Programming with dplyr

Most dplyr functions use non-standard evaluation (NSE). This is a catch-all term that means they don't follow the usual R rules of evaluation. Instead, they capture the expression that you typed and evaluate it in a custom way. This has two main benefits for dplyr code:

- Operations on data frames can be expressed succinctly because you don't need to repeat the name of the data frame. For example, you can write `filter(df, x == 1, y == 2, z == 3)` instead of `df[df$x == 1 & df$y ==2 & df$z == 3, ]`.
- dplyr can choose to compute results in a different way to base R. This is important for database backends because dplyr itself doesn't do any work, but instead generates the SQL that tells the database what to do.

Unfortunately these benefits do not come for free. There are two main drawbacks:

- Most dplyr arguments are not **referentially transparent**. That means you can't replace a value with a seemingly equivalent object that you've defined elsewhere. In other words, this code:

  ```
  df <- tibble(x = 1:3, y = 3:1)
  filter(df, x == 1)
  #> # A tibble: 1 x 2
  #>       x     y
  #>   <int> <int>
  #> 1     1     3
  ```

  Is not equivalent to this code:

  ```
  my_var <- x
  #> Error in eval(expr, envir, enclos): object 'x' not found
  filter(df, my_var == 1)
  #> Error: object 'my_var' not found
  ```

  nor to this code:

  ```
  my_var <- "x"
  filter(df, my_var == 1)
  #> # A tibble: 0 x 2
  #> # … with 2 variables: x <int>, y <int>
  ```

  This makes it hard to create functions with arguments that change how dplyr verbs are computed.
- dplyr code is ambiguous. Depending on what variables are defined where, `filter(df, x == y)` could be equivalent to any of:

  ```
  df[df$x == df$y, ]
  df[df$x == y, ]
  df[x == df$y, ]
  df[x == y, ]
  ```

  This is useful when working interactively (because it saves typing and you quickly spot problems) but makes functions more unpredictable than you might desire.

Fortunately, dplyr provides tools to overcome these challenges. They require a little more typing, but a small amount of upfront work is worth it because they help you save time in the long run.

This vignette has two goals:

- Show you how to use dplyr's **pronouns** and **quasiquotation** to write reliable functions that reduce duplication in your data analysis code.
- To teach you the underlying theory including **quosures**, the data structure that stores both an expression and an environment, and **tidyeval**, the underlying toolkit.

We'll start with a warmup, tying this problem to something you're more familiar with, then move on to some practical tools, then dive into the deeper theory.

Warm up

You might not have realised it, but you're already accomplished at solving this type of problem in another domain: strings. It's obvious that this function doesn't do what you want:

```
greet <- function(name) {
  "How do you do, name?"
}
greet("Hadley")
#> [1] "How do you do, name?"
```

That's because `"` "quotes" its input: it doesn't interpret what you've typed, it just stores it in a string. One way to make the function do what you want is to use `paste()` to build up the string piece by piece:

```
greet <- function(name) {
  paste0("How do you do, ", name, "?")
}
greet("Hadley")
#> [1] "How do you do, Hadley?"
```

Another approach is exemplified by the **glue** package: it allows you to "unquote" components of a string, replacing the string with the value of the R expression. This allows an elegant implementation of our function because `{name}` is replaced with the value of the `name` argument.

```
greet <- function(name) {
  glue::glue("How do you do, {name}?")
}
greet("Hadley")
#> How do you do, Hadley?
```

Programming recipes

The following recipes walk you through the basics of tidyeval, with the nominal goal of reducing duplication in dplyr code. The examples here are somewhat inauthentic because we've reduced them down to very simple components to make them easier to understand. They're so simple that you might wonder why we bother writing a function at all. But it's a good idea to learn the ideas on simple examples, so that you're better prepared to apply them to the more complex situations you'll see in your own code.

Different data sets

You already know how to write functions that work with the first argument of dplyr verbs: the data. That's because dplyr doesn't do anything special with that argument, so it's referentially transparent. For example, if you saw repeated code like this:

```
mutate(df1, y = a + x)
mutate(df2, y = a + x)
mutate(df3, y = a + x)
mutate(df4, y = a + x)
```

You could already write a function to capture that duplication:

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

Unfortunately, there's a drawback to this simple approach: it can fail silently if one of the variables isn't present in the data frame, but is present in the global environment.

```
df1 <- tibble(x = 1:3)
a <- 10
mutate_y(df1)
#> # A tibble: 3 x 2
#>       x     y
#>   <int> <dbl>
#> 1     1    11
#> 2     2    12
#> 3     3    13
```

We can fix that ambiguity by being more explicit and using the `.data` pronoun. This will throw an informative error if the variable doesn't exist:

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

mutate_y(df1)
#> Column `a` not found in `.data`
```

If this function is in a package, using `.data` also prevents `R CMD check` from giving a NOTE about undefined global variables (provided that you've also imported `rlang::.data` with `@importFrom rlang .data`).

Different expressions

Writing a function is hard if you want one of the arguments to be a variable name (like `x`) or an expression (like `x + y`). That's because dplyr automatically "quotes" those inputs, so they are not referentially transparent. Let's start with a simple case: you want to vary the grouping variable for a data summarization.

