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
```

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

- **Greater & Less**: These statements should be familiar, with a bit of a notation twist. To write a greater than or equal to statement you would you >= in your statement, and similarly with less than or equal to (the inequality goes before the = in R.

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

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

```
# 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 (a vector). The relational statement is applied element wise (a step-wise progression through the entries).

```
# These dating data, of messages received per week, have been created for you

okcupid <- c(16, 9, 13, 5, 2, 17, 14)
match <- c(17, 7, 5, 16, 8, 13, 14)
messages <- matrix(c(okcupid, match), nrow = 2, byrow = TRUE)

# Use the messages matrix to return a logical matrix
# When were the messages equal to 13?
messages >= 13
```

```
##      [,1]  [,2]  [,3]  [,4]  [,5] [,6] [,7]
## [1,] TRUE FALSE  TRUE FALSE FALSE TRUE TRUE
## [2,] TRUE FALSE FALSE  TRUE FALSE TRUE TRUE
```

```
# For which days 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 the first relational statement is `FALSE`
- **"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.

```
# Using the messages data from above, answer the following questions.
    # Is last under 5 or above 10?
    # Is last between 15 and 20, excluding 15 but including 20?
```

Logicals in Your Daily (Statistics) Life

Many of you may be wondering how the topics above relate to your daily lives in statistical practices, but wait! Let's play with some data and some functions that are likely very familiar to you. Import the *BlackfootFish* dataset and carry out the following questions:

- Which fish do no have both length and weight recorded? (hint: use relational operators, logicals, and the `which` or `subset` functions)
- Remove these fish you found from the dataset, renaming the new dataset `BlackfootFish2`.
- Subset these data so that only the Rainbow Trout (RBT) and Brown Trout (Brown) remain.

Conditional Statements

Conditional statements utilize relational statements and logicals 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 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.

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 outline above, but also ifelse statements using the built-in ifelse() function. This function accepts a vector of inputs and is structured as follows:

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

where the first argument is the conditional statement, the second argument is the statement that is evaluated when the condition is TRUE, and the third statement is the statement that is evaluated when the condition is FALSE.

Else If Statements

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

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

- 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 elseif condition,
- Now the second condition 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.

```
# Using the message dataset, write an if statement that prints 'You're
# popular!' if the number of messages exceeds 10
```

```
# Using the if statements above add the following: When the if-condition on
# messages is not met, R prints out 'Send more messages!'
```

```
# Using the ifelse statement above add the following: 'Your number of views
# is average' is printed if messages is between 15 (inclusive) and 10
# (exclusive).
```

Combine It All!

```
# Variables related to your last day of recordings
ma <- tail(match, 1)
ok <- tail(okcupid, 1)

# Code the control-flow construct
if (___ & ___) {
  sms <- 2 * (ma + ok)
} elseif (___) {
  sms <- 0.5 * (ma + ok)
} else {
  sms <- ___
}

