Functional_R_Coding

Marco Salvo

2025-03-26

What is Functional R Coding?

Functional R coding is a coding concept which focuses on the creation of functions to perform repetitive tasks. One of the main reasons we use complex programming languages, such as R, is the availability of high level functions which allow us to perform complex tasks quickly. Many of us rely on functions found in base R and packages such as dplyr, glmer, etc to perform fairly complex coding tasks in one line of code. Unfortunately, there isn't always a function to do every task we need, as our data and analyses are often unique for each of us. We however can take the concepts used by the people who built R and R packages to make our lives easier and more streamlined.

Function based coding is best suited for tasks which are repeated often and/or need to be reproducible for others to use. For simple tasks which only need to be done once, it may be faster to just write code that works through your data.

Function Basics in R

To start thinking like a functional programmer, we need to start at the very basics: What is a function? Why do we use them?

Functions are pieces of code that take an input, perform some kind of operation on them, and return an output. One of the most basic functions, is the print() function.

```
print("Hello World")
```

```
## [1] "Hello World"
```

The print function takes a charcter string, and returns it as output in your console. In this case we are not saving anything, but print can help us check progress on code, or see what an object says. We can see the documentation behind a function by putting a ? before the function name, like below.

```
?print
```

This will explain to us what the function does, what arguments it accepts, and what output it may give. But what is this function doing behind the hood? We can see that by using View on the function:

```
View(print)
```

In this case it's not super informative because print is a very simple function, but it gives us an idea of what's going on in the background.

Let's look at a higher level function: lm(). lm lets us build linear models in R.

```
View(lm)
```

We can see now what a more complex function contains, and that there's a lot which goes into it! Let's build a simple linear model to just reiterate how we use a function. We will use the iris dataset, everyone's favorite introductory data to build a model.

```
#load the data
data(iris)

#run a model
model <- lm(Sepal.Length ~ Species, data = iris)

#model summary
summary(model)
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Species, data = iris)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.6880 -0.3285 -0.0060  0.3120  1.3120
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)         5.0060     0.0728  68.762  < 2e-16 ***
## Speciesversicolor   0.9300     0.1030   9.033 8.77e-16 ***
## Speciesvirginica    1.5820     0.1030  15.366  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5148 on 147 degrees of freedom
## Multiple R-squared:  0.6187, Adjusted R-squared:  0.6135
## F-statistic: 119.3 on 2 and 147 DF,  p-value: < 2.2e-16
```

We used three functions here: 1) data: loaded the iris dataset into R 2) lm: ran a linear model explaining sepal width as a function of plant species 3) summary: gave us the model summary of our model

Each of these functions took arguments we gave it, did something to the data associated with those arguments, and then returned something to us.

This is great! But what happens when there isn't a function that does what you want?

Making Functions in R

Sometimes, R just doesn't have a built in function to do what we want it to. One example that comes to mind is calculating the standard error around your data. The formula for standard error can be seen below:

$SE = \frac{sd}{\sqrt{n}}$

Let's say we want to report the standard error around each value in our iris dataset. We could do it this way:

```r
sd(iris$Sepal.Length)/sqrt(length(iris$Sepal.Length))
```

```
## [1] 0.06761132
```

```r
sd(iris$Sepal.Width)/sqrt(length(iris$Sepal.Width))
```

```
## [1] 0.03558833
```

```r
sd(iris$Petal.Length)/sqrt(length(iris$Petal.Length))
```

```
## [1] 0.144136
```

```r
sd(iris$Petal.Width)/sqrt(length(iris$Petal.Width))
```

```
## [1] 0.06223645
```

As you can see, this takes four lines of code and a ton of repetition, which frankly is a waste of our valuable time. Instead we can make a function to do this! First, before we make this slightly more complex function, we can make a simple function such as a function to square a value, just to show how we build a function.

Below we do a few things. the **function** function tells R we want to create a function. Within the parentheses, we provide R the arguments we want the function to accept. Inside the function **return** says what we want the function to give back to us. So in this case, we give the function a number x and it gives us back x to the second power.

```r
squared <- function(x){
  return(x^2)
}

squared(2)
```

```
## [1] 4
```

Now let's do something a little more complex and build a standard error function!

We're going to do this in a few more parts than needed to make sure we can see how functions can be useful:

```r
SE <- function(values) # a vector of numeric values
  {
  #calculate sd
  fun.sd <- sd(values)
  #calculate n
  fun.n <- length(values)
  #calculate se
  se <- fun.sd/sqrt(fun.n)

  return(se)
}
```

So here, we give our function, **SE** a vector of values. We then find the standard deviation of those values, and the number of values in the vector. We then calculate SE, and the function returns the SE value. Let's use it!

```r
SE(iris$Sepal.Length)
```

```
## [1] 0.06761132
```

