COMP 120: Lab 5 Worksheet

In this lab, we will explore some of the basic data type conversions you will frequently encounter in everyday problems. We will also demonstrate some of the routines provided by the tidyverse packages that ease the manipulation of data in data frames (or tibbles).

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.

# Basic Data Type Conversion

The elementary type conversions were introduced in the lectures this week. Here we will focus on: cleaning up character data prior to data conversion; converting text data into numeric and factor data; and converting (binning) numeric data into factors.

## Checking Data Type

The data type of any piece of data can be examined through the typeof() function. For example:

a <- 1

b <- "1"

c <- "one"

d <- 5 > 3

typeof(a)

[1] "double"

typeof(b)

[1] "character"

typeof(c)

[1] "character"

typeof(d)

[1] "logical"

typeof(NA)

[1] "logical"

typeof(1 + 3)

[1] "double"

If you are ever performing a calculation in R, and the result is not what you anticipated or seems erroneous, then you should start your error checking by applying the typeof() function to your inputs.

## Converting Data Types

As discussed in the lectures, explicit type conversion can be done using the as.\*() functions. For example, to convert our variables from their current type into numeric values, we use the as.numeric() function:

as.numeric(a)

[1] 1

as.numeric(b)

[1] 1

as.numeric(c)

[1] NA

Warning message:

NAs introduced by coercion

as.numeric(d)

[1] 1

typeof(as.numeric(a))

[1] "double"

typeof(as.numeric(b))

[1] "double"

typeof(as.numeric(c))

[1] "double"

Warning message:

In typeof(as.numeric(c)) : NAs introduced by coercion

typeof(as.numeric(d))

[1] "double"

Note that the conversion of c from the character value "one" returned an NA value and gave us a warning, but did not fail with a clear error. This is an important thing to note – you will need to handle such cases manually and carefully!

## Cleaning Character Data

Many times, data will be loaded into R as character data, when its natural form is another type, such as a factor or a number. For example, measurement data is often stored with the corresponding units (e.g., height in metres), and this has implications on how it can be used. Consider the following:

distance <- "125km"

time <- "1.5hr"

speed <- distance / time

Error in distance/time : non-numeric argument to binary operator

Clearly, the intended result (125 / 1.5 = 83.333km/h) could not be computed because of the inputs being character data types (remember, you could confirm this by using the typeof() function).

Once you have identified that the values are character data, and not the intended numeric data, your initial response might be to apply the as.numeric() function to your variables to get them into the right data type for computation. However, without additional consideration, this will lead to errors:

as.numeric(distance)

[1] NA

Warning message:

NAs introduced by coercion

Clearly, this is not the intended result. The reason for this is that R has no way of knowing the meaning of the units – we have to be explicit and help it by cleaning up the data prior to converting its type.

Cleaning up the data can be a long and difficult task, and could arguably be an entire course on its own. Here, we will provide a single example of using the str\_replace() function from the *stringr* package (which will be covered in detail in a later lecture) to remove the units from our data:

distance <- str\_replace(distance, "km", "")

time <- str\_replace(time, "hr", "")

With the units removed from the data, we can now convert the data as intended and perform the calculation:

as.numeric(distance) / as.numeric(time)

[1] 83.33333

It is very important to realise that no data cleaning can be performed without a proper understanding of the domain from which the data is sampled. Therefore, you must be very careful with your data cleaning, and seek advice on what are acceptable operations to perform!

# Managing Dates

As mentioned in the lectures, dates are very frequently encountered in human-centric data sources, and are frequently very difficult to interpret (for example, is 12/11/2018 the 12th of November 2018, or the 11th of December?). Date values are typically entered into R using character strings, and then need to be explicitly converted using functions. The lubridate package provides a number of functions to make date manipulation easier, as long as you are sure of the convention being used to represent the date. For example, if you know the date will be in the format of day-month-year, then you can use the dmy() function:

library(lubridate)

battle.of.hastings <- dmy("14-10-1066")

battle.of.hastings

[1] "1066-10-14"

There are equivalent functions for year-month-day (ymd()), hours:minutes:seconds (hms()), and so on. Again, if you are sure of the format that the date is represented in, then these serve as quick and effective conversion tools.

