Programming with dplyr

Introduction

Most dplyr verbs use **tidy evaluation** in some way. Tidy evaluation is a special type of non-standard evaluation used throughout the tidyverse. There are two basic forms found in dplyr:

- `arrange()`, `count()`, `filter()`, `group_by()`, `mutate()`, and `summarise()` use **data masking** so that you can use data variables as if they were variables in the environment (i.e. you write `my_variable` not `df$my_variable`).
- `across()`, `relocate()`, `rename()`, `select()`, and `pull()` use **tidy selection** so you can easily choose variables based on their position, name, or type (e.g. `starts_with("x")` or `is.numeric`).

To determine whether a function argument uses data masking or tidy selection, look at the documentation: in the arguments list, you'll see `<data-masking>` or `<tidy-select>`.

Data masking and tidy selection make interactive data exploration fast and fluid, but they add some new challenges when you attempt to use them indirectly such as in a for loop or a function. This vignette shows you how to overcome those challenges. We'll first go over the basics of data masking and tidy selection, talk about how to use them indirectly, and then show you a number of recipes to solve common problems.

This vignette will give you the minimum knowledge you need to be an effective programmer with tidy evaluation. If you'd like to learn more about the underlying theory, or precisely how it's different from non-standard evaluation, we recommend that you read the Metaprogramming chapters in *Advanced R*.

```
library(dplyr)
```

Data masking

Data masking makes data manipulation faster because it requires less typing. In most (but not all[1]) base R functions you need to refer to variables with `$`, leading to code that repeats the name of the data frame many times:

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

The dplyr equivalent of this code is more concise because data masking allows you to need to type `starwars` once:

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

Data- and env-variables

The key idea behind data masking is that it blurs the line between the two different meanings of the word "variable":

- **env-variables** are "programming" variables that live in an environment. They are usually created with `<-`.

- **data-variables** are "statistical" variables that live in a data frame. They usually come from data files (e.g. `.csv`, `.xls`), or are created manipulating existing variables.

To make those definitions a little more concrete, take this piece of code:

```
df <- data.frame(x = runif(3), y = runif(3))
df$x
#> [1] 0.08075014 0.83433304 0.60076089
```

It creates a env-variable, `df`, that contains two data-variables, `x` and `y`. Then it extracts the data-variable `x` out of the env-variable `df` using `$`.

I think this blurring of the meaning of "variable" is a really nice feature for interactive data analysis because it allows you to refer to data-vars as is, without any prefix. And this seems to be fairly intuitive since many newer R users will attempt to write `diamonds[x == 0 | y == 0, ]`.

Unfortunately, this benefit does not come for free. When you start to program with these tools, you're going to have to grapple with the distinction. This will be hard because you've never had to think about it before, so it'll take a while for your brain to learn these new concepts and categories. However, once you've teased apart the idea of "variable" into data-variable and env-variable, I think you'll find it fairly straightforward to use.

Indirection

The main challenge of programming with functions that use data masking arises when you introduce some indirection, i.e. when you want to get the data-variable from an env-variable instead of directly typing the data-variable's name. There are two main cases:

- When you have the data-variable in a function argument (i.e. an env-variable that holds a promise[2]), you need to **embrace** the argument by surrounding it in doubled braces, like `filter(df, {{ var }})`.

  The following function uses embracing to create a wrapper around `summarise()` that computes the minimum and maximum values of a variable, as well as the number of observations that were summarised:

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

- When you have an env-variable that is a character vector, you need to index into the `.data` pronoun with `[[`, like `summarise(df, mean = mean(.data[[var]]))`.

  The following example uses `.data` to count the number of unique values in each variable of `mtcars`:

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

Note that `.data` is not a data frame; it's a special construct, a pronoun, that allows you to access the current variables either directly, with `.data$x` or indirectly with `.data[[var]]`. Don't expect other functions to work with it.

