The tidyverse is making the life of a data scientist a lot easier. That’s why we at [STATWORX](https://www.statworx.com/de/) love to execute our analytics and data science with the tidyverse. Its user-centered approach has many advantages. Instead of the base R version df\_test[df\_test$x > 10], we can write df\_test %>% filter(x>10)), which is a lot more readable – especially if our data pipeline gets more complex and nested. Also, as you might have noticed, we can use the column names directly instead of referencing the Data Frame before. Because of those advantages, we want to use dplyr-verbs for writing our function. Imagine we want to write our own summary-function my\_summary(), which takes a grouping variable and calculates some descriptive statistics. Let’s see what happens when we wrap a dplyr-pipeline into a function:

my\_summary <- function(df, grouping\_var){

df %>%

group\_by(grouping\_var) %>%

summarise(

avg = mean(air\_time),

sum = sum(air\_time),

min = min(air\_time),

max = max(air\_time),

obs = n()

)

}

my\_summary(airline\_df, origin)

Error in grouped\_df\_impl(data, unname(vars), drop) :

Column `grouping\_var` is unknown

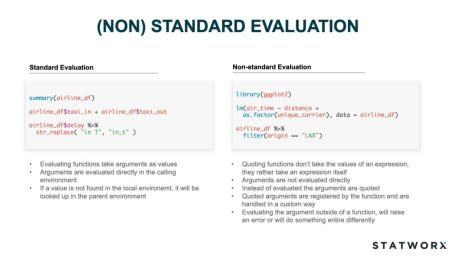
Our new function uses group\_by(), which is searching for a grouping variable *grouping\_var*, and not for *origin*, as we intended. So, what happened here? group\_by() is searching within its scope for the variable *grouping\_var*, which it does not find. group\_by() is quoting its arguments, *grouping\_var* in our example. That’s why dplyr can implement custom ways of handling its operation. Throughout the tidyverse, tidy evaluation is used. Therefore we can use column names, as it is a variable. However, our data frame has no column *grouping\_var*.

**Non-Standard Evaluation**

Talking about whether an argument is quoted or evaluated is a more precise way of stating whether or not a function uses non-standard evaluation (NSE).

*– Hadley Wickham*

The quoting used by group\_by() means, that it uses non-standard evaluation, like most verbs you can find in dplyr. Nonetheless, non-standard evaluation is not only found and used within dplyr and the tidyverse.



Because dplyr quotes its arguments, we have to do two things to use it in our function:

* First, we have to quote our argument
* Second, we have to tell dplyr, that we already have quoted the argument, which we do with unquoting

We will see this quote-and-unquote pattern consequently through functions which are using tidy evaluation.

my\_summary <- function(df, grouping\_var){

df %>%

group\_by(!!grouping\_var) %>%

summarise(

avg = mean(air\_time),

sum = sum(air\_time),

min = min(air\_time),

max = max(air\_time),

obs = n()

)

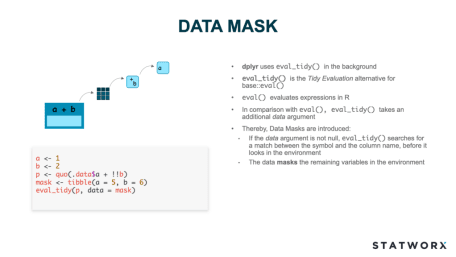
}

my\_summary(airline\_df, quo(origin))

Therefore, as input in our function, we quote the origin-variable, which means that R doesn’t search for the symbol origin in the global environment, but holds evaluation. The quotation takes place with the quo() function. In order to tell group\_by(), that the variable was already quoted we need to use the !!-Operator; pronounced Bang-Bang (if you wondered about the title).

If we are not using !!, group\_by() at first searches for the variable within its scope, which are the columns of the given data frame. As mentioned before, throughout the tidyverse, tidy evaluation is used with its eval\_tidy()-function. Whereby, it also introduces the concept of data mask, which makes data a first class object in R.

