Extra R programming 1

Jingwei Xiong

Contents:

- Conditional Programming: if
- iteration: for
- Write the R function
- Some examples Link to the examples

Conditional Programming in R Using if

- if is a powerful tool for implementing conditional programming in R
- The basic syntax of if in R is as follows:

```
if (condition) {
  # code to execute if the condition is TRUE
}else {
  # code to execute if the condition is FALSE
}
```

- The condition can be any logical expression that evaluates to TRUE or FALSE
- If the condition is TRUE, the code within the braces will be executed; otherwise, it will be skipped, and the else code within the braces will be executed.

- If - else if - else

```r
x <- 10

if (x < 5) {
  print("x is less than 5")
} else if (x > 15) {
  print("x is greater than 15")
} else {
  print("x is between 5 and 15")
}
```

```
[1] "x is between 5 and 15"
```

In this example, if the value of x is less than 5, the message "x is less than 5" will be printed. If the value of x is greater than 15, the message "x is greater than 15" will be printed. If neither of these conditions is true (i.e., if x is between 5 and 15), the message "x is between 5 and 15" will be printed.

For Loop in R

- For Loop is a programming construct that allows iterating over a sequence of values, such as a vector or a list.
- The general syntax of the for loop in R is as follows:

```
for (variable in sequence) {
  statements
}
```

- In the for loop, variable is a new variable that takes on each value in sequence in turn, and statements is a sequence of R code to execute each time the loop is executed.

- In lecture 15 we will see how important for loop is in R simulation.

- Here's an example of using a for loop to calculate the sum of the first ten integers:

```r
sum = 0
for (i in 1:10) {
  sum = sum + i
}
print(sum)
```

```
[1] 55
```

- In this example, i takes on the values of 1, 2, 3, ..., 10 in turn, and each time the loop is executed, the value of i is added to sum.
- After the loop is finished, the value of sum is printed to the console, which is the sum of the first ten integers, i.e., 55.
- The for loop is useful when we need to repeat a block of code for a fixed number of times or when we need to iterate over a sequence of values.

What is function

An R function is a self-contained block of code that performs a specific task and can be executed repeatedly with different inputs.

Functions are defined using the **function** keyword and have a specific structure, including a name, a set of arguments, and a code block.

The basic structure of an R function looks like this:

```r
function_name = function(arg1, arg2, ...){
  # code block
  return(output)
}
```

```r
function_name = function(arg1, arg2, ...){
  # code block
  return(output)
}
```

- **function_name** is the name of the function and can be any valid R object name.
- **arg1, arg2, ...** are the arguments or inputs of the function, which can be used to pass data into the function and specify how it should behave.
- The code block is the set of instructions that the function will execute when it is called.
- The return statement is used to specify the output or result of the function, which can be used in other parts of your code. If there is no return statement, the last line will be returned.

Benefits of using functions

One of the best ways to improve your reach as a data scientist is to write functions. Functions allow you to automate common tasks in a more powerful and general way than copy-and-pasting. Writing a function has three big advantages over using copy-and-paste:

1. You can give a function an evocative name that makes your code easier to understand.

2. As requirements change, you only need to update code in one place, instead of many.

3. You eliminate the chance of making incidental mistakes when you copy and paste (i.e. updating a variable name in one place, but not in another).

When should you write a function?

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). For example, take a look at this code. What does it do?

```
df = data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a = (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b = (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c = (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d = (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

You might be able to puzzle out that this rescales each column to have a range from 0 to 1. But did you spot the mistake? I made an error when copying-and-pasting the code for **df$b**: I forgot to change an **a** to a **b**. Extracting repeated code out into a function is a good idea because it prevents you from making this type of mistake.

To write a function you need to first analyse the code. How many inputs does it have?

```
(df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
```

This code only has one input: df$a. (If you’re surprised that TRUE is not an input, you can explore why in the exercise.)

To make the inputs more clear, it's a good idea to rewrite the code using temporary variables with general names. Here this code only requires a single numeric vector, so I'll call it x:

```
x = df$a
(x - min(x, na.rm = TRUE)) / (max(x, na.rm = TRUE) - min(x, na.rm = 
```

There is some duplication in this code. We're computing the range of the data three times, so it makes sense to do it in one step:

```
x = df$a
xmin = min(x, na.rm = TRUE)
xmax = max(x, na.rm = TRUE)
(x - xmin) / (xmax - xmin)
```

Or use the `range` function:

```
rng <- range(x, na.rm = TRUE)
(x - rng[1]) / (rng[2] - rng[1])
```

Remarks: **Pulling out intermediate calculations into named variables** is a **good practice** because it makes it more clear what the code is doing.

Now that I’ve simplified the code, and checked that it still works, I can turn it into a function:

