

CSC-20053 Statistical Techniques for Data Analytics

Practical Worksheet – Week 2 (w/c March ¹⁴~~9~~, 202~~1~~)
(Session on Getting started with R/RStudio)

1. Introduction

R is a statistical programming language that has rapidly gained popularity in many scientific fields. Originally developed by *Ross Ihaka* and *Robert Gentleman* as an open source implementation of the “S” programming language, the language has a large online support community worldwide, with dedicated packages that provide extra functionality for many applications and different fields of study. Importantly, unlike several statistical software/programming platforms, R has no **GUI** (*Graphical User Interface*) and can be run entirely by typing commands into a text interface. This may seem a little daunting, but it also means a whole lot more flexibility, as you are not relying on a specific toolkit for your analyses.

This workshop aims to provide sufficient information for setting up your machine and troubleshooting the process. In short, you will need to install the following programs:

1. **R:** a programming language commonly used for working with data. This is the primary programming language used throughout this book. “Installing R” actually means installing tools that will let your computer understand and run R code.
2. **RStudio:** An graphical editor for writing and running R code. This will soon become our primary development application.
3. **Git:** A set of tools for tracking changes to computer code (especially when collaborating with others). This program is already installed on Macs.
4. **Bash:** A command-line interface for controlling your computer. git is a command-line program so you’ll need a command shell to use it. Macs already have a Bash program called Terminal. On Windows, installing git will also install a Bash shell called Git Bash, or you can try the Linux subsystem for Windows 10.

2. Setting up

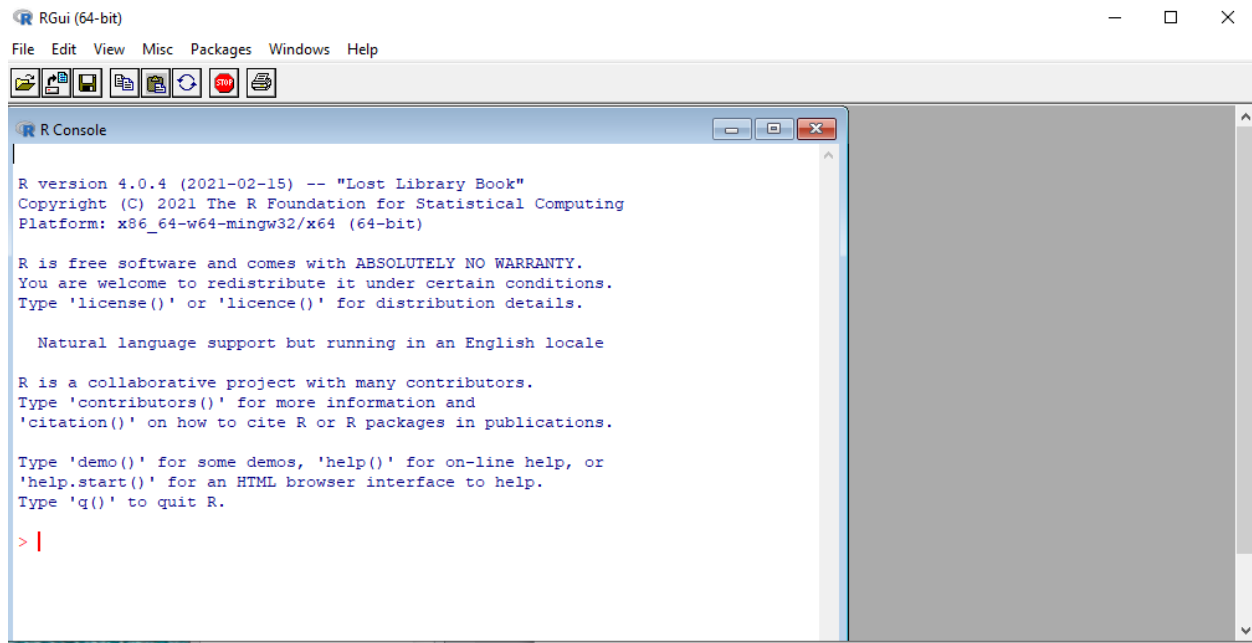
If you are using your own machine (**recommended**), you may need admin privileges to install the software described above. However, the lab machines in the School should have all appropriate software already installed and ready to use.

2.1 R Language

The primary programming language you will use in this module is called R. It is a very powerful statistical programming language that is built to work well with large and diverse datasets. In order to program with R, you will need to install the R Interpreter on your machine. This is a piece of software that is able to “read” code written in R and use that code to control your computer, thereby “programming” it.

