

Intermediate R

Allison Theobald

Relational Operators

Relational operators tell how one object relates to another, where the output is a logical value. 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 types of 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 (`!=`).

```
w <- 10.2
x <- 1.3
y <- 2.8
z <- 17.5
dna1 <- "attattaggaccaca"
dna2 <- "attattaggaacaca"

dna1 == dna2

## [1] FALSE
dna1 != dna2

## [1] TRUE
```

- **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 `<=`).

```
w > 10

## [1] TRUE
x > y

## [1] FALSE
```

- **Inclusion:** This type of statement checks if a character, number, or factor are included in a vector. The `%in%` operator tells you whether a value is included in an object. These values can be linked together into a vector, and the output will be a vector of logical values.

```
colors <- c("green", "pink", "red")

"blue" %in% colors

## [1] FALSE
numbers <- c(1, 2, 3, 4, 5)

5 %in% numbers
```

```
## [1] TRUE
letters <- c("a", "b", "c", "d")

c("a", "b") %in% letters

## [1] TRUE TRUE
```

Exercise 1:

```
# write R code to see if
# 2 * x + 0.2 is equal to y

# "hello" is greater than or equal to "goodbye"

# TRUE is greater than FALSE

# dna1 is longer than 5 bases (use nchar() to figure out how long a string is)
```

- **Which:** The `which` statement returns the *indices* of a vector, where a relational statement is satisfied (evaluates to TRUE).

```
x <- c(3, 5, 7, 9, 11, 13, 15)

which(x > 1)

## [1] 1 2 3 4 5 6 7

which(x == 8)

## integer(0)
```

Challenge: How would you use the indices from these which statements to extract the elements of x that are greater than 10?

- **Matrices:** The above relational and which statements can be applied to a matrix, or to a subset of the matrix (e.g. a vector). The relational and `which` statements are applied element wise (a step-by-step progression through the matrix/vector 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 <- data.frame(okcupid = okcupid, match = match)
# This makes the data from OkCupid the first column and the data from Match the second column
row.names(messages) <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday")
```

Exercise 2: Use the messages matrix to return a matrix of *logical* values that answers the following question:

```
# Create a logical matrix to answer the question:  
# For what days were the number of messages at either site greater than 12?
```

Exercise 3: Use the messages matrix to return the *indices* of **messages** that answers the following questions:

```
# Use indices of the matrix to answer:  
# when were the messages at OkCupid equal to 13?
```

```
# when were the number of messages at Match greater than the messages at OkCupid?
```

```
# Challenge: Use the indices from above to extract the rows of the message matrix that meet the criterion
```

- **Subset:** The **subset** command takes in an object (vector, matrix, data frame) and returns the subset of that object where the entries meet the relational conditions specified (evaluates to **TRUE**).

```
x <- c(3, 5, 7, 9, 11, 13, 15)
```

```
subset(x, x > 2)
```

```
## [1] 3 5 7 9 11 13 15
```

```
# Challenge: Change the which() statement code to a subset() statement, extracting the days that the number of messages at either site was greater than 12
```

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**
- **not** statements negate the statement they proceed.
 - `is.numeric(5)` would evaluate to **TRUE** because 5 is a number
 - `!is.numeric(5)` would evaluate to **FALSE** as it negates the statement it proceeds (`!TRUE = 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.

```
c(TRUE, TRUE, FALSE) & c(TRUE, FALSE, FALSE)
```

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

```
c(TRUE, TRUE, FALSE) | c(TRUE, FALSE, FALSE)
```

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

```
c(TRUE, TRUE, FALSE) && c(TRUE, FALSE, FALSE)
```

```
## [1] TRUE
```

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) could be helpful)
```

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

The `subset` command can from before, can also accept more than one relational conditions, joined by logicals.

```
bad_days <- subset(messages, okcupid < 6 | match < 6)
```

```
bad_days
```

```
##           okcupid match
```

```
## Wednesday      13     5
```

```
## Thursday       5     16
```

```
## Friday         2     8
```

```
good_days <- subset(messages, okcupid > 10 & match > 10)
```

```
good_days
```

```
##           okcupid match
```

```
## Monday        16     17
```

```
## Saturday      17     13
```

```
## Sunday        14     14
```

Conditional Statements

Conditional statements utilize relational 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 check its logical value,
 - 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).

```
y <- -3  
  
if(y < 0){  
  "y is a negative number"  
}  
  
## [1] "y is a negative number"  
  
y <- 5
```

Exercise 5: Using the `last` number from above, write an `if` statement that prints “You’re popular!” if the number of messages exceeds 10.

```
# hint: use the last day of the week for okcupid that you made above
```

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

```
y <- -3  
  
if(y < 0){  
  "y is a negative number"  
}else{  
  "y is either positive or zero"  
}
```

```
## [1] "y is a negative number"
```

```
y <- 5
```

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 that is evaluated when the condition is `TRUE` (`statement1`), and the third statement is the statement that is evaluated when the condition is `FALSE` (`statement2`).

```
ifelse(y < 0, "y is a negative number", "y is either positive or zero")
```

```
## [1] "y is either positive or zero"
```

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!”.