```r
rescale01 <- function(x) {
  rng = range(x, na.rm = TRUE)
  return(
    (x - rng[1]) / (rng[2] - rng[1])
    )
}
rescale01(c(0, 5, 10))
```

```
[1] 0.0 0.5 1.0
```

There are three key steps to creating a new function:

1. You need to pick a name for the function. Here I’ve used rescale01 because this function rescales a vector to lie between 0 and 1.

2. You list the inputs, or arguments, to the function inside function. Here we have just one argument. If we had more the call would look like function(x, y, z).

3. You place the code you have developed in body of the function, a { block that immediately follows function(...).

Note the overall process: I only made the function after I'd figured out how to make it work with a simple input. **It's easier to start with working code and turn it into a function**; it's harder to create a function and then try to make it work.

At this point it's a good idea to check your function with a few different inputs:

```r
rescale01(c(-10, 0, 10))
```

```
[1] 0.0 0.5 1.0
```

```r
rescale01(c(1, 2, 3, NA, 5))
```

```
[1] 0.00 0.25 0.50   NA 1.00
```

Now we can simplify the original example now that we have a function:

```
df$a = rescale01(df$a)
df$b = rescale01(df$b)
df$c = rescale01(df$c)
df$d = rescale01(df$d)
```

Compared to the original, this code is easier to understand and we've eliminated one class of copy-and-paste errors. There is still quite a bit of duplication since we're doing the same thing to multiple columns. We'll learn how to eliminate that duplication in **iteration**.

Another advantage of functions is that if our requirements change, we only need to make the **change in one place**. For example, we might discover that some of our variables include infinite values, and rescale01() fails:

```
x <- c(1:10, Inf)
rescale01(x)
```

```
 [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

Because we've extracted the code into a function, we only need to make the fix in one place:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  #because when finite = TRUE, range will return the max which is no
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(x)
```

```
 [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
 [8] 0.7777778 0.8888889 1.0000000       Inf
```

This is an important part of the "**do not repeat yourself**" (or DRY) principle. The more repetition you have in your code, the more places you need to remember to update when things change (and they always do!), and the more likely you are to create bugs over time.

Function are for humans and computers

It’s important to remember that functions are not just for the computer, but are also for humans. R doesn’t care what your function is called, or what comments it contains, but **these are important for human readers**.

The name of a function is important. Ideally, the name of your function will be short, but clearly evoke what the function does. That’s hard! But **it’s better to be clear than short**, as RStudio’s autocomplete makes it easy to type long names.

- Generally, function names should be verbs, and arguments should be nouns.

- There are some exceptions: nouns are ok if the function computes a very well known noun (i.e. `mean()` is better than `compute_mean()`), or accessing some property of an object (i.e. `coef()` is better than `get_coefficients()`).

- A good sign that a noun might be a better choice is if you’re using a very broad verb like "get", "compute", "calculate", or "determine".

- Use your best judgement and don’t be afraid to rename a function if you figure out a better name later.

```
# Too short
f()   # (Don't run this code because we call functions not defined)

# Not a verb, or descriptive
my_awesome_function()

# Long, but clear
impute_missing()
collapse_years()
```

If your function name is composed of multiple words, I recommend using “**snake_case**”, where each lowercase word is separated by an underscore.

camelCase is a popular alternative. It doesn’t really matter which one you pick, the important thing is to be consistent: pick one or the other and stick with it.

R itself is not very consistent, but there’s nothing you can do about that. Make sure you don’t fall into the same trap by making your code as consistent as possible.

```
# Never do this!
col_mins <- function(x, y) {}
rowMaxes <- function(y, x) {}
```

If you have a family of functions that do similar things, make sure they have consistent names and arguments. Use a **common prefix** to indicate that they are connected. That's better than a common suffix because autocomplete allows you to type the prefix and see all the members of the family.

```r
# Good
input_select()
input_checkbox()
input_text()

# Not so good
select_input()
checkbox_input()
text_input()
```

Where possible, avoid **overriding existing functions and variables**. It's impossible to do in general because so many good names are already taken by other packages, but avoiding the most common names from base R will avoid confusion.

```r
# Don't do this!
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

- Use comments, lines starting with #, to explain the “why” of your code.
- You generally should avoid comments that explain the “what” or the “how”. If you can’t understand what the code does from reading it, you should think about how to rewrite it to be more clear.
- Do you need to add some intermediate variables with useful names? Do you need to break out a subcomponent of a large function so you can name it?
- However, your code can never capture the reasoning behind your decisions: why did you choose this approach instead of an alternative? What else did you try that didn’t work?

