Intermediate R

Allison Theobold

Relational Operators

Relational operators tell how one object relates to another. There are a variety of relational operators that you have used before in mathematics but take on a slightly different feel in computer science. We will walk through a few examples of different relational operators and the rest will be left as exercises.

- **Equality & Inequality**: This type of operator tells whether an object is equivalent to another object (==), or its negation (!=).

```
TRUE == FALSE
```

```
## [1] FALSE
```

```
-6*14 != 17-101
```

```
## [1] FALSE
```

Exercise 1

```
# write R code to see if
    # "useR" and "user" are equal.
    # TRUE and 1 are equal.
```

- **Greater & Less**: These statements should be familiar, with a bit of a notation twist. To write a strict greater than or less than statement you would use > or < in your statement. To add an equality constraint, you would add a = sign after the inequality statement (>= or <=).

```
-6 * 5 + 2 >= -10 + 1
```

```
## [1] FALSE
```

Exercise 2

```
# write R code to see if
    # "raining" is less than or equal to "raining dogs".
    # TRUE is greater than FALSE.
```

- **Matrices**: For matrices, the above relational statements can be applied across the entire matrix, or to a subset of the matrix (e.g. a vector). The relational statement is applied element wise (a step-wise progression through the entries).

```
# The following are dating data, of one person's messages received per day for one week

okcupid <- c(16, 9, 13, 5, 2, 17, 14)
match <- c(17, 7, 5, 16, 8, 13, 14)
messages <- matrix(c(okcupid, match), ncol = 2, byrow = FALSE)
# This makes the data from OkCupid the first column and the data from Match the second column
```

Exercise 3

```
# Use the messages matrix to return a logical matrix that answers the following questions:


# For which days and sites were the messages equal to 13?


# For which days and sites were the number of messages less than or equal to 14?
```

Logicals

These statements allow for us to change or combine the results of the relational statements we discussed before, using "and", "or", or "not".

- **"and"** statements evaluate to `TRUE` only if every relational statement evaluates to `TRUE`.
  - `(3 < 5) & (9 > 7)` would evaluate to `TRUE` because both relational statements are `TRUE`
  - `(3 > 5) & (9 > 7)`, this would evaluate to `FALSE` as only one of the relational statements is `FALSE` (the first one)
- **"or"** statements evaluate to `TRUE` if at least one relational statement evaluates to `TRUE`.
  - `(3 > 5) | (9 > 7)` would evaluate to `TRUE` because one of the relational statements is `TRUE` (the second one)
  - `(3 > 5) | (9 < 7)` would evaluate to `FALSE` as both relational statements evaluate to `FALSE`
- **Remark:** The `&&` and `||` logical statements do not evaluate the same as their single counterparts. Instead, these logical operators evaluate to `TRUE` or `FALSE` based only on the first element of the statement, vector, or matrix.

Exercise 4

```
# Using the okcupid vector from above, answer the following questions:

# Is the last day of the week under 5 messages or above 10 messages? (hint: last <- tail(okcupid, 1) co

# Is the last day of the week between 15 and 20 messages, excluding 15 but including 20?
```

Conditional Statements

Conditional statements utilize relational statements and logical statements to change the results of your `R` code. You may have encountered an `ifelse` statement before (or not), but let's breakdown exactly what `R` is doing when it evaluates them.

If Statements

First, let's start with an `if` statement, the often overlooked building block of the `ifelse` statement. The `if` statement is structured as follows:

```
if(condition){
              statement
}
```

- the condition inside the parenthesis (a relational statement) is what the computer executes to verify if it is `TRUE`,
  - if the condition evaluates to `TRUE` then the statement inside the curly braces (`{}`) is output, and
  - if the condition is `FALSE` nothing is output.

Remark: In `R` the `if` statement, as described above, will only accept a **single** value (not a vector or matrix).

Else Statements

Since whenever an `if` statement evaluates to `FALSE` nothing is output, you might see why an `else` statement could be beneficial! An else statement allows for another statement to be output whenever the `if` condition evaluates to `FALSE`. The `ifelse` statement is structured as follows:

```
if(condition){
              statement1
}
else{
              statement2
}
```

- again, the `if` condition is executed first,
  - if it evaluates to `TRUE` then the first statement (`statement1`) is output,
  - if the condition is `FALSE` the computer moves on to the `else` statement, and the second statement (`statement2`) is output.

