Session 2-Functions and Loops

Ya-Feng Wen & Shen Cheng

2020-06-19

Contents

Objectives

- Functions
- Loops
  - For loops
  - Nesting
  - While loops

Functions

When should you write a function?

When you have copied and pasted a block of code more than twice! It's called the DRY ("Do not repeat yourself") principle.

Benefits of a function

- An evocative function name makes code easier to understand
- Only need to update code in one place
- Reduce incidental mistakes when copying and pasting code

Three key steps to creating a new function:

1. Choose an evocative function name
   - Ideally should be verbs
   - Use snake_case or camelCase
   - Use consistent names and arguments if you have a family of functions that do similar things
   - Avoid overriding existing functions and variables
2. List the inputs, or **arguments** (should be nouns), to the function inside `function`
3. Place the code you have developed in body of the function, a `{` block that immediately follows `function(...)`

It is easier to start with working code and turn it into a function; it is harder to create a function and then try to make it work.

4. Unit testing to turn informal, interactive tests into formal, automated tests

How to write a function?

- Basic structure: `function(arglist) {body}`
- One line function (drop curly braces): `function(arglist) body`. For example, `function(x) file.info(x)$isdir` to check whether path `x` is a directory or not.
- Using snippet. Type `fun` + Tab

Practice Function Writing

1. Create a function which calculate the proportion of NA values in a vector.

```
# create a testing vector called test
test <- c(1:5, NA, 6:8, NA)

# check which element is NA
is.na(test)

# count the proportion of NA
as.numeric(is.na(test)) # convert Boolean to numeric
sum(as.numeric(is.na(test))) # sum up number of NA
sum(as.numeric(is.na(test)))/length(test) # calculate the proportion
```

```r
# alternatively, use the following expression to replace three lines of code above
mean(is.na(test))


prop_NA <- function(x) {
  mean(is.na(x))
}

# testing your function
prop_NA(test)
```

2. Write a function to calculate variance of a vector.

$$Var(x) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

```r
# number of element
length(test) # including NA
length(na.omit(test))

# mean of the vector
mean(test) # returns NA
mean(test, na.rm = TRUE)

# mean square error
(test - mean(test, na.rm = TRUE))^2

# sum of mean square error
sum((test - mean(test, na.rm = TRUE))^2) # returns NA
sum((test - mean(test, na.rm = TRUE))^2, na.rm = T)


variance <- function(x, na.rm=TRUE) {
  n <- length(na.omit(x))
  m <- mean(x, na.rm = na.rm)
  mean_sqaure_error <- (x - m)^2

  sum(mean_sqaure_error, na.rm = na.rm) / (n-1)
}


variance(test)
variance(test) == var(test, na.rm = T)
```

3. Write a function to calculate drug concentrations until time T and plot the concentration time curve. This function allows you to calcuate drug concentrations following either the one-compartment or two-compartment PK (given as IV) with linear elimination. Recall that concentration can be described as follow:

For One-Compartment:

$$C(t) = C_0 \times e^{-kt}$$

For Two-Compartment:

$$C(t) = Ae^{-\alpha t} + Be^{-\beta t}$$

```
Dose = 100 # mg
Vc = 2 # L
k = 0.1 # hr-1

t = seq(0,24,1)

C0 = Dose/Vc

Conc = C0 * exp(-k*t)
#plot(Conc~t)

oneCmtIV <- function(Dose,Time) {
  # define parameters
  Vc = 2 # L
  k = 0.1 # hr-1
  # simulation time
  t = seq(0, Time, 1)
  # calculate concentration
  C0 = Dose/Vc
  Conc = C0 * exp(-k*t)
  # plot
  plot(Conc ~ t)
}

#oneCmtIV(Dose = 100, Time = 24)


twoCmtIV <- function(Dose, Time) {

  A = 30 # mg/L
  B = 20 # mg/L
  alpha = 1.5 # hr-1
  beta = 0.1 # hr-1

  t = seq(0, Time, 1)

  C0 = Dose/Vc
  Conc = A*exp(-alpha*t) + B*exp(-beta*t)

  plot(Conc ~ t)
}

#twoCmtIV(100, 24)


simulateOneTwoCmtIV <- function(Dose, Time, Cmt, ...) {
  # parameters for 1-CMT
  Vc = 2 # L
  k = 0.1 # hr-1

  # parameters for 2-CMT
  A = 30 # mg/L
  B = 20 # mg/L
```

```
  alpha = 1.5 # hr-1
  beta = 0.1 # hr-1

  t = seq(0, Time, 0.1)

  C0 = Dose/Vc

  if (Cmt == 1) {

    Conc = C0 * exp(-k*t)

  } else if (Cmt == 2) {

    Conc = A*exp(-alpha*t) + B*exp(-beta*t)

  } else {

    stop("Cmt must be 1 or 2")

  }

  plot(Conc ~ t)

}

#simulateOneTwoCmtIV(100, 24, 1)
#simulateOneTwoCmtIV(100, 24, 1, Vc=3)
#simulateOneTwoCmtIV(100, 24, 2)
#simulateOneTwoCmtIV(100, 24, 3)
```