*It’s a great idea to capture that sort of thinking in a comment.

Another important use of comments is to break up your file into easily readable chunks. Use long lines of - and = to make it easy to spot the breaks.

```
# Load data --------------------------------------

# Plot data --------------------------------------
```

https://www.dataquest.io/blog/rstudio-tips-tricks-shortcuts/ for R studio shortcuts

Example 1: weighted summaries

```
wt_mean <- function(x, w) {
  sum(x * w) / sum(w)
}
wt_var <- function(x, w) {
  mu <- wt_mean(x, w)
  sum(w * (x - mu) ^ 2) / sum(w)
}
wt_sd <- function(x, w) {
  sqrt(wt_var(x, w))
}
```

Example 2: R as an integration calculator

Suppose we want to find out:

$$\int_0^\infty \frac{1}{(x+1)\sqrt{x}}\,dx$$

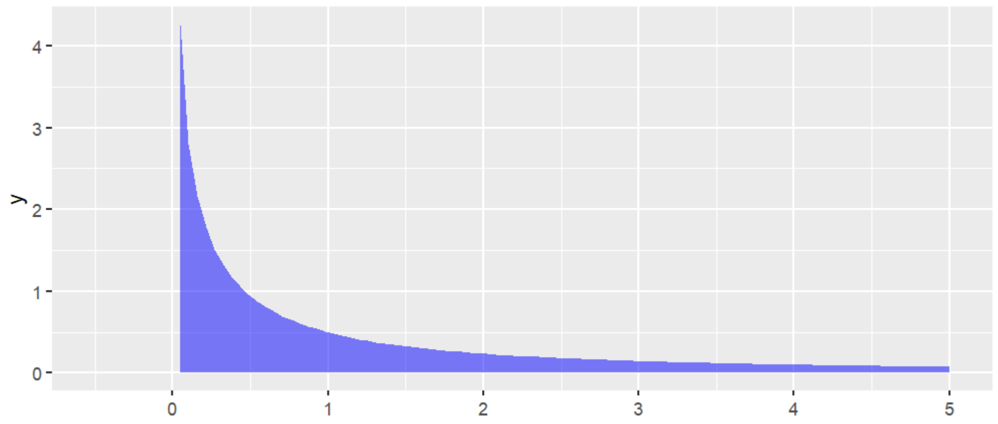
```r
integrand = function(x) {
  1/((x+1)*sqrt(x))
}
integrate(integrand, lower = 0, upper = Inf)
```

```
3.141593 with absolute error < 2.7e-05
```

Using previous knowledge, we can draw the density plot and integrate and show the result:

```r
library(ggplot2)
integrand = function(x) {
  1/((x+1)*sqrt(x))
}

base = ggplot() + xlim(-0.5, 5)
base + stat_function(fun =  integrand ,
    geom = "area", fill = "blue", alpha = 0.5)
```



```r
integrate(integrand, lower = 0, upper = Inf)
```

```
3.141593 with absolute error < 2.7e-05
```