**Data Mask**



Generally speaking, the data mask approach is much more convenient. However, on the programming site, we have to pay attention to some things, like the quote-and-unquote pattern from before.

As a next step, we want the quotation to take place inside of the function, so the user of the function does not have to do it. Sadly, using quo() inside the function does not work.

my\_summary <- function(df, grouping\_var){

quo(grouping\_var)

df %>%

group\_by(!!grouping\_var) %>%

summarise(

avg = mean(air\_time),

sum = sum(air\_time),

min = min(air\_time),

max = max(air\_time),

obs = n()

)

}

my\_summary(airline\_df, origin)

Error in quos(...) : object 'origin' not found

We are getting an error message because quo() is taking it too literal and is quoting *grouping\_var* directly instead of substituting it with *origin* as desired. That’s why we use the function enquo() for **en**riched **quo**tation, which creates a quosure. A quosure is an object which contains an expression and an environment. Quosures redefine the internal promise object into something that can be used for programming. Thus, the following code is working, and we see the quote-and-unquote pattern again.

my\_summary <- function(df, grouping\_var){

grouping\_var <- enquo(grouping\_var)

df %>%

group\_by(!!grouping\_var) %>%

summarise(

avg = mean(air\_time),

sum = sum(air\_time),

min = min(air\_time),

max = max(air\_time),

obs = n()

)

}

my\_summary(airline\_df, origin)

# A tibble: 2 x 6

origin avg sum min max obs

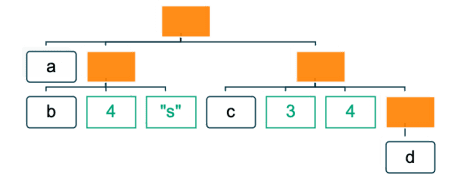
1 JFK 166. 587966 19 415 3539

2 LAX 132. 850259 1 381 6461

**All R code is a tree**

To better understand what’s happening, it is useful to know that every R code can be represented by an Abstract Syntax Tree (AST) because the structure of the code is strictly hierarchical. The leaves of an AST are either symbols or constants. The more complex a function call gets, the deeper an AST is getting with more and more levels. Symbols are drawn in dark-blue and have rounded corners, whereby constants have green borders and square corners. The strings are surrounded by quotes so that they won’t be confused with symbols. The branches are function calls and are depicted as orange rectangles.

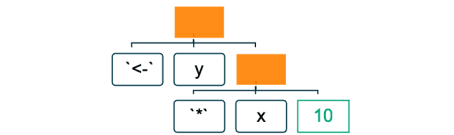
a(b(4, "s"), c(3, 4, d()))



To understand how an expression is represented as an AST, it helps to write it in its prefix form.

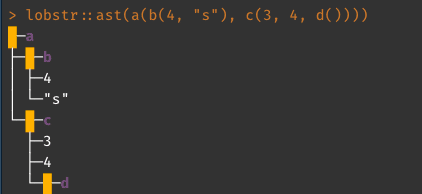
y <- x \* 10

`<-`(y, `\*`(x, 10))



There is also the R package called *lobstr*, which contains the function ast() to create an AST from R Code.

The code from the first example lobstr::ast(a(b(4, "s"), c(3, 4, d()))) results in this:

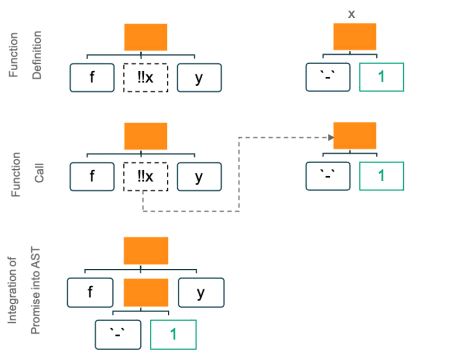


It looks as expected and just like our hand-drawn AST. The concept of ASTs helps us to understand what is happening in our function. So, if we have the following simple function, !!` introduces a placeholder (promise) for *x*.

