Loops and Functions workshop

Mauro & JEPA

November 27, 2019

Contents

The idea of this workshop is to help make your code more clean and enficient by reducing code duplication. This has three main benefits:

1. It's easier to see the intent of your code, because your eyes are drawn to what's different, not what stays the same.
2. It's easier to respond to changes in requirements. As your needs change, you only need to make changes in one place, rather than remembering to change every place that you copied-and-pasted the code.
3. You're likely to have fewer bugs because each line of code is used in more places.

Loops in R

"Looping", "cycling", "iterating" or just replicating instructions is an old practice that originated well before the invention of computers. It is nothing more than automating a multi-step process by organizing sequences of actions and grouping the parts that need to be repeated.

The main ideia is to look for regularities in your code. As a general rules-of-thumb: if you repeat the same line of code 3 or more times (just changing a few parameters) then you should use a loop!

For example, let's estimate the mean for all the columns of a given data frame:

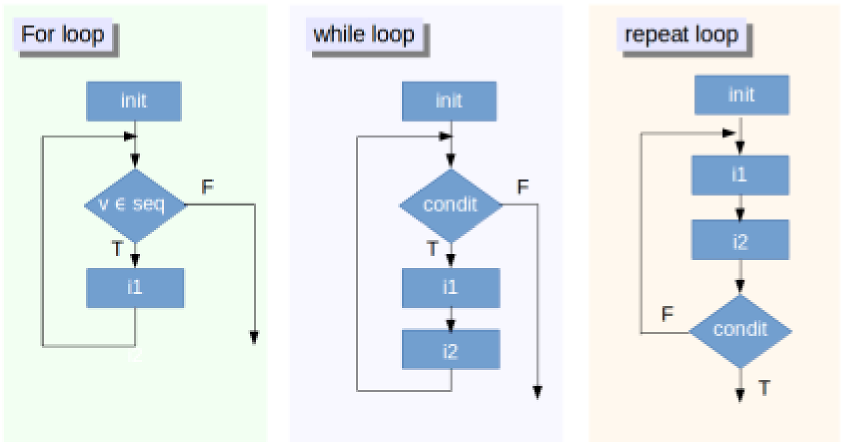
```r
# generates a dataset with 4 columns and 10 rows
# with numbers drawn from a normal distribution
set.seed(666) # for consistency
my_df = data.frame(a = rnorm(10), b = rnorm(10),
                   c = rnorm(10), d = rnorm(10))

# mean of the column a
mean(my_df$a)
# mean of the column b
mean(my_df$b)
# mean of the column c
```

```r
mean(my_df$c)
# mean of the column d
mean(my_df$a)
```

Can you find the repetitive pattern in this example? Did you notice the error in the code? This kind of error is pretty common when copying and pasting…

There are basically 3 kinds of loops:

- `for( variable in sequence ){ body }`; useful when one knows exactly how many times (ie, `sequence`) to run a chunk of code (ie, `body`).
- `while( condition ){ body }`; useful when one needs to run a chunk of code (ie, `body`) for as long as the `condition` is `TRUE`. (Really useful to run simulations that must meet a certain criteria!)
- `repeat{ body }`; similar to `while` (see below for details).



NOTE: Use `help(Control)` to see a summary of the main functions used in loops in R.

`for` loops

Every `for` loop has three components:

1. **output**: before you start the loop, you should allocate sufficient space for the output. This is very important for efficiency: if you grow the `for` loop at each iteration using `c()` (for example), your `for` loop will be very slow.
2. **sequence**: this determines what to loop over: each run of the `for` loop will assign `variable` to a different value from `sequence`.
3. **body**: this is the code that does the work. It's run repeatedly, each time with a different value for `variable`.

That's all there is to the `for` loop! Now, can you write a loop to estimate the mean for all the columns of a given data frame?

```r
set.seed(666) # for consistency
my_df = data.frame(a = rnorm(10), b = rnorm(10),
                   c = rnorm(10), d = rnorm(10))


output <- numeric( ncol(my_df) )      # 1. output
for (i in 1:ncol(my_df)) {            # 2. sequence
  output[i] <- mean(my_df[ , i])      # 3. body
}
```

```
output
# [1] -0.09607573  0.09895272 -0.76666574  0.25983311
```

NOTE: `for` loops are slow in R. When you feel confortable enough with loops, it's worthwhile to migrate to the `apply` functions. This is a family of functions to replace `for` loops that differ in the input and output: `apply` receives a table and returns a vector; `lapply` receives and returns a list; `sapply` receives a list and returns a vector; etc. I suggest this post for the interested.

