I think it’s fair to say that the pipe operator %>% from the {magrittr} package—made famous by its use in {dplyr}—revolutionized R. Using %>% you can chain together multiple functions to create pipelines for your data. That way you can write highly expressive code and mitigate the need to define intermediate variables for each step of you data processing. But how does %>% actually work? If you think about it, %>% does something quite astonishing: it applies the function on its right to the expression of its left. Achieving this is not really straightforward which is why I decided to write this post.

Before I start, though, a word of caution and a disclaimer. First, I will cover some advanced R in this post. If you are new to R then this probably isn’t for you. Second, in this post I will *not* look at the actual code of %>% as defined in {magrittr}. Why? Because {magrittr} does some fancy stuff to take care of edge cases that blur the essence of what %>% is all about. Instead I will use this minimal definition of %>% which really captures the essence of this operator and can replace the ‘real’ pipe operator in probably more than 95% of cases. Without further ado, here it is:

`%>%` <- function(lhs, rhs) {

lhs <- substitute(lhs)

rhs <- substitute(rhs)

building\_blocks <- c(

rhs[[1L]],

lhs,

as.list(rhs[-1L])

)

call <- as.call(building\_blocks)

eval(call, envir = parent.frame())

}

Take a deep breath. I understand that this might look frightening. But don’t worry, I will walk you through it step-by-step.

Let’s start with the very first line. lhs is short for left-hand side and rhs—as you might guess—stands for right-hand side. So, %>% is defined as a function of something to its left and something to its right. If you’ve ever used %>% this should make sense. But isn’t %>% an operator rather than a function? Yes and no. %>% is an operator but it also is a function. In fact, every operator in R be it \*, + or %\*% is a function. That’s why you can use %>% like any other function, e.g.

`%>%`("This actually works!", print())

[1] "This actually works!"

"Just like this!" %>% print()

[1] "Just like this!"

Granted this defies the purpose of %>% but just so you know.

Next, let’s move on to the function body, i.e. the code between { and }. What does substitute() do? Usually, if you pass an argument to a function it is *evaluated*.

identity <- function(x) {

x

}

y <- 1

identity(y)

[1] 1

identity(mean(1:10))

[1] 5.5

Using substitute() you can infer the actual *expression* that was passed to an argument. Put differently, you bypass evaluating the expression but instead *quote* it.

identity2 <- function(x) {

substitute(x)

}

identity2(y)

y

identity2(mean(1:10))

mean(1:10)

Why is this useful? Because you can actually modify this quoted expression which is exactly what %>% needs to do. Let’s move on to line 3 of %>% to see what I mean.

`%>%` <- function(lhs, rhs) {

lhs <- substitute(lhs)

rhs <- substitute(rhs)

building\_blocks <- c(

rhs[[1L]],

lhs,

as.list(rhs[-1L])

)

building\_blocks

}

mtcars %>% subset(cyl == 4, select = c(hp, cyl))

[[1]]

subset

[[2]]

mtcars

[[3]]

cyl == 4

$select

c(hp, cyl)

This looks interesting. Apparently you can subset a quoted expression like a list using [[. When you have a quoted expression expr that involves a function, expr[[1L]] will get you the name of the function and expr[-1L] will get you the arguments of the function.

expr <- identity2(mean(x = 1:10, na.rm = TRUE))

expr[[1L]]

mean

expr[-1L]

(1:10)(na.rm = TRUE)

But wait, (1:10)(na.rm = TRUE) looks almost like a function, doesn’t it? Indeed. R will always interpret the first element in a quoted expression as a function. To circumvent this I used as.list().

as.list(expr[-1L])

$x

1:10

$na.rm

[1] TRUE

Going back to the current definition of %>% we actually already have all we need.

mtcars %>% subset(cyl == 4, select = c(hp, cyl))

[[1]]

subset

[[2]]

mtcars

[[3]]

cyl == 4

$select

c(hp, cyl)

The first element in the list is the function on the right-hand side that should be applied to the left-hand side. The left-hand side appears second in this list, i.e. as the first argument of rhs, just as needed. The next elements are the remaining argument to rhs in the order initially specified.

A list is not what we need, though. We somehow have to turn this list into a function call. This is where as.call() comes into play.

