Flow Control Statements

Chapter 4

Stats 20: Introduction to Statistical Programming with R

UCLA

Contents

Acknowledgements: Jake Elmstedt, Vivian Lew, and Juana Sanchez

Learning Objectives

After studying this chapter, you should be able to:

- Create loops using `for()`, `while()`, and `repeat`.
- Understand the difference between `for`, `while`, and `repeat` loops.
- Understand and construct `if()` and `if-else` statements.

1 The `for()` Loop

One of the main tools in a statistician's toolkit is simulation of random experiments. Since simulations are done to observe patterns in behavior, not just a single instance, we often need to repeat operations over and over again. Rather than writing repetitive code, we can use a **loop** that repeats a chunk of code multiple times. We will discuss several **flow control** statements that control how many times commands in a loop are repeated.

The **for loop** is a common coding procedure in most (if not all) programming languages that repeats a set of commands a fixed number of times. The `for()` statement is used to create for loops in R.

The syntax of a `for()` loop is given by:

```
for (name in vector) {
  # Commands go here
}
```

The `for()` statement performs one iteration of the loop for each entry in `vector`, with the `name` variable being assigned to the values in those entries: The first iteration assigns `name <- vector[1]`, the second iteration assigns `name <- vector[2]`, etc.

There are some general properties of loops in R that will apply to all the loops we will cover:

- The body of the loop is the code that is repeated in each iteration of the loop. Indenting the code in the body is highly recommended, as it helps make it clear that the code in those lines are inside of a loop.
- The curly braces `{}` allow for each iteration of the loop to contain multiple commands. Curly braces are not strictly necessary if there is only command in the loop, but they are still recommended for clarity.
- The result from any of the commands in the body of the loop will not be printed automatically.
- Loops do *not* create local environments. Any objects created within a loop will appear in the global environment.

As a simple example, we can create a `for()` loop that squares each entry in a vector.

```
lost_nums <- c(4, 8, 15, 16, 23, 42)

# Let n cycle over the entries in lost_nums.
for (n in lost_nums) {
  # For each iteration, compute the square of n.
  n^2
}
```

In many settings, it is useful to set `vector` to be an **indexing set**, where the values of `vector` tell R which numbers to use as an index for the loop to cycle over. For example, if the indexing set represents the row numbers of a data frame, then the `for()` loop will only apply the code in the body of the loop to the rows specified by the indexing set.

When `vector` represents an indexing set, it is common to use the letter `i` as the index, since it is consistent with using i as an index in mathematics (like when we write "the ith entry of a vector").

Note: As written, the loop above is not particularly useful, since the computed squared values are not stored or even printed. To save the output we get from each iteration of a loop, we need to make an empty object *outside* the loop first as a place to store our output. Using the iteration number as the indexing set, we then can use subsetting to save the output from the ith iteration of the loop into the ith entry of the storage object.

For example, to save the squares of the `lost_nums` vector, the loop above can be rewritten as follows:

```
# Create a vector to store the output from the for loop.
lost_squares <- numeric(6)

# Let i cycle over the numbers 1 to 6 (the length of lost_nums).
for (i in seq_len(6)) {
  # For the i-th iteration of the for loop, square the i-th entry of lost_nums
  # and save the output into the i-th entry of the lost_squares vector.
  lost_squares[i] <- lost_nums[i]^2
}
# Print the output from the for loop.
lost_squares
```

```
[1]   16   64  225  256  529 1764
```

1.1 Example: The Fibonacci Sequence

Iterations in a loop sometimes depend on the results from previous iterations of the loop.

The **Fibonacci sequence** is a famous and well-studied sequence in mathematics. The first two terms in the sequence are 1 and 1, and each subsequent number is the sum of the previous two terms. For example, the third term is $1 + 1 = 2$, the fourth term is $1 + 2 = 3$, the fifth term is $2 + 3 = 5$, etc.