Loops

Concepts

What is loop?

A loop is a sequence of instructions that is repeated until a certain condition is reached. It allows you to repeat operations multiple times without copy-pasting sections of your code.

Why we need loops?

Similar to function writting, using loops can avoid writing same or similar codes repeatedly.

for-loops

Three essential components to write a for loop

- Output
- Sequence
- Body

Syntax in general (example 1)

```
output <- c()
for (val in sequence){body}
output
```

Example 1

```
#Assemble dataset for example 1
test_df <- data.frame(
  v1 = c(1:5, NA, 6:8, NA),
  v2 = seq(1,10,1),                          #seq(from, to, by)
  v3 = c(letters[1:9], NA),                  #letter() generate lower case letters
  v4 = c("apple", "banana", NA, "cherry", NA, NA, NA, "orange", NA, NA)
  )
test_df
```

- purpose: Using the `prop_NA()` created to calculate the proportion of NAs in each column.

```
prop_NA(test_df$v1)
prop_NA(test_df$v2)
prop_NA(test_df$v3)
prop_NA(test_df$v4)
```

Alternatively, we can use a for-loop

```
output <- c()                          #output
for (i in 1:4){                        #sequence
  output[i] <- prop_NA(test_df[,i])    #body
}
output
```

You can write it in a fancier way which also improves its generalizability.

```
output <- c()
for (i in seq_along(test_df)){
  output[[i]] <- prop_NA(test_df[[i]])
}
output
```

- seq_along() creates a sequence from 1 to the length of the input
- [[ ]] can be used for elements selection for both vector and dataframe otherwise will have to use different syntax

You can also add names to the output generated to make it more recognizable.

```
col_prop_NA <- c()
for (i in seq_along(test_df)){
  col_prop_NA[[i]] <- prop_NA(test_df[[i]])
  names(col_prop_NA)[i] <- names(test_df)[i]
}
col_prop_NA
```

Practice right away! Write the above code as a function.

```
col_prop_NA <- function(df){                #name <- function(argument){
  output <- c()                             #output
  for (i in seq_along(df)){                 #sequence
    output[[i]] <- prop_NA(df[[i]])         #body
    names(output)[i] <- names(df)[i]        #}
  }
  output
}

col_prop_NA(test_df)
```

You can the apply function above to other dataframes.

```
col_prop_NA(ChickWeight)
col_prop_NA(Theoph)
```

Writting a for-loop is actually not the easiest way to do this in r...

```
#lapply
apply(test_df, 2, prop_NA)
lapply(test_df, prop_NA)
```

A group of `apply()` function can be useful alternatives for "for loops". A concise summary of `apply()` based syntax was presented by Dr.Sam Callisto in 2018 and Dr. Ashwin Karanam in 2019 during the R for PMX student session.

*`apply()`: evaluate a function over the margins of a matrix or array

*`lapply()`: evaluate a function on each element in a list, and return the results as a list

*`sapply()`: evaluate a function on each element in a list, and return the results in a simplified format (not always predictable, but can be convenient)

*`vapply()`: similar to sapply(), but with more consistent return types

*`tapply()`: evaluate a function on subsets of a vector; alternative to group_by() for dealing with subsets

"Purr" package within "Tidyverse" provide similar functions but is suggested to be more powerful.

break and next

`break` enables you to stop the loop when encounter certain elements.

```
output <- c()
for (i in seq_along(test_df)){
  output[[i]] <- prop_NA(test_df[[i]])
  names(output)[i] <- names(test_df)[i]
  if (i == 3) {
    break
  }
}
output

#Similarly we can create this as a function
col_prop_NA <- function(df, end){
  output <- c()
  for (i in seq_along(df)){
    output[[i]] <- prop_NA(df[[i]])
    names(output)[i] <- names(df)[i]
    if (i == end) break
  }
output
}

col_prop_NA(df = test_df, end = 3)
```

