Intro to R for Decision Modeling

Loops and Functions

SickKids and DARTH

11/2/2020

Change `eval` to `TRUE` if you want to knit this document.

This worksheet introduces the programming capabilities of `R`. Programming in `R` can make it easier to perform repetitive analyses such as sensitivity analysis and allow you to create your own functions to perform specific data analysis for example. This session is split into the following sections:

1. Loops
2. Functions
3. The 'apply' family

Throughout the course, we will demonstrate code and leave some empty *code chunks* for you to fill in. We will also provide solutions after the session.

Feel free to modify this document with your own comments and clarifications.

0. Load data into `R`

Before we begin this session, we need to load our Framingham dataset into `R`.

```
data <- read.csv('framingham.csv', header = TRUE)
```

We will modify a few important variables so that we can understand and visualize the results better.

Nested `ifelse` statements should be read from 'outside-in'. That is, start with the most outer `ifelse` statement and work your way in. For the purpose of working through these exercises, you do not have to fully understand the code in the below code chunk.

```
library(dplyr)
data1 <- data %>%
             mutate(SEX = ifelse(!is.na(SEX),
                                 ifelse(SEX == 1,
                                        'male', 'female'),
                              NA)) %>%
             mutate(PREVSTRK = ifelse(!is.na(PREVSTRK),
                                      ifelse(PREVSTRK == 1,
                                             'yes', 'no'),
                                   NA)) %>%
```

```
             mutate(PREVMI = ifelse(!is.na(PREVMI),
                                     ifelse(PREVMI == 1,
                                            'yes', 'no'),
                                  NA)) %>%
             mutate(DIABETES = ifelse(!is.na(DIABETES),
                                       ifelse(DIABETES== 1,
                                              'yes', 'no'),
                                    NA)) %>%
             mutate(CURSMOKE = ifelse(!is.na(CURSMOKE),
                                       ifelse(CURSMOKE== 1,
                                              'yes', 'no'),
                                    NA)) %>%
             mutate(BPMEDS = ifelse(!is.na(BPMEDS),
                                     ifelse(BPMEDS== 1,
                                            'yes', 'no'),
                                  NA))
```

1. Loops

When programming in `R`, we often wish to execute an operation or a combination of operations multiple times. If you are copying the same code multiple times it may be easier to use loops to iterate. Both for speed and readibility.

For creating loops we will be looking at `for`, `while`, `if` and `else` statements.

`for` loop

A `for` loop in R allows us to run a piece of code multiple times and is structured as follows.

```
for(value in sequence){
 do something
}
```

The `value` is a character (we often use `i`), and the `sequence` is a vector. For example, the following `for` loop prints (outputs) the sqaure of each element in the vector `Ages`:

```r
Ages <- c(5, 10, 12)
# For each age, square it
for(i in Ages){
  print(i ^ 2)
}
```

```
## [1] 25
## [1] 100
## [1] 144
```

Note that we used the `print()` function in the code above. Typically `R` does not produce any output in the console or R Markdown document when using a `for` loop. The `print()` function is used to give output from the loop.

There are two key functions that are often used in conjunction with `for` loops:

- The colon operator `:`, which creates a sequence from the number left of the `:` to the number right, increasing by 1 or -1, e.g.

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
4:1
```

```
## [1] 4 3 2 1
```

- The `length()` function that returns the number of elements in a vector or list.

```
length(Ages)
```

```
## [1] 3
```

A common structure of a `for` loop uses the colon operator and `length()` to perfom an operation for the i^{th} element in a vector,

```
# For each age, square it
for(i in 1:length(Ages)){
  print(Ages[i] ^ 2)
}
```

```
## [1] 25
## [1] 100
## [1] 144
```

The number of functions and mathematical operations that can be used within a `for` loop is very large. So, `for` loops can get very complex as you perform more complex operations.

EXERCISE 1 Print the BMI multiplied by 2 for each of the first ten subjects in the Framingham dataset.

