Session 3_3. Programming basics

Table of contents

Learning Objectives

- Understand three key programming concepts in R

Conditional expressions

Conditional expressions are one of the basic features of programming.

if-else

Here is a very simple example showing the general structure of an if-else statement. The basic idea is to print the reciprocal of `a` unless `a` is 0:

```
a <- 0

if (a != 0) {
  print(1/a)
} else {
  print("No reciprocal for 0.")
}
```

```
[1] "No reciprocal for 0."
```

Let's look at one more example using the US murders data frame, `murders`:

```
# if(!"dslabs" %in% installed.packages()) {install.packages("dslabs")}
library(dslabs)
murder_rate <- murders$total / murders$population * 100000
```

Here is a very simple example that tells us which states, if any, have a murder rate lower than 0.5 per 100,000. The if-else statement protects us from the case in which no state satisfies the condition.

```
ind <- which.min(murder_rate)

if (murder_rate[ind] < 0.5) {
  print(murders$state[ind])
} else {
  print("No state has murder rate that low")
}
```

```
[1] "Vermont"
```

If we try it again with a rate of 0.25, we get a different answer:

```
if (murder_rate[ind] < 0.25) {
  print(murders$state[ind])
} else {
  print("No state has a murder rate that low.")
}
```

```
[1] "No state has a murder rate that low."
```

ifelse

A related function that is very useful is `ifelse`. This function takes three arguments: a logical and two possible answers. If the logical is `TRUE`, the value in the second argument is returned and if `FALSE`, the value in the third argument is returned. Here is an example:

a	is_a_positive	answer1	answer2	result
0	FALSE	Inf	NA	NA
1	TRUE	1.00	NA	1.0
2	TRUE	0.50	NA	0.5
-4	FALSE	-0.25	NA	NA
5	TRUE	0.20	NA	0.2

```
a <- 0
ifelse(a > 0, 1/a, NA)
```

```
[1] NA
```

The function is particularly useful because it works on vectors. It examines each entry of the logical vector and returns elements from the vector provided in the second argument, if the entry is `TRUE`, or elements from the vector provided in the third argument, if the entry is `FALSE`.

```
a <- c(0, 1, 2, -4, 5)
result <- ifelse(a > 0, 1/a, NA)
```

This table helps us see what happened:

Here is an example of how this function can be readily used to replace all the missing values in a vector with zeros:

```
no_nas <- ifelse(is.na(na_example), 0, na_example)
sum(is.na(no_nas))
```

```
[1] 0
```

any, all

Two other useful functions are `any` and `all`. The `any` function takes a vector of logicals and returns `TRUE` if any of the entries is `TRUE`. The `all` function takes a vector of logicals and returns `TRUE` if all of the entries are `TRUE`. Here is an example:

```
z <- c(TRUE, TRUE, FALSE)
any(z)
```

```
[1] TRUE
```

```
all(z)
```

```
[1] FALSE
```

Functions

During data analysis, you will find yourself needing to perform the same operations over and over. For the repetitive tasks, it is much more efficient to write a function that performs the operation. Lots of common tasks are already available as functions (e.g., there is `mean()` function, so you don't need to write one for that, `sum(x)/length(x)`). However, you will encounter situations where the function does not already exist and there is a way to make your own function! A simple version of a function that computes the average can be defined like this:

```
avg <- function(x) {
  s <- sum(x)
  n <- length(x)
  s/n
}
```

```
avg(1:10)
```

```
[1] 5.5
```

In general, functions are objects, so we assign them to variable names with `<-`. The function `function` tells R you are about to define a function. The general form of a function definition looks like this:

```
my_function <- function(VARIABLE_NAME) {
  perform operations on VARIABLE_NAME and calculate VALUE
  VALUE
}
```

The functions you define can have multiple arguments as well as default values.

For-loops

For-loop iterates over a collection of objects, such as a vector, a list, a matrix, or a data frame, and apply the same set of operations on each item of a given data structure.

Basic for loop structure is:

```
for (item in list_of_items) {
  do_something(item)
}
```

For example, we can calculate masses from volumes using a for-loop. We include `print()` function to display values inside a loop or function.

```
## Looping by values
volumes <- c(1.6, 3, 8)
for (volume in volumes) {
  mass <- 2.65 * volume ^ 0.9
  print(mass)
}
```

In the above example, we loop by values. We can also perform looping by index:

```
## Looping by index
volumes <- c(1.6, 3, 8)
for (i in 1:length(volumes)) { # i stands for "index"
   mass <- 2.65 * volumes[i] ^ 0.9
   print(mass)
}
```

With looping by index, we can easily store the results calculated in the loop:

```
## Store results from the loop
masses <- vector(mode = "numeric", # the type of data we are going to store
                 length = length(volumes)) # the length of the vector
masses

for (i in 1:length(volumes)) {
   mass <- 2.65 * volumes[i] ^ 0.9
   masses[i] <- mass
}
masses
```

Although for-loops are an important concept to understand, in R *vectorization* is preferred over for-loops since it results in more concise, easy to read, and less error prone code. Most of R's functions are vectorized, meaning that the function will operate on all elements of a vector without needing to loop through and act on each element one at a time.

Functionals

Functionals are functions that help us apply the same function to each entry in a vector, matrix, data frame, or list. Check `sapply()`, `apply()`, and `replicate()`.

Tip

- `seq_along(x)`, where `x` is a vector, generates a vector of numbers from 1 to `length(x)`
- `seq_len(y)`, where `y` is a integer, generated a vector of numbers from 1 to `y`

Reference
http://rafalab.dfci.harvard.edu/dsbook-part-1/R/programming-basics.html
https://datacarpentry.org/semester-biology/materials/for-loops-R/