
>   <chr>       <dbl>  <dbl>
> 1 Alderaan       64   176.
> 2 Aleen Minor    15    79
> 3 Bespin         79   175
> 4 Bestine IV    110   180
> # ... with 45 more rows

starwars %>% my_summarise(sex, gender)
```

```
> `summarise()` has grouped output by 'sex'. You can override using the `.groups`
    argument.
> # A tibble: 6 x 4
> # Groups:   sex [5]
>   sex          gender       mass height
>   <chr>        <chr>       <dbl>  <dbl>
> 1 female       feminine     54.7   169.
> 2 hermaphrodit masculine 1358     175
> 3 male         masculine    81.0   179.
> 4 none         feminine    NaN      96
> # ... with 2 more rows
```

Cuando se usa `...` de esta manera, asegúrese que los otros argumentos empiecen con `.` para reducir la probabilidad de conflictos entre argumentos; para más detalles, visite esta página.

4.5. Transformando variables proporcionadas por el usuario

Use *across()* si se busca que el usuario provea un conjunto de variables de conjunto de datos que van a ser transformadas:

```
my_summarise <- function(data, summary_vars) {
  data %>%
    summarise(across({{ summary_vars }}, ~ mean(., na.rm = TRUE)))
}

starwars %>%
  group_by(species) %>%
  my_summarise(c(mass, height))
> # A tibble: 38 x 3
>   species    mass height
>   <chr>     <dbl>  <dbl>
> 1 Aleena       15     79
> 2 Besalisk    102    198
> 3 Cerean       82    198
> 4 Chagrian    NaN    196
> # ... with 34 more rows
```

Se puede usar la misma idea para múltiples conjuntos de variables de datos:

```
my_summarise <- function(data, group_var, summarise_var) {
  data %>%
    group_by(across({{ group_var }})) %>%
    summarise(across({{ summarise_var }}, mean))
}
```

Use el argumento *.names* de la función *across()* para controlar los nombres del resultado.

```
my_summarise <- function(data, group_var, summarise_var) {
  data %>%
    group_by(across({{ group_var }})) %>%
    summarise(across({{ summarise_var }}, mean, .names = "mean_{.col}"))
}
```

4.6. Bucle sobre múltiples variables

Si se tiene un vector caracter de nombres de variables, y se desea operar en ellas con un lazo for, indexamos en el pronombre especial *.data*:

```
for (var in names(mtcars)) {
  mtcars %>%
    count(.data[[var]]) %>%
    print()
}
```

La misma técnica funciona con las alternativas al lazo for, como la familia de funciones *apply()* de R base, o la familia de funciones *map()* del paquete purrr.

```
mtcars %>%
  names() %>%
  purrr::map(~ count(mtcars, .data[[.x]]))
```

4.7. Usar una variable de una entrada de Shiny

Muchas entradas de control de Shiny devuelven vectores caracter, así que se puede usar la misma aproximación anterior *.data*[[*input*$*var*]] :

```
library(shiny)
ui <- fluidPage(
  selectInput("var", "Variable", choices = names(diamonds)),
  tableOutput("output")
)
server <- function(input, output, session) {
  data <- reactive(filter(diamonds, .data[[input$var]] > 0))
  output$output <- renderTable(head(data()))
}
```

Visite el siguiente link para más detalles y casos de estudio.