The `for()` loop below computes the first 12 Fibonacci numbers.

```
# Create a vector to store the output from the for loop.
fib <- numeric(12)
fib[1:2] <- c(1, 1) # Initialize the sequence

# Let i cycle over the numbers 3 to 12 (the indices for the remaining entries in fib).
for (i in 3:12) {
  # For the i-th term in the Fibonacci sequence,
  # compute the sum of the (i-2) and (i-1) terms in the fib vector
  # and save the output into the i-th entry of the fib vector.
  fib[i] <- fib[i - 2] + fib[i - 1]
}
# Print the output from the for loop.
fib
```

```
 [1]   1   1   2   3   5   8  13  21  34  55  89 144
```

2 The `if()` Statement

Recall that relational and Boolean operators produce logical vectors. A common use of logical vectors is as an index to extract only certain elements from an object that satisfy some condition or criterion. Using subsetting, we can apply different commands to different parts of an object.

A general and often more convenient way to control which commands are executed is to use an **`if()`** statement, which only runs certain commands if specified conditions are met.

The syntax of an `if()` statement is given by:

```r
if (condition) {
  # Commands when TRUE
}
```

The `condition` is a logical expression that produces either `TRUE` or `FALSE`. The `if` statement will execute a set of commands if `condition` is `TRUE`. A missing value (`NA`) for will throw an error. A numeric input for `condition` will be coerced into a logical value using the `as.logical()` function, but it is always preferred to explicitly use a logical `condition`.

If there is an alternative set of commands to execute when `condition` is `FALSE`, then an `else` statement is added to the `if` statement. The syntax for an `if-else` statement is given by:

```r
if (condition) {
  # Commands when TRUE
} else {
  # Commands when FALSE
}
```

Caution: Notice that the right curly brace that closes the `if` statement is placed on the same line as the `else`. The right curly brace before the `else` is a signal to R that this is an `if-else` statement rather than just an `if`. Starting the `else` on the same line as the closing brace makes it clear (to R and to the reader) that the statement is incomplete until the `else` statement is closed. The `else` statement cannot be used without an `if()` statement.

Note: The logical expression in `condition` should evaluate to a single logical value. If a logical vector with more than one element is used, R will throw an error. In earlier versions of R (prior to R Version 4.2.0), R would throw a warning and only use the first element as the `condition`.

For a simple example, consider the following `if-else` statement:

```r
x <- 1
if (x > 1) {
  y <- x
} else {
  y <- x + 1
}
```

Question: What is the value of `y`?

Note: An `if` or `if-else` statement is similar to a function call in that the output from the last command in the body will be returned (if the command is executed). However, unlike functions, `if` and `if-else` statements do not create local environments.

Using this fact, the above `if-else` statement can alternatively be written as follows:

```r
y <- if (x > 1) {
  x
} else {
  x + 1
}
```

The `if()` statement is often used inside functions to allow for optional features. As an example, we can create a function that computes the correlation between two numeric vectors that optionally draws a scatterplot of the data.

```r
cor_plot <- function(x, y, scatter = TRUE, ...) {
  if (scatter) {
    # If scatter = TRUE, draw a scatterplot of x and y.
    plot(x, y, ...)
```

```
  }
  # Compute the correlation between x and y.
  cor(x, y)
}
```

Side Note: The `...` argument is used to pass optional arguments to functions used inside the main function. In this example, the `...` enables the `cor_plot()` function to pass arguments to the `plot()` function.

2.1 Basic Error Handling

The `if()` statement enables us to throw error or warning messages when function inputs are not specified as expected. For a motivating example, we will revisit the variance function we wrote in Chapter 2.

```
var_fn <- function(x) {
  sum((x - mean(x))^2) / (length(x) - 1)
}
var_fn(lost_nums)
```

```
[1] 182
```

What if the input vector contains `NA` values?

```
incomplete_nums <- c(4, 8, NA, 16, 23, NA)
var_fn(incomplete_nums)
```

```
[1] NA
```

It would be helpful if our function could throw an error or a warning message if it cannot compute the variance properly.