Remark: `R` accepts both `ifelse` statements structured as outlined above, but also `ifelse` statements using the built-in `ifelse()` function. This function accepts **both singular and vector** inputs and is structured as follows:

```
ifelse(condition, statement1, statement2),
```

where the first argument is the conditional (relational) statement, the second argument is the statement (`statement1`) that is evaluated when the condition is `TRUE`, and the third statement (`statement2`) is the statement that is evaluated when the condition is `FALSE`.

Else If Statements

On occasion, you may want to add a third (or fourth, or fifth, ...) condition to your `ifelse` statement, which is where the `elseif` statement comes in. The `elseif` statement is added to the `ifelse` as follows:

```
if(condition1){
              statement1
}
elseif(condition2){
              statement2
}
else{
              statement3
}
```

- The `if` condition (`condition1`) is executed first,
  - if it evaluates to `TRUE` then the first statement (`statement1`) is output,
  - if the condition is `FALSE` the computer moves on to the `elseif` condition,

- Now the second condition (`condition2`) is executed,
  - if it evaluates to `TRUE` then the second statement (`statement2`) is output,
  - if the condition is `FALSE` the computer moves on to the `else` statement, and
- the third statement (`statement3`) is output.

Exercise 5

```
# Using the okcupid vector from above, write an if statement that prints
# 'You're popular!' if the number of messages exceeds 10
```

Exercise 6

```
# Using the if statement from Exercise 5 add the following else statement:
# When the if-condition on messages is not met, R prints out 'Send more
# messages!'
```

Loops

Loops are a popular way to iterate or replicate a set of instructions many times. It is no more than creating an automated process by organizing a sequence of steps and grouping together the steps that need to be repeated. In general, the advice of many `R` users would be to learn about loops, but once you have a clear understanding of them to get rid of them. Loops will give you a detailed view of what is happening and the data you are manipulating. Once you have this understanding, you should put your effort into learning about vectorized alternatives as they pay off in efficiency. These loop alternatives (the `apply` and `purrr` families) will be covered in the *Simulation Workshop*.

Typically in computer science, we separate loops into two types. The loops that execute a process a specified number of times, where the "index" or "counter" is incremented after each cycle are part of the `for loop` family. Other loops that only repeat themselves until a conditional statement is evaluated to be `FALSE` are part of the `while` family.

