COMP 120: Lab 2 Worksheet

In this lab, we will explore some more fundamentals of R, which were discussed in the lectures this week including: vector manipulation; conditions, branching, and looping; and user-defined functions. As with the previous lab, there are a number of tasks for you to complete, and these have been embedded into this document. Your job today is to work through these examples, and then work on the mastery tasks (in your own time). Once you have completed your mastery tasks, upload the finished script to Blackboard.

Remember that anything in a code block should be copied and pasted into the RStudio console so that you can see the desired result for yourself.

# Remember to set your working directory and library path (if working on the Student Desktop) at the start of your session. See the Lab 01 for advice on how to do this.

# Vector and matrices

We covered vectors in the previous lab and in the lectures this week. Here, we’ll delve into vectors just a little more, and then look briefly at matrices. It’s likely that you won’t use **all** of these commands in COMP 120, but they’re useful to know, anyway.

Once declared, you can do quite a lot with vectors. Consider the following chunk of R:

vec1 <- c(1,4,6,8,10)

vec1

vec1[5]

vec1[3] <- 12

vec1

vec2 <- seq(from=0, to=1, by=0.25)

vec2

sum(vec1)

vec1 + vec2

* In line 1, a vector vec1 is explicitly constructed by the combine function c(), which was introduced in the previous lab. Elements in vectors can be addressed by standard [i] indexing, as shown in lines 3-4. Note the first element of the vector starts at index 1.
* In line 3, we inspect the value stored in the fifth element of vec1
* In line 4, one of the elements (the 3rd element) of vec1 is replaced with a new number (12).
* Line 6 demonstrates another useful way of constructing a vector: the seq() (sequence) function.
* Lines 8-9 show some typical vector-oriented calculations. Note that the function *sum* (in line 8) sums up the elements within a vector, leading to one number (vector of length 1). If you add up two vectors of the same length (as shown in line 9), the first elements of both vectors are summed, and the second elements, etc., leading to a new vector of length 5.

The result of executing the chunk on the previous page (i.e., what you’d see in the console output) is:

## [1] 1 4 6 8 10

## [1] 10

## [1] 1 4 12 8 10

## [1] 0.00 0.25 0.50 0.75 1.00

## [1] 35

## [1] 1.00 4.25 12.50 8.75 11.00

Question: What would be the result of:

sum (vec1 + vec2)

mean(vec1 + vec2)

## Matrices

Matrices are nothing more than 2-dimensional vectors. To define a matrix, use the function matrix:

mat <- matrix(data=c(9,2,3,4,5,6),ncol=3)

mat

## [,1] [,2] [,3]

## [1,] 9 3 5

## [2,] 2 4 6

The argument **data** specifies which values should be in the matrix. Use either ncol to specify the number of columns or nrow to specify the number of rows. Note that the data in a matrix will be filled in a column-wise fashion.

Matrix-operations are similar to vector operations:

mat[1,2]

## [1] 3

mat[2,3] <- 10

mat[2,]

## [1] 2 4 10

mean(mat)

## [1] 5.5

* Elements of a matrix can be addressed in the usual way: [row,column] (line 1). The result of executing line 1 is shown in line 2.
* Line 3: One value is replaced by another.
* Line 4: When you want to select a whole row, you leave the spot for the column number empty (the other way around for columns of course). Line 5 shows the result of executing line 4.
* Line 6 shows that many functions also work with matrices as argument. Line 7 shows the output.

Tasks:

1. Print the values stored in the third column.
2. Change the value stored in the last column of the last row to 100.
3. To change the filling order by row, you should add a third argument when creating the matrix: byrow = TRUE. Modify the code that creates the matrix mat to fill values in a row wise fashion. Then, print the values in the matrix.

