Lab-Functions_in_R

May 22, 2021

Writing your own CUSTOM FUNCTIONS in R

0.1 Table of Contents

Estimated Time Needed: 25 min

About the Dataset

Imagine you got many movie recomendations from your friends and compiled all of the recomendations in a table, with specific info about each movie.

The table has one row for each movie and several columns

- **name** - The name of the movie
- **year** - The year the movie was released
- **length_min** - The lenght of the movie in minutes
- **genre** - The genre of the movie
- **average_rating** - Average rating on Imdb
- **cost_millions** - The movie's production cost in millions
- **sequences** - The amount of sequences
- **foreign** - Indicative of whether the movie is foreign (1) or domestic (0)
- **age_restriction** - The age restriction for the movie

You can see part of the dataset below

Lets first download the dataset that we will use in this notebook:

[ ]:
```
# code to download the dataset
download.file("https://ibm.box.com/shared/static/
 ↪n5ay5qadfe7e1nnsv5s01oe1x62mq51j.csv", destfile="movies-db.csv")
```

What is a Function?

A function is a re-usable block of code which performs operations specified in the function.

There are two types of functions :

- **Pre-defined functions**
- **User defined functions**

Pre-defined functions are those that are already defined for you, whether it's in R or within a package. For example, **`sum()`** is a pre-defined function that returns the sum of its numeric inputs.

User-defined functions are custom functions created and defined by the user. For example, you can create a custom function to print **Hello World**.

Pre-defined functions

There are many pre-defined functions, so let's start with the simple ones.

Using the **`mean()`** function, let's get the average of these three movie ratings:

- **Star Wars (1977)** - rating of 8.7
- **Jumanji** - rating of 6.9
- **Back to the Future** - rating of 8.5

[ ]:
```
ratings <- c(8.7, 6.9, 8.5)
mean(ratings)
```

We can use the **`sort()`** function to sort the movies rating in *ascending order*.

[ ]:
```
sort(ratings)
```

You can also sort by *decreasing* order, by adding in the argument **`decreasing = TRUE`**.

[ ]:
```
sort(ratings, decreasing = TRUE)
```

[Tip] How do I learn more about the pre-defined functions in R?

We will be introducing a variety of **pre-defined functions** to you as you learn more about R. There are just too many functions, so there's no way we can teach them all in one sitting. But if you'd like to take a quick peek, here's a short reference card for some of the commonly-used pre-defined functions:
https://cran.r-project.org/doc/contrib/Short-refcard.pdf

User-defined functions

Functions are very easy to create in R:

[ ]:
```
printHelloWorld <- function(){
    print("Hello World")
}
printHelloWorld()
```

To use it, simply run the function with **`()`** at the end:

[ ]:
```
printHelloWorld()
```

But what if you want the function to provide some **output** based on some **inputs**?

[ ]:
```
add <- function(x, y) {
    x + y
}
add(3, 4)
```

As you can see above, you can create functions with the following syntax to take in inputs (as its arguments), then provide some output.

```
f <- function(<arguments>) {
Do something
Do something
return(some_output)
}
```

Explicitly returning outputs in user-defined functions

In R, the last line in the function is automatically inferred as the output the function.

You can also explicitly tell the function to return an output.

[ ]:
```
add <- function(x, y){
    return(x + y)
}
add(3, 4)
```

It's good practice to use the `return()` function to explicitly tell the function to return the output.

Using IF/ELSE statements in functions

The **`return()`** function is particularly useful if you have any IF statements in the function, when you want your output to be dependent on some condition:

[ ]:
```
isGoodRating <- function(rating){
    #This function returns "NO" if the input value is less than 7. Otherwise it␣
 ↪returns "YES".

    if(rating < 7){
        return("NO") # return NO if the movie rating is less than 7

    }else{
        return("YES") # otherwise return YES
    }
}

isGoodRating(6)
isGoodRating(9.5)
```

Setting default argument values in your custom functions

You can a set a default value for arguments in your function. For example, in the **`isGoodRating()`** function, what if we wanted to create a threshold for what we consider to be a good rating?

Perhaps by default, we should set the threshold to 7:

[ ]:
```
isGoodRating <- function(rating, threshold = 7){
    if(rating < threshold){
        return("NO") # return NO if the movie rating is less than the threshold
    }else{
        return("YES") # otherwise return YES
    }
}

isGoodRating(6)
isGoodRating(10)
```

Notice how we did not have to explicitly specify the second argument (threshold), but we could specify it. Let's say we have a higher standard for movie ratings, so let's bring our threshold up to 8.5:

[ ]:
```
isGoodRating(8, threshold = 8.5)
```

Great! Now you know how to create default values. **Note that** if you know the order of the arguments, you do not need to write out the argument, as in:

[ ]:
```
isGoodRating(8, 8.5) #rating = 8, threshold = 8.5
```

Using functions within functions

Using functions within functions is no big deal. In fact, you've already used the **`print()`** and **`return()`** functions. So let's try making our **`isGoodRating()`** more interesting.