```
# Your turn
for (i in 1:10) {
   print(data1$BMI[i] * 2)
}
```

```
## [1] NA
## [1] 57
## [1] 49.22
## [1] 62.34
## [1] 44.04
## [1] 51.44
## [1] 58.22
## [1] 43.96
## [1] 53.24
## [1] 49.54
```

Within the `for` loop it is also possible to assign values to different elements in a vector. In the following code, we calculate the mean age for males and females in Framingham dataset and save it to a vector called `mean_sex_age`. **Notice that *before* saving the mean age for males and females in the vector `mean_sex_age`, we have to create a vector called `mean_sex_age` so `R` knows where to save the calculations.** We usually create an empty vector of the right length to store our results, e.g.,

```
# a vector containing the categories of the SEX variable for us to iterate over later
sex_index <- c("male", "female")
# create an empty vector store the result of our for loop
mean_sex_age <- vector(length = length(sex_index))
# the names() function gives names to the elements in a vector
names(mean_sex_age) <- sex_index

# set up and run the for loop
for (i in 1:length(sex_index)) {
  mean_sex_age[i] <- mean(
  data1$AGE[data1$SEX == sex_index[i]]
  )
}

# display the results
mean_sex_age
```

```
##     male   female
## 60.34895 60.86940
```

if and else statement

An `if` statement is used to perform an action when a specific statement is true. The syntax for an `if` statement is:

```
if (expression) {
statement
}
```

If the `expression` is `TRUE`, the code in the `statement` is run. If the `expression` is `FALSE`, nothing happens.

EXERCISE 2 The following code prints "negative number" if `x` is negative, test this statement with different values for `x`.

```
# Your turn
x <- 5
if (x < 0) {
    print("negative number")
}
```

It is possible to follow `if` by `else` to create a `if...else` statement. The syntax is:

```
if (expression) {
statement1
} else {
statement2
}
```

The `else` component is optional and is only evaluated if the expression is FALSE.

ExERCISE 3 Create an `if...else` statement that prints "negative number" if `x` is less than 0 and "not a negative number" if `x` is not. Test your statement with different values for `x`.

```
# Your turn
x <- -10
if (x < 0) {
   print("negative number")
} else {
  print("not a negative number")
}
```

```
## [1] "negative number"
```

You can also have another `if` with an expression followed by `else` as follows:

```
if (expression 1) {
statement1
} else if (expression 2) {
statement2
}
```

ExERCISE 4 Create an `if...else` statement that prints "negative number" if `x` is less than 0 and "positive number" if `x` is greater than 0. Test your statement with different values for `x`.

```
# Your turn
x <- 10
if (x < 0) {
   print("negative number")
} else if (x > 0) {
  print("positive number")
}
```

```
## [1] "positive number"
```

We can combine `if...else` statements with `for` loops to create powerful tools for data analysis and data manipulation. For example,

```
y <- c(1, 3, 10, -1, 122, -9, -200)
for (i in 1:length(y)) {
  if (y[i] > 0) {
  print('positive')
  }
  else if (y[i] < 0) {
  print('negative')
  }
}
```

```
## [1] "positive"
## [1] "positive"
## [1] "positive"
```

```
## [1] "negative"
## [1] "positive"
## [1] "negative"
## [1] "negative"
```

The code above loops through the vector `y` and prints 'positive' if the `i`-th element is greater than 0 and 'negative' if it is less than 0.

EXERCISE 5 Store the subject id (`RANDID`) of all subjects over 70 years old in a vector and display it using `for`, `if` and `else`. Check your answer by using `filter()` and `select()` from the `dplyr` package.

```
# Your turn
age_70_index <- vector(length = length(data1$AGE))
for (i in 1:nrow(data1)) {
  if (data1$AGE[i] == 70) {
     age_70_index[i] <- 1
  }
}
age_70 <- data$RANDID[age_70_index == 1]
age_70[1:6]
```

```
## [1] 390449 491826 571377 610146 814971 968768
```

```
# check answer
data1 %>%
     filter(AGE == 70) %>%
     select(RANDID) %>%
     head()
```

```
##   RANDID
## 1 390449
## 2 491826
## 3 571377
## 4 610146
## 5 814971
## 6 968768
```

2. Functions

Almost all operations in `R` are achieved using *functions*. The general syntax for *functions* is `function_name(arguments)` We have seen many of them until now (e.g. `mean()`, `max()`, `quantile()`). These are pre-specified (built-in) functions in `R`.

In some settings, pre-specified functions in `R` are not sufficient for the analysis we would like to perform. In such cases, it is good practice to define your own functions to perform analyses. This can help you reproduce your analysis at a later date and repeat the same analysis on another dataset.

The generally syntax for writing your own function is as follows:

```
function_name <- function(arguments) {
   ...
   return(output)
}
```

The function can have a large number of `arguments`, or function inputs, separated by commas. The `output` from a function can have multiple elements that should be returned as a list.