# Other methods of indexing into vectors

Another way of indexing into one vector is to use the elements of a second (numerical or logical) vector. Consider the following:

X <- seq(25, 30)

X

## [1] 25 26 27 28 29 30

i <- c(1, 3, 5)

j <- c(FALSE, FALSE, TRUE, TRUE, FALSE, FALSE)

X[i]

## [1] 25 27 29

X[j]

## [1] 27 28

* Line 1: Creates vector X comprising five numbers (25 to 30).
* Line 4: Creates vector i comprising three numbers: 1, 3 and 5.
* Line 5: Creates vector j comprising five logical values.
* Line 6: The elements in X that have indices corresponding the values of i are printed (i.e., values of X[1], X[3] and X[5] are printed).
* Line 7: The elements in X for which the logical values are true in j are printed (i.e., third and fourth elements).

The notion of checking for conditions that result in TRUE or FALSE decisions (i.e., logical tests) was introduced in lecture 4. Let us look at some examples.

marks <- c(100, 89, 34, 48, 56, 76)

marks > 50

## [1] TRUE TRUE FALSE FALSE TRUE TRUE

marks == 100

## [1] TRUE FALSE FALSE FALSE FALSE FALSE

* Line 2 above checks whether each value in marks is greater than 50. If the test is successful for a value being tested, it returns TRUE, otherwise FALSE. The result is a logical vector of length five.
* Line 4: Checks whether each value in marks is equal to 100. Note == checks for equality of values.

**Logical tests** on variables (such as the ones shown above) can be very useful for indexing. For example, we can use the result from the comparison of one vector to index a related vector:

age <- c(25, 30, 22, 40, 33, 24)

sex <- c("M", "F", "M", "F", "U", "U")

males <- sex == "M"

males

## [1] TRUE FALSE TRUE FALSE FALSE FALSE

mean(age[males])

## [1] 23.5

* Lines 1 and 2 above should be easy to understand. In line 2, M, F and U stand for males, females and unspecified respectively.
* Line 3: The variable males contains logical values (TRUE or FALSE) corresponding to the evaluation of the test which checks whether the sex variable contains the value M. The result of this computation is shown in line 5.
* Line 6: prints the mean age of all males.

Questions: How will you compute the mean ages of all females?

How will you print the ages of all those people whose ages are above 30?

**NOTE:** the double equals sign (==) in the condition in line 3. Other conditions (also called logical or Boolean operators) are <, >, != (not equal), <= (less or equal) and >= (greater or equal). To test more than one condition in one if-statement, use && if both conditions have to be met (“and”) and || if one of the conditions has to be met (“or”).

Answer the following self-assessment questions based on the variables that you have created above:

1. What is the result of age > 25? Explain what the code does.
2. What is the result of age[age > 25]? Explain what the code does.
3. What is the result of sum(age[age > 25])? Explain what the code does.
4. What is the result of sex[age > 25]? Explain what the code does.
5. What is the result of sum(males)? Explain what the code does. Note: this is the same as: sum (sex == "M")
6. Write a single line of code that prints the ages of people who are than or equal to 30. The output of this line of code should be 25, 30, 22, 24.
7. Write a single line of code that prints the number of people who are than or equal to 30. The output of this line of code should be 4.
8. What is the result of ages\_1 <- c(age, c(90, 35, 75, 89, 98, 100))? Explain what the code does.
9. What is the result of ages\_2 <- age + c(90, 35, 75, 89, 98, 100)? Explain what the code does.
10. Do you understand the difference between the lines of code in lines 5 and 6? Which of these two lines of code achieves the objective of storing the ages of 12 people?

# Flow Control in Scripts

When writing scripts, you’ll often want to vary what parts of the script are executed based upon the state of variables in the R environment. Alternatively, you may want to run the same part of your script multiple times in response to the state of variables in your environment (e.g., to apply the same operation to all the elements in a vector). We call these two concepts **branching** and **looping** and they provide a simple but effective structure for creating flexible programs.

