COMP 120: Lab 3 Worksheet

In this lab, we will explore some more fundamentals of R, specifically looking at creating and using lists, manipulating data in data frames, reading and writing files, creating tibbles (a more user-friendly version of data frames) and finally creating factors.

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.

# Data Structures in R - lists and data frames

## Lists

Lists are the structure used in R to hold arbitrary elements of related data. The elements may have different data types (i.e., heterogenous), be different lengths, or even other lists themselves. This makes lists very powerful and a flexible way of returning data from a function. In COMP 120, you’ll only encounter lists a few times, but it is still useful to know how they are created and (more importantly) accessed once created.

Creation of a list is done through the list() function, in which you simply pass the required elements of the list as named arguments:

my.list <- list(paper="COMP 120", year=2019, data=rnorm(50))

my.list

which would produce output similar to the following:

## $paper

## [1] "COMP 120"

##

## $year

## [1] 2019

##

## $data

## [1] -0.5948051 2.1544573 -0.6553934 -0.2511621 -1.0285264 -0.2924925

## [7] 0.8521612 0.6583955 0.1469885 1.4587802 -0.5523227 0.3350335

## [13] -0.7050069 0.2916799 -0.6951939 -0.3905599 -0.1257651 0.6519869

## [19] 1.3169142 0.4443961 -0.0916359 -0.1109366 1.7851183 -0.6521358

## [25] -1.0181422 -1.3035747 -1.1642061 0.2031387 -1.0524980 0.6854540

## [31] 1.3167648 -1.2861468 -1.8371779 1.4186738 -0.1998487 0.9870309

## [37] 0.6709705 -0.4731205 -1.1875298 0.9415232 -0.4440412 -1.9125416

## [43] -0.7509610 1.5265595 0.4145217 0.2380918 0.2907978 -0.2048037

## [49] -0.1109350 0.1588699

Now, use the structure function that was introduced in the lectures that presents the details of the list more compactly – i.e., str(my.list). Compare the result with output for my.list.

If we do not supply any arguments to list(), then it creates an empty structure into which elements can be later added:

empty.list <- list()

empty.list

## list()

Once we have created a list and assigned to a variable name, we can access and modify its contents through the dollar ($) operator:

empty.list$X <- "New Element"

empty.list

## $X

## [1] "New Element"

my.list$data <- rnorm(10)

my.list$sequence <- seq(0, 1, 0.2)

my.list

## $paper

## [1] "COMP 120"

##

## $year

## [1] 2019

##

## $data

## [1] 0.57996417 -0.89012463 0.02328478 0.65877827 1.49402031

## [6] -0.90832468 -1.96383616 -0.01803388 0.84255014 -0.62633300

##

## $sequence

## [1] 0.0 0.2 0.4 0.6 0.8 1.0

Finally (for lists), we can remove an element in a list by setting it to NULL, for example:

my.list$year <- NULL

my.list$data <- NULL

my.list

## $paper

## [1] "COMP 120"

##

## $sequence

## [1] 0.0 0.2 0.4 0.6 0.8 1.0

Apart from using element names as shown above to access and modify values, you can also use indices. However, to access an element of a list you will use the [[]] syntax (double square brackets). If you just use one square bracket [] it returns a sub-list, but that is not what we want in many cases.

my.list[[1]]

## [1] "COMP 120"

my.list[[2]]

## [1] 0.0 0.2 0.4 0.6 0.8 1.0

my.list[[2]][1]

## [1] 0.0

Note that my.list[[2]] in line 3 above returns a vector of values (shown in line 4). To access the first element of this vector, we use the vector indexing symbol, the [] syntax (square bracket) as shown in line 5. Now, a couple of questions:

1. How would you access the third element (0.4) inside the sequence element in my.list?
2. What is the ouput of my.list[["paper"]]? Hint: this is alternative to the code in line 1 in the block above.

The examples above demonstrate that there are a few ways to access the elements in a list. If the double square brackets are confusing, stick to accessing your lists by names (e.g. my.list$paper). However, you may realize that if you are using somebody else’s code and they haven’t provided a name for an element, then you may have to resort to accessing the value of an element in the list using the [[]] syntax. Ideally, you’ll need to remember different ways of accessing elements in a list.