x <- expr(-1)

f(!!x, y)

Due to R’s lazy evaluation, the function f() is not evaluated immediately, but once we called it. At the moment of the function call, the placeholder *x* is replaced by an additional AST, which can get arbitrary complex. Furthermore, it keeps the order of the operators correct, which is not the case when we use parse() and paste() with strings. So the resulting AST of our code snippet is the following:



Furthermore, !! also works with symbols, functions, and constants.

**Perfecting our function**

Now, we want to add an argument for the variable we are summarizing to refine our function. At the moment we have air\_time hardcoded into it. Thus, we want to replace it with a general *summary\_var* as an argument in our function. Additionally, we want the column names of the final output data frame to be adjusted dynamically, depending on the input variable. For adding *summary\_var*, we follow the quote and unquote pattern from above. However, for the column-naming, we need two additional functions.

Firstly, quo\_name(), which converts a quoted symbol into a string. Therefore, we can use normal string operations on it and, e.g. use the base paste command for manipulating it. However, we also need to unquote it, which would be on the Left-Hand-Side, where R is not allowing any computations. Thus, we need the second function, the vestigial operator := instead of the normal =.

my\_summary <- function(df, grouping\_var, summary\_var){

grouping\_var <- enquo(grouping\_var)

summary\_var <- enquo(summary\_var)

summary\_nm <- quo\_name(summary\_var)

summary\_nm\_avg <- paste0("avg\_",summary\_nm)

summary\_nm\_sum <- paste0("sum\_",summary\_nm)

summary\_nm\_obs <- paste0("obs\_",summary\_nm)

df %>%

group\_by(!!grouping\_var) %>%

summarise(

!!summary\_nm\_avg := mean(!!summary\_var),

!!summary\_nm\_sum := sum(!!summary\_var),

!!summary\_nm\_obs := n()

)

}

my\_summary(airline\_df, origin, air\_time)

# A tibble: 2 x 4

origin avg\_air\_time sum\_air\_time obs\_air\_time

1 JFK 166. 587966 3539

2 LAX 132. 850259 6461

**Tidy Dots**

In the next step, we want to add the possibility to summarize an arbitrary number of variables. Therefore, we need to use *tidy dots* (or *dot-dot-dot*) …. E.g. if we call the documentation for select(), we get

**Usage**

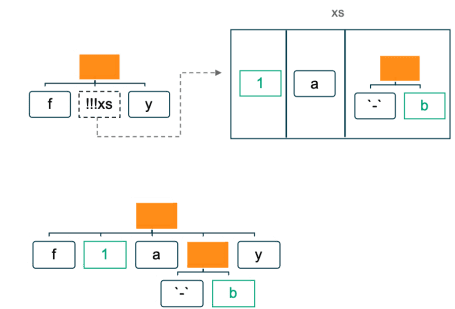
select(.data, ...)

**Arguments**

... One or more unquoted expressions separated by commas.

In select() we can use any number of variables we want to select. We will use tidy dots ... in our function. However, there are some things we have to account for.

Within the function, ... is treated as a list. So we cannot use !! or enquo(), because these commands are made for single variables. However, there are counterparts for the case of .... In order to quote several arguments at once, we can use enquos(). enquos() gives back a list of quoted arguments. In order to unquote several arguments we need to use !!!, which is also called the big bang-Operator. !!! replaces arguments one-to-many, which is called unquote-splicing and respects hierarchical orders.