## Branching

The concept of branching was introduced in Lecture 4. The main construct of branching is the if () statement, which conditionally executes a block of instructions when a given test returns a TRUE result. For example, we may test the state of a variable (e.g., being equal to a required value) and execute the underlying block if the test is TRUE:

sex <- "M"

if (sex == "F") {

message("The test was TRUE, so this message was printed")

}

message("End of script.")

If the above block of code was executed in R, then the resulting output would be:

## End of script.

as test of the sex variable (which checks to see if its value is equal to "F") would equate to FALSE. In comparison the following modification to this code (which changes the test performed within the if statement to check if the sex variable is equal to "M"):

sex <- "M"

if (sex == "M") {

message("The test was TRUE, so this message was printed")

}

message("End of script.")

would produce a different result:

## The test was TRUE, so this message was printed

## End of script.

The if () statement becomes very useful when we want to choose between several blocks of code to execute as the result of multiple different tests. We can selectively execute these blocks through the use of if (<TEST>) { } else { } statements. For example, consider the following block of code:

age <- 25

if (age < 16) {

message("Person is too young to have a NZ driving licence")

} else if (age < 75) {

message("Person can apply for a NZ driving licence")

} else {

message("Person needs medical clearance for a NZ driving licence")

}

The output of this block of code would be:

## Person can apply for a NZ driving licence

Question: What would be the output of that block of code had the age variable been set to 85?

## Looping using the for loop

There are many times where you may want to execute the same operations many times in your code. For example, you may want to apply a given operation to all the elements in a vector to compute a statistic (e.g., the mean). A naïve solution would be to copy and paste the operation multiple times in your script to achieve the required result. For example:

sum <- 0

X <- c(5, 1, 4, 3, 2)

sum <- sum + X[1]

sum <- sum + X[2]

sum <- sum + X[3]

sum <- sum + X[4]

sum <- sum + X[5]

mean <- sum / length(X)

would compute the mean of the elements of the vector X. However, doing this creates a few problems:

1. It is error-prone (e.g., it’s easy to make a mistake on a given line)
2. It is tedious to update (e.g., imagine if the vector X had 1000 elements!)
3. It assumes that you know how large the vector is while you are writing the script (and maybe the size of the vector won’t be known until the script is run).

To overcome this R provides a simple constructs called loops to make the code easier to read and maintain. Two common types of loops are the for() and while() loops. The same code to compute the mean of X using a for loop could be:

sum <- 0

X <- c(5, 1, 4, 3, 2)

for (Xi in X) {

sum <- sum + Xi

}

mean <- sum / length(X)

Such an approach would be easier to maintain, and much easier to read for long vectors.[[1]](#footnote-1)

Now, modify the code given above so that you only sum those values that are greater than 3. Hint: use the if construct you have learnt already within the for loop. What is the new value of mean?

For loops become quite useful when used in conjunction with the if () statement as you have already seen above. Another example of combining of loops and branching is to compute per-group means (averages) of ages for three groups – males (M), females (F) and unspecified (U). The lengthy(!) way to compute the means is given below. Execute the code below (except the very last line) to produce the result given in the last line. Look at the code carefully to work out the various constructs of programming used in the code (the use of variables, for loop and branching). If you do not understand, do not panic. Consult with the staff to understand what the code does.

age <- c(25, 30, 22, 40, 33, 24)

sex <- c("M", "F", "M", "F", "U", "U")

# The following three variables are used to store the counts of males, females and unspecified.

n.M <- 0

n.F <- 0

n.U <- 0

# The following three variables are used to store the sum of ages of males, females and unspecified.

m.M <- 0

m.F <- 0

m.U <- 0

for (i in 1:length(sex)) {

if (sex[i] == "F") {

n.F <- n.F + 1

m.F <- m.F + age[i]

} else if (sex[i] == "M") {

n.M <- n.M + 1

m.M <- m.M + age[i]

} else {

n.U <- n.U + 1

m.U <- m.U + age[i]

}

}

m.F <- m.F/ n.F # Average age of females (sum of ages/count)

m.M <- m.M/ n.M # Average age of males (sum of ages/count)

m.U <- m.U/ n.U # Average age of unspecified (sum of ages/count)

c(m.F, m.M, m.U)

## [1] 35.0 23.5 28.5

As you progress throughout the semester, you’ll make use of these constructs (combination of loops and if statements) several times to inspect your data and perform per-group analysis.[[2]](#footnote-2) You’ll be pleasantly surprised in the mastery task that you are able to rewrite this lengthy code in a few lines!

