

Chapter 3

3. Writing functions

Objectives

After completing this unit, students will be able to:

- Understand Grouping, loops and conditional execution in R
- Write simple functions and programs in R

3.1. Grouping, loops and conditional execution

Grouped expressions

R is an expression language in the sense that its only command type is a function or expression which returns a result. Commands may be grouped together in braces, {expr 1, . . . , expr m}, in which case the value of the group is the result of the last expression in the group evaluated.

Control statements

Control structures in R allow you to control the flow of execution of a series of R expressions. Basically, control structures allow you to put some “logic” into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly.

Commonly used control structures are:

- `if` and `else`: testing a condition and acting on it
- `for`: execute a loop a fixed number of times
- `while`: execute a loop while a condition is true
- `repeat`: execute an infinite loop (must break out of it to stop)
- `break`: break the execution of a loop
- `next`: skip an iteration of a loop

Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions.

Conditional execution: if statements/ if-else

The **if-else** combination is probably the most commonly used control structure in R (or perhaps any language). This structure allows you to test a condition and act on it depending on whether it's true or false.

The language has available a conditional construction of the form

```
if (expr_1) expr_2 else expr_3
```

where `expr_1` must evaluate to a logical value and the result of the entire expression is then evident.

That is,

```
if(condition) {
  ## do something
}
else {
  ## do something else
}
```

Example:

```
y <- if(x > 3) {
  10
} else {
  0
}
```

if...else Ladder/nested if...else

The if...else ladder (if...else...if) statement allows you execute a block of code among more than 2 alternatives

The syntax of if...else statement is:

```
if ( test_expression1) {
statement1
} else if ( test_expression2) {
statement2
} else if ( test_expression3) {
statement3
```

```

} else {
statement4
}

```

Only one statement will get executed depending upon the test_expressions.

Example of nested if...else

```

x <- 0
if (x < 0) {
  print("Negative number")
} else if (x > 0) {
  print("Positive number")
} else
  print("Zero")
[1] "Zero"

```

A vectorized version of the if/else construct, the **ifelse** function. This has the form

ifelse(condition, a, b)

Example: ifelse() function

```

x <- 1:10 # Creates sample data
ifelse(x<5 | x>8, x, 0)
[1] 1 2 3 4 0 0 0 0 9 10

```

Repetitive execution: for loops, repeat and while

Looping: is the repeated evaluation of a statement or block of statements.

R has three statements that provide explicit looping. They are **for**, **while** and **repeat**.

The two built-in constructs, **next** and **break**, provide additional control over the evaluation. Each of the three statements returns the value of the last statement that was evaluated.

There are two statements that can be used to explicitly control looping. They are **break** and **next**.

- The **break** statement causes an exit from the innermost loop that is currently being executed.

- The **next** statement immediately causes control to return to the start of the loop.

for Loops

In R, for loops take an iterator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

The syntax of the **for** loop is

```
for(name in vector) expr
```

Example

```
for(i in 1:10) {  
  print(i)  
}
```

Example -The following four loops all have the same behavior.

1.

```
x <- c("a", "b", "c", "d")  
for(i in 1:4) {  
  ## Print out each element of 'x'  
  print(x[i])  
}
```

2.

```
## Generate a sequence based on length of 'x'  
> for(i in seq_along(x)) {  
+   print(x[i])  
+ }
```

3.

```
> for(letter in x) {  
+   print(letter)  
+ }
```

4.

```
# For one line loops, the curly braces are not strictly necessary.  
> for(i in 1:4) print(x[i])
```

The output is

```
[1] "a"  
[1] "b"  
[1] "c"  
[1] "d"
```

Nested for loops

Nested loops are commonly needed for multidimensional or hierarchical data structures (e.g. matrices, lists). for loops can be nested inside of each other.

```
x <- matrix(1:6, 2, 3)
for(i in seq_len(nrow(x))) {
  for(j in seq_len(ncol(x))) {
    print(x[i, j])
  }
}
```

while Loops

while loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits.

while (Condition) expr

where condition is evaluated and if its value is TRUE then expr is evaluated. This process continues until condition evaluates to FALSE.