Challenge: Rewrite the if, else function from above using R's built in ifelse() function.

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  
}  
else if(condition2){  
    statement2  
}  
else{  
    statement3  
}
```

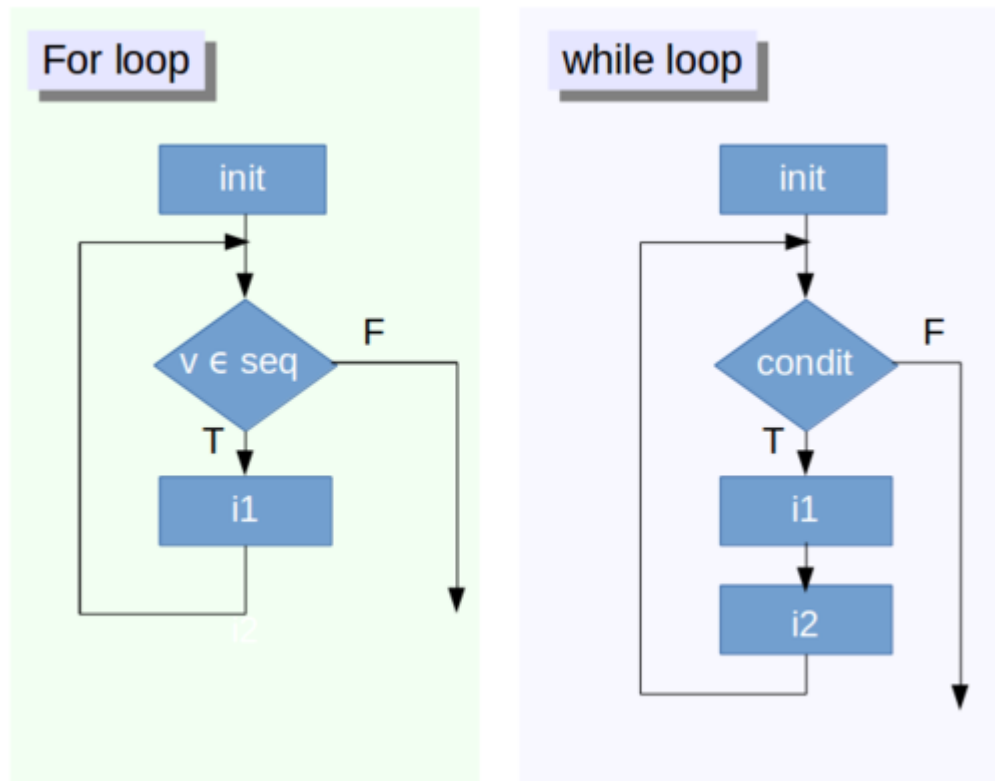
- The `if` condition (`condition1`) is executed first,
 - if it evaluates to `TRUE` then the first statement (`statement1`) is output,
 - if the condition is `FALSE` the computer moves on to the `elseif` condition,
- Now the second condition (`condition2`) is executed,
 - if it evaluates to `TRUE` then 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.

```
y <- -3  
  
if(y < 0){  
  "y is a negative number"  
}else if(y == 0){  
  "y is zero"  
}else{  
  "y is positive"  
}  
  
## [1] "y is a negative number"  
  
y <- 5
```

- **Remark:** Conditional statements should be written so that the components are mutually exclusive (an observation can only belong to one piece).

```
x <- 6  
  
if(x %% 2 == 0){  
  "x is divisible by 2"  
}else if(x %% 3 == 0){  
  "x is divisible by 3"  
}else{  
  "x is not divisible by 2 or 3"  
}  
  
## [1] "x is divisible by 2"
```

Loops



Fanara, C. (2018, November 20). Content Flowchart 1 [A flow chart showing R loop structures.]. Retrieved February 12, 2019, from http://community.datacamp.com.s3.amazonaws.com/community/production/ckeditor_assets/pictures/132/content_flowchart1.png

Loops are a popular way to iterate or replicate the same set of instructions many times. It is no more than creating an automated process by organizing a sequence of steps in your code 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 instead use vectorization when you can. Loops will give you a detailed view of the process that is happening and how it relates to 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 a later 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 loop** family.

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.
- 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 either defined in the initialization or within the repeat decision (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 repeat decision evaluates to `FALSE` (v not in `seq`).

Notation

The `for` loop repeat statement is placed between parentheses after the `for` statement. Directly after the repeat statement, in curly braces, are the the instructions that are to be repeated. You will find the formatting that you like best for things such as loops and functions, but here is my preferred syntax:

```
seq <- 1:100
for(i in seq){
    statement
}
```

OR

```
num <- 100
for(i in 1:num){
    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)

print("This loop calculates the square of elements drawn from a t-distribution")

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]))
}

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
.
.
.
```

Exercise 7: How could the above calculation be done outside of a loop?