Super quick and much more interpretable than our non-function-based code! Now let's see how we can build on this to say, calculate the mean and se of all numeric columns in the iris dataframe.

```r
mean_se_summary <- function(df){

  #initialize vectors
  means <- c()
  SEs <- c()
  names <- c()

  #for every column in the dataframe
  for(i in 1:ncol(df)){
    #if the class of the column is numeric
    if(class(df[,i]) == 'numeric'){
      #extract the:
      means <- c(means, mean(df[,i])) #mean
      SEs <- c(SEs, SE(df[,i])) #SE
      names <- c(names, colnames(df)[i]) #name
    } else { #if not numeric
      next() #skip the column
    }
  }

  #combine to a data frame
  output <- data.frame(value = names,
                       mean = means,
                       se = SEs)

  #return output
  return(output)

}
```

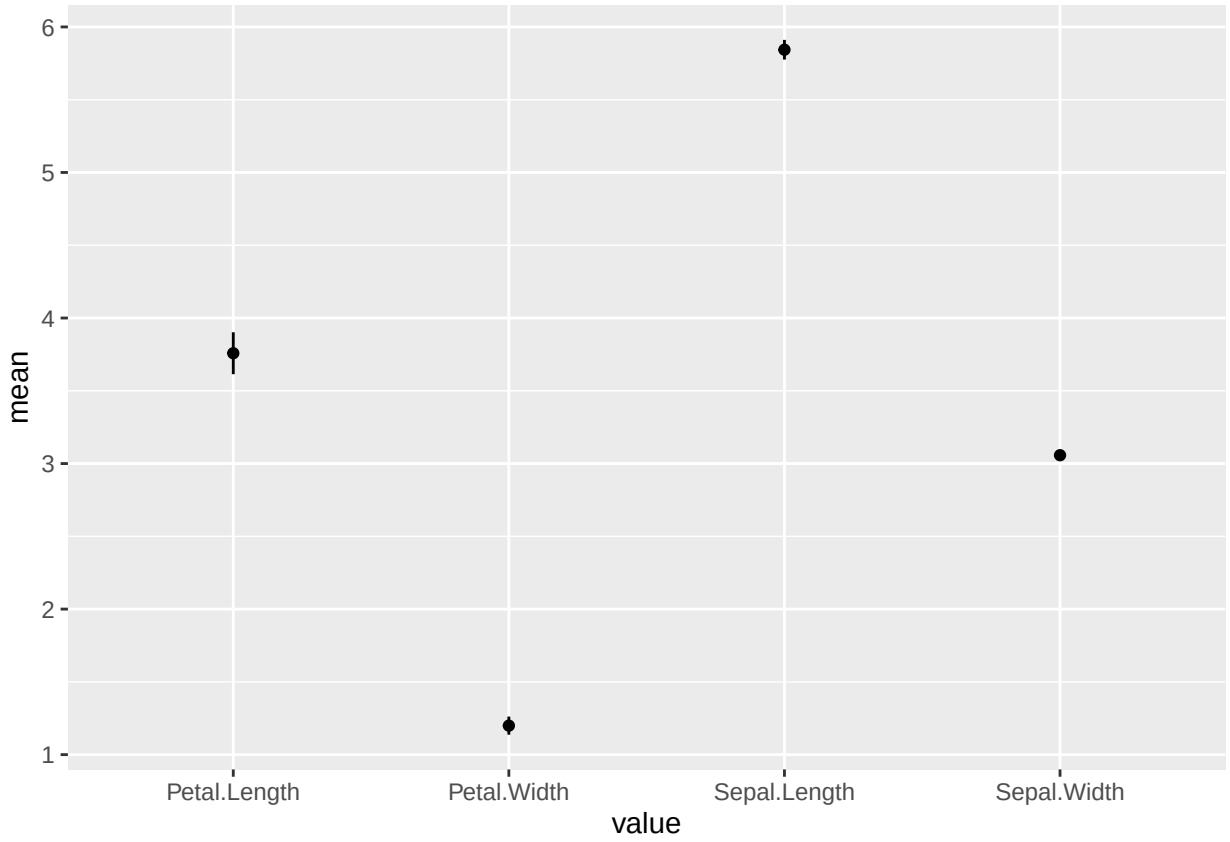
We can now use this function to get a summary of all numeric variables in iris.

```r
iris.sum <- mean_se_summary(iris)
iris.sum
```

```
##          value     mean         se
## 1 Sepal.Length 5.843333 0.06761132
## 2  Sepal.Width 3.057333 0.03558833
## 3 Petal.Length 3.758000 0.14413600
## 4  Petal.Width 1.199333 0.06223645
```

Awesome! Think about how long that would have taken if we tried to do that manually. And we can now repeat it for as many data frames as we want. For those of you more familiar with R, we know that tidyverse may help us do this even quicker, but for illustrative purposes we can see how this is useful. Let's now plot this just to show how it can be helpful.

```r
ggplot(iris.sum, aes(value, mean)) +
  geom_point() +
  geom_linerange(aes(ymin = mean - se, ymax = mean + se))
```



This starts to give us a good idea of how function based coding can be used to simplify tasks, and how we can combine custom built functions to allow our code to work for us.