## Data Frames

Data frames are easily the most common structure that you will deal with in R. They are the framework through which data is loaded and exported in R, and form the basis from which most modelling takes place. Therefore, you need to become comfortable with creating and using data frames.

Data frames are created using the data.frame() function:

new.df <- data.frame(id=rep(seq(5), times=2), value=rnorm(10),

constant=1)

new.df

## id value constant

## 1 1 -1.2997589 1

## 2 2 2.3356838 1

## 3 3 -1.1694459 1

## 4 4 -0.4065512 1

## 5 5 1.7660844 1

## 6 1 -1.2552799 1

## 7 2 2.3001079 1

## 8 3 -1.3390491 1

## 9 4 0.8870049 1

## 10 5 1.0964537 1

Unlike generic lists, the elements of a data frame ***must be the same length*** (or a length that can be multiplied by a whole number to become the same length as the biggest element). This has happened to the constant column in the example above.

As with lists, elements (columns) can be added, modified, or removed through the $ operator:

new.df$constant <- NULL

new.df$value <- rnorm(10)

new.df$other <- sample(1000, 10) # samples 10 values from 1:1000

new.df

## id value other

## 1 1 -2.5125718 497

## 2 2 1.6191584 734

## 3 3 1.3219554 510

## 4 4 -0.7913144 808

## 5 5 -0.1086636 484

## 6 1 -0.3072889 15

## 7 2 1.3504565 651

## 8 3 0.1552573 944

## 9 4 -0.4047019 911

## 10 5 -1.3658272 412

In a similar approach to vectors, specific rows and/or columns of data frames can also be accessed like an array using square brackets:

new.df[c(1,3,5), ] ## notice the comma! Nothing AFTER the comma is

## shorthand for "all columns"

## id value other

## 1 1 -2.5125718 497

## 3 3 1.3219554 510

## 5 5 -0.1086636 484

new.df[ , c(1,3)] ## notice the comma! Nothing BEFORE the comma is

## shorthand for "all rows"

## id other

## 1 1 497

## 2 2 734

## 3 3 510

## 4 4 808

## 5 5 484

## 6 1 15

## 7 2 651

## 8 3 944

## 9 4 911

## 10 5 412

new.df[c(1, 10), 3]

## [1] 497 412

Here are a couple of questions to make things more concrete:

1. Print the contents of the other column in the new.df data frame (try different ways of doing this – using $ operator and []).
2. Print the contents of the value column for all even numbered rows. There are different ways of achieving this result (e.g. using c() function, or using a seq() function or a loop as a part of the code).

You can use functions such as nrow() and ncol() to find the number of rows and columns in a data frame by passing the data frame as the argument. rownames() and colnames() functions show the row and column names of a data frames that are passed as arguments to these functions respectively. Try these functions out on the new.df data frame.

# Loading and Writing Data Frames using read\_delim() and write\_delim()

The previous examples have all dealt with data frames that you have created by hand using the data.frame() function. Realistically, the majority of times that you will use data frames will be for importing data from external sources (e.g., files) into R. There are a handful of ways to perform this: in COMP 120 we demonstrate the use of the readr package that is part of Hadly Wickham’s tidyverse package. This package offers a very simple mechanism that should be sufficiently flexible to open most of the typical file types you will encounter.

Before proceeding with these exercises, make sure that you’ve downloaded the file INFO204-2017.tsv from Blackboard, and saved it to your working directory. Also, you should ensure that the tidyverse package is loaded into R:

library(tidyverse)

## Loading data with read\_delim()

Open the provided INFO204-2017.tsv file in a text editor (preferably something like notepad++ or notepad on Windows) and get a general feel for the structure of the flat file. In particular, you should note the following:

1. Is there a clear field (column) separator in the file?
2. Does the file have column headings?
3. Is there a special symbol (or multiple symbols) in the file to denote missing (NA) values?

Once you have worked out these three things, you typically have enough information about the structure of the file to be able to load it into R. In this case, the provided file is **delimited by tab characters**, **has column headings**, and **uses a question mark** (?) to indicate that a value is missing. Now, to read the file, we issue the following command:

INFO204 <- read\_delim("INFO204-2017.tsv", delim="\t", na="?")