```
# code to execute squaring of the u vector NOT in a loop
```

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!

Exercise 8: Recursive for-loop!

Read in the BlackfootFish dataset and modify the code to write a `for`-loop to find the indices needed to sample every 7th row from the dataset, starting with the 1st row, until you've sampled 1900 rows.

```
BlackfootFish <- read.csv("data/BlackfootFish.csv", header = TRUE)

size <- 1900

samps <- NA
## initializing the samps vector, for storing indices

samps[1] <- 1
## setting first sample index to 1 (first row)

# code snippet:
# create the code for the process will be executed at each step
# e.g. how will you get the next sample after 1?

for(i in 2:#ending index here){
  samps[i] <- # process you execute at every index
}

testing <- BlackfootFish[samps, ]
training <- BlackfootFish[-samps, ]
```

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 = 10)

# verify the dimensions of mymat
dim(mymat)

## [1] 30 10

# 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] 10

# 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 13x10 piece
mymat[1:13, ]

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

But when should you use a loop? Couldn't you just repeat the set of instructions the number of times you want to do it?

A rule of thumb is that if you need to perform the same action in one place of your code three or more times, then you are better served by using a loop. A loop makes your code more compact, readable, maintainable, and saves you some typing! If you have the same process multiple places in your code, that's where functions come in handy!

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 an output.

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 a single output, but the best way to learn about the inner workings of functions is to write your own!

User Defined Functions

Similar to loops, you will often find that you have performed the same task multiple times. A good rule of thumb is that if you've copied and pasted the same code twice (so you have three copies), you should write a function instead!

Example: If I wanted to rescale every variable in my dataset so that the quantitative variables have values between 0 and 1. I could use the following formula:

$$y_{scaled} = \frac{y_i - \min\{y_1, y_2, \dots, y_n\}}{\max\{y_1, y_2, \dots, y_n\} - \min\{y_1, y_2, \dots, y_n\}}$$

The following R code would carry out this rescaling procedure for the `length` and `weight` columns of my data:

```
BlackfootFish$length <- (BlackfootFish$length - min(BlackfootFish$length, na.rm = TRUE))/  
  (max(BlackfootFish$length, na.rm = TRUE) - min(BlackfootFish$length, na.rm = TRUE))  
  
BlackfootFish$weight <- (BlackfootFish$weight - min(BlackfootFish$weight, na.rm = TRUE))/  
  (max(BlackfootFish$weight, na.rm = TRUE) - min(BlackfootFish$length, na.rm = TRUE))
```

I could continue to copy and paste for other columns of my data, but you can probably get the idea. This process of duplicating an action multiple times makes it difficult to understand the intent of the process. Additionally, it makes it very difficult to spot the mistakes. *Did you spot the mistake in the fourth line?*

Often you will find yourself in the position of needing to find a function that performs a specific 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!

Example: Let's transform the repeated process above into a rescaling function.

The following snippet of code rescales a column to be between 0 and 1:

```
(BlackfootFish$length - min(BlackfootFish$length, na.rm = TRUE)) /  
(max(BlackfootFish$length, na.rm = TRUE) - min(BlackfootFish$length, na.rm = TRUE))
```

Our goal is to turn this snippet of code into a general rescale function that we can apply to any numeric vector.

- The first step is to examine the process to determine how many inputs there are.
- The second step is to change the code snippet to instead refer to these inputs using temporary variables.

For this process there is one input, the numeric vector to be rescaled (currently `BlackfootFish$length`). We instead want to refer to this input using a temporary name. It is common (in R) to refer to a vector of data as `x`.

Exercise 9: Modify the code snippet above to instead refer to a temporary variable `x`.