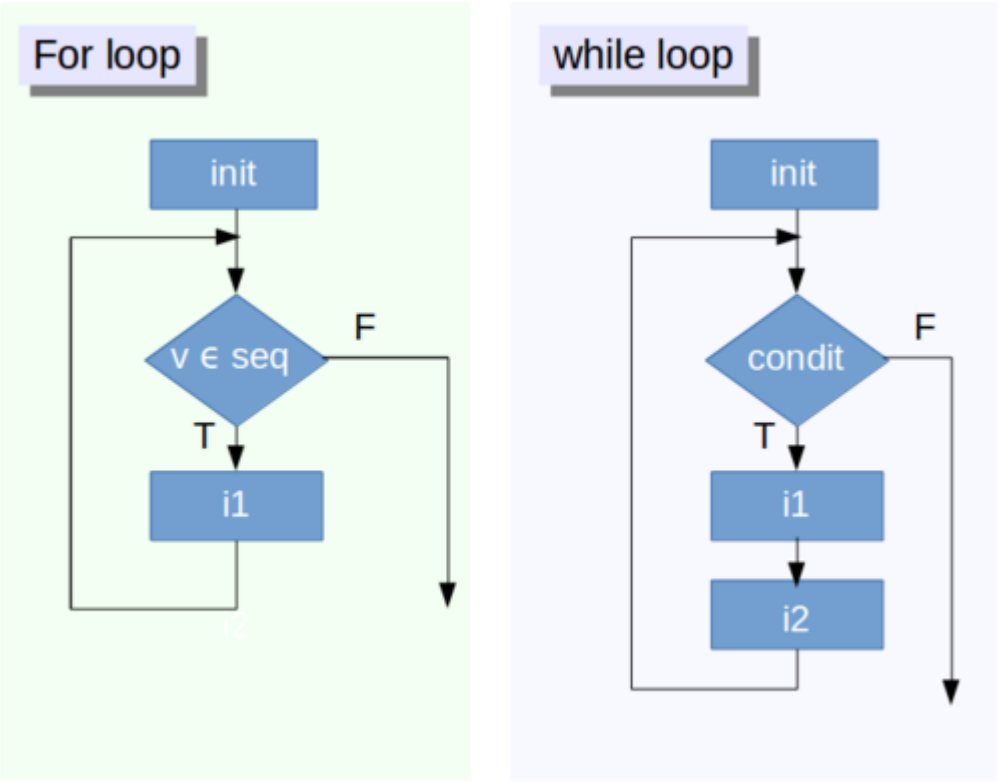

Figure 1: Loop Flowchart

For Loops

In the for loop figure above:

- The rectangular `init` box represents the initialization of the object being used in the `for loop` (i.e. the variable). This initialization is required in `R`, as opposed to other languages, such as Python, where an initialization does not have to occur.
- The diamond shapes represent the repeat decision the computer is required to make. The computer evaluates the conditional statement (v in seq) as either `TRUE` or `FALSE`. In other terms, you are testing if in the current value of v is within the specified range of values seq, where this range is defined in the initialization (like 1:100).
- The second `i1` rectangle represents the set of instructions to execute for every iteration. This could be a simple statement, a block of instructions, or another loop (nested loops).

The computer marches through this process until the the statement evaluates to `FALSE` (v not in seq).

Notation

The `for loop` instructions are placed between curly braces, which are placed directly after the test condition or beneath it. You will find the formatting that you like best for things such as loops and functions, but here is my preferred syntax:

```
for(i in 1:n){
              statement
}
```

Example:

```
sim <- 100
# Define a counter variable to determine how many t-values to use

u <- rt(sim, 32)
# Draw sim random t-values, from a t-distribution with 32 degrees of freedom

# Initialize `usq` (could have used a single NA or 0 initialization)
usq <- rep(NA, sim)

for(i in 1:sim) {
  # i-th element of `u1` squared into i-th position of `usq`
  usq[i] <- u[i]^2
  print(c(u[i], usq[i]))
}

print(i) ## should be 100 (the same as sim), unless the loop had issues!
```

```
[1] 0.13559322 0.01838552
[1] 0.21768730 0.04738776
[1] -0.3459573  0.1196864
.
.
.
```

Remark: It is often useful, when writing a loop to add a `print()` output inside. This allows for you to verify that the process is executing correctly and can save you some major headaches!

Nesting For Loops

Now that you know that for loops can also be nested, you're probably wondering when and why you would be using this in your code.

Well, suppose you wish to manipulate a matrix by setting its elements to specific values, based on their row and column position. You will need to use nested `for loops` in order to assign each of the matrix's entries a value.

- The index `i` runs over the rows of the matrix
- The index `j` runs over the columns of the matrix

```
# Create a 30 x 30 matrix (of 30 rows and 30 columns)
mymat <- matrix(nrow = 30, ncol = 30)

# verify the dimensions of mymat
dim(mymat)
```

```
## [1] 30 30
```

```
# pull off number of rows for first loop
dim(mymat)[1]
```

```
## [1] 30
```

```
# pull off number of columns for second loop
dim(mymat)[2]
```

```
## [1] 30
```

```
# For each entry in the matrix: assign values based on its position (the product of two indices)
for(i in 1:dim(mymat)[1]){
  for(j in 1:dim(mymat)[2]){
    mymat[i,j] <- i*j
  }
}

# Just show the upper left 13x13 piece
mymat[1:13, 1:13]
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
##  [1,]    1    2    3    4    5    6    7    8    9    10    11    12    13
##  [2,]    2    4    6    8   10   12   14   16   18    20    22    24    26
##  [3,]    3    6    9   12   15   18   21   24   27    30    33    36    39
##  [4,]    4    8   12   16   20   24   28   32   36    40    44    48    52
##  [5,]    5   10   15   20   25   30   35   40   45    50    55    60    65
##  [6,]    6   12   18   24   30   36   42   48   54    60    66    72    78
##  [7,]    7   14   21   28   35   42   49   56   63    70    77    84    91
##  [8,]    8   16   24   32   40   48   56   64   72    80    88    96   104
##  [9,]    9   18   27   36   45   54   63   72   81    90    99   108   117
## [10,]   10   20   30   40   50   60   70   80   90   100   110   120   130
## [11,]   11   22   33   44   55   66   77   88   99   110   121   132   143
## [12,]   12   24   36   48   60   72   84   96  108   120   132   144   156
## [13,]   13   26   39   52   65   78   91  104  117   130   143   156   169
```

