Week-5: Functions

NM2207: Computational Media Literacy

Narayani Vedam, Ph.D.

Department of Communications and New Media

NUS National University of Singapore | Faculty of Arts & Social Sciences

This week

Table of contents


```
# Example of a function

circle_area <- function(r){

  pi*r^2

}
```

I. Functions and their use

What are functions?

- In mathematics, a function is a rule that maps the input to an output
- In computing, a function is a sequence of instructions within a larger computer program
- We have used so many functions so far, some of them being,
  - `read_csv()`
  - `mean()`
  - `summarize()`
  - `ggplot()`
  - `count()`
- These are functions that are either provided by R or by packages
- But there are times when such an inbuilt function or a package function won't be available

Code repetitions

"You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code)" - Hadley Wickham, R for Data Science

Instead of repeating code

```
data %>%
mutate(a_new = (a_old - min(a_old)) / (max(a_old) - min(a_old)), # <-- Notice how the only change
       b_new = (b_old - min(b_old)) / (max(b_old) - min(b_old)), # line is the change in variables
       c_new = (c_old - min(c_old)) / (max(c_old) - min(c_old)), # a_old to b_old
       d_new = (d_old - min(d_old)) / (max(d_old) - min(d_old))) # b_old to c_old, c_old to d_old
```

Write a function!

Code repetitions

Write a function x is the input

```
# Generic function
rescale_01 <- function(x) {

  (x - min(x)) / (max(x) - min(x)) #<-- Notice how a,b,c,d are replaced by x

}

data %>%
  mutate(a_new = rescale_01(a_old),

         b_new = rescale_01(b_old),

         c_new = rescale_01(c_old),

         d_new = rescale_01(d_old))
```

Function anatomy

A function consists of,

- Function arguments — data sets/values of the variables
  - They are parameters used by instructions in the body of the function
- Function body
  - They contain statements that are executed when the function is called
- Return value
  - The output inside `return()`
  - Could be a vector, list, data frame, another function, or even nothing
  - If unspecified, will be the last thing calculated

```
# Structure of a function
function_name <- function(arguments) {

  body_of_the_function

  return(output)

}
```

Note: We can assign the function to a name like any other object in R.

Function anatomy: example

- arguments: `x`
- body with instructions: `(x - min(x)) / (max(x) - min(x))`
- assign output to a variable: `rescale_01`
- Note that we don't need to explicitly call `return()`
- the last line of the code will be the value returned by the function.

```
rescale_01 <- function(x) {
  (x - min(x)) / (max(x) - min(x))
  }
```

II. Writing functions

Writing a function: printing output

You start writing code to say "Hello" to all of your friends.

```
print("Hello Kashif!")
```

```
## [1] "Hello Kashif!"
```

```
print("Hello Zach!")
```

```
## [1] "Hello Zach!"
```

```
print("Hello Deniz!")
```

```
## [1] "Hello Deniz!"
```

Writing a function: parameterize the code

body shld contain the repeated stuff given you're not using a function; eg print hello

- **Start** with the body of the function
- **Ask**: What part of the code is changing?
  - Make this an argument
- **Rewrite** the code to accommodate the parameterization
- Check several **potential inputs** to avoid future headaches
- Insert the body of the code within `{}`
- Name the function, `function_name`
- `?function_name` tells you only in the case of readily avail functions
  - what arguments the function expects
  - what value it produces

```
# print("Hello Kashif!") becomes ...

name <- "Kashif"

print(paste0("Hello ", name, "!"))
```

body

```
## [1] "Hello Kashif!"
```

Writing a function: add the structure

of the function slide 8

```
# name <- "Kashiif"

# print(paste0("Hello ", name, "!"))

function(name) {

print(paste0("Hello ", name, "!"))

}
```

```
## function(name) {
##
## print(paste0("Hello ", name, "!"))
##
## }
```

Writing a function: assign to a name

Try to use names that actively tell the user what the code does

- We recommend `verb_thing()`
- good: `calc_size()` or `compare_prices()`
- bad: `prices()`, `calc()`, or `fun1()`

```
# name <- "Kashif"
# print(paste0("Hello ", name, "!"))
say_hello_to <- function(name) {
  print(paste0("Hello ", name, "!"))
}
```

Simple example: printing output

Test out different inputs!

```
say_hello_to('Kashif')
```

```
## [1] "Hello Kashif!"
```

```
say_hello_to('Zach')
```

```
## [1] "Hello Zach!"
```

```
say_hello_to('Deniz')
```

```
## [1] "Hello Deniz!"
```