2.1.1 The `stop()` Function

The **`stop()`** function stops the execution of the current expression and throws an error message.

```
var_fn2 <- function(x) {
  if (sum(is.na(x)) > 0) {
    stop("The input has NA values!")
  }
  sum((x - mean(x))^2) / (length(x) - 1)
}
var_fn2(incomplete_nums)
```

```
Error in var_fn2(incomplete_nums): The input has NA values!
```

2.1.2 The `warning()` Function

The **`warning()`** function throws a warning message but does not stop the execution of the current expression.

```
var_fn3 <- function(x) {
  if (sum(is.na(x)) > 0) {
    warning("The input has NA values!")
  }
  sum((x - mean(x))^2) / (length(x) - 1)
}
var_fn3(incomplete_nums)
```

```
Warning in var_fn3(incomplete_nums): The input has NA values!
```

```
[1] NA
```

Think about other things that can go wrong with this function, or what other functionality you might want:

- What if the input object is not numeric?
- What if the input object is not a vector?
- What if you wanted to first check for `NA` values, throw a warning if necessary, then remove the `NA` values before computing the variance?

2.1.3 The `message()` Function

A related function is the **`message()`** function, which is used for printing diagnostic messages.

```
var_fn4 <- function(x) {
  message("Computing variance, unless there are NAs...")
  sum((x - mean(x))^2) / (length(x) - 1)
}
var_fn4(incomplete_nums)
```

```
Computing variance, unless there are NAs...
```

```
[1] NA
```

The behavior of `message()` and `warning()` are similar, but the purposes are distinct. The purpose of the `message()` function is to notify the user of what the code is doing, but typically the code is working as intended. The `warning()` function is reserved for warning when the result may not be what the user is expecting.

3 The `while()` Loop

The `for()` loop repeats a set of commands a fixed number of times. In some scenarios, the number of times to repeat the commands is not known in advance, so a different type of loop needs to be used.

The **`while()`** statement creates a loop that repeats a set of commands for as long as a certain condition holds.

The syntax of a `while()` loop is given by:

```
while (condition) {
  # Commands go here
}
```

The `while()` loop is essentially a repeating `if` statement:

1. The logical `condition` expression is evaluated.
2. If `condition` is `TRUE`, the commands are executed.
3. Repeat steps 1 and 2 until the `condition` evaluates to `FALSE`, when the loop stops.

A small example of a `while()` loop is given below.

```
# Start with num = 1.
num <- 1
# While num is less than or equal to 20, execute the following commands.
while (num <= 20) {
  # Add 6 to num and assign the sum to num (replace the old num value).
  num <- num + 6
}
# Print the output from the while loop.
num
```

```
[1] 25
```

Question: Why is the result larger than 20 if the loop only runs when `num` is less than or equal to 20?

3.1 Example: Fibonacci Revisited

For a more complete example, we revisit the Fibonacci sequence.

Suppose we want to list all Fibonacci numbers less than 500. Without prior knowledge, we do not know how long this list is, so a `for()` loop would not be ideal.