```
df <- tibble(
  g1 = c(1, 1, 2, 2, 2),
  g2 = c(1, 2, 1, 2, 1),
  a = sample(5),
  b = sample(5)
)

df %>%
  group_by(g1) %>%
  summarise(a = mean(a))
#> # A tibble: 2 x 2
#>      g1     a
#>   <dbl> <dbl>
#> 1     1  4
#> 2     2  2.33

df %>%
  group_by(g2) %>%
  summarise(a = mean(a))
#> # A tibble: 2 x 2
#>      g2     a
#>   <dbl> <dbl>
#> 1     1  3.67
#> 2     2  2
```

You might hope that this will work:

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

my_summarise(df, g1)
#> Error: Column `group_var` is unknown
```

But it doesn't.

Maybe providing the variable name as a string will fix things?

```
my_summarise(df, "g2")
#> Error: Column `group_var` is unknown
```

Nope.

If you look carefully at the error message, you'll see that it's the same in both cases. `group_by()` works like `"`: it doesn't evaluate its input; it quotes it.

To make this function work, we need to do two things. We need to quote the input ourselves (so `my_summarise()` can take a bare variable name like `group_by()`), and then we need to tell `group_by()` not to quote its input (because we've done the quoting).

How do we quote the input? We can't use `""` to quote the input, because that gives us a string. Instead we need a function that captures the expression and its environment (we'll come back to why this is important later on). There are two possible options we could use in base R, the function `quote()` and the operator `~`. Neither of these work quite the way we want, so we need a new function: `quo()`.

`quo()` works like `"`: it quotes its input rather than evaluating it.

```
quo(g1)
#> <quosure>
#> expr: ^g1
#> env:  global
quo(a + b + c)
#> <quosure>
#> expr: ^a + b + c
#> env:  global
quo("a")
#> <quosure>
#> expr: ^"a"
#> env:  empty
```

`quo()` returns a **quosure**, which is a special type of formula. You'll learn more about quosures later on.

Now that we've captured this expression, how do we use it with `group_by()`? It doesn't work if we just shove it into our naive approach:

```
my_summarise(df, quo(g1))
#> Error: Column `group_var` is unknown
```

We get the same error as before, because we haven't yet told `group_by()` that we're taking care of the quoting. In other words, we need to tell `group_by()` not to quote its input, because it has been pre-quoted by `my_summarise()`. Yet another way of saying the same thing is that we want to **unquote** `group_var`.

In dplyr (and in tidyeval in general) you use `!!` to say that you want to unquote an input so that it's evaluated, not quoted. This gives us a function that actually does what we want.

```
my_summarise <- function(df, group_var) {
  df %>%
    group_by(!! group_var) %>%
    summarise(a = mean(a))
}

my_summarise(df, quo(g1))
#> # A tibble: 2 x 2
#>      g1     a
#>   <dbl> <dbl>
#> 1     1  4
#> 2     2  2.33
```

Huzzah!

There's just one step left: we want to call this function like we call `group_by()`:

```
my_summarise(df, g1)
```

This doesn't work because there's no object called `g1`. We need to capture what the user of the function typed and quote it for them. You might try using `quo()` to do that:

```
my_summarise <- function(df, group_var) {
  quo_group_var <- quo(group_var)
  print(quo_group_var)

  df %>%
    group_by(!! quo_group_var) %>%
    summarise(a = mean(a))
}

my_summarise(df, g1)
#> <quosure>
#> expr: ^group_var
#> env:  0x7fc08f920820
#> Error: Column `group_var` is unknown
```

I've added a `print()` call to make it obvious what's going wrong here: `quo(group_var)` always returns `~group_var`. It is being too literal! We want it to substitute the value that the user supplied, i.e. to return `~g1`.