INFO204

# A tibble: 53 x 7

StudentID FormalName A1 A2 A3 Tutorials Exam

<int> <chr> <dbl> <dbl> <dbl> <dbl> <dbl>

1 364497 Barnett, Patricia 13.5 8.00 14.2 4.38 45.5

2 460664 Lambert, Alfonso 13.5 5.25 13.0 4.22 29.0

3 153697 Chandler, Roosevelt 12.5 5.25 10.5 0.630 14.5

4 236466 Watkins, Marc 12.5 4.50 10.5 1.41 18.5

5 583165 Brady, Kayla 13.0 10.0 13.2 5.00 52.0

6 837258 Cole, Dana 10.5 8.75 10.5 5.00 40.0

7 563138 May, Kevin 9.50 3.50 4.00 0.630 17.0

8 676795 Bowers, Alma 12.5 NA NA NA NA

9 905474 Mclaughlin, Gretchen 9.50 8.75 11.8 4.38 35.5

10 833059 Farmer, Sherri 12.5 8.00 10.5 NA 36.0

# ... with 43 more rows

There are a couple of things to note about this function call. The first is that the result of the function call is a special type of data frame (called a tibble) designed to make the display of the data easier to read (as discussed in the lectures – we’ll also discuss this here shortly). Second, you should see that the read\_delim() function was called with three arguments:

1. the first argument is the path (relative to the working directory) of the file that we wish to read into a data frame;
2. the second argument tells the function that we’re using tab characters to separate columns (\t is the code used to represent tab characters); and
3. the third argument tells the function that missing values are represented by a question mark.

Naturally, if any of the above three arguments changed (e.g., columns were separated using semicolons), then the call to the read\_delim() function would change accordingly. This gives us a lot of flexibility in using the read\_delim() function and allows it to load many different files types.

## Writing data frames to files using write\_delim()

The read\_delim() function is intuitively matched with a corresponding function to write data from a data frame out to a file. The format for the corresponding write\_delim() function is almost identical, with the exception that it takes one more argument (the name of the data frame that we wish to export). Therefore, we could write out the INFO204 data frame from the previous step using the following command:

write\_delim(INFO204, "INFO204-revised.tsv", delim="\t", na="?")

## Shortcut functions for reading tabular data

There are a number of well-known and frequently used file types for storage of tabular data (e.g., comma-separated, or CSV, files). Rather than having to memorise how to tweak the read\_delim() function for each of the most popular file types, the readr package provides a helper function to wrap up the required details, for example:

* read\_csv(), to read data in which columns are separated by commas
* read\_tsv(), to read data in which columns are separated by tabs
* read\_table(), to read data in which columns are separated by spaces

These are just a few examples. There are also examples for Excel spreadsheets and for other file types from common statistical packages (e.g., SPSS). Finally, to close the loop for every read function, there is a corresponding write function to help export the data frame to a file of the required type ( write\_csv(), write\_tsv() etc.). Now, a couple of tasks for you to do:

1. Write the data in the INFO204 data frame to a file named INFO204-CSV.csv using the appropriate helper function.
2. Write the data in the INFO204 data frame to a file named INFO204-TSV.tsv using the appropriate helper function.

# Easier data frames using tibble

This lab has explored the use of data frames, which are the default mechanism for storing tabular data in R. However, you should have noticed that the examples in the previous section related to reading data produced a different kind of structure called a tibble. Tibbles are a relatively recent addition to R through the tidyverse. They are essentially just a data frame, but they provide a couple of useful improvements, mostly in the area of printing, where they provide additional information about columns and truncate output to make it easier to read (and these were discussed in the lectures). Tibbles are created in three ways:

1. by replacing the data.frame() function with the tibble() function;
2. by using the as\_tibble() function, passing in an existing data frame; or
3. by using almost any of the functions in the tidyverse that produce data.

Given the simplicity with which tibbles are created, there really doesn’t seem to be too much benefit to using raw data frames, so we would recommend their use wherever practical.

# Factors

As discussed in the lectures, factor is a data structure used to represent *categorical variables* that contain finite number of values. For example, marital status of person can have one of five values - single, married, separated, divorced and widowed. Typically, marital status variable is created as a character vector comprising five values. However, sometimes, we need these to be internally stored as integers (even though they appear as characters). This is where we need factors. We will create a factor variable based on marital status variable. Once a factor is created, the values in a factor will be integers. While factors look (and often behave) like character vectors, *they are actually integers under the hood, and you need to be careful when treating them like strings*. Factors are very useful for statistical analysis and plotting graphs, and you will appreciate this better in the coming labs. We will return the martial status variable towards the end of this lab.