## Looping using the while loop

The for loop construct is great if you can determine the exact number of times the loop needs to execute (e.g., you know the size of the vector). However, there are times when you may want to execute the same instructions multiple times, but don’t know exactly how many iterations you need. In such situations, if you can perform a test to determine when to stop performing a loop, you can use a while () loop. For example, the code on the next page uses a while loop to compute the greatest common divisor of two integer values. The [greatest common divisor](https://en.wikipedia.org/wiki/Greatest_common_divisor) (GCD) is the largest number that can divide both the numbers without any remainder (i.e. remainder of 0). The code to compute GCD is given below:

A <- 124

B <- 44

while (A != B) {

if (A > B) {

A <- A - B

} else {

B <- B - A

}

message("A = ", A, " B = ", B)

}

which will produce the following output in the console:

## A = 80 B = 44

## A = 36 B = 44

## A = 36 B = 8

## A = 28 B = 8

## A = 20 B = 8

## A = 12 B = 8

## A = 4 B = 8

## A = 4 B = 4

**Observe that the code doesn’t explicitly state how many times the inner block will execute**, and indeed for different combinations of A and B, you should observe a different number of messages (and so a different number of times the loop is executed). Execute the code for different combinations of A and B (e.g. 99 and 18, 1024 and 7).

# User-defined Functions

Functions you program yourself work in the same way as pre-programmed (built-in) R functions. While it is easy to use existing functions for common tasks such as computing *sum* and *mean* since they are readily available (using sum() and mean() functions), one must create new functions to instruct the computer to perform specific tasks. Once created, these functions can be reused just like the built-in functions.

A simple function is created as follows:

fun1 <- **function**(arg1, arg2) { # Takes two arguments arg1 and arg2

w <- arg1^2 # w is arg1 \* arg2

**return**(arg2+w) # returns arg2 + w

}

fun1(arg1=3, arg2=5)

## [1] 14

fun1(arg1=4, arg2=5)

## [1] 21

fun1(arg1=2, arg2=3)

## [1] 7

fun1(2, 3)

## [1] 7

Typically, you’d create your own functions by first writing the statements for the function in a script (and testing that they work correctly!) and then wrapping them in a function () statement. For example, you could take the code written in the previous section to compute the greatest common divisor and easily turn it into a function by using the following (where the newly added lines of code are highlighted in red):

**GCD <- function(A, B) {**

while (A != B) {

if (A > B) {

A <- A - B

} else {

B <- B - A

}

}

**return(A)**

**}**

# The following line of code calls the function by passing two values

GCD(124, 44)

## [1] 4

Now, call the GCD function with different values for the two arguments.

# Extra tasks (optional)

Here are a couple of extra tasks that may help you with completing certain mastery tasks.

1. Write a simple function called add\_two\_numbers that adds two numbers which are provided as arguments to the function. The function returns the sum of two numbers. To test the function call add\_two\_numbers(10, 20) and the function should return 30.
2. Write a function that takes two arguments V and T. The function should return the occurrences of T in V. Name the function count\_numberOf\_instances. To test the function call count\_numberOf\_instances(c(80, 40, 20, 20), 20) and the function should return 2. This is because 20 appears twice in the first argument.