Technical aside: typeof(your_function)

LIKE variables, there are also diff types of functions. Functions only hv 2 types: built-in/primitive (provided by R wo downloading any packages) & closure

Like all R objects, functions have types;

- Primitive functions are of type “built-in”

```
typeof(`+`)
```

```
## [1] "builtin"
```

```
typeof(sum)
```

```
## [1] "builtin"
```

Technical aside: typeof(your_function)

Like all R objects, functions have types;

- The following functions are of type "closure"
  - user-defined functions — made ourselves (Not provided to us by R)
  - functions loaded with packages
  - many base R functions

```
typeof(say_hello_to)
```

```
## [1] "closure"
```

```
typeof(mean)
```

```
## [1] "closure"
```

Second example: mean of a sample

eg of nested functions

- For the sake of simplicity, assume that a sample represents a small collection - sample of height of a population, sample of grades of students, etc.
- This collection, numerically, could either be **random** or **normal** (most common kinds!)

```
mean(rnorm(100))
```

rnorm generates normal dist + use ? to know more about this function
argument of rnorm -> size of sample

```
## [1] 0.1392151
```

```
mean(rnorm(3000))
```

```
## [1] 0.00477852
```

Second example: calculating the mean of a sample

The number is changing, so it becomes the argument.

- The number is the sample size, so I call it `sample_size`.
- `n` would also be appropriate.
- The body code is otherwise identical to the code in the previous slide.

```
calc_sample_mean <- function(sample_size) {
  mean(rnorm(sample_size))
  }
```

1. identify the constant in the prob we're solving: instruction of the command being repeated -> mean(rnorm(sample_size))
2. identify input of argument -> sample size
3. no need return
4. put within {}
5. make sure hv function(argument)
6. assign a name

Second example: calculating the mean of a sample

Alternate code to calc mean of sample

For added clarity you can unnest your code, and assign the intermediate results to meaningful names.

- `return()` explicitly tells R what the function will return
  - The last line of code run is returned by default.

```
calc_sample_mean <- function(sample_size) {
  random_sample <- rnorm(sample_size)
  sample_mean <- mean(random_sample)
  return(sample_mean)
}
```

need return here, otherwise dk which output of which command in {} to return

but if u don't assign the 2nd command to a variable called " sample_mean", no need return

Second example: calculating the mean of a sample

If the function can be fit in one line, then you can write it without the curly brackets like so:

```
calc_sample_mean <- function(n) mean(rnorm(n))
```

have >1 instruction in defn of function -> get error bc no curly brackets

Some settings call for *anonymous* functions, where the function has no name.

```
function(n) mean(rnorm(n))
```

```
## function(n) mean(rnorm(n))
```

Always test your code

Try to foresee the kind of input you expect to use.

```
calc_sample_mean(1000)
```

```
## [1] -0.06605835
```

We see below that this function is not vectorized. We might hope to get 3 sample means out but only get 1.

```
# read ?rnorm to understand how rnorm
# inteprets vector input.
calc_sample_mean(c(100, 300, 3000))
```

impt to know what the expected types of inputs & outputs of a functions are eg vector?

```
## [1] 0.5082238
```

How to deal with unvectorized functions

If we don't want to change our function, but we want to use it to deal with vectors, then we have a couple options:

- Here we are going to use the function `group_by` from `tidyverse` package

Create tibble w column name called sample-Size assigned to vector shown

tibble is like a list, but unlike a list, all the columns/var shld hv the same no. of entries

group_by from tidyverse package

```
library(tidyverse)

#creating a vector to test our function
sample_tibble <- tibble(sample_sizes =
                          c(100, 300, 3000))

#using rowwise groups the data by row,
# allowing calc_sample_mean
sample_tibble %>%
group_by(sample_sizes) %>%
mutate(sample_means =
      calc_sample_mean(sample_sizes))
```

```
## # A tibble: 3 × 2
## # Groups:   sample_sizes [3]
##   sample_sizes sample_means
##          <dbl>        <dbl>
## 1          100      0.0502
## 2          300     -0.00606
## 3         3000      0.00573
```

Adding additional arguments

If we want to be able to adjust the details of how our function runs we can add arguments

- typically, we put "data" arguments first
  - and then "detail" arguments after
  - Mind the order of arguments/parameters, because it can't change when you call the function in the future

```
calc_sample_mean <- function(sample_size,
                             our_mean, our_sd)
{

sample <- rnorm(sample_size,
                mean = our_mean,
                mean(sample),
                sd = our_sd)

}
```

