I. The Fundamentals of R (R codes for presentation)

14 June, 2022

Working directory and paths

This presentation goes though the fundamentals of R. Before we can begin using R for any analysis, it is important to be familiarized with key concepts such as: directory paths, scrips, how to install, load and remove packages, the concept of functions, and importantly where to find help.
Firstly, lets have a look at how we locate and move between our directories (i.e. folders) from within R/Rstudio.

The term **path** refers to the trajectory you need to follow from the place you are "located" on your computer to the place you want to work from. You can use the command `getwd()` to see where you are currently working from and `setwd()` to move to your directory of choice. As you will see in the code chunk below a path can be either relative or absolute.

Show the path of the current working directory:

```
getwd()
```

```
## [1] "/Users/kgx936/Desktop/HeaDS/GitHub_repos/FromExceltoR/Presentations"
```

Set working directory:

```
# set working directory (absolute path)
setwd('/Users/kgx936/Desktop/HeaDS/GitHub_repos/FromExceltoR/Presentations')
setwd('~/Desktop/HeaDS/GitHub_repos/FromExceltoR/Presentations')

# where am I now?
getwd()
```

```
## [1] "/Users/kgx936/Desktop/HeaDS/GitHub_repos/FromExceltoR/Presentations"
```

```
# move one directory back
setwd('..')

# set working directory (relative path)
setwd('./Presentations')
```

```
# show folders in wd
list.dirs(path = '.', recursive = FALSE)
```

```
## [1] "./HTMLs" "./PDFs"
```

Functions & Arguments

A function is a chunk of code wrapped in a way which makes the code inside reusable. A function takes an input(s) (arguments) and returns an output(s). A concept example could be the one below

```
function(x,y) {
  z <- x + y
```

```
  return(z)
}
```

Lets have a look at some simple functions. We will use the `mean()` and `sum()` functions as examples:

```
# Get the mean of some numbers:
(5+1+11+6+4)/5
```

```
## [1] 5.4
```

```
# A vector of numbers named num1
num1 <- c(5,1,11,6,4)
num1
```

```
## [1]  5  1 11  6  4
```

```
# Mean: x (which is some numbers/values) is the input and mean is the output.
mean(x=num1)
```

```
## [1] 5.4
```

```
# Functions makes code reusable:
num2 <- c(0,3,4,9,1,2,7,10,2,11)
num2
```

```
##  [1]  0  3  4  9  1  2  7 10  2 11
```

```
mean(x=num2)
```

```
## [1] 4.9
```

R packages

R packages are collections of functions written by R developers and super users and they make our lives much easier.
An example of a highly used function is `mean()`, which takes a vector of numbers and returns the mean of these. This saves us time as we do not need to write the code to perform this calculation ourselves, we simply call this function, and we can do it as many times as we want on different objects.

Functions used in the same type of R analysis/pipeline are bundled and organized in packages - the package name often denotes the purpose of the functions within it. There is a help page for each package to tell us which functions it contains and which arguments go into these.

In order to use a package we need to download and install it on our computer. Most R packages are stored and maintained on the CRAN{https://cran.r-project.org/mirrors.html} repository. When we call the function `install.packages('')` we are automatically querying this repository for our package of choice, downloading and installing.

```
install.packages('tidyverse')
```

To use the package we do not only need to have it installed, we need to `load()` it into our R working environment.
N.B. you need to `load()` packages every time you start your R/Rstudio session (unless you saved a session). Packages are not automatically loaded into R as this would be very strenuous for our computers working memory, i.e. some people may have hundreds of packages installed on their computer, instead, we only load what we need for a specific R session.

```
library(tidyverse)
```

```
# Query package
?tidyverse

# Query function from package
?select()
?dplyr::filter()
?mean()
```

We can also unload a package from an R session:

```
detach('package:tidyverse')
```

If we would like to remove a package we can do so with `remove.packages('')`, *N.B* this will remove the package from your computer, not just from the R session.

```
remove.packages('tidyverse')
```

