library(purrr)

library(magrittr)

I first started using R long before the RStudio and tidyverse days… I remember writing chunks of code in a text editor and copy/pasting it into the R console! Yes I know, shocking. Nonetheless, most of us will have written code over the years that works just fine in base R, however in my case, the ever-growing adoption of the tidyverse packages (and my personal aspiration to improve my coding skills) has created a sense of necessity to re-write parts of it to fit within the tidyverse setting.

In this blog post I explore the purrr package (part of tidyverse collection) and its use within a data  
scientist’s toolset. I aim to present the case for using the purrr functions and through the use of examples compare them with base R functionality. To do this, we will concentrate on two typical coding scenarios in base R: 1) loops and 2) the suite of apply functions and then compare them with their relevant counterpart map functions in the purrr package.

However, before I start, I wanted to make it clear that I do sympathise with those of you whose first reaction to purrr is *“but I can do all this stuff in base R”*. Putting that aside, the obvious first obstacle for us to overcome is to lose the notion of *“if it’s not broken why change it”* and open our ‘coding’ minds to change. At least, I hope you agree with me that the silver lining of this kind of exercise is to satisfy ones curiosity about the purrr package and maybe learn something new!

Let us first briefly describe the concept of functional programming (FP) in case you are not familiar with it.

**Functional programming (FP)**

R is a functional programming language which means that a user of R has the necessary tools to create and manipulate functions. There is no need to go into too much depth here but it suffices to know that FP is the process of writing code in a structured way and through functions remove code duplications and redundancies. In effect, computations or evaluations are treated as mathematical functions and the output of a function only depends on the values of its inputs – known as arguments. FP ensures that any side-effects such as changes in state do not affect the expected output such that if you call the same function twice with the same arguments the function returns the same output.

For those that are interested to find out more, I suggest reading Hadley Wickham’s *Functional Programming* chapter in the “Advanced R” book. The companion website for this can be found [here.](http://adv-r.had.co.nz/)

The purrr package, which forms part of the tidyverse ecosystem of packages, further enhances the functional programming aspect of R. It allows the user to write functional code with less friction in a complete and consistent manner. The purrr functions can be used, among other things, to replace loops and the suite of apply functions.

**Let’s talk about loops**

The motivation behind the examples we are going to look at involve iterating in R for various scenarios. For example, iterate over elements of a vector or list, iterate over rows or columns of a matrix … the list (pun intended) can go on and on!

One of the first things that one gets very excited to ‘play’ when learning to use R – at least that was the case for me – is loops! Lot’s of loops, elaborate, complex… dare I say never ending infinite loops (queue hysteric laughter emoji). Joking aside, it is usually the default answer to a problem that involves iteration of some sort as I demonstrate below.

# Create a vector of the mean values of all the columns of the mtcars dataset

# The long repetitive way

mean\_vec <- c(mean(mtcars$mpg), mean(mtcars$cyl), mean(mtcars$disp), mean(mtcars$hp),

mean(mtcars$drat), mean(mtcars$wt), mean(mtcars$qsec), mean(mtcars$vs),

mean(mtcars$am), mean(mtcars$gear), mean(mtcars$carb))

mean\_vec

# The loop way

mean\_vec\_loop <- vector("double", ncol(mtcars))

for (i in seq\_along(mtcars)) {

mean\_vec\_loop[[i]] <- mean(mtcars[[i]])

}

mean\_vec\_loop

The resulting vectors are the same and the difference in speed (milliseconds) is negligible. I hope that we can all agree that the long way is definitely not advised and actually is bad coding practice, let alone the frustration (and error-prone task) of copy/pasting. Having said that, I am sure there are other ways to do this – I demonstrate this later using lapply – but my aim was to show the benefit of using a for loop in base R for an iteration problem.

Now imagine if in the above example I wanted to calculate the variance of each column as well…

# Create two vectors of the mean and variance of all the columns of the mtcars dataset

# For mean

mean\_vec\_loop <- vector("double", ncol(mtcars))

for (i in seq\_along(mtcars)) {

mean\_vec\_loop[[i]] <- mean(mtcars[[i]])

}

mean\_vec\_loop

#For variance

var\_vec\_loop <- vector("double", ncol(mtcars))

for (i in seq\_along(mtcars)) {

var\_vec\_loop[[i]] <- var(mtcars[[i]])

}

var\_vec\_loop

# Or combine both calculations in one loop

for (i in seq\_along(mtcars)) {

mean\_vec\_loop[[i]] <- mean(mtcars[[i]])

var\_vec\_loop[[i]] <- var(mtcars[[i]])

}

mean\_vec\_loop

var\_vec\_loop

Now let us assume that we know that we want to create these vectors not just for the mtcars dataset but for other datasets as well. We could in theory copy/paste the for loops and just change the dataset we supply in the loop but one should agree that this action is repetitive and could result to mistakes. Instead we can generalise this into functions. This is where FP comes into play.

# Create two functions that returns the mean and variance of the columns of a dataset

# For mean

col\_mean <- function(df) {

output <- vector("double", length(df))

for (i in seq\_along(df)) {

output[[i]] <- mean(df[[i]])

}

output

}

col\_mean(mtcars)

#For variance

col\_variance <- function(df) {

output <- vector("double", length(df))

for (i in seq\_along(df)) {

output[[i]] <- var(df[[i]])

}

output

}

col\_variance(mtcars)

Why not take this one step further and take full advantage of R’s functional programming tools by creating a function that takes as an argument a function! Yes, you read it correctly… a function within a function!