Name injection

Many data masking functions also use dynamic dots, which gives you another useful feature: generating names programmatically by using `:=` instead of `=`. There are two basics forms, as illustrated below with `tibble()`:

- If you have the name in an env-variable, you can use glue syntax to interpolate in:

```
name <- "susan"
tibble("{name}" := 2)
#> # A tibble: 1 × 1
#>   susan
#>   <dbl>
#> 1     2
```

- If the name should be derived from a data-variable in an argument, you can use embracing syntax:

```
my_df <- function(x) {
  tibble("{{x}}_2" := x * 2)
}
my_var <- 10
my_df(my_var)
#> # A tibble: 1 × 1
#>   my_var_2
#>      <dbl>
#> 1       20
```

Learn more in `?rlang::`dyn-dots``.

Tidy selection

Data masking makes it easy to compute on values within a dataset. Tidy selection is a complementary tool that makes it easy to work with the columns of a dataset.

The tidyselect DSL

Underneath all functions that use tidy selection is the tidyselect package. It provides a miniature domain specific language that makes it easy to select columns by name, position, or type. For example:

- `select(df, 1)` selects the first column; `select(df, last_col())` selects the last column.
- `select(df, c(a, b, c))` selects columns `a`, `b`, and `c`.
- `select(df, starts_with("a"))` selects all columns whose name starts with "a"; `select(df, ends_with("z"))` selects all columns whose name ends with "z".
- `select(df, where(is.numeric))` selects all numeric columns.

You can see more details in `?dplyr_tidy_select`.

Indirection

As with data masking, tidy selection makes a common task easier at the cost of making a less common task harder. When you want to use tidy select indirectly with the column specification stored in an intermediate variable, you'll need to learn some new tools. Again, there are two forms of indirection:

- When you have the data-variable in an env-variable that is a function argument, you use the same technique as data masking: you **embrace** the argument by surrounding it in doubled braces.

  The following function summarises a data frame by computing the mean of all variables selected by the user:

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

- When you have an env-variable that is a character vector, you need to use `all_of()` or `any_of()` depending on whether you want the function to error if a variable is not found.

  The following code uses `all_of()` to select all of the variables found in a character vector; then `!` plus `all_of()` to select all of the variables *not* found in a character vector:

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

How-tos

The following examples solve a grab bag of common problems. We show you the minimum amount of code so that you can get the basic idea; most real problems will require more code or combining multiple techniques.

User-supplied data

If you check the documentation, you'll see that `.data` never uses data masking or tidy select. That means you don't need to do anything special in your function:

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

One or more user-supplied expressions

If you want the user to supply an expression that's passed onto an argument which uses data masking or tidy select, embrace the argument:

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

This generalises in a straightforward way if you want to use one user-supplied expression in multiple places:

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

If you want the user to provide multiple expressions, embrace each of them:

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

If you want to use the name of a variable in the output, you can embrace the variable name on the left-hand side of `:=` with `{{`:

```
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 }})
    )
}
```

Any number of user-supplied expressions

If you want to take an arbitrary number of user supplied expressions, use `...`. This is most often useful when you want to give the user full control over a single part of the pipeline, like a `group_by()` or a `mutate()`.

```
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 × 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
#> # ℹ 45 more rows
starwars %>% my_summarise(sex, gender)
#> `summarise()` has grouped output by 'sex'. You can override using the `.groups`
#> argument.
#> # A tibble: 6 × 4
#> # Groups:   sex [5]
#>   sex            gender      mass height
#>   <chr>          <chr>      <dbl>  <dbl>
#> 1 female         feminine    54.7   172.
#> 2 hermaphroditic masculine 1358     175
#> 3 male           masculine   80.2   179.
#> 4 none           feminine   NaN      96
#> # ℹ 2 more rows
```

When you use `...` in this way, make sure that any other arguments start with `.` to reduce the chances of argument clashes; see https://design.tidyverse.org/dots-prefix.html for more details.