Now, let us create a factor to represent the credit risk of banking customers using the code given below. First, we create a variable called credit\_risk that stores the credit risk of five customers (stored as a character vector). Then, we create a factor called credit\_risk\_factor, by calling the factor() function and passing credit\_risk as the argument. Then, we examine what is stored in the credit\_risk\_factor as shown in the snippet given below.

credit\_risk <- c("high", "low", "medium", "medium", "low")

credit\_risk\_factor <- factor(credit\_risk)

credit\_risk\_factor

[1] high low medium medium low

Levels: high low medium

Once a factor is created, it contains predefined set of values called *levels*. The levels are shown in the output printed in the last line in code block shown above. There are three values in levels which are high, low and medium. Each level has an underlying integer value. You can see those values by using the str() function as shown below. The value for *high* is 1, *low* is 2 and *medium* is 3. These numbers are assigned based on alphabetical ordering (i.e., in alphabetical ordering, *high* comes before *low* which in turn comes before *medium*). The numbers in the output (i.e., 1 2 3 3 2), representing the underlying integer values, correspond to the credit\_risk character vector that contains: high, low, medium, medium and low.

str(risk\_factor)

Factor w/ 3 levels "high","low","medium": 1 2 3 3 2

We can change the alphabetical ordering, to the order that we want, by specifying the *labels* argument when a factor is created, as shown in the code below (see code highlighted in green). As can be seen in the output below, *low* now gets the value 1, *medium* gets the value 2, and *high* gets the value 3. These values were assigned based on the order we specified in the *levels* argument (i.e., low appears medium which appears before high). The numbers in the output (i.e., 3 1 2 2 1), representing the integer values, correspond to the credit\_risk character vector that contains: high, low, medium, medium and low.

credit\_risk\_factor <- factor(credit\_risk, levels = c("low","medium","high"))

str(credit\_risk\_factor)

Factor w/ 3 levels "low","medium",..: 3 1 2 2 1

A factor can be *ordered* or *unordered*. An unordered factor is one where there is no preference ordering. For example, if you create a factor for gender values (say male and female), there is no preference ordering between genders (i.e., one gender is not less than or greater than the other, and the ordering does not make any sense!). Similarly, there is no preference ordering for the marital status variable also. However, in some cases, we want to create an ordered factor where a value of a factor is expected to be less than or greater than another value. An example is the credit risk factor where lower risk value is less than medium risk. To create ordered factors, we just need to add the argument ordered = TRUE to the factor function as shown below (see code highlighted in green). When str() is used to print the output, the ordering details are presented (i.e., "low"<"medium"< "high"). This output shows that the value for low is less than that of medium and so on. Creating ordered factors allows for comparisons of values to be made. For example, the code credit\_risk\_factor[1] < credit\_risk\_factor[2] will result in FALSE because the value for the first element (high, which is 3) is greater than the value for the second element (low, which is 1).

credit\_risk\_factor <- factor(credit\_risk, levels = c("low","medium","high"), ordered = TRUE)

str(credit\_risk\_factor)

Ord.factor w/ 3 levels "low"<"medium"<..: 3 1 2 2 1

credit\_risk\_factor[1] < credit\_risk\_factor[2]

FALSE

A question – what will be result of credit\_risk\_factor[3] > credit\_risk\_factor[5]?

We can also set the labels for a factor using the *labels* argument within the factor function. This was discussed in the lectures. We can use the table() function to print a table of the counts of each combination of factor levels – try table(credit\_risk\_factor). Do you understand what the output shows?

We will use factors in the later labs when we get to plotting (i.e., creating graphs). It is now time to get to the mastery tasks for this week.

# Mastery Tasks

By this point, you should have a basic grasp of the structures used to maintain data in R. You must now complete the mastery tasks to demonstrate your understanding of the concepts. For this week, we have provided an *incomplete script* (mastery-03.R) for this week’s tasks that you can download from Blackboard. Put all your completed mastery tasks into the provided script (i.e., mastery-03.R). When you have completed the tasks, **submit your work on Blackboard before 4pm on Thursday the 30th of July**.