# 1. If both ma and ok are 15 or higher, set sms equal to double the sum of ma and ok.
# 2. If both ma and ok are strictly below 10, set sms equal to half the sum of ma and ok.
# 3. In all other cases, set sms equal to ma + ok.
# 4. Print the resulting sms variable to the console.
```

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 family) 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. On the other hand, other loops only repeat themselves until a conditional statement is evaluated to be `FALSE`. These conditional loops belong to the `while` family.

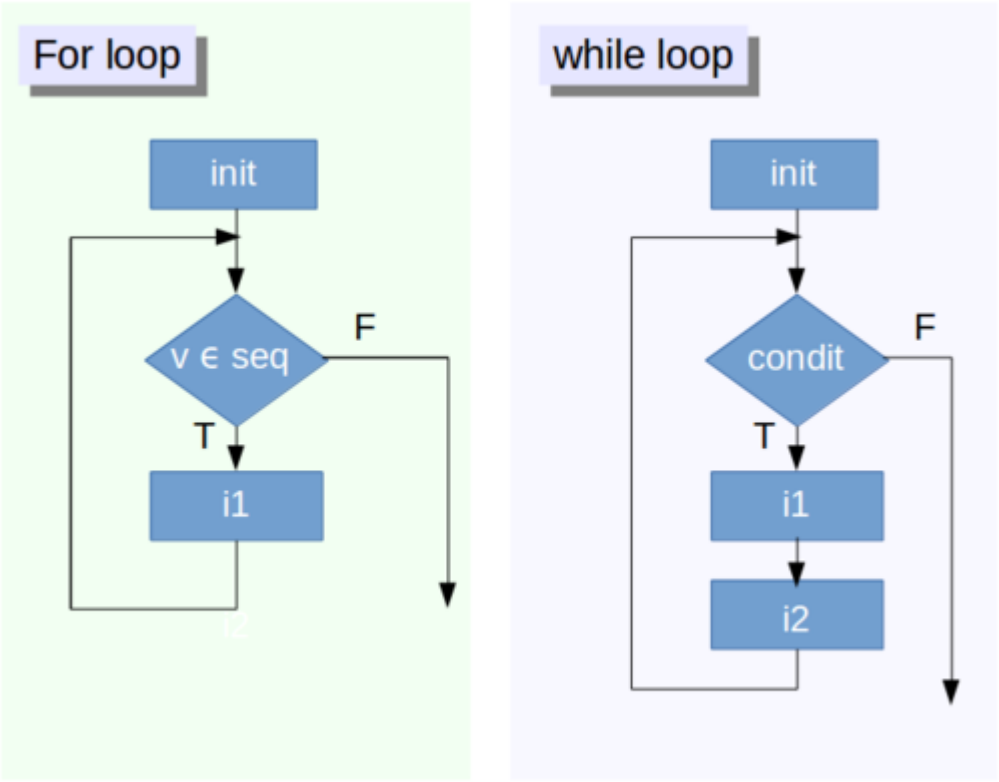

Figure 1: Loop Flowchart

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 conditional statement evaluates to `FALSE` ($v \notin seq$).

Notation

The `for loop` is placed between curly braces, which can be placed either 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
}
```

```
# Create a vector filled with random t-values, from a distribution with 32 degrees of freedom

sim <- 1:100
u <- rt(sim, 32)

# Initialize `usq` (could have used a 0 initialization)
usq <- NA

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

```
## [1] -0.028795885  0.000829203
## [1] 0.055016799 0.003026848
## [1] 2.418806 5.850625
## [1] 0.5395341 0.2910971
## [1] 0.7778744 0.6050886
## [1] 0.23106954 0.05339313
## [1] -0.20758016  0.04308952
## [1] 0.4862042 0.2363946
## [1] -0.5939806  0.3528129
## [1] -0.26370721  0.06954149
## [1] 0.8822383 0.7783445
## [1] -0.3489892  0.1217935
## [1] -0.4064924  0.1652361
## [1] -0.3770630  0.1421765
```

```
## [1] 0.6805635 0.4631667
## [1] -0.20792994  0.04323486
## [1] -0.3990674  0.1592548
## [1] -0.27522901  0.07575101
## [1] 0.099481690 0.009896607
## [1] 1.052567 1.107898
## [1] -0.7286304  0.5309023
## [1] 0.6798865 0.4622457
## [1] -1.130048  1.277009
## [1] 1.073302 1.151978
## [1] 0.4654940 0.2166847
## [1] 1.111939 1.236408
## [1] -0.7782160  0.6056202
## [1] 1.306631 1.707285
## [1] 0.15396447 0.02370506
## [1] -0.6072704  0.3687774
## [1] 1.597893 2.553262
## [1] 1.366023 1.866019
## [1] -0.3617990  0.1308985
## [1] -0.092216943  0.008503965
## [1] 0.6685516 0.4469612
## [1] -0.3595696  0.1292903
## [1] 0.7560618 0.5716294
## [1] 0.17851966 0.03186927
## [1] -1.643936  2.702525
## [1] -0.4439664  0.1971062
## [1] -1.605972  2.579146
## [1] 0.8086082 0.6538472
## [1] 0.3956244 0.1565187
## [1] -1.020476  1.041371
## [1] -0.24937809  0.06218943
## [1] -0.11225520  0.01260123
## [1] -1.354616  1.834984
## [1] 0.6907232 0.4770986
## [1] 1.925792 3.708675
## [1] -0.5634166  0.3174383
## [1] -0.05786052  0.00334784
## [1] 0.22968206 0.05275385
## [1] -0.9547805  0.9116059
## [1] 1.865256 3.479179
## [1] 0.7227834 0.5224159
## [1] 1.973485 3.894644
## [1] -1.538764  2.367794
## [1] 0.3320009 0.1102246
## [1]  3.770714 14.218282
## [1] -3.907683 15.269987
## [1] -2.167402  4.697631
## [1] 1.306538 1.707043
## [1] 0.076348593 0.005829108
## [1] -1.859362  3.457226
## [1] 1.007167 1.014386
## [1] -1.518985  2.307317
## [1] -0.5949694  0.3539886
## [1] -1.025112  1.050854
```