```
# code modification with temporary variables
```

But now take a closer look at our process. Is there any duplication? An obvious duplicated statement is `min(x, na.rm = TRUE)`. It would make more sense to calculate the minimum of the data once, store the result, and then refer to it when needed. We also notice that we use the maximum value of `x`, so we could instead calculate the range of `x` and refer to the first (minimum) and second (maximum) elements when they are needed.

The `range` function in R takes a single numeric input and returns the minimum and maximum:

```
range(seq(1, 5))
```

```
## [1] 1 5
```

What should we call this variable containing the range of `x`? Some important advice on naming variables in R:

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

I suggest we call this intermediate variable `rng` (for range).

Exercise 10: Create an intermediate variable called `rng` that contains the range of `x`, using the `range()` function. Make sure that you specify the `na.rm()` option to ignore any NAs in the input vector.

```
# Create rng
```

```
# Rewrite the code snippet from Exercise 8 to now refer to rng
```

How does this process end up with us writing our own function? What do you need in order to write a function?

- The problem the function will solve (what you've been copying and pasting) **and**
- The inputs of the function (the intermediate variables)

We now have all these pieces ready to put together. It's time to write the function!

In R the function you define will have the following construction:

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

Exercise 11: Using the template above, write a function that rescales a vector to be between 0 and 1. The function should take a single argument, `x`.

```
# Use the function template to create a function named rescale01 that perfoms the process outlined above  
  
myfunction <- function(arg1,...){  
  # body  
}
```

Once you have written your function, the next step is to test it out. The simplest place to start is to set value(s) for the function's argument(s) (e.g. `x <- c(1, 2, 3, 4, 5)`). You can then run the code *inside* the function, line by line, to make sure that each line successfully executes for this variable. This is what we call unit testing in computer science. This is a similar process to testing out a `for-loop`, by setting the value of the index (`i <- 1`) and running the inside of the loop.

Exercise 12: Test your function on a simple vector, with the same name as your function's input. What do you expect the values of the test to be after you input it into your function?

```
# Test your function out on a simple vector named x
```

If your unit test was successful, now you can move on to testing your function with your data.

Exercise 13: Test your function on the `length` column of the `BlackfootFish` dataset. Inspect the rescaled values. Do they look correct?

```
# Test your function out on the BlackfootFish data
```

Process of Creating a Function

You should **never** start writing a function with the function template. Instead, you start with a problem that you need to solve and work through the following steps:

1. Define the problem you need to solve.
2. Get a working snippet of code that solves the simple problem. (*Where you start when you're copying and pasting*)
3. Rewrite the code snippet to use temporary variables.
4. Rewrite the code for clarity (remove redundancy or multiple calculations of the same value).
5. Turn everything into a function! The code snippet forms the body of the function, the temporary variables are the arguments, and you choose the name of your function.
6. Test your function! Start with a simple example and then go big!

Function Scoping

When writing and using functions, you should be careful to consider where each of the variables you are using are defined. Variables that are defined *within* a function belong to a local environment, and are not accessible from outside of the function. Take this example:

```
x <- 10
## globally defined variable

f <- function(){
  ## locally defined x and y
  x <- 1
  y <- 2
  c(x, y)
}

f()

## [1] 1 2
x

## [1] 10
```

We notice that the globally defined value of `x` is not called inside the function, as the function defines `x` within itself, so it does not search for a value of `x`! Now look at another example, where `x` is not defined within the function.

```
g <- function(){
  ## locally defined y
  y <- 2
  c(x, y)
}

x <- 15

g()
```

```
## [1] 15  2
x <- 20

g()

## [1] 20  2
```

Summary:

1. When you call a function, a new environment is created for the function to do its work in.
2. The new environment is populated with the argument values passed in to the function.
3. Objects are looked for *first* in the function environment.
4. If objects are not found in the function environment, they are then looked for in the global environment.

Exercise 14: Putting it All Together!

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

Now, given the matrix of fish lengths and weights above, use the tools from the workshop (`for` loop, function, conditional and relational statements) to:

- compute the condition index of each fish ($\text{condition} = \frac{\text{weight}^{1/3}}{\text{length}} * 50$)
- remove the fish from the dataset whose condition index is NA or less 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>

Workshop Materials & Recordings Available:

- through RStudio Cloud at: <http://bit.ly/intermediate-R>
- through Allison's personal website at: http://www.math.montana.edu/allison_theobold
- through the MSU Library YouTube channel: <https://www.youtube.com/watch?v=xV7JNWTfsww>

How to Learn More About R

This material is intended to provide you with an introduction to using R for scientific analyses of data. The best way for you to continue to learn more about R is to use it in your research! This may sound daunting, but writing R scripts is the best way to become familiar with the syntax. This will help you progress through more advanced operations, such as cleaning your data, using statistical methods, or creating graphics.

The best place to start is playing around with the code from today's workshop. Change parts of the code and see what happens! Better yet, use the code from the workshop to investigate your own data!

Workshop build on: 2019-08-29 09:43:34