```
function_name <- function(argument1, argument2, argument3) {
  ...
  return(list(output1, output2, output3)
}
```

Once a function has been defined using the code above, it can be used as a standard function in `R`.

For example, the following function calculates the mean of a vector:

```
# Write our own function to calculate the mean
mean_own <- function(data){
  mean_calc <- sum(data) / length(data)
  return(mean_calc)
}
mean_own(1:10)
```

```
## [1] 5.5
```

```
# test
mean(1:10)
```

```
## [1] 5.5
```

EXERCISE 6 Write a function that converts Fahrenheit to Celsius. The formula is (F-12) * 5 / 9. Test your function on 100 degrees Fahrenheit.

```
# Your turn
F_to_C <- function(temperature_F) {
  temperature_C <- (temperature_F - 32) * 5 / 9
  return(temperature_C)
}
# test
F_to_C(100)
```

```
## [1] 37.77778
```

EXERCISE 7 Write a function that calculates the square root of the sum of the squares of two numbers. Test your function on 3 and 4. The answer should be 5.

```
# Your turn
root_sum_of_squares <- function (x, y) {
  root_sum_squares <- sqrt(x ^ 2 + y ^ 2)
  return(root_sum_squares)
}

# test
root_sum_of_squares(3,4)
```

```
## [1] 5
```

```r
sqrt(3 ^ 2 + 4 ^ 2)
```

```
## [1] 5
```

You can write any number of operations between the left curly bracket { and the right curly bracket }. All these operations will be included in the function so you can perform the same operations on different datasets.

Functions can be combined with loops to perform complex operations but as these functions and loops become more complex, it is good practice to insert comments within your functions.

Below we create our own summary function that calculates the mean, standard deviation, minima and maxima of a vector. We use it on the age variable in the framingham dataset.

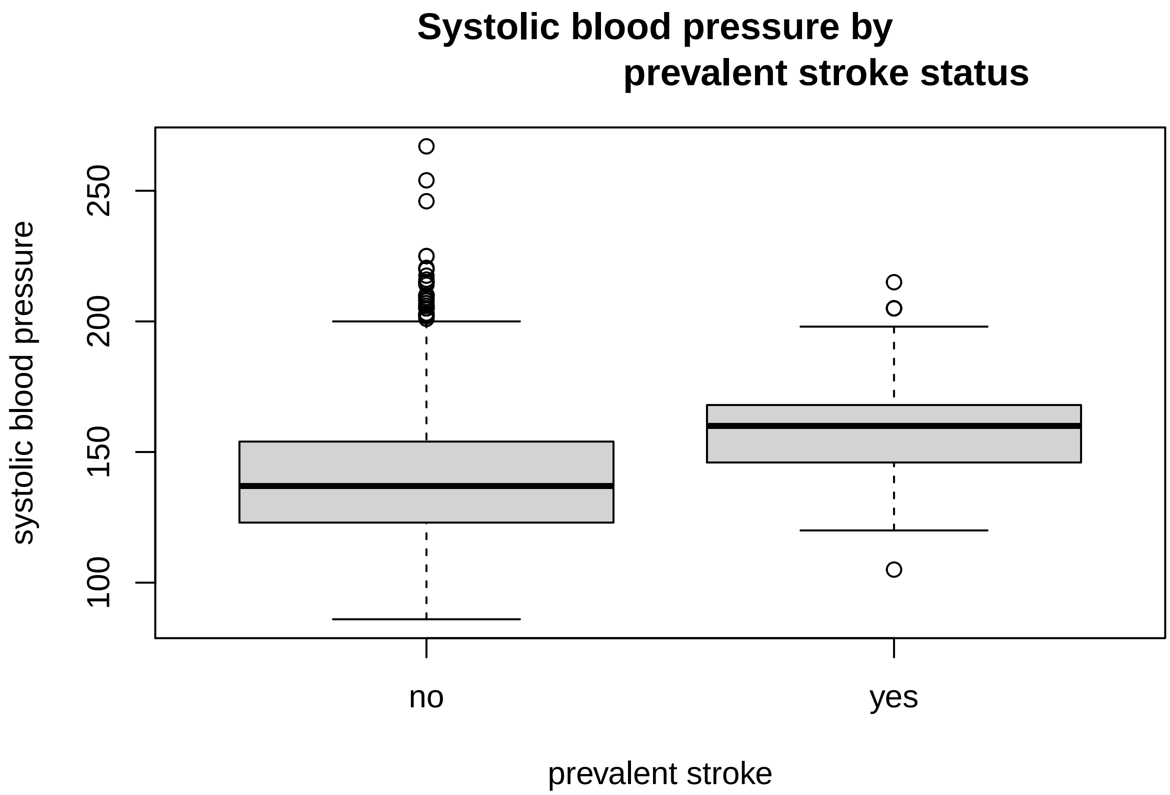
```r
# Write a function to summarize data
summary_own <- function(data) {
  # summary statistics
   mean_calc <- sum(data) / length(data)
   sd_calc <- sd(data)
   min_calc <- min(data)
   max_calc <-max(data)
   # name the summary statistics
   summaries <- c(mean_calc,sd_calc,min_calc,max_calc)
   names(summaries) <- c('mean', 'sd', 'min', 'max')
   return(summaries)
}