Data structures: `vector`, `tibble` and `data.frame`

We will need to work with a variety of data types and structures in R and often convert between these. In he examples below, we will look at three types of objects, a `vector`, a `tibble` and a 'data.frame. There are of course many more, but these three are the ones we will mainly be working with in this course.

```
# A vector of characters:
groupMembers <- c("Diana", "Tugce", "Henrike", "Thilde", "Alex", "Jennie", "Viki", "Yuhu", "Inigo", "Jo
groupMembers
```

```
##  [1] "Diana"    "Tugce"    "Henrike"  "Thilde"   "Alex"     "Jennie"
##  [7] "Viki"     "Yuhu"     "Inigo"    "Jonas"    "Conor"    "Marilena"
## [13] "Chloe"    "Anders"
```

```
# A vector of numeric values:
cakeDegrees <- c(44.67, 35.43, 30.13, 44.94, 45.0, 39.37, 32.79, 43.92, 44.61, 40.88, 32.28, 42.79, 39.
cakeDegrees
```

```
##  [1] 44.67 35.43 30.13 44.94 45.00 39.37 32.79 43.92 44.61 40.88 32.28 42.79
## [13] 39.17 50.41
```

Want to know what data type or structure you have, try the function class:

```
class(cakeDegrees)
```

```
## [1] "numeric"
```

```
class(groupMembers)
```

```
## [1] "character"
```

cakeDegrees is a vector of numerical numbers (double), but we would like to convert these to integers (whole numbers). Luckily, there are a range of functions implemented in base R (i.e. no package need to be loaded for these to work) which helps you convert from one data type to another. Common for these functions is that they begin with `as.` followed by the type you would like to convert to, and example would be `as.integer()` which we use below:

```
# Convert cakeDegrees to character values
cakeDegrees <- as.character(cakeDegrees)
cakeDegrees
```

```
##  [1] "44.67" "35.43" "30.13" "44.94" "45"    "39.37" "32.79" "43.92" "44.61"
## [10] "40.88" "32.28" "42.79" "39.17" "50.41"
```

```
# Convert cakeDegrees back to numeric values
cakeDegrees <- as.numeric(cakeDegrees)
cakeDegrees
```

```
##  [1] 44.67 35.43 30.13 44.94 45.00 39.37 32.79 43.92 44.61 40.88 32.28 42.79
## [13] 39.17 50.41
```

```
# Convert cakeDegrees to integer values (whole numbers)
cakeDegrees <- as.integer(cakeDegrees)
cakeDegrees
```

```
##  [1] 44 35 30 44 45 39 32 43 44 40 32 42 39 50
```

```
# Other examples of 'as.' functions for type conversion
# as.numeric()
# as.integer()
# as.character()
# as.factor()
# ...
```

Now, lets make a `tibble` and `data.frame`. **N.B** Here we are using the two vectors we created above to make our dataframe and tibble. We could also have typed out the values in the vector notation i.e. `cakeDegrees = c(44.67, 35.43,...)` directly in the tibble and data.frame function, however this would be double work, as we have already made two vectors and assigned them names:

```
# Make a dataframe
CakeDF <- data.frame(groupMembers = groupMembers,
           cakeDegrees = cakeDegrees)

# Check class
class(CakeDF)
```

```
## [1] "data.frame"
```

```
# Make a tibble
CakeT <- tibble(groupMembers = groupMembers,
                     cakeDegrees = cakeDegrees)

# Check class
class(CakeT)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

Just like we can convert between different data types with some variant of an `as.` function, we can also convert between data structures/objects:

```
# Convert tibble to dataframe:
as.data.frame(CakeT)
```

```
##    groupMembers cakeDegrees
## 1         Diana          44
## 2         Tugce          35
## 3       Henrike          30
## 4        Thilde          44
## 5          Alex          45
## 6        Jennie          39
## 7          Viki          32
```

```
## 8          Yuhu          43
## 9         Inigo          44
## 10        Jonas          40
## 11        Conor          32
## 12     Marilena          42
## 13        Chloe          39
## 14       Anders          50
```

```
# Convert dataframe to matrix:
as.matrix(CakeDF)
```

```
##       groupMembers cakeDegrees
##  [1,] "Diana"      "44"
##  [2,] "Tugce"      "35"
##  [3,] "Henrike"    "30"
##  [4,] "Thilde"     "44"
##  [5,] "Alex"       "45"
##  [6,] "Jennie"     "39"
##  [7,] "Viki"       "32"
##  [8,] "Yuhu"       "43"
##  [9,] "Inigo"      "44"
## [10,] "Jonas"      "40"
## [11,] "Conor"      "32"
## [12,] "Marilena"   "42"
## [13,] "Chloe"      "39"
## [14,] "Anders"     "50"
```