With using *purrr*, we can neatly handle the computation with our list entries provided by ... ([for more information ask your Purrr-Macist](https://www.statworx.com/blog/about-risks-and-side-effects-consult-your-purrr-macist/)). So, putting everything together, we finally arrive at our final function.

my\_summary <- function(df, grouping\_var, ...) {

grouping\_var <- enquo(grouping\_var)

smry\_vars <- enquos(..., .named = TRUE)

smry\_avg <- purrr::map(smry\_vars, function(var) {

expr(mean(!!var, na.rm = TRUE))

})

names(smry\_avg) <- paste0("avg\_", names(smry\_avg))

smry\_sum <- purrr::map(smry\_vars, function(var) {

expr(sum(!!var, na.rm = TRUE))

})

names(smry\_sum) <- paste0("sum\_", names(smry\_sum))

df %>%

group\_by(!!grouping\_var) %>%

summarise(!!!smry\_avg, !!!smry\_sum, obs = n())

}

my\_summary(airline\_df, origin, dep\_delay, arr\_delay)

# A tibble: 2 x 6

origin avg\_dep\_delay avg\_arr\_delay sum\_dep\_delay sum\_arr\_delay obs

1 JFK 12.9 11.8 45792 41625 3539

2 LAX 8.64 5.13 55816 33117 6461

**And the tidy evaluation goes on and on**

As mentioned in the beginning, tidy evaluation is not only used within dplyr but within most of the packages in the tidyverse. Thus, to know how tidy evaluation works is also helpful if one wants to use ggplot in order to create a function for a styled version of a grouped scatter plot. In this example, the function takes the data, the values for the x and y-axes as well as the grouping variable as inputs:

scatter\_by <- function(.data, x, y, z=NULL) {

x <- enquo(x)

y <- enquo(y)

z <- enquo(z)

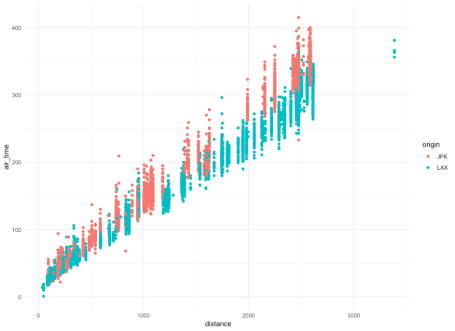
ggplot(.data) +

geom\_point(aes(!!x, !!y, color = !!z)) +

theme\_minimal()

}

scatter\_by(airline\_df, distance, air\_time, origin)



Another example would be to use R Shiny Inputs in a sparklyr-Pipeline. input$ cannot be used directly within sparklyr, because it would try to resolve the input list object on the spark side.

**server.R**

library(shiny)

library(dplyr)

library(sparklyr)

# Define server logic required to filter numbers

shinyServer(function(input, output) {

tbl\_1 <- tibble(a = 1:5, b = 6:10)

sc <- spark\_connect(master = "local")

tbl\_1\_sp <-

sparklyr::copy\_to(

dest = sc,

df = tbl\_1,

name = "tbl\_1\_sp",

overwrite = TRUE

)

observeEvent(input$select\_a, {

number\_b <- tbl\_1\_sp %>%

filter(a == !!input$select\_a) %>%

collect() %>%

pull()

output$text\_b <- renderText({

paste0("Selected number : ", number\_b)

})

})

})

**ui.R**

library(shiny)

library(dplyr)

library(sparklyr)

# Define UI for application t

shinyUI(fluidPage(

# Application title

titlePanel("Select Number Example"),

# Sidebar with a slider input for number

sidebarLayout(sidebarPanel(

sliderInput(

"select\_a",

"Number for 1:",

min = 1,

max = 5,

value = 1

)

),

# Show a text as output

mainPanel(textOutput("text\_b")))

))

**Conclusion**

There are many use cases for tidy evaluation, especially for advanced programmers. With the tidyverse getting bigger by the day, knowing tidy evaluation gets more and more useful. For getting more information about the metaprogramming in R and other advanced topics, I can recommend the book [Advanced R by *Hadley Wickham*](https://adv-r.hadley.nz/).