Setting defaults

We usually set default values for "detail" arguments.

```
calc_sample_mean <- function(sample_size,
                             our_mean=0,
                             our_sd=1) {

  sample <- rnorm(sample_size,
                  mean = our_mean,
                  sd = our_sd)

   mean(sample)
}
```

Still can change value of arguments w defaults, the defaults just means that if you don't assign any values to them, default values will show

```
# uses the defaults
calc_sample_mean(sample_size = 10)
```

```
## [1] -0.2621734
```

Setting defaults

```
# we can change one or two defaults.
# You can refer by name, or use position
calc_sample_mean(10, our_sd = 2)
```

```
## [1] 0.3151067
```

```
calc_sample_mean(10, our_mean = 6)
```

```
## [1] 5.978619
```

```
calc_sample_mean(10, 6, 2)
```

```
## [1] 5.903527
```

Setting defaults

This won't work though:

BC sample size not assigned default value

```
calc_sample_mean(our_mean = 5)
```

```
## Error in rnorm(sample_size, mean = our_mean, sd = our_sd): argument "sample_size" is missing, wi
```

Some more examples

```r
# Add 2 to the input x
add_two <- function(x) {
  x+2
}
```

```r
add_two(4)
```

```
## [1] 6
```

```r
add_two(-34)
```

```
## [1] -32
```

```r
add_two(5.784)
```

III. Common mistakes to avoid

What could go wrong?

Mismatch in the argument in the definition of the function (`x`) and the variable name used inside the function (`y`)

```
# Erroneous code
add_two <- function(x) {

  y+2

}

# Function call
add_two(4)
```

```
## Error in add_two(4): object 'y' not found
```

```
# Correct code
add_two <- function(x) {

  x+2

}

# Function call

add_two(4)
```

```
## [1] 6
```

Another example: adding two numbers

Mismatch in the names of the function in the definition and in the function call

```
# function definition
add_numbers <- function(x,y) {
  x+y
}

# function call
add_numers(45, 12)
```

```
## Error in add_numers(45, 12): could not find function "add_numers"
```

Another example: adding two numbers

Mismatch in the number of arguments in the function definition and in the function call

```
# function definition
add_numbers <- function(x,y) {
  x+y
}

# function call
add_numbers(45, 12, 72)
```

```
## Error in add_numbers(45, 12, 72): unused argument (72)
```

Another example: adding two numbers

You cannot return more than one value using the `return()` command

```
# function definition
add_numbers <- function(x,y) {
  z = 20
  x+y
  return(x+y,z)
}
# Function call
add_numbers(1,2)
```

```
## Error in return(x + y, z): multi-argument returns are not permitted
```

Another example: adding two numbers

Variables declared inside a function, cannot be accessed outside of it

```
# function definition
add_numbers <- function(x,y) {
  z = 20
  x+y
}
print(z)
```

```
## Error in print(z): object 'z' not found
```

IV. Scope of variables

Scoping

```
# Initialize z
z <- 1
sprintf("The value assigned to z outside the function is %d",z)
```

```
## [1] "The value assigned to z outside the function is 1"
```

Set default value of 2 to z

```
# declare a function, notice how we pass a value of 2 for z
foo <- function(z = 2) {
  # reassigning z
  z <- 3
  return(z+3)
}
foo()
```

note: bad naming convention

value defined outside the function was overridden by the value defined inside the function (z = 2),

which was again overridden by the reassignment of value within the function body

```
## [1] 6
```

Scoping

```r
# Initialize z
z <- 1
# declare a function, notice how we pass a value of 2 for z
foo <- function(z = 2) {
  # reassigning z
  z <- 3
  return(z+3)
}
# another reassignment of z
foo(z = 4)
```

```
## [1] 6
```

```r
# Accessing z outside the function
sprintf("The final value of z after reassigning it to a different value inside the function is %d"
```

```
## [1] "The final value of z after reassigning it to a different value inside the function is 1"
```

Scoping

> The scope of a variable is decided by two factors, (a) location of initialization and (b) where we can access it when required

There are mainly two types of variable scopes,

Global

- They are declared outside functions
- They can be accessed from anywhere in the program
- Hence, **global**
- In the code, `z<-1` is the global variable
- Type `z` in the console and check the output

Local

- They are declared inside functions
- They cannot be accessed outside the functions
- Hence, **local**
- In the code, `z<-3` is the local variable

Thanks!

Slides created via the R packages:

xaringan
gadenbuie/xaringanthemer.

Faculty of Arts
& Social Sciences