Why do we want to do that? Well, the code for the two functions above, as clean as it might look, is still repetitive and the only real difference between col\_mean and col\_var is the mathematical function that we are calling. So why not generalise this further?

# Create a function that returns a computational value (such as mean or variance)

# for a given dataset

col\_calculation <- function(df, fun) {

output <- vector("double", length(df))

for (i in seq\_along(df)) {

output[[i]] <- fun(df[[i]])

}

output

}

col\_calculation(mtcars, mean)

col\_calculation(mtcars, var)

**Did someone say *apply?***

I mentioned earlier that an alternative way to solve the problem is to use the apply function (or suite of apply functions such as lapply, sapply, vapply, etc). In fact, these functions are what we call Higher Order Functions. Similar to what we did earlier, these are functions that can take other functions as an argument.

The benefit of using higher order functions instead of a for loop is that they allow us to think about what code we are executing at a higher level. Think of it as: “apply this to that” rather than “take the first item, do this, take the next item, do this…”

I must admit that at first it might take a little while to get used to but there is definitely a sense of pride when you can improve your code by eliminating for loops and replace them with *apply-type* functions.

# Create a list/vector of the mean values of all the columns of the mtcars dataset

lapply(mtcars, mean) %>% head # Returns a list

sapply(mtcars, mean) %>% head # Returns a vector

Once again, speed of execution is not the issue and neither is the common misconception about loops being slow compared to apply functions. As a matter of fact the main argument in favour of using lapply or any of the purrr functions as we will see later is the pure *simplicity* and *readability* of the code. Full stop.

**Enter the purrr**

The best place to start when exploring the purrr package is the map function. The reader will notice that these functions are utilised in a very similar way to the apply family of functions. The subtle difference is that the purrr functions are consistent and the user can be assured of the output – as opposed to some cases when using for example sapply as I demonstrate later on.

# Create a list/vector of the mean values of all the columns of the mtcars dataset

map(mtcars, mean) %>% head # Returns a list

map\_dbl(mtcars, mean) %>% head # Returns a vector - of class double

Let us introduce the iris dataset with a slight modification in order to demonstrate the inconsistency that sometimes can occur when using the sapply function. This can often cause issues with the code and introduce mystery bugs that are hard to spot.

# Modify iris dataset

iris\_mod <- iris

iris\_mod$Species <- ordered(iris\_mod$Species) # Ordered factor levels

class(iris\_mod$Species) # Note: The ordered function changes the class

# Extract class of every column in iris\_mod

sapply(iris\_mod, class) %>% str # Returns a list of the results

sapply(iris\_mod[1:3], class) %>% str # Returns a character vector!?!? - Note: inconsistent object type

Since by default map returns a list one can ensure that an object of the same class is returned without any unexpected (and unwanted) surprises. This is inline with FP consistency.

# Extract class of every column in iris\_mod

map(iris\_mod, class) %>% str # Returns a list of the results

map(iris\_mod[1:3], class) %>% str # Returns a list of the results

To further demonstrate the consistency of the purrr package in this type of setting, the map\_\*() functions (see below) can be used to return a vector of the expected type, otherwise you get an informative error.

* map\_lgl() makes a logical vector.
* map\_int() makes an integer vector.
* map\_dbl() makes a double vector.
* map\_chr() makes a character vector.

# Extract class of every column in iris\_mod

map\_chr(iris\_mod[1:4], class) %>% str # Returns a character vector

map\_chr(iris\_mod, class) %>% str # Returns a meaningful error

# As opposed to the equivalent base R function vapply

vapply(iris\_mod[1:4], class, character(1)) %>% str # Returns a character vector

vapply(iris\_mod, class, character(1)) %>% str # Returns a possibly harder to understand error

It is worth noting that if the user does not wish to rely on tidyverse dependencies they can always use base R functions but need to be extra careful of the potential inconsistencies that might arise.

**Multiple arguments and neat tricks**

In case we wanted to apply a function to multiple vector arguments we have the option of mapply from base R or the map2 from purrr.

# Create random normal values from a list of means and a list of standard deviations

mu <- list(10, 100, -100)

sigma <- list(0.01, 1, 10)

mapply(rnorm, n = 5, mu, sigma, SIMPLIFY = FALSE) # I need SIMPLIFY = FALSE because otherwise I get a matrix

map2(mu, sigma, rnorm, n = 5)

The map2 function can easily extend to further arguments – not just two as in the example above – and that is where the pmap function comes in.

I also thought of sharing a couple of neat tricks that one can use with the map function.

1) Say you want to fit a linear model for every cylinder type in the mtcars dataset. You can avoid code duplication and do it as follows:

# Split mtcars dataset by cylinder values and then fit a simple lm

models <- mtcars %>%

split(.$cyl) %>% # Split by cylinder into 3 lists

map(function(df) lm(mpg ~ wt, data = df)) # Fit linear model for each list

2) Say we are using a function, such as sqrt (calculate square root), on a list that contains a non-numeric element. The base R function lapply throws an error and execution stops without knowing what caused the error. The safely function of purrr completes execution and the user can identify what caused the error.

x <- list(1, 2, 3, "e", 5)

# Base R

lapply(x, sqrt)

# purrr package

safe\_sqrt <- safely(sqrt)

safe\_result\_list <- map(x, safe\_sqrt) %>% transpose

safe\_result\_list$result

**Conclusion**

Overall, I think it is fair to say that using higher order functions in R is a great way to improve ones code. With that in mind, my closing remark for this blog post is to simply re-iterate the benefits of using the purrr package. That is:

* The output is consistent.
* The code is easier to read and write.