By analogy to strings, we don't want `""`, instead we want some function that turns an argument into a string. That's the job of `enquo()`. `enquo()` uses some dark magic to look at the argument, see what the user typed, and return that value as a quosure. (Technically, this works because function arguments are evaluated lazily, using a special data structure called a **promise**.)

```
my_summarise <- function(df, group_var) {
  group_var <- enquo(group_var)
  print(group_var)

  df %>%
    group_by(!! group_var) %>%
    summarise(a = mean(a))
}

my_summarise(df, g1)
#> <quosure>
#> expr: ^g1
#> env:  global
#> # A tibble: 2 x 2
#>      g1     a
#>   <dbl> <dbl>
#> 1     1  4
#> 2     2  2.33
```

(If you're familiar with `quote()` and `substitute()` in base R, `quo()` is equivalent to `quote()` and `enquo()` is equivalent to `substitute()`.)

You might wonder how to extend this to handle multiple grouping variables: we'll come back to that a little later.

Different input variable

Now let's tackle something a bit more complicated. The code below shows a duplicate `summarise()` statement where we compute three summaries, varying the input variable.

```
summarise(df, mean = mean(a), sum = sum(a), n = n())
#> # A tibble: 1 x 3
#>    mean   sum     n
#>   <dbl> <int> <int>
#> 1     3    15     5
summarise(df, mean = mean(a * b), sum = sum(a * b), n = n())
#> # A tibble: 1 x 3
#>    mean   sum     n
#>   <dbl> <int> <int>
#> 1   8.6    43     5
```

To turn this into a function, we start by testing the basic approach interactively: we quote the variable with `quo()`, then unquoting it in the dplyr call with `!!`. Notice that we can unquote anywhere inside a complicated expression.

```
my_var <- quo(a)
summarise(df, mean = mean(!! my_var), sum = sum(!! my_var), n = n())
#> # A tibble: 1 x 3
#>    mean   sum     n
#>   <dbl> <int> <int>
#> 1     3    15     5
```

You can also wrap `quo()` around the dplyr call to see what will happen from dplyr's perspective. This is a very useful tool for debugging.

```
quo(summarise(df,
  mean = mean(!! my_var),
  sum = sum(!! my_var),
  n = n()
))
#> <quosure>
#> expr: ^summarise(df, mean = mean(^a), sum = sum(^a), n = n())
#> env:  global
```

Now we can turn our code into a function (remembering to replace `quo()` with `enquo()`), and check that it works:

```
my_summarise2 <- function(df, expr) {
  expr <- enquo(expr)

  summarise(df,
    mean = mean(!! expr),
    sum = sum(!! expr),
    n = n()
  )
}
my_summarise2(df, a)
#> # A tibble: 1 x 3
#>    mean   sum     n
#>   <dbl> <int> <int>
#> 1     3    15     5
my_summarise2(df, a * b)
#> # A tibble: 1 x 3
#>    mean   sum     n
#>   <dbl> <int> <int>
#> 1   8.6    43     5
```

Different input and output variable

The next challenge is to vary the name of the output variables:

```
mutate(df, mean_a = mean(a), sum_a = sum(a))
#> # A tibble: 5 x 6
#>      g1    g2     a     b mean_a sum_a
#>   <dbl> <dbl> <int> <int>  <dbl> <int>
#> 1     1     1     5     4      3    15
#> 2     1     2     3     2      3    15
#> 3     2     1     4     1      3    15
#> 4     2     2     1     3      3    15
#> # … with 1 more row
mutate(df, mean_b = mean(b), sum_b = sum(b))
#> # A tibble: 5 x 6
#>      g1    g2     a     b mean_b sum_b
#>   <dbl> <dbl> <int> <int>  <dbl> <int>
#> 1     1     1     5     4      3    15
#> 2     1     2     3     2      3    15
#> 3     2     1     4     1      3    15
#> 4     2     2     1     3      3    15
#> # … with 1 more row
```

This code is similar to the previous example, but there are two new wrinkles:

- We create the new names by pasting together strings, so we need `quo_name()` to convert the input expression to a string.
- `!! mean_name = mean(!! expr)` isn't valid R code, so we need to use the `:=` helper provided by rlang.

```
my_mutate <- function(df, expr) {
  expr <- enquo(expr)
  mean_name <- paste0("mean_", quo_name(expr))
  sum_name <- paste0("sum_", quo_name(expr))

  mutate(df,
    !! mean_name := mean(!! expr),
    !! sum_name := sum(!! expr)
  )
}

my_mutate(df, a)
#> # A tibble: 5 x 6
#>      g1    g2     a     b mean_a sum_a
#>    <dbl> <dbl> <int> <int>  <dbl> <int>
#> 1      1     1     5     4      3    15
#> 2      1     2     3     2      3    15
#> 3      2     1     4     1      3    15
#> 4      2     2     1     3      3    15
#> # … with 1 more row
```