# Mastery Tasks

Through the lab tasks so far, you have learnt the basic set of skills to perform R programming. You must now complete the following mastery tasks to demonstrate your understanding of the concepts. Put all your completed mastery tasks into mastery-02.R given to you in the order that they appear below. Place a short comment before each block of code that you use to complete a given task to indicate which example it corresponds to (e.g. code block 3 of page 3 of lab 2 document or slide 3 of the lecture 3). When you have completed the tasks, **submit your work on Blackboard before 4pm on Thursday the 23rd of July**.

1. Rewrite the code example that starts at the end on page 6 of this document (relating to per-group computing of means) so that it performs the same work using the mean function and logical indexing (as discussed on page 3). Your code should print the means of ages for males, females and unspecified. Hint: If you have completed the lab tasks for today, you would have already written the code to print the means of ages for males and females.
2. Create a vector using the c() function (discussed in the previous lab) containing the elements 1, 10, 2, 9, 3, 8, 4, 7, 5, 6. Assign this vector to the variable name A. Now, answer the following questions.
3. Write a comment explaining the result of the following line of code.

A!=10

1. Write a comment explaining the result of the following line of code.

A[A==10]

1. Write a *single line of code* that prints whether the value of each element in A that is greater than or equal to 8.
2. Write a *single line of code* that prints the total number of elements in A that are less than 5.
3. Write a *single line* of code that #e.(14, 19, 23, 25, 99) to A. (i.e., A originally has 10 elements, now we want to add five more elements indicated above to the vector such that A contains 15 elements – i.e., 10, 2, 9, 3, 8, 4, 7, 5, 6, 14, 19, 23, 25, 99).
4. Write a *single line* of code that prints the elements whose index is odd (i.e., first, third, fifth etc. all the way to element 15). The output should be: 1, 2, 3, 4, 5, 14, 23 and 99.

***Note: The answers to questions 2c-f above should not use loops.***

1. This task involves writing several lines of code, one corresponding to each sub-task described below.
   1. Using sequence function create a vector that contains all odd numbers (whole numbers) that are less than 24 and assign it to a variable called odd12.
   2. Using odd12, create a 3 by 4 matrix (i.e., 3 rows and 4 columns)
   3. #bcalled odd\_matrix.
   4. Print the average of all values of the matrix.
   5. Replace all values in the third row by log(-5).
   6. Replace the value in the second row, second column by a NA.
   7. Write code that checks whether each element in the matrix is a NaN.
   8. Print all the values in the first column.
   9. Print the value stored in the second element of the fourth column.

1. Create a function called count\_elements that takes two arguments: a vector V and a target T. The function uses *for loop* to count the number of elements in V that are greater than T. After you have written this function, test this function by calling count\_elements(c(97, 48, 26, 57, 64), 50). The result of this test should be 3.
2. Implement a function called fizzbuzz. It takes a single number as input called num. If the number is divisible by three, it returns “**fizz**”. If it is divisible by five it returns “**buzz**”. If it is divisible by three and five, it returns “**fizzbuzz**”. Otherwise, it returns the number that is provided as the input. Hint: To check whether a number (say A) is divisible by another number (say B), you should use the mod operator (given by the symbol %%). A %% B gives the remainder of division. If A is 5 and B is 3, the result of A %% B will be 2 (it is the remainder when we divide 5 by 3). If A is 6 and B is 3, then A %% B will be 0. If A is 7 and B is 3, then A %% B will be 1.

When the fizzbuzz function is tested it produces the following outputs.

fizzbuzz(1) produces 1

fizzbuzz(3) produces fizz

fizzbuzz(5) produces buzz

fizzbuzz(15) produces fizzbuzz

fizzbuzz(120) produces fizzbuzz

1. Of course, an even easier and more appropriate approach would be to use the built-in mean() function to compute the mean of X. The resulting code would simply be: m <- mean(X) [↑](#footnote-ref-1)
2. Actually, this is such a common construct that R provides a simple function to make this much simpler. The code could be replaced with tapply(age, sex, mean) and you’d get the same result. You can use the online help to learn about this function: ?tapply. [↑](#footnote-ref-2)