Unlike `break`, `next` will enable you to skip certain elements without stopping the entire loop.

```
output <- c()
for (i in seq_along(test_df)){
  if (i == 3) next
  output[[i]] <- prop_NA(test_df[[i]])
  names(output)[i] <- names(test_df)[i]
}

output <- output[!is.na(output)]      #remove column skipped
output

#Create a function for this as well
col_prop_NA <- function(df, skip){
  output <- c()
  for (i in seq_along(test_df)){
    if (i == skip) next
    output[[i]] <- prop_NA(test_df[[i]])
    names(output)[i] <- names(test_df)[i]
  }
  output <- output[!is.na(output)]
  output
}

col_prop_NA(df_test_df, skip = 3)
```

Nesting for-loops (example 2)

Sometimes, you may need to loop over multiple levels instead just one level. In these situations, you can use multiple for loops nested with each other

Example 2

```
#assemble dataset for example 2

#Description: 10 patients, each patients go through 3 occasions, at each occation, 3 systolic blood pres

set.seed(0720)

sbp <- data.frame(
  ID  = c(rep(1:10,9)),
  OCC = c(rep(rep(1:3,each = 10), 3)),
  SBP  = rnorm(90, mean = 130, sd = 13)
)
```

*purpose: calculate average systolic blood pressure (SBP) for each patients at each occasion

```
var <- data.frame(NULL)

for (i in sbp$ID){
  temp1 <- subset(sbp, ID == i)

  for (j in temp1$OCC){
    temp2 <- subset(temp1, OCC == j)
    var[i,j] = variance(temp2$SBP)             #function created above
    rownames(var)[i] = paste("ID" , i, " ")
    colnames(var)[j] = paste("OCC", j, " ")
  }
}

var
```

*Question: We have a dataset which includes 10 patients. Each patients have 3 occasions. Each occasion have 3 time points. At each time point, 3 sbp measurements are taken. If you want to write a for loop to calculate the variance of sbp at each time point of each occasion for each patient. How many levels of nesting will you need?

While-loops (example 3)

This type of looping structure is useful for situations when the number of iterations necessary for the task to finish is unknown. It should be noted that all for-loops can be re-written as as a while-loop, but the opposite is not true.

Example 3
*Purpose: Using trapezoidal rules, calculate AUC from 0 to 8hrs using PK profile simulated by one compartment model.

$AUC_i = \frac{(Cp_i - Cp_{i-1}) \times (t_i - t_{i-1})}{2}$

$AUC_{0-t} = \sum_{i=1}^{n} AUC_i$

```
#code for example 3

AUC <- 0
```

```r
i <- 1

while (i <= 8){
  AUC <- AUC + ((Conc[i+1] + Conc[i]) * (t[i+1] - t[i]))/2
  i = i + 1
}

paste0("AUC_0-", i-1, "=", AUC, sep=" ")
```

*Question: Write a function to calculate AUC from 0 to t for any t specified

Repeat

A modifed version of the while loop is repeat, which will keep running until it reaches a break statement.

Write example 3 with repeat:

```r
AUC <- 0
i <- 1

repeat{
  AUC <- AUC + ((Conc[i+1] + Conc[i]) * (t[i+1] - t[i]))/2
  if (i == 8) break
  i = i + 1
}

paste0("AUC_0-", i, "=", AUC, sep=" ")
```

Resources:

1. R for Data Science by Grlemund and Wickham (2017), online at https://r4ds.had.co.nz/
2. Advanced R by Wickham, online at http://adv-r.had.co.nz/

Session Info:

```r
sessionInfo()
```

```
## R version 3.6.1 (2019-07-05)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 18363)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
```

```
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
##  [1] compiler_3.6.1  magrittr_1.5    tools_3.6.1     htmltools_0.4.0
##  [5] yaml_2.2.1      Rcpp_1.0.4.6    stringi_1.4.6   rmarkdown_1.16
##  [9] knitr_1.28      stringr_1.4.0   xfun_0.14       digest_0.6.25
## [13] rlang_0.4.6     evaluate_0.14
```