Sometimes, you will encounter a date representation that cannot be easily handled by the default functions. For example, the date “14-Oct-1066 09:00” (i.e., day-monthname-year 24hours:minutes) is not handled by the functions available in lubridate. To parse this type of date, we need to use the parse\_date\_time() function, and explicitly provide the markup of the date:

battle.of.hastings <- parse\_date\_time("14-Oct-1066 09:00",

orders="d-b-Y H:M")

battle.of.hastings

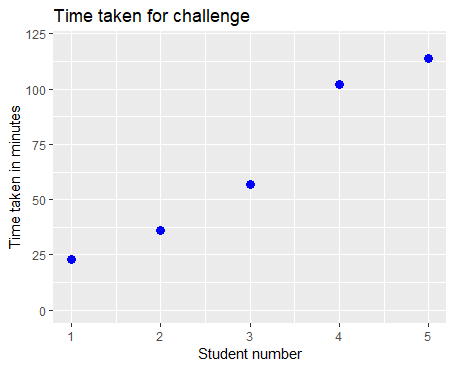
[1] "1066-10-14 09:00:00 UTC"

as\_date(battle.of.hastings)

[1] "1066-10-14"

The orders parameter is quite complex and can take a number of different options. These are comprehensively covered in the parse\_date\_time() help: ?parse\_date\_time.

**Task for you to do**: The data from the challenge task (covered in lecture 9) is available in challenge\_solution.csv on Blackboard. Your task is to follow the steps outlined in lecture 9 (including the creation of new columns and data conversion) and create the graph shown below. The size of blue data point in the graph is set to 4.



# Creating Factors (revisited)

You may recollect that we discussed factors in lecture 5 and lab 3 (and you were given the code to create one in the last lab). Here we revisit factors again (quickly, though!). Remember, factors are categorical data types and are very useful for grouping rows of data (e.g., grouping subjects in a clinical trial). When we load data in from a file using read\_\* functions, in the form of tibbles, factors are loaded in as character strings, and so we frequently need to create factors manually.

Factors are created using the factor() function. For example, the previous lab mentioned a gender variable that could easily be turned into a factor:

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

factor(gender)

[1] M U F F M U

Levels: F M U

By default, the levels (categories) in the factor are introduced in alphabetical order. To change this, we can specify the levels as a parameter, and control their order (which may be useful for plotting and printing, for example):

factor(gender, levels=c("U", "F", "M"))

[1] M U F F M U

Levels: U F M

Finally, to make the factors more meaningful, we can supply a list of labels to use in place of the (possibly cryptic) codes that were used for levels:

factor(gender, levels=c("U", "F", "M"),

labels=c("Unspecified", "Female", "Male"))

[1] Male Unspecified Female Female Male Unspecified

Levels: Unspecified Female Male

## Binning Numeric Data into Factors Using cut()

Often, we would like to take a collection of numeric data, and “bin” it into meaningful groups. For example, in a previous lab, we took a number of final marks for a paper (out of 100) and turned them into a corresponding letter grade (A+, A, …). In R, this is done using the cut() function (not discussed in the lectures). For example, we might group ages into preteen (0-12), teenager (13-19) and adult (20+). We would do this in the following way:

ages <- c(26, 41, 16, 9, 44, 43, 49, 23, 29, 3)

cut(ages, breaks=c(0, 12, 19, Inf),

labels=c("Preteen", "Teenager", "Adult"))

[1] Adult Adult Teenager Preteen Adult Adult Adult Adult Adult Preteen

Levels: Preteen Teenager Adult

Note that the vector supplied for the breaks parameter starts with zero (to denote the **start** of the first group) and ends with infinity (to denote then **end** of the final group). If you want to bin into N levels, you will need to define (N+1) breaks.

**Task for you to do:** Assume you have created a vector called *marks* (shown below).

marks <- c(96, 41, 61, 56, 47, 73, 66, 28, 35, 83)

Now, create a variable called *marks\_cut* using the cut function you have learnt, that cuts the *marks* into 4 bins: 0 to 49, 50 to 74, 75 to 89 and 90 to 100. The labels for these bins should be Fail, Pass, Merit and Excellent. Print the contents of marks\_cut.