`while` loops

Every `while` loop has three components:

1. **output**
2. **condition**: this determines when to stop: the loop will run for as long as the condition is `TRUE`.
3. **body**

Now can you re-write the previous loop using `while`?

```
set.seed(666) # for consistency
my_df = data.frame(a = rnorm(10), b = rnorm(10),
                   c = rnorm(10), d = rnorm(10))


output <- numeric( ncol(my_df) )    # 1. output
i = 1
while (i < 5) {                     # 2. condition
  output[i] <- mean(my_df[ , i])    # 3. body
  i = i + 1
}
output
# [1] -0.09607573  0.09895272 -0.76666574  0.25983311
```

`repeat` loops

Every `repeat` loop has three components:

1. **output**
2. **body**
3. **condition**: instead of appearing in the begging of the loop as in `while`, the condition is evaluated within the **body** of the loop. It uses auxialiary functions like `if( condition )` and `break()`.

Now can you re-write the previous loop using `repeat`?

```
set.seed(666) # for consistency
my_df = data.frame(a = rnorm(10), b = rnorm(10),
                   c = rnorm(10), d = rnorm(10))


output <- numeric( ncol(my_df) )    # 1. output
i = 1
repeat {
  output[i] <- mean(my_df[ , i])    # 2. body
  i = i + 1
  if (i < 5){                       # 3. condition
    break()
  }
}
output
# [1] -0.09607573  0.09895272 -0.76666574  0.25983311
```

NOTE: `repeat` loops are useful in cases where one wants (or needs) to run the code in **body** before evaluating the condition. Suppose one wants to find the best linear model to your dataset. One starts with a full model that contains all the 10 independent variables. Then you discard one of the variables and compares the fit of the models: if the full model is better, you stop; if the new model is better, you **repeat** the procedure.

Challenges

1. Can you write a loop that uses the dataset `USArrests` and estimates the total number of arrests for violent crimes (Murder+Assault+Rape) for each state?
2. Find the body mass index (BMI) for the `women` dataset. BMI is a measurement obtained by dividing a person's weight by the square of the person's height.
3. If you sum all the number of lynx caught on traps along the years (`lynx` dataset). How many years did it take to capture 50,000 lynx? [37 years]
4. Can you write a loop to estimate the 6th element of the Fibonnacci sequence?

Functions in R

In programming, you use functions to incorporate sets of instructions that you want to use repeatedly or that, because of their complexity, are better self-contained in a sub program and called when needed. A function is a piece of code written to carry out a specified task; it can or can not accept arguments or parameters and it can or can not return values.

Similarly to loops, functions can be used to reduce repetition (looking for regularities in your code). The main advantage of functions, though, is to find tasks that can be applied in different contexts. As a general rules-of-thumb: if you believe you'll have to perform the same task in more than one script/project, then you should write a function!

Components of a function

All R functions have three parts:

1. **body**, the code inside the function.
2. **formals**, the list of arguments which controls how you can call the function.
3. **environment**, the "map" of the location of the function's variables (usually, the package the function comes from).

NOTE: You can print a function in R by calling it without the parentheses (try `data.frame` or `lm`).

My first functions!

To write your very own function you'll have to use the function `function`! The sintax is as follow: `function_name = function( arguments ){ body }`.

When building functions I always consider two main aspects:

1. **Input data**, what does the function receives: a data.frame, a list, a vector, etc. One can test (using `if` statements) and convert the data if necessary. Inputs (and parameters in general) are not necessary for a function!
2. **Output data**, what does my function returns? To choose the best option, it's good to consider what you want to do after the function: plot the results, fit a model, save as csv, etc. Functions in R can only return ONE object. To return multiple objects, one must create a list.

Let's create a function that receives two numbers and returns the sum of their squared values.

```
sum_of_squared = function(x, y){
  out = x*x + y*y
  return(out)
}
sum_of_squared(3, 2) # 13
```

What happens if you call the function without an input? Is there a way around it?

```
sum_of_squared = function(x = 3, y = 2){
  out = x*x + y*y
  return(out)
}
sum_of_squared()  # 13
sum_of_squared(4) # 20
```

TIP1: Good functions are **short**, perform a **single operation**, have **intuitive** names, and are as **general** as possible!

TIP2: Don't start to write your functions directly on R. It's better to think about the algorithm first! (The same way one must think of the conceptual variables before the operational ones.) A good pratice for beginners is to write the comments before the actual line of code!

Organizing your functions!