Let's create a function that can help us decide on which movie to watch, based on its rating. We should be able to provide the name of the movie, and it should return **NO** if the movie rating is below 7, and **YES** otherwise.

First, let's read in our movies data:

[ ]:
```
my_data <- read.csv("movies-db.csv")
head(my_data)
```

Next, do you remember how to check the value of the **average_rating** column if we specify a movie name?
Here's how:

[ ]:
```
# Within myData, the row should be where the first column equals "Akira"
# AND the column should be "average_rating"

akira <- my_data[my_data$name == "Akira", "average_rating"]
```

```
akira

isGoodRating(akira)
```

Now, let's put this all together into a function, that can take any **moviename** and return a **YES** or **NO** for whether or not we should watch it.

[ ]:
```
watchMovie <- function(data, moviename){
    rating <- data[data["name"] == moviename,"average_rating"]
    return(isGoodRating(rating))
}

watchMovie(my_data, "Akira")
```

Make sure you take the time to understand the function above. Notice how the function expects two inputs: `data` and `moviename`, and so when we use the function, we must also input two arguments.

But what if we only want to watch really good movies? How do we set our rating threshold that we created earlier? Here's how:

[ ]:
```
watchMovie <- function(data, moviename, my_threshold){
    rating <- data[data$name == moviename,"average_rating"]
    return(isGoodRating(rating, threshold = my_threshold))
}
```

Now our watchMovie takes three inputs: **data**, **moviename** and **my__threshold**

[ ]:
```
watchMovie(my_data, "Akira", 7)
```

What if we want to still set our default threshold to be 7? Here's how we can do it:

[ ]:
```
watchMovie <- function(data, moviename, my_threshold = 7){
    rating <- data[data[,1] == moviename,"average_rating"]
    return(isGoodRating(rating, threshold = my_threshold))
}

watchMovie(my_data,"Akira")
```

As you can imagine, if we assign the output to a variable, the variable will be assigned to **YES**

[ ]:
```
a <- watchMovie(my_data, "Akira")
a
```

While the **watchMovie** is easier to use, I can't tell what the movie rating actually is. How do I make it *print* what the actual movie rating is, before giving me a response? To do so, we can simply add in a **print** statement before the final line of the function.

We can also use the built-in **`paste()`** function to concatenate a sequence of character strings together into a single string.

[ ]:
```r
watchMovie <- function(moviename, my_threshold = 7){
    rating <- my_data[my_data[,1] == moviename,"average_rating"]

    memo <- paste("The movie rating for", moviename, "is", rating)
    print(memo)

    return(isGoodRating(rating, threshold = my_threshold))
}

watchMovie("Akira")
```

Just note that the returned output is actually the resulting value of the function:

[ ]:
```r
x <- watchMovie("Akira")
```

[ ]:
```r
print(x)
```

Global and local variables

So far, we've been creating variables within functions, but did you notice what happens to those variables outside of the function?

Let's try to see what **memo** returns:

[ ]:
```r
watchMovie <- function(moviename, my_threshold = 7){
    rating <- my_data[my_data[,1] == moviename,"average_rating"]

    memo <- paste("The movie rating for", moviename, "is", rating)
    print(memo)

    isGoodRating(rating, threshold = my_threshold)
}

watchMovie("Akira")
```

[ ]:
```r
memo
```

We got an error: `object 'memo' not found`. **Why?**

It's because all the variables we create in the function remain within the function. In technical terms, this is a **local variable**, meaning that the variable assignment does not persist outside the function. The `memo` variable only exists within the function.

But there is a way to create **global variables** from within a function – where you can use the global variable outside of the function. It is typically *not* recommended that you use global variables, since it may become harder to manage your code, so this is just for your information.

To create a **global variable**, we need to use this syntax:

```r
x <<- 1
```

Here's an example of a global variable assignment:

[ ]:
```
myFunction <- function(){
    y <<- 3.14
    return("Hello World")
    }
myFunction()
```

[ ]:
```
y #created only in the myFunction function
```

Scaling R with big data As you learn more about R, if you are interested in exploring platforms that can help you run analyses at scale, you might want to sign up for a free account on IBM Watson Studio, which allows you to run analyses in R with two Spark executors for free.

0.1.1 About the Author:

Hi! It's Aditya Walia, the author of this notebook. I hope you found R easy to learn! There's lots more to learn about R but you're well on your way. Feel free to connect with me if you have any questions.