Capturing multiple variables

It would be nice to extend `my_summarise()` to accept any number of grouping variables. We need to make three changes:

- Use `...` in the function definition so our function can accept any number of arguments.
- Use `enquos()` to capture all the `...` as a list of formulas.
- Use `!!!` instead of `!!` to **splice** the arguments into `group_by()`.

```
my_summarise <- function(df, ...) {
  group_var <- enquos(...)

  df %>%
    group_by(!!! group_var) %>%
    summarise(a = mean(a))
}

my_summarise(df, g1, g2)
#> # A tibble: 4 x 3
#> # Groups:   g1 [2]
#>       g1    g2     a
#>    <dbl> <dbl> <dbl>
#> 1      1     1     5
#> 2      1     2     3
#> 3      2     1     3
#> 4      2     2     1
```

`!!!` takes a list of elements and splices them into to the current call. Look at the bottom of the `!!!` and think `...`.

```
args <- list(na.rm = TRUE, trim = 0.25)
quo(mean(x, !!! args))
#> <quosure>
#> expr: ^mean(x, na.rm = TRUE, trim = 0.25)
#> env:  global

args <- list(quo(x), na.rm = TRUE, trim = 0.25)
quo(mean(!!! args))
#> <quosure>
#> expr: ^mean(^x, na.rm = TRUE, trim = 0.25)
#> env:  global
```

Now that you've learned the basics of tidyeval through some practical examples, we'll dive into the theory. This will help you generalise what you've learned here to new situations.

Quoting

Quoting is the action of capturing an expression instead of evaluating it. All expression-based functions quote their arguments and get the R code as an expression rather than the result of evaluating that code. If you are an R user, you probably quote expressions on a regular basis. One of the most important quoting operators in R is the *formula*. It is famously used for the specification of statistical models:

```
disp ~ cyl + drat
#> disp ~ cyl + drat
```

The other quoting operator in base R is `quote()`. It returns a raw expression rather than a formula:

```
# Computing the value of the expression:
toupper(letters[1:5])
#> [1] "A" "B" "C" "D" "E"
```

```
# Capturing the expression:
quote(toupper(letters[1:5]))
#> toupper(letters[1:5])
```

(Note that despite being called the double quote, `"` is not a quoting operator in this context, because it generates a string, not an expression.)

In practice, the formula is the better of the two options because it captures the code and its execution **environment**. This is important because even simple expression can yield different values in different environments. For example, the `x` in the following two expressions refers to different values:

```
f <- function(x) {
  quo(x)
}

x1 <- f(10)
x2 <- f(100)
```

It might look like the expressions are the same if you print them out.

```
x1
#> <quosure>
#> expr: ^x
#> env:  0x7fc090aaed08
x2
#> <quosure>
#> expr: ^x
#> env:  0x7fc090afb8d8
```

But if you inspect the environments using `rlang::get_env()` — they're different.

```
library(rlang)

get_env(x1)
#> <environment: 0x7fc090aaed08>
get_env(x2)
#> <environment: 0x7fc090afb8d8>
```

Further, when we evaluate those formulas using `rlang::eval_tidy()`, we see that they yield different values:

```
eval_tidy(x1)
#> [1] 10
eval_tidy(x2)
#> [1] 100
```

This is a key property of R: one name can refer to different values in different environments. This is also important for dplyr, because it allows you to combine variables and objects in a call:

```
user_var <- 1000
mtcars %>% summarise(cyl = mean(cyl) * user_var)
#>      cyl
#> 1 6187.5
```

When an object keeps track of an environment, it is said to have an enclosure. This is the reason that functions in R are sometimes referred to as closures:

```
typeof(mean)
#> [1] "closure"
```

For this reason we use a special name to refer to one-sided formulas: **quosures**. One-sided formulas are quotes (they carry an expression) with an environment.

Quosures are regular R objects. They can be stored in a variable and inspected:

```
var <- ~toupper(letters[1:5])
var
#> ~toupper(letters[1:5])
```

```
# You can extract its expression:
get_expr(var)
#> toupper(letters[1:5])

# Or inspect its enclosure:
get_env(var)
#> <environment: R_GlobalEnv>
```

Quasiquotation

> Put simply, quasi-quotation enables one to introduce symbols that stand for a linguistic expression in a given instance and are used as that linguistic expression in a different instance. — Willard van Orman Quine

Automatic quoting makes dplyr very convenient for interactive use. But if you want to program with dplyr, you need some way to refer to variables indirectly. The solution to this problem is **quasiquotation**, which allows you to evaluate directly inside an expression that is otherwise quoted.