Creating multiple columns

Sometimes it can be useful for a single expression to return multiple columns. You can do this by returning an unnamed data frame:

```
quantile_df <- function(x, probs = c(0.25, 0.5, 0.75)) {
  tibble(
    val = quantile(x, probs),
    quant = probs
  )
}

x <- 1:5
quantile_df(x)
#> # A tibble: 3 × 2
#>     val quant
#>   <dbl> <dbl>
#> 1     2  0.25
```

```
#> 2     3  0.5
#> 3     4  0.75
```

This sort of function is useful inside `summarise()` and `mutate()` which allow you to add multiple columns by returning a data frame:

```
df <- tibble(
  grp = rep(1:3, each = 10),
  x = runif(30),
  y = rnorm(30)
)

df %>%
  group_by(grp) %>%
  summarise(quantile_df(x, probs = .5))
#> # A tibble: 3 × 3
#>     grp   val quant
#>   <int> <dbl> <dbl>
#> 1     1 0.361   0.5
#> 2     2 0.541   0.5
#> 3     3 0.456   0.5

df %>%
  group_by(grp) %>%
  summarise(across(x:y, ~ quantile_df(.x, probs = .5), .unpack = TRUE))
#> # A tibble: 3 × 5
#>     grp x_val x_quant   y_val y_quant
#>   <int> <dbl>   <dbl>   <dbl>   <dbl>
#> 1     1 0.361     0.5  0.174      0.5
#> 2     2 0.541     0.5 -0.0110     0.5
#> 3     3 0.456     0.5  0.0583     0.5
```

Notice that we set `.unpack = TRUE` inside `across()`. This tells `across()` to *unpack* the data frame returned by `quantile_df()` into its respective columns, combining the column names of the original columns (`x` and `y`) with the column names returned from the function (`val` and `quant`).

If your function returns multiple *rows* per group, then you'll need to switch from `summarise()` to `reframe()`. `summarise()` is restricted to returning 1 row summaries per group, but `reframe()` lifts this restriction:

```
df %>%
  group_by(grp) %>%
  reframe(across(x:y, quantile_df, .unpack = TRUE))
#> # A tibble: 9 × 5
#>     grp x_val x_quant  y_val y_quant
#>   <int> <dbl>   <dbl>  <dbl>   <dbl>
#> 1     1 0.219    0.25 -0.710    0.25
#> 2     1 0.361    0.5   0.174    0.5
#> 3     1 0.674    0.75  0.524    0.75
#> 4     2 0.315    0.25 -0.690    0.25
#> # ℹ 5 more rows
```

Transforming user-supplied variables

If you want the user to provide a set of data-variables that are then transformed, use `across()` and `pick()`:

```
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 × 3
#>   species   mass height
#>   <chr>    <dbl>  <dbl>
#> 1 Aleena      15     79
#> 2 Besalisk   102    198
#> 3 Cerean      82    198
#> 4 Chagrian   NaN    196
#> # ℹ 34 more rows
```

You can use this same idea for multiple sets of input data-variables:

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

Use the `.names` argument to `across()` to control the names of the output.

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

Loop over multiple variables

If you have a character vector of variable names, and want to operate on them with a for loop, index into the special `.data` pronoun:

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

This same technique works with for loop alternatives like the base R `apply()` family and the purrr `map()` family:

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

(Note that the `x` in `.data[[x]]` is always treated as an env-variable; it will never come from the data.)

Use a variable from an Shiny input

Many Shiny input controls return character vectors, so you can use the same approach as above: `.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()))
}
```

See https://mastering-shiny.org/action-tidy.html for more details and case studies.

---

1. dplyr's `filter()` is inspired by base R's `subset()`. `subset()` provides data masking, but not with tidy evaluation, so the techniques described in this chapter don't apply to it.↩
2. In R, arguments are lazily evaluated which means that until you attempt to use, they don't hold a value, just a **promise** that describes how to compute the value. You can learn more at https://adv-r.hadley.nz/functions.html#lazy-evaluation↩