`%>%` <- function(lhs, rhs) {

lhs <- substitute(lhs)

rhs <- substitute(rhs)

building\_blocks <- c(

rhs[[1L]],

lhs,

as.list(rhs[-1L])

)

call <- as.call(building\_blocks)

call

}

mtcars %>% subset(cyl == 4, select = c(hp, cyl))

subset(mtcars, cyl == 4, select = c(hp, cyl))

At this point %>% returns the code we want to run. But this code is still quoted and not evaluated yet. We can manually run it using the eval() function.

code <- mtcars %>% subset(cyl == 4, select = c(hp, cyl))

eval(code)

hp cyl

Datsun 710 93 4

Merc 240D 62 4

Merc 230 95 4

Fiat 128 66 4

Honda Civic 52 4

Toyota Corolla 65 4

Toyota Corona 97 4

Fiat X1-9 66 4

Porsche 914-2 91 4

Lotus Europa 113 4

Volvo 142E 109 4

But that’s obviously cumbersome. Instead we will evaluate the quoted expression directly inside %>%.

`%>%` <- function(lhs, rhs) {

lhs <- substitute(lhs)

rhs <- substitute(rhs)

building\_blocks <- c(rhs[[1L]], lhs, as.list(rhs[-1L]))

call <- as.call(building\_blocks)

eval(call)

}

If you’ve paid close attention to the definition of %>% I displayed in the beginning of this post, you will realize that this one here is a bit different. Let’s try to run it.

mtcars %>% subset(cyl == 4, select = c(hp, cyl))

hp cyl

Datsun 710 93 4

Merc 240D 62 4

Merc 230 95 4

Fiat 128 66 4

Honda Civic 52 4

Toyota Corolla 65 4

Toyota Corona 97 4

Fiat X1-9 66 4

Porsche 914-2 91 4

Lotus Europa 113 4

Volvo 142E 109 4

This seems to work just fine. However, let me make a little change to make you aware of a subtle bug in this function.

`%>%` <- function(lhs, rhs) {

mtcars <- NULL

lhs <- substitute(lhs)

rhs <- substitute(rhs)

building\_blocks <- c(rhs[[1L]], lhs, as.list(rhs[-1L]))

call <- as.call(building\_blocks)

eval(call)

}

mtcars %>% subset(cyl == 4, select = c(hp, cyl))

Error in subset.default(mtcars, cyl == 4, select = c(hp, cyl)): object 'cyl' not found

By default, eval() looks for the variables inside the expression given to it in the current [environment](http://adv-r.had.co.nz/Environments.html). In this case that’s the environment of %>%. Only if the variable is not defined in the current environment it will go up one environment and look in the environment where the function was called. In this case that’s the global environment. Since I defined mtcars locally it will never look in the global environment, though. If you think about it, if we use the %>% operator in the global environment it should always start to look there not inside its own environment. To do so, I will make a little change to the definition of %>%.

`%>%` <- function(lhs, rhs) {

mtcars <- NULL

lhs <- substitute(lhs)

rhs <- substitute(rhs)

building\_blocks <- c(rhs[[1L]], lhs, as.list(rhs[-1L]))

call <- as.call(building\_blocks)

eval(call, envir = parent.frame())

}

mtcars %>% subset(cyl == 4, select = c(hp, cyl))

hp cyl

Datsun 710 93 4

Merc 240D 62 4

Merc 230 95 4

Fiat 128 66 4

Honda Civic 52 4

Toyota Corolla 65 4

Toyota Corona 97 4

Fiat X1-9 66 4

Porsche 914-2 91 4

Lotus Europa 113 4

Volvo 142E 109 4

parent.frame() returns the environment from which a function—in this case %>%—was called. By passing parent.frame() to the envir argument of eval(), it will start looking for variables not in the local function environment but one level above. That way even if mtcars is defined locally it won’t do any harm. Still, let’s remove it and have a finally look at %>%.

`%>%` <- function(lhs, rhs) {

lhs <- substitute(lhs)

rhs <- substitute(rhs)

building\_blocks <- c(

rhs[[1L]],

lhs,

as.list(rhs[-1L])

)

call <- as.call(building\_blocks)

eval(call, envir = parent.frame())

}

As stated initially, this is not the same as %>% in {magrittr}. The following code would work with the real %>% but doesn’t with the definition I presented here.

"error\_message" %>%

gsub(pattern = "\_", replacement = " ", x = .)

Error in gsub("error\_message", pattern = "\_", replacement = " ", x = .): object '.' not found

For {magrittr}’s %>%, . is a special symbol that acts as a placeholder for lhs. This is useful when a function takes the data not as its first argument as is the case with gsub(). The pipe I wrote in this post is not that ‘smart’ and always places its lhs as the first argument of rhs. Nevertheless, the %>% presented here is functional. {dplyr} could use this pipe operator and everything would work just fine. In fact, the {poorman} package—a base R clone of {dplyr}—uses more-or-less this definition fo %>%.