While Loops

The `while loop` takes on a similar structure as the `for loop`, except that the diamond block can now be **any** conditional statement. In a `for loop`, the only job of this block was to verify that the "counter" of the loop (i or j) was still in the specified range, and did not require you to construct a conditional statement.

Instead, we now have a relational statement to check, which is typically expressed using a relational statement between a "control variable" and a value.

- If the result of the relational statement is `FALSE`, the loop is never executed (similar to the "counter" being outside the sequence of numbers).
- If the result of the relational statement is `TRUE`, the block of instructions `i1` is executed, and
  - the "control variable" is updated in `i2`.

For example, here is how we could rewrite the first for loop as a while loop:

```
sim <- 100
u <- rt(sim, 32)


usq <- 0  # Initialize `usq`

i <- 1  # Initialize the "control variable"

# verifying a relational statement, checking if i is STRICTLY less than 100 (sim)
while(i < sim){
  # i-th element of `u1` squared into `i`-th position of `usq`
  usq[i] <- u[i]^2
  i <- i + 1
}
```

How is the `usq` output from the above `while loop` different from the `usq` from the previous `for loop`?

Functions

What is a function? In rough terms, a function is a set of instructions that you would like repeated (similar to a loop), but are more efficient to be self-contained in a sub-program and called upon when needed. A function is a piece of code that carries out a specific task, accepting arguments (inputs), and returning values (outputs). Many different programming languages fuss over function terminology, but in `R`, a function is a function!

In `R` the function you define (or use) will have the following construction:

```
functionName <- function(argument1, argument2, ...){
                body
}
```

At this point, you may be familiar with a smattering of `R` functions in a few different packages (e.g. `lm`, `glm`, `ggplot`, `diffmean`, etc.). Some of these functions take multiple arguments and return multiple outputs, but the best way to learn about the inner workings of functions is to write your own!

User Defined Functions

Often, you will be in the position where you need to perform a task, but you do not know of a function or a library that would help you. You could spend time Googling for a solution, but in the amount of time it takes you to find something you could have already written your own function!

The function you build will have the same form as described above, where you select the name of the function, the arguments, and the outputs. Some important advice on writing your own functions:

- Make sure that the name you choose for your function is not a reserved word in `R` (`mean`, `sd`, `hist`, etc.).
  - A way to avoid this is to use the help directory to see if that function name already exists (`?functionName`).
- It is possible to overwrite the name of an existing function, but it is not recommended!

Examples:
Here are some functions that I wrote. Let's look them over and find each of the following:

- what are the arguments?
- what are the outputs?
- what is being performed in the body of the function?

Example 7

```
fahr_to_kelvin <- function(temp){
                    kelvin <- ((temp - 32) * (5 / 9)) + 273.15
                    return(kelvin)
}
```

- Arguments:
- Outputs:
- Body:

Example 8

```
dataCenter <- function(data, new_center){
            # returns a new vector containing the original data centered around the desired value
            new_data <- (data - mean(data)) + new_center
            return(new_data)
}
```

- Arguments:
- Outputs:
- Body:

Example 9 Write a function that computes the condition index for a given fish

$$\text{condition index} = \frac{\text{weight}^{\frac{1}{3}}}{\text{length}} * 50$$

The function should take two arguments (a length and a weight)

```
## function code here!
```

Example 10: Putting it All Together!

```
matrix <- data.frame(NA, l = BlackfootFish$length, w = BlackfootFish$weight)
```

Now, given the matrix of integers above,
1. use a `for loop`,
2. an `ifelse` statement, and
3. the function you defined above,

to compute the condition number of each fish, and remove that fish from the dataset if it's condition is NA or greater than 2.

References

https://www.datacamp.com/community/tutorials/functions-in-r-a-tutorial#gs.5ySzPg0

https://www.datacamp.com/community/tutorials/tutorial-on-loops-in-r