# Manipulating Data Frames (and Tibbles) with mutate() and filter() and the Pipe (%>%)

Once you have some external data loaded into a data frame, you will usually want to add additional columns/calculations to the data frame or want to restrict (filter) the rows of the data frame to match some criteria. We showed you some basic ways of doing this “statically” in lab 3 (using the $ operator and logical vectors for indexing). However, it is often more convenient to do this “on the fly” as part of a bigger computation – to do this, the pipe operator (%>%, discussed in the last lecture) can be used in conjunction with the mutate() function to add/modify columns, or the filter() function to meet specific criteria. For example, we may create the following data frame:

age <- c(28, 3, 8, 48, 14, 2, 32, 42, 16, 28)

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

people <- tibble(age=age, gender=gender)

We may want to then manipulate this data frame to make the gender column a factor, and then add another column that indicates a target demographic of teenagers:

people %>%

mutate(gender=factor(gender, levels=c("U", "M", "F")),

Target.Demographic=age < 20 & age > 12)

# A tibble: 10 x 3

age gender Target.Demographic

<dbl> <fctr> <lgl>

1 28.0 U FALSE

2 3.00 M FALSE

3 8.00 F FALSE

4 48.0 U FALSE

5 14.0 F TRUE

6 2.00 F FALSE

7 32.0 U FALSE

8 42.0 F FALSE

9 16.0 F TRUE

10 28.0 U FALSE

Note the use of the ampersand (&) operator to test two conditions and return TRUE when both conditions are met and FALSE otherwise (a logical AND operation).

Rather than add a target demographic column, we could have alternatively filtered our data frame to select just the rows that match. For example:

people %>%

filter(age < 20 & age > 12)

# A tibble: 2 x 2

age gender

<dbl> <chr>

1 14.0 F

2 16.0 F

And finally, we could have combined both a mutation and filtering in a single command using multiple pipes:

people %>%

filter(age < 20 & age > 12) %>%

mutate(gender=factor(gender, levels=c("U", "F", "M"),

labels=c("Unspecified", "Female", "Male")))

# A tibble: 2 x 2

age gender

<dbl> <fctr>

1 14.0 Female

2 16.0 Female

Note that the pipe operator (%>%) must go at the end of the line if you intend to carry the command over multiple lines!

There are additional dplyr functions that were covered in the lectures such as select() and arrange(). ***It would be a good idea to run all the code in the demo files supplied with lecture 9 and 10 material.*** In particular, answering the questions in lecture 10 will help you develop a deeper understanding of the dplyr functions. In particular, have you worked out the answers for the four homework questions from lecture 10? Answering these questions will prepare you well for the upcoming test. Once you have completed the tasks above, you are ready to go on to the mastery tasks for this week.

# Mastery Tasks

This lab has scratched the surface of data manipulation and type conversion. The best way to become comfortable with these techniques is practice, practice, practice! Therefore, the mastery task this week is to download the file BloodPressure.csv from Blackboard and answer two questions.

**Question 1:** For the first question, you will perform a number of steps to make it ready for analysis. You will perform the following:

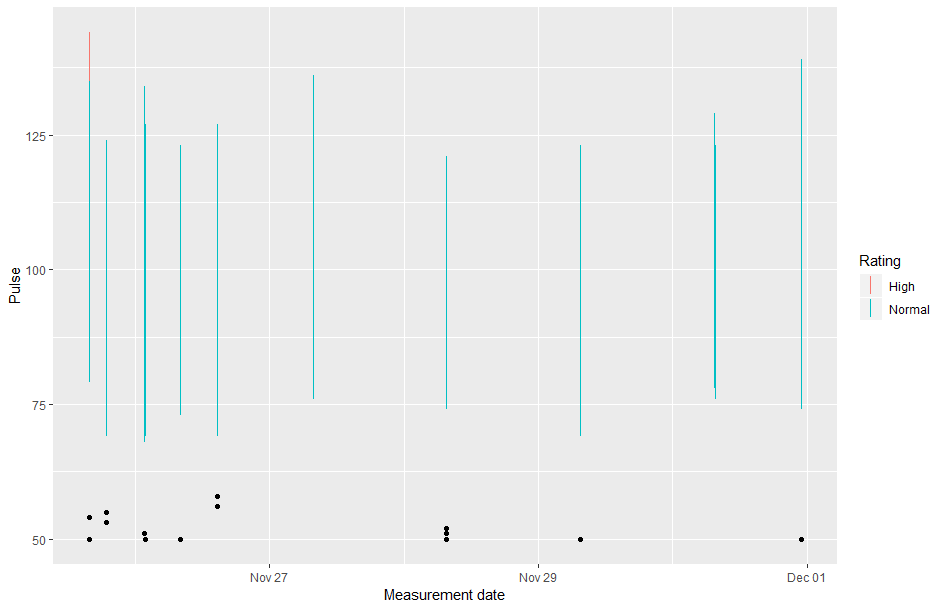
1. Open the file in Excel or a text editor (notepad or notepad++), and examine the data. The data given to you contains four columns: MeasurementDate, SYS, DIA and Pulse. These are the blood pressure readings of a patient recorded on different dates. SYS stands for Systolic pressure and DIA stands for Diastolic pressure. Now, examine the data for any peculiar values in the data. You may observe some English characters in a column which is meant to have only numbers. You will need to clean the data by removing them. You’ll find entries such as TH, TL and EST in a column. These stand for *too high*, *too low* and *estimated* values respectively. The details about how to go about cleaning those are given in the steps below (2-4 below).
2. Load the file into a tibble in R, paying attention to the type of data for each column. As you load the data replace TH and TL to NAs (i.e., every entry that has TH or TL should become a NA). Hint: Previously you have used na = “?” as an argument to the “read\_” functions to replace ? with NA. It is now a matter of using the same argument with a character vector containing the two mentioned elements.
3. an appropriate function (e.g. parse\_date\_time()) to fix this. Hint: the examples from this lab and/or the lectures should help here.
4. You will find that the Pulse column has not been imported as intended (it should be a *number*, but it is likely to be a character string). Replace “ EST” by an empty string “”. At the end of cleaning up, the Pulse column should be a number. Hint: The functions you require for string replacement and conversion to a numeric data type have already been covered either in the lectures or the labs.
5. Use the mutate() function to add three new columns:
   1. SYS.Class, which uses the cut() function to bin the SYS column into the following levels: 0-90 = “Low”, 90-120 = “Normal”, 120-140 = “Elevated”, 140-Inf = “High”
   2. DIA.Class, which uses the cut() function to bin the DIA column into the following levels: 0-60 = “Low”, 60-80 = “Normal”, 80-90 = “Elevated”, 90-Inf = “High”
   3. Rating, which uses the ifelse() function to return the character string “High” when either SYS.Class = “High” or DIA.Class = “High”, and the string “Normal” otherwise. Use the OR operator given by the symbol | to chain the two conditions.
6. Use the filter() function to restrict the data frame to rows with a MeasurementDate before the 1st of December 2017 (as\_date(MeasurementDate) < dmy("01-12-2017") will do this nicely). Store the result in a variable called filtered\_data.
7. Finally, use the result from the step above with ggplot to create the plot (shown below) using the code given below:

ggplot(data = filtered\_data, aes(x = MeasurementDate, y = Pulse)) +

geom\_linerange(aes(ymin = DIA, ymax = SYS, colour = Rating)) +

geom\_point() +

xlab("Measurement date")



Note that to get the vertical lines in the plot above we have used geom\_linerange(aes(ymin=DIA, ymax=SYS)) in the plot. The plot also uses geom\_point() to show the data points representing the pulse values.

Note there are no marks associated with the last step (step 7), but a penalty of 0.15 will apply if you do not include the code above as a part of your solution.

**Question 2:** Based on the data you have created at the end of step 6 for the previous question (i.e., filtered\_data dataset) answer the following two sub-questions. You must use dplyr functions (e.g. select, filter, mutate and arrange) wherever appropriate. Also, the code should use the pipe operator to chain different functions.

1. Write code to display all rows where the value in the SYS column is greater than or equal to 100 and the value for DIA column is greater than 75. The result must be sorted based on descending order of the Pulse column.
2. Write code to display values of just two columns (MeasurementDate and Rating) where the value of Rating is High. When the value of MeasurementDate is displayed, it should be in the date format (Hint: MeasurementDate column is a date-time column. Convert this *date-time* column to *date* using the *as\_date()* function).

Put all the required code into a single script provided (mastery-05.R), including the loading of appropriate libraries. Don’t forget to add your student id and the name in the file (see placeholders at the top of the file). As usual, place a short comment before each block of code that you use to complete a given task that explains what the code does. Not writing comments will attract a penalty of 5% (0.15 out of 3). When you have completed the tasks, **submit your work on Blackboard before 4pm on Thursday the 13th of August.**