Example 3: Math 24

Brute force: We examine all possible permutation of the numbers and all possible permutation of the operators.

```r
math24 <- function(nums) {
  # nums should be a numeric vector of length 4
  # returns a character string of the solution, or NULL if no solution exists

  # all possible permutations of the four numbers,
  # it will return a list, each element is one permutation
  perms <- unique(combinat::permn(nums))

  # all possible combinations of operations (+, -, *, /)
  # each combination is one row of a data.frame
  ops <- expand.grid(replicate(3, c("+", "-", "*", "/"), simplify = FALSE))

  # iterate through all permutations and operations
  for (i in 1:length(perms)) {
    for (j in 1:nrow(ops)) {
      a <- perms[[i]][1]
      b <- perms[[i]][2]
      c <- perms[[i]][3]
      d <- perms[[i]][4]
      op1 <- ops[j, 1]
      op2 <- ops[j, 2]
      op3 <- ops[j, 3]

      # evaluate the expression
      expr <- paste0("(", a, op1, b, ")", op2, "(", c, op3, d, ")")
      result <- eval(parse(text = expr))

      # check if result is 24 (with some tolerance for floating point error)
      if (abs(result - 24) < 1e-10) {
        return(expr)
      }
    }
  }
  return(NULL)  # no solution found
}
```

```r
# all possible permutations of the four numbers,
# it will return a list, each element is one permutation
nums = c(4, 7, 8, 8)
(perms <- unique(combinat::permn(nums)))
```

```
[[1]]
[1] 4 7 8 8

[[2]]
[1] 4 8 7 8

[[3]]
[1] 8 4 7 8

[[4]]
[1] 8 4 8 7

[[5]]
[1] 4 8 8 7

[[6]]
[1] 8 8 4 7

[[7]]
[1] 8 8 7 4

[[8]]
[1] 8 7 8 4

[[9]]
[1] 8 7 4 8
```

```r
# all possible combinations of operations (+, -, *, /)
# each combination is one row of a data.frame
head(ops <- expand.grid(replicate(3, c("+", "-", "*", "/"), simplify
```

```
   Var1 Var2 Var3
1     +    +    +
2     -    +    +
3     *    +    +
4     /    +    +
5     +    -    +
6     -    -    +
7     *    -    +
8     /    -    +
9     +    *    +
10    -    *    +
```

```r
# iterate through all permutations and operations
for (i in 1:length(perms)) {
  for (j in 1:nrow(ops)) {
    a <- perms[[i]][1]
    b <- perms[[i]][2]
    c <- perms[[i]][3]
    d <- perms[[i]][4]
    op1 <- ops[j, 1]
    op2 <- ops[j, 2]
    op3 <- ops[j, 3]

    # evaluate the expression
    expr <- paste0("(", a, op1, b, ")", op2, "(", c, op3, d, ")")
    result <- eval(parse(text = expr))

    # check if result is 24 (with some tolerance for floating point error)
    if (abs(result - 24) < 1e-10) {
      return(expr)
    }
  }
}
```

```r
length(perms)
```

```
[1] 12
```

```r
nrow(ops)
```

```
[1] 64
```

```r
i = 3
j = 4

a <- perms[[i]][1]
b <- perms[[i]][2]
c <- perms[[i]][3]
d <- perms[[i]][4]
op1 <- ops[j, 1]
op2 <- ops[j, 2]
op3 <- ops[j, 3]

# evaluate the expression
expr <- paste0("(", a, op1, b, ")", op2, "(", c, op3, d, ")")
expr
```

```
[1] "(8/4)+(7+8)"
```

```r
result <- eval(parse(text = expr))
result
```

```
[1] 17
```

```r
math24(c(4, 7, 8, 8))
```

```
[1] "(8*7)-(8*4)"
```

```r
math24(c(3, 9, 8, 2))
```

```
[1] "(8/2)*(9-3)"
```

```r
math24(c(1, 2, 10, 3))
```

```
[1] "(1+3)+(2*10)"
```

```r
math24(c(1, 1, 1, 4))
```

```
NULL
```

```r
# returns NULL (no solution)
```

Conditional execution

An if statement allows you to conditionally execute code. It looks like this:

```r
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

Here's a simple function that uses an if statement. The goal of this function is to return a logical vector describing whether or not each element of a vector is named.

```r
has_name <- function(x) {
  nms <- names(x)
  if (is.null(nms)) {
    rep(FALSE, length(x))
  } else {
    !is.na(nms) & nms != ""
    # only name is not NA or blank
  }
}
```

```r
has_name <- function(x) {
  nms <- names(x)
  if (is.null(nms)) {
    rep(FALSE, length(x))
  } else {
    !is.na(nms) & nms != ""
    # only name is not NA or blank
  }
}
```

This function takes advantage of the standard return rule: a function returns the last value that it computed. Here that is either one of the two branches of the if statement.

Conditions

The condition must evaluate to either TRUE or FALSE. If it's a vector, you'll get a warning message; if it's an NA, you'll get an error. Watch out for these messages in your own code:

```
if (c(TRUE, FALSE)) {}
#> Error in if (c(TRUE, FALSE)) {: the condition has length > 1

if (NA) {}
#> Error in if (NA) {: missing value where TRUE/FALSE needed
```

You can use || (or) and && (and) to combine multiple logical expressions.

These operators are "short-circuiting": as soon as || sees the first TRUE it returns TRUE without computing anything else. As soon as && sees the first FALSE it returns FALSE.

You should never use | or & in an if statement: these are vectorised operations that apply to multiple values (that's why you use them in filter()).

Multiple conditions

You can chain multiple if statements together:

```
if (this) {
  # do that
} else if (that) {
  # do something else
} else {
  #
}# QUIZ how to write it to determine A,B,C,D,E based on score?
```

But if you end up with a very long series of chained if statements, you should consider rewriting. One useful technique is the switch() function. It allows you to evaluate selected code based on position or name.

But if you end up with a very long series of chained if statements, you should consider rewriting. One useful technique is the switch() function. It allows you to evaluate selected code based on position or name.

```r
calculator = function(x, y, op) {
  switch(op,
     plus = x + y,
     minus = x - y,
     times = x * y,
     divide = x / y,
     stop("Unknown op!") # if the op is not any of them
   )
}
```