# use it on AGE and store the ouptut in a object
age_summary <- summary_own(data$AGE)
# print the results
age_summary
```

```
##      mean        sd       min       max
## 60.648177  8.296766 44.000000 81.000000
```

EXERCISE 8 Write a function that tests whether the mean systolic blood pressure (SYSBP) is different for those who have prevalent stroke (PREVSTRK) and those who do not and produces a box plot of systolic blood pressure stratified by prevalent stroke status with an appropriate title and labels. The function takes our framingham dataset as the only argument.

```r
# Your turn
test_sysbp <- function (data) {
                no_PREVSTRK_SYSBP <- data$SYSBP[data$PREVSTRK
                                                == 'no']
                PREVSTRK_SYSBP <- data$SYSBP[data$PREVSTRK
                                             == 'yes']
                boxplot(SYSBP ~ PREVSTRK, data = data,
                        main = 'Systolic blood pressure by
                                prevalent stroke status',
                        xlab = 'prevalent stroke',
                        ylab = 'systolic blood pressure')
                return(t.test(no_PREVSTRK_SYSBP, PREVSTRK_SYSBP))
}
test_sysbp(data1)
```




```
##
##  Welch Two Sample t-test
##
## data:  no_PREVSTRK_SYSBP and PREVSTRK_SYSBP
## t = -6.9149, df = 71.276, p-value = 1.654e-09
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -23.56921 -13.01937
## sample estimates:
## mean of x mean of y
##  139.8289  158.1232
```

3. The 'apply' family

R can be quite slow if you use a `for` loop to repeatedly use the same function across different elements of a dataset, or multiple datasets. R is much faster when we use a family of functions called the `apply` function. These functions, such as `sapply()` function or `lapply()` are used to repeatedly *apply* the same function to a dataset or a list of different datasets.

`sapply()` and `lapply()` take a list as its 1st argument and a function as its 2nd argument. It calls the function on each item in the list. The first *code chunk* demonstrates the `for` loop that we could use to calculate the summary statistics using our function for the age of both males and females.

```
sex_index <- c("male", "female")
# create an dummy matrix to store the summary statistics
summary_sex_age <- matrix(0, nrow = 4, ncol = length(sex_index))
# name the columns of the matrix
```

```r
colnames(summary_sex_age) <- sex_index
# use for loop to calculate the statistics
for (i in 1:length(sex_index)) {
  summary_sex_age[,i] <- summary_own(
  data1$AGE[data1$SEX == sex_index[i]]
  )
}
# display the results
summary_sex_age
```

```
##           male    female
## [1,] 60.348955 60.869403
## [2,]  8.191481  8.369054
## [3,] 45.000000 44.000000
## [4,] 80.000000 81.000000
```

The `sapply()` function can be used to avoid the `for` loop:

```r
# create a list containing the ages for males and females separately
age_list <- list(data1$AGE[data1$SEX == sex_index[1]],
                 data1$AGE[data1$SEX == sex_index[2]])
# use sapply to calculate the summary statistics
summary_sex_age1 <- sapply(age_list, summary_own)
# name the rows
colnames(summary_sex_age1) <- sex_index
# display the results
summary_sex_age1
```

```
##           male    female
## mean 60.348955 60.869403
## sd    8.191481  8.369054
## min  45.000000 44.000000
## max  80.000000 81.000000
```

We can also use the `apply()` to apply a function to the rows or columns (or both) of a data frame. While `lapply()` requires a list input and returns a list instead of a vector as we saw for `sapply()`.

EXERCISE 9 We have created a list of temperatures in Fahrenheit. Use the `sapply()` or `lapply()` functions to convert them to Celsius.

```r
# Your turn
F_temperatures <- list(50, 62, 81, 102, 157)
lapply(F_temperatures, F_to_C)
```

```
## [[1]]
## [1] 10
##
## [[2]]
## [1] 16.66667
##
## [[3]]
## [1] 27.22222
```

```
##
## [[4]]
## [1] 38.88889
##
## [[5]]
## [1] 69.44444
```

```
sapply(F_temperatures, F_to_C)
```

```
## [1] 10.00000 16.66667 27.22222 38.88889 69.44444
```