**The Selectificator 1000**

Before v0.2.0, {poorman} was using a form of non-standard evaluation which converts function inputs to character strings and then uses those to figure out which columns to select. Specifically, {poorman} includes the following helper function.

deparse\_dots <- function(...) {

vapply(substitute(...()), deparse, NA\_character\_)

}

This is an unexported function since it is not user facing. Let’s dig into what this function does a little, but first we will define a temporary function to show by example.

dummy\_select <- function(...) {

deparse\_dots(...)

}

Now of course in the real select() function, select() takes both data (passed to .data) and column names (passed to ...) as inputs and then returns a subset of the data object containing only those columns. In our temporary function, we are just interested in the evaluation of the column names. So let’s try it out.

dummy\_select(x, y)

# [1] "x" "y"

Here we passed two objects, x and y which are intended to represent our column names. These objects are in fact symbols which we would expect to be evaluated, but they weren’t; they are instead turned into character strings. This is thanks to the deparse() and substitute() combination. deparse\_dots() first uses substitute() to return the unevaluated expression (in our case ...()) and substitutes any variables bound in the environment (here x and y). deparse\_dots() then loops over each of these inputs and deparses them. From the help page, ?deparse:

Turn unevaluated expressions into character strings.

So now we have our function column inputs as character strings. This is a good start as we can now match on those character strings with the column names of our data (match(deparse\_dots(...), colnames(.data))) to get the integer column positions of x and y. But what if the user inputs, for example, an integer?

dummy\_select(1, 3, 5)

# [1] "1" "3" "5"

This now poses a problem. Does the user here mean that they want the first, third and fifth columns returned? Or do they mean that there are columns within their data.frame called “1”, “3” and “5” – which aren’t necessarily in the first, third and fifth positions – and those are the columns they would like? So the problem with this approach is that when everything is converted to characters, it is almost impossible to know whether the user input to the function is a column name, a column integer position. The function has to try() certain things by making certain assumptions and of course making an assumption makes…well, you’ve heard the saying.

**The Selectificator 2000**

This is where the upgrades to the poorman::select() function come in. As a user I want to be able to pass integers, numerics, character strings, symbols and even functions to be interpreted by the select() function correctly and return the columns I desire. This is where the next line of code – which is absolutely brilliant – comes in.

eval(substitute(alist(...)))

Let’s break this function down and see what it does. We will work with a function and build it up bit by bit.

dummy\_select <- function(...) {

alist(...)

}

dummy\_select(x, y)

# [[1]]

# ...

Ok, not so exciting, right? What about if we wrap this in substitute?

dummy\_select <- function(...) {

substitute(alist(...))

}

dummy\_select(x, y)

# alist(x, y)

Now we can see that we have managed to pass in our inputs but they seem to be wrapped in an unevaluated call to alist(), so let’s try and evaluate it.

dummy\_select <- function(...) {

eval(substitute(alist(...)))

}

dummy\_select(x, y)

# [[1]]

# x

#

# [[2]]

# y

Perfect! Now we have our unevaluated function inputs stored in a list. But what is so great about that? Well, let’s take a look at the structure of this output.

str(dummy\_select(x, y))

# List of 2

# $ : symbol x

# $ : symbol y

So we can see that our inputs have been stored as their appropriate class; symbols! As a matter of fact, this occurs no matter what we use as our inputs.

str(dummy\_select(1L, 2, "x", y, starts\_with("R")))

# List of 5

# $ : int 1

# $ : num 2

# $ : chr "x"

# $ : symbol y

# $ : language starts\_with("R")

This is fantastic. What this means is that we can now define functionality that can handle each of the separate types we typically expect being used in a call to select(). All of this magic is owed to the alist() function, let’s take a look at the documentation.

alist handles its arguments as if they described function arguments. So the values are not evaluated, and tagged arguments with no value are allowed whereas list simply ignores them.

So if we had used the list() function, it would have attempted to evaluate our inputs to the dummy\_select() function whereas alist() does not – it stores them as unevaluated objects.

Now, this starts to get really interesting when we start to take a look at the language inputs. When using dplyr::select(), the user is able to select columns in a number of ways. Let’s take a look at some of them and what the structure of those look like.

str(dummy\_select(!w, x:y, -z))

# List of 3

# $ : language !w

# $ : language x:y

# $ : language -z

So selecting columns through the use of an exclamation mark (negation), a colon or a minus sign actually gives us a language object. Ok, so what? What is so special about this? Well what makes this so useful to us is how we can interact with the objects. We can actually access parts of the language object, much in the same way we do with a list(). Let’s take a look.

obj <- dummy\_select(x:y)

obj[[1]][1]

# `:`()

So operators are actually functions. In fact, in R, everything is an object and we interact with those objects using functions, so this makes sense. Let’s look a bit deeper into this object.

obj[[1]][[1]]

# `:`

obj[[1]][[2]]

# x

obj[[1]][[3]]

# y

So our function input, x:y, can be broken down into it’s individual components. We know that the components are made up of the colon function as well as x and y, but what is the structure of these final two components?

str(obj[[1]][[2]])

# symbol x

str(obj[[1]][[3]])

# symbol y

They are symbols! So now we can handle these much in the same way as we did with the symbol inputs in the first instance, because remember, just passing x or y to our dummy\_select() function returned a symbol.

str(dummy\_select(x, y))

# List of 2

# $ : symbol x

# $ : symbol y

So there you have it! This is how you, as the user, are now able to perform select() calls with {poorman} like this:

library(poorman, warn.conflicts = FALSE)

mtcars %>%

select(drat, mpg:hp, starts\_with("g"), everything()) %>%

head()

# drat mpg cyl disp hp gear wt qsec vs am carb

# Mazda RX4 3.90 21.0 6 160 110 4 2.620 16.46 0 1 4

# Mazda RX4 Wag 3.90 21.0 6 160 110 4 2.875 17.02 0 1 4

# Datsun 710 3.85 22.8 4 108 93 4 2.320 18.61 1 1 1

# Hornet 4 Drive 3.08 21.4 6 258 110 3 3.215 19.44 1 0 1

# Hornet Sportabout 3.15 18.7 8 360 175 3 3.440 17.02 0 0 2

# Valiant 2.76 18.1 6 225 105 3 3.460 20.22 1 0 1

**Conclusion**

This post has taken a look at two separate non-standard evaluation approaches – using nothing but {base} – deployed within the {poorman} package. We have seen how to break down language objects to pick out key pieces of information from them. We also saw how to determine the object types of function inputs and why this is important to consider. In particular, this showed how I was able to transform the select() function into the omega selectificator function…of doom.