Example:

```
> count <- 0
> while(count < 10) {
  print(count)
  count <- count + 1
}
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

Remark: While loops can potentially result in infinite loops if not written properly. Use with care!

repeat Loops

repeat initiates an infinite loop right from the start. These are not commonly used in statistical

or data analysis applications but they do have their uses. The only way to exit a repeat loop is to call break.

repeat expr

Example

```
z=0
repeat{
  z=z+1
  print(z)
  if (z>10) break()
}
```

next, break

next is used to skip an iteration of a loop.

Example

```
for(i in 1:100) {
  if(i <= 20) {
    ## Skip the first 20 iterations
    next
  }
  ## Do something here
}
```

break is used to exit a loop immediately, regardless of what iteration the loop may be on.

Example

```
for(i in 1:100) {
  print(i)
  if(i > 20) {
    ## Stop loop after 20 iterations
    break
  }
}
```

3.2. Writing functions

Writing functions is a core activity of an R programmer. It represents the key step of the transition from a mere “user” to a developer who creates new functionality for R. Functions are often used to encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions. Functions are also often written when code must be shared with others or the public.

Functions in R are “first class objects”, which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions. This is very handy for the various apply functions, like `lapply()` and `sapply()`.
- Functions can be nested, so that you can define a function inside of another function

If you’re familiar with common language like C, these features might appear a bit strange. However, they are really important in R and can be useful for data analysis.

Functions are defined using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class “function”.

A function is defined by an assignment of the form

```
> name <- function(arg_1, arg_2, ...) expression
```

The `expression` is an R expression, (usually a grouped expression), that uses the arguments, `arg i`, to calculate a value

A call to the function then usually takes the form `name(arg_1, arg_2, ...)` and may occur anywhere a function call is legitimate.

Note: Functions arguments can be specified by name or by position in the argument list

3.3. Simple examples

1. Write a function to calculate the two sample t-statistic, showing “all the steps”(assuming equality of variance)

```

twosamp <- function(x1, x2) {
  n1 <- length(x1); n2 <- length(x2)
  xbar1 <- mean(x1); xbar2 <- mean(x2)
  s1 <- var(x1); s2 <- var(x2)
  sp <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
  tst <- (xbar1 - xbar2)/sqrt(sp*(1/n1 + 1/n2))
  tst;round(tst,4)
}
## Here is the data
A<-c(79.98, 80.04, 80.02, 80.04, 80.03, 80.03, 80.04,
79.97, 80.05, 80.03, 80.02, 80, 80.02)
B<-c(80.02, 79.94, 79.98, 79.97, 79.97, 80.03, 79.95,
79.97)
twosamp(A,B)

```

2. Write a function to calculate the variance of a given observation

```

standard<-function(x) {
  n<-length(x)
  a<-mean(x)
  sum<-0
  for(i in 1:n){
    sum<-sum+((x[i]-a)^2)
  }
  variance<-sum/(n-1)
  print(variance)
}
## here is an example, the data is given below
x<-c(80.02, 79.94, 79.98, 79.97, 79.97, 80.03, 79.95,
79.97)
standard(x)

```

3.4. Binary operators

Most of the operators that we use in R are binary operators (having two operands). Hence, they are infix operators, used between the operands. Actually, these operators do a function call in the background.

For example, the expression `a+b` is actually calling the function `'+'()` with the arguments `a` and `b`, as `'+'(a, b)`.

Note: the back tick (`), this is important as the function name contains special symbols.

Example: User defined infix operator

```
`%divisible%` <- function(x,y)
{
  if (x%%y==0) return (TRUE)
  else return (FALSE)
}
```

This function can be used as infix operator `a %divisible% b` or as a function call `%divisible%(a,b)`. Both are the same.

```
> 10 %divisible% 3
[1] FALSE
> 10 %divisible% 2
[1] TRUE
> '%divisible%'(10,5)
[1] TRUE
```

Things to remember while defining your own infix operators are that they must start and end with %. Surround it with back tick (`) in the function definition and escape any special symbols.

Exercise

1. Write a function that converts temperatures from Kelvin to Celsius
2. Write a function to calculate the two sample t-statistic, showing “all the steps”(assume not equal variance)
3. Write a function that calculate or solve quadratic equations
4. Write a function to calculate transpose of a matrix