```
# Convert dataframe to tibble:
as_tibble(CakeDF)
```

```
## # A tibble: 14 x 2
##    groupMembers cakeDegrees
##    <chr>              <int>
##  1 Diana                 44
##  2 Tugce                 35
##  3 Henrike               30
##  4 Thilde                44
##  5 Alex                  45
##  6 Jennie                39
##  7 Viki                  32
##  8 Yuhu                  43
##  9 Inigo                 44
## 10 Jonas                 40
## 11 Conor                 32
## 12 Marilena              42
## 13 Chloe                 39
## 14 Anders                50
```

```
# Other examples of 'as.' function for structure/object conversion
# as.data.frame()
# as.matrix()
# as.list()
# as.table()
# ...
# as_tibble()
```

Fundamental operations & summary statistics

It is continuously necessary to inspect R objects so ensure that whatever operation was performed on the object was done correctly. If the object is small we can simply call it and it will be printed to the console. However, when we have a large object it impractical to have the whole thing printed out, instead we would just like to view the first part or last part of it - this can be done with `head()` and `tail()`.
If you would like to see your data in a tabular excel style format you can use `view()` which will open a new tap in Rstudio:

```
# Look at the "head" of an object, default is print first 6 lines:
head(CakeDF)
```

```
##   groupMembers cakeDegrees
## 1        Diana          44
## 2        Tugce          35
## 3      Henrike          30
## 4       Thilde          44
## 5         Alex          45
## 6       Jennie          39
```

```
head(CakeDF, n=8)
```

```
##   groupMembers cakeDegrees
## 1        Diana          44
## 2        Tugce          35
## 3      Henrike          30
## 4       Thilde          44
## 5         Alex          45
## 6       Jennie          39
## 7         Viki          32
## 8         Yuhu          43
```

```
# print (is almost always default, but there are times it is not)
print(CakeDF)
```

```
##    groupMembers cakeDegrees
## 1         Diana          44
## 2         Tugce          35
## 3       Henrike          30
## 4        Thilde          44
## 5          Alex          45
## 6        Jennie          39
## 7          Viki          32
## 8          Yuhu          43
## 9         Inigo          44
## 10        Jonas          40
## 11        Conor          32
## 12     Marilena          42
## 13        Chloe          39
## 14       Anders          50
```

```
# opens table as a table, excel style
# view(CakeDF)
```

Another useful function is `dim()`, short for dimensions, which returns the number of rows and columns of an

R object.

```
dim(CakeDF)
```

```
## [1] 14  2
```

Often we would like to pull out a single column from a dataframe or tibble to work with. This may be done with the '$' symbol:

```
# Extract variable with $ symbol
cakeDegrees <- CakeDF$cakeDegrees
groupMembers <- CakeDF$groupMembers
```

If we are only interested in a specific subset of either rows or columns from an R data object, we can extract these using the *slice* annotation. A slice is defined as square parentheses with the selection of rows/columns inside it. You of course need to define where to slice from, so you will first write the name of the object followed by the slice, i.e. `x[row, column]`.

```
# Specific rows/columns of an R object (a slice):
CakeDF[1:5, 1:2]
```

```
##   groupMembers cakeDegrees
## 1        Diana          44
## 2        Tugce          35
## 3      Henrike          30
## 4       Thilde          44
## 5         Alex          45
```

```
groupMembers[3:9]
```

```
## [1] "Henrike" "Thilde"  "Alex"    "Jennie"  "Viki"    "Yuhu"    "Inigo"
```

```
groupMembers[c(1:3, 5, 9, 11)]
```

```
## [1] "Diana"   "Tugce"   "Henrike" "Alex"    "Inigo"   "Conor"
```

Lastly, it is easy to get basic summary statistics on object in R. The R-base has a variety of simple stats functions build in (i.e. no package needs to be loaded), see some examples below:

```
# Simple summary statistics:
length(cakeDegrees)
```

```
## [1] 14
```

```
mean(cakeDegrees)
```

```
## [1] 39.92857
```

```
sd(cakeDegrees)
```

```
## [1] 5.82388
```

```
median(cakeDegrees)
```

```
## [1] 41
```

```
min(cakeDegrees)
```

```
## [1] 30
```

```
max(cakeDegrees)
```

```
## [1] 50
```

```
summary(CakeDF)
```

```
##  groupMembers        cakeDegrees
##  Length:14          Min.   :30.00
##  Class :character   1st Qu.:36.00
##  Mode  :character   Median :41.00
##                     Mean   :39.93
##                     3rd Qu.:44.00
##                     Max.   :50.00
```