As mentioned earlier, one of the main advantages of functions is to use it in other analyses. No one wants to copy-and-paste your function between different scripts - it'd have the same problems as any other repetition! So you must save your functions in a differente script, separated from your analysis!

There are two main strategies to organize your functions (but a quick google will find you thousands of other options!):

- create a file with all the functions. Then you always have your functions in one place, but if you want to change a specific one you'll have to look for it. To load the file use `source( "path_to_file" )`.
- create a separate file to each function. It's easier to find each function, but you have a lot of files. One can call each function individually, like indicated above - which sucks! Another option is to put all the files in a specific folder (that has the files with the functions and nothing else), then use `sourceDirectory( "path_to_folder" )` to load all functions.

Challenges

1. Can you write a function to estimate the Nth element of the Fibonnacci sequence?

2. Write down a function that creates a data frame where each entry is the ratio between its row and its column. For example, a data frame with 3 rows and 5 columns would look like:

```
##   X1  X2        X3   X4  X5
## 1  1 0.5 0.3333333 0.25 0.2
## 2  2 1.0 0.6666667 0.50 0.4
## 3  3 1.5 1.0000000 0.75 0.6
```

3. Can you rescale all the columns of the `anscombe` dataset to vary between 0 and 1?

```
data(anscombe)
# rescale the first column to vary between 0 and 1
anscombe[ , 1] = ( anscombe[ , 1] - min(anscombe[ , 1]) ) / ( max(anscombe[ , 1]) - min(anscombe[ , 1])
# rescale the second column to vary between 0 and 1
anscombe[ , 2] = ( anscombe[ , 2] - min(anscombe[ , 2]) ) / ( max(anscombe[ , 2]) - min(anscombe[ , 2])
# rescale the third column to vary between 0 and 1
anscombe[ , 3] = ( anscombe[ , 3] - min(anscombe[ , 3]) ) / ( max(anscombe[ , 3]) - min(anscombe[ , 3])
# ....
```

4. Make a function that generates a random integer between 0 and 9.

Auxiliary functions

Now let's go over some functions that you might need!

`if` / `else` statements

These functions help decision making.

The syntax of `if` statement is: `if(condition) { body }`. Basically, `if` the **condition** is `TRUE` do **body**.

`if` statements might be followed by `else{ body }` statements that specify what to do in case the **condition** is `FALSE`. The sintax is:

```
if (condition) {
  body1
} else {
  body2
}
```

R has a couple of logical operators that you'll need to build your **condition**:

- `==`, is equal to?
- `>` / `<`, is bigger / smaller than?
- `>=` / `<=`, is bigger /smaller than or equal to?
- `!`, changes the signal of the condition. Eg, `(5 >= 1)` is `TRUE`, but `!(5 >= 1)` is `FALSE`.
- `!=`, is different from?
- `a %in% b`, is `a` an element of `b`?
- `is.na(obj)` / `is.null(obj)`, is `obj` equal to `NA` / `NULL`? (There's a family of function `is.`, to see the methods available run `methods("is")`).

One can build a complex **condition** (ie, more than one comparison) using the `|` and `&` operators, that indicate the logical operations OR and AND. For example, to test if x equals 1 AND y equals 2: `if(x == 1 & y == 2){body}`.

`next` statements

`next()` skips the current repetition of the loop and goes to the beggining of the loop. The sintax might look like this:

```
for(i in 1:10){
  if (i == 3) { # skips the 3rd interaction
    next()
  }
  body
}
```

`break` statements

`break()` stops the loop and continues the rest of the code.

NOTE: In case of nested loops (loop within loop whithin loop…), it `breaks` the inner loop (the last one called)!

`stop` statements

`stop( message )` stops the function and returns the error **message** (OPTIONAL).

NOTE: Do not mix `stop` and `break`. The former is used within **functions**, while the latter is used in **loops**.

`try` function and `try-error` class

Sometimes one of the iterations of a loop (or a function) will encounter an error (eg, `6/"banana"`), then it'll immediately stop and return an error message. The function `try( expression )` exists to help you handle such cases. When the **expression** being evaluated by `try` fails, it returns an object of class `try-error`. For example:

```
# list with 5 elements: 4 numeric and 1 character
my_list = list(a = 33, b = -9.3, c = "banana", d = 903, e = 7)
output = numeric(5)
for(i in 1:5){
  output[i] = 6 / my_list[[i]] # fails
}
output

for(i in 1:5){
  aux = try(6 / my_list[[i]])
  if(class(aux) != "try-error"){
    output[i] = aux
  }
}
output
```