The `while()` loop below computes all the Fibonacci numbers less than 500. Since the number of iterations is not fixed, it is not clear how to use indexing to refer to the current and previous terms in the sequence. Instead, notice the use of multiple variables to keep track of the two latest terms in the sequence.

```
# fib1 and fib2 will represent the two latest terms in the sequence.
fib1 <- 1 # Initialize fib1
fib2 <- 1 # Initialize fib2
# Create the vector to store the output from the while loop.
full_fib <- c(fib1, fib2)

# While the sum of the last two terms is less than 500, execute the following commands.
while (fib1 + fib2 < 500) {
  # Save the latest term to old_fib2.
  old_fib2 <- fib2
  # Compute the sum of the latest two terms and assign the sum to be the new latest term.
  fib2 <- fib1 + fib2
  # Append the latest term to the end of the full_fib vector with all previous terms.
  full_fib <- c(full_fib, fib2)
  # Save the previously latest term (now the second to last term) to fib1.
  fib1 <- old_fib2
}
# Print the output from the while loop.
full_fib
```

```
 [1]   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

The loop may appear complicated, but the main idea is explained here:

- Before each iteration of the loop is executed, the `condition` of `fib1 + fib2 < 500` is evaluated.
- If the next term in the sequence (i.e., `fib1 + fib2`) is still less than 500, then `fib1 + fib2 < 500` will be `TRUE`, and the commands are repeated.
- The `full_fib <- c(full_fib, fib2)` command within the body appends the latest term in the sequence to the growing `full_fib` vector of Fibonacci numbers.

Note: Even though this technique is used here for intuition and illustrative purposes, increasing the length of a vector element by element (e.g., `full_fib <- c(full_fib, fib2)`) in a `for()` or `while()` loop should generally be avoided. Technically, R replaces the entire vector by a vector with one more entry each time. New storage space is allocated in memory for each new vector. This vector memory allocation is time consuming, inefficient, and can considerably slow down the loop, especially when the numbers of iterations is large. It is generally preferable to minimize (or at least reduce) the number of memory allocations done in large loops, such as by using a preallocated storage vector of a fixed size whenever possible.

4 The `repeat` Loop and `break` Statements

The conditions for exiting a `for()` or `while()` loop are specified at the top of the loop: In a `for()` loop, the loop repeats until the fixed number of iterations are completed. In a `while()` loop, the loop repeats until the `condition` evaluates to `FALSE`. For some loops, it may be clearer not to include the exit condition at the top.

The **repeat** statement creates a loop that executes a set of commands repeatedly without a built-in `condition` to exit the loop. The loop will repeat indefinitely until a **break** statement is executed.

The syntax of a `repeat` loop is given by:

```
repeat{
  # Commands go here
}
```

A `break` statement immediately exits (or breaks out of) a loop. The syntax for `repeat` does not require a `break` statement, but not including one will result in an infinite loop (i.e., a loop that is repeated infinitely many times). A `break` statement is typically inside of an `if()` statement so that the `break` is executed only if a certain `condition` (the exit condition) is satisfied.

The syntax of a typical `break` statement is given by:

```
if (condition) {
  break
}
```

Note: `break` statements can also be used in `for()` and `while()` loops to specify alternative exit conditions.

The small `while()` loop example can be written as a `repeat` loop:

```
# Start with num = 1.
num <- 1

# Repeat the following commands.
repeat{
  # Add 6 to num and assign the sum to num (replace the old num value).
  num <- num + 6
  # If num is greater than 20, break out of the loop.
  if (num > 20) {
    break
  }
}
# Print the output from the repeat loop.
num
```

```
[1] 25
```

Side Note: `repeat` loops can also be written as `while()` loops by using a `condition` that is always `TRUE` and including a `break` statement.

```
# Start with num = 1.
num <- 1

# While TRUE is TRUE (which is always), execute the following commands.
while (TRUE) {
  # Add 6 to num and assign the sum to num (replace the old num value).
  num <- num + 6
  # If num is greater than 20, break out of the loop.
  if (num > 20) {
    break
  }
}
# Print the output from the repeat loop.
num
```

```
[1] 25
```

5 When To Use Which Loop (and When Not To)

When executing a chunk of code multiple times, or iterating over an index, it is often natural to consider using a loop. However, depending on the problem, a loop may not be necessary.

If the order in which the iterations are executed does not matter, vectorization should be used instead of a loop. A vectorized approach is typically simpler, shorter, and more computationally efficient than a loop.

In general, a loop is necessary if the order in which the iterations of the repeated code are executed matters. For example, the 10th Fibonacci number cannot be calculated before the 8th and the 9th are, so the order matters and a loop is needed.

A `for()` loop is used when the number of times a chunk of code is repeated is known in advance. If the number of iterations is not known in advance, then a `while()` loop is more appropriate.

In practice, the `repeat` loop is rarely used, but it may be useful when it is advantageous to exit a loop somewhere other than the top of the loop.