Quasiquotation was coined by Willard van Orman Quine in the 1940s, and was adopted for programming by the LISP community in the 1970s. All expression-based functions in the tidyeval framework support quasiquotation. Unquoting cancels quotation of parts of an expression. There are three types of unquoting:

- basic
- unquote splicing
- unquoting names

Unquoting

The first important operation is the basic unquote, which comes in a functional form, `UQ()`, and as syntactic-sugar, `!!`.

```
# Here we capture `letters[1:5]` as an expression:
quo(toupper(letters[1:5]))
#> <quosure>
#> expr: ^toupper(letters[1:5])
#> env:  global

# Here we capture the value of `letters[1:5]`
quo(toupper(!! letters[1:5]))
#> <quosure>
#> expr: ^toupper(<chr: "a", "b", "c", "d", "e">)
#> env:  global
quo(toupper(UQ(letters[1:5])))
#> <quosure>
#> expr: ^toupper(<chr: "a", "b", "c", "d", "e">)
#> env:  global
```

It is also possible to unquote other quoted expressions. Unquoting such symbolic objects provides a powerful way of manipulating expressions.

```
var1 <- quo(letters[1:5])
quo(toupper(!! var1))
#> <quosure>
#> expr: ^toupper(^letters[1:5])
#> env:  global
```

You can safely unquote quosures because they track their environments, and tidyeval functions know how to evaluate them. This allows any depth of quoting and unquoting.

```
my_mutate <- function(x) {
  mtcars %>%
    select(cyl) %>%
    slice(1:4) %>%
    mutate(cyl2 = cyl + (!! x))
}

f <- function(x) quo(x)
expr1 <- f(100)
```

```
expr2 <- f(10)

my_mutate(expr1)
#>   cyl cyl2
#> 1   6  106
#> 2   6  106
#> 3   4  104
#> 4   6  106
my_mutate(expr2)
#>   cyl cyl2
#> 1   6   16
#> 2   6   16
#> 3   4   14
#> 4   6   16
```

The functional form is useful in cases where the precedence of `!` causes problems:

```
my_fun <- quo(fun)
quo(!! my_fun(x, y, z))
#> Error in my_fun(x, y, z): could not find function "my_fun"
quo(UQ(my_fun)(x, y, z))
#> <quosure>
#> expr: ^`~`(fun)(x, y, z)
#> env:  global

my_var <- quo(x)
quo(filter(df, !! my_var == 1))
#> <quosure>
#> expr: ^filter(df, (^x) == 1)
#> env:  global
quo(filter(df, UQ(my_var) == 1))
#> <quosure>
#> expr: ^filter(df, (^x) == 1)
#> env:  global
```

Unquote-splicing

The second unquote operation is unquote-splicing. Its functional form is `UQS()` and the syntactic shortcut is `!!!`. It takes a vector and inserts each element of the vector in the surrounding function call:

```
quo(list(!!! letters[1:5]))
#> <quosure>
#> expr: ^list("a", "b", "c", "d", "e")
#> env:  global
```

A very useful feature of unquote-splicing is that the vector names become argument names:

```
x <- list(foo = 1L, bar = quo(baz))
quo(list(!!! x))
#> <quosure>
#> expr: ^list(foo = 1L, bar = ^baz)
#> env:  global
```

This makes it easy to program with dplyr verbs that take named dots:

```
args <- list(mean = quo(mean(cyl)), count = quo(n()))
mtcars %>%
  group_by(am) %>%
  summarise(!!! args)
#> # A tibble: 2 x 3
#>      am  mean count
#>   <dbl> <dbl> <int>
#> 1     0  6.95    19
#> 2     1  5.08    13
```

Setting variable names

The final unquote operation is setting argument names. You've seen one way to do that above, but you can also use the definition operator `:=` instead of `=`. `:=` supports unquoting on both the LHS and the RHS.

The rules on the LHS are slightly different: the unquoted operand should evaluate to a string or a symbol.

```
mean_nm <- "mean"
count_nm <- "count"

mtcars %>%
  group_by(am) %>%
  summarise(
    !! mean_nm := mean(cyl),
    !! count_nm := n()
  )
#> # A tibble: 2 x 3
#>      am  mean count
#>   <dbl> <dbl> <int>
#> 1     0  6.95    19
#> 2     1  5.08    13
```