```
## [1] 0.053533878 0.002865876
## [1] -0.9277511  0.8607221
## [1] 0.7967780 0.6348551
## [1] -0.6265905  0.3926156
## [1] 0.11554902 0.01335158
## [1] 0.18306949 0.03351444
## [1] -0.7443527  0.5540610
## [1] -0.8525130  0.7267784
## [1] -0.05306797  0.00281621
## [1] -0.3378099  0.1141155
## [1] 0.24818516 0.06159587
## [1] 0.4334807 0.1879055
## [1] 0.6895517 0.4754815
## [1] -1.055963  1.115058
## [1] 0.13574624 0.01842704
## [1] 1.022113 1.044714
## [1] -1.486127  2.208572
## [1] -0.8132639  0.6613981
## [1] -0.6841997  0.4681292
## [1] 0.5146455 0.2648600
## [1] -0.17010989  0.02893737
## [1] -0.22387797  0.05012135
## [1] -1.352805  1.830082
## [1] 1.042223 1.086229
## [1] 2.496606 6.233040
## [1] -0.21078620  0.04443082
## [1] 0.4510325 0.2034303
## [1] -1.518831  2.306846
## [1] -0.7643228  0.5841893
## [1] -0.0203482037  0.0004140494
## [1] 0.4456230 0.1985798
## [1] -1.859000  3.455882
```

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

```
## [1] 100
```

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.

```
# Create a matrix probabilities of drawing binomials 0-5, from a population
# of 5, a sample size of 2, and probabilities going from 0-1 by 0.1
x <- 0:5
theta <- seq(0, 1, by = 0.1)

probs <- matrix(NA, nrow = length(x), ncol = length(seq(0, 1, by = 0.1)))

# For each row and for each column, assign values based on position: product
# of two indexes
for (i in 1:length(x)) {
    for (j in 1:length(theta)) {
        probs[i, j] = dbinom(x[i], 5, theta[j])
    }
}

probs <- as.data.frame(probs)

colnames(probs) <- NA

for (i in 1:length(theta)) {
    colnames(probs)[i] <- paste(expression(theta), "=", 0 + 0.1 * (i - 1))
}

rownames(probs) <- apply(expand.grid(replicate(1, "x"), c(0, 1, 2, 3, 4, 5)),
    1, paste, collapse = "=")

probs
```

```
##     theta = 0 theta = 0.1 theta = 0.2 theta = 0.3 theta = 0.4 theta = 0.5
## x=0         1     0.59049     0.32768     0.16807     0.07776     0.03125
## x=1         0     0.32805     0.40960     0.36015     0.25920     0.15625
## x=2         0     0.07290     0.20480     0.30870     0.34560     0.31250
## x=3         0     0.00810     0.05120     0.13230     0.23040     0.31250
## x=4         0     0.00045     0.00640     0.02835     0.07680     0.15625
## x=5         0     0.00001     0.00032     0.00243     0.01024     0.03125
##     theta = 0.6 theta = 0.7 theta = 0.8 theta = 0.9 theta = 1
## x=0     0.01024     0.00243     0.00032     0.00001         0
## x=1     0.07680     0.02835     0.00640     0.00045         0
## x=2     0.23040     0.13230     0.05120     0.00810         0
## x=3     0.34560     0.30870     0.20480     0.07290         0
## x=4     0.25920     0.36015     0.40960     0.32805         0
## x=5     0.07776     0.16807     0.32768     0.59049         1
```

While Loops

The `while loop` takes on a similar structure to the `for loop`, except that the diamond block now **explicitly** represents a conditional statement. In a `for loop`, the only job of this block was to verify that the "counter" of the loop (i) was still in the specified range, and did not require you to construct a conditional statement. Now this conditional statement 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 in the range of 1 to sim
while(i %in% 1:sim){
  # i-th element of `u1` squared into `i`-th position of `usq`
  usq[i] <- u[i]^2
  i <- i + 1
}
```

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 better 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
}
```

But, at this point you may only be familiar with a smattering of R functions in a few different packages. 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 some solution, but often 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, etc.).
- A way to avoid this is to use the help directory to see if that function name already exists (`?function name`).
- 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?

1.

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

2.

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

Remark: It is often useful, when writing a function or 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!

Practice: Write a function that computes the condition index for a given fish (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!
```

Now, given the matrix of integers below,
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.

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

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