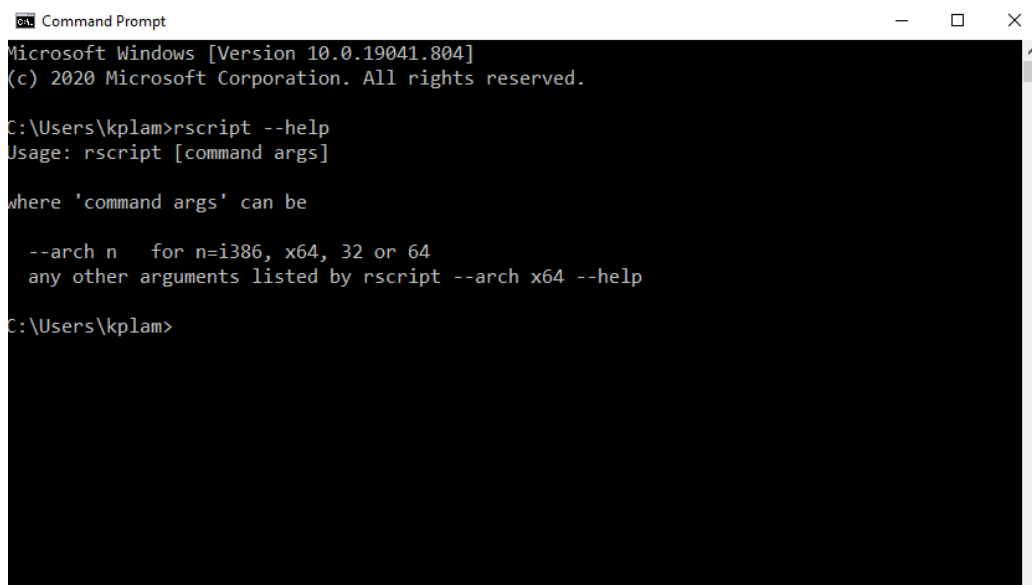
The easiest way to install R is to download it from the Comprehensive R Archive Network (CRAN) at <https://cran.rstudio.com/>. On Windows, follow the link to the base subdirectory (or follow the link to “install R for the first time”), then click the link to download the latest version of R for Windows. You will need to double-click on the .exe file to install the software. On a Mac, you are looking for the .pkg file—get the latest version supported by your computer.

On successful installation, you should see the following window/R Console opened on your screen:



Double click on either of these icons ( or ) on your desktop if this is not the case.

On Windows, you should also be able to configure your command shell (cmd) to run R or an R-script directly once the R Interpreter is installed, by adding to the (default) \$PATH of your environmental variable as: C:\Program Files\R\R-4.0.4\bin



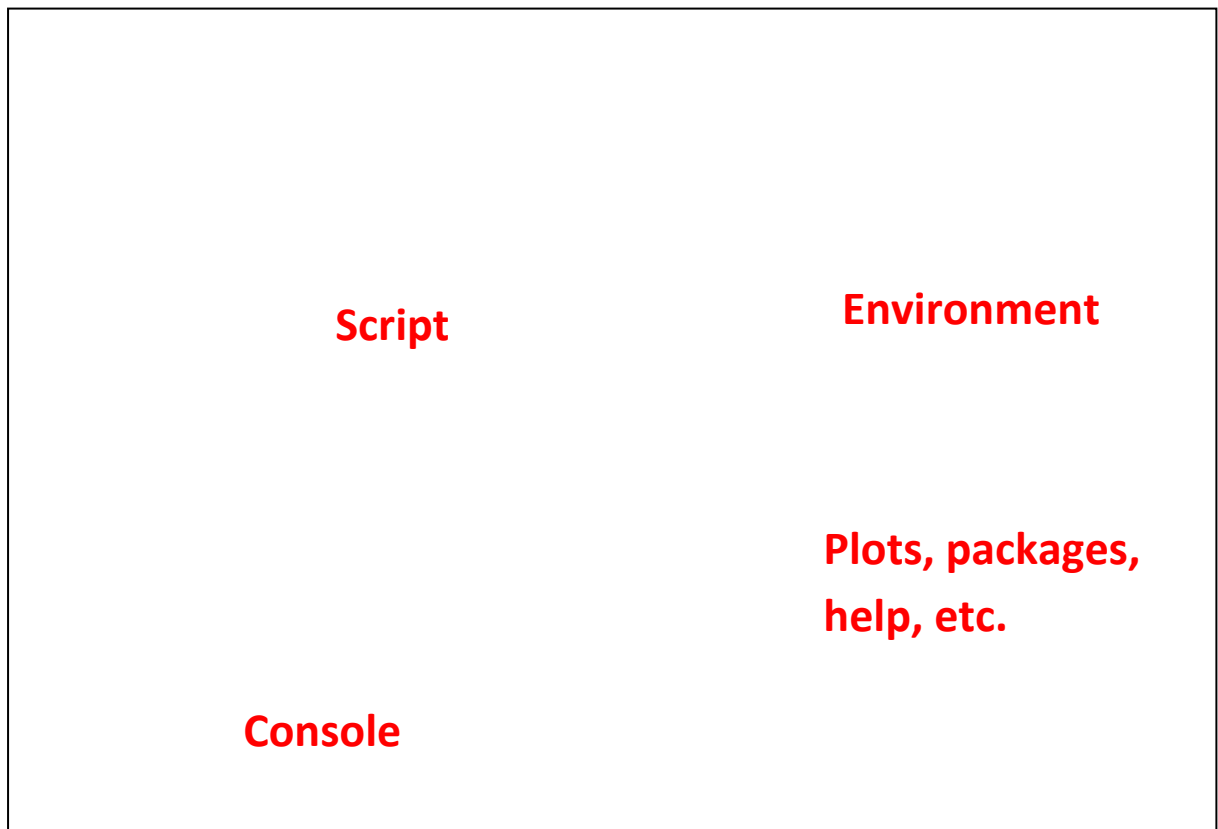
See also the “Terminal” Tab in R-Studio described below.

2.2 R-Studio

While you are able to execute R scripts without a dedicated application, the RStudio program is an open-source integrated development environment (IDE) that provides an informative user interface for interacting with the R interpreter. IDEs provide a platform for writing and executing code, including viewing the results of the code you have run.

To install the RStudio program, select the installer for your operating system from the [downloads](https://www.rstudio.com/products/rstudio/download/) page (at <https://www.rstudio.com/products/rstudio/download/>). (Make sure to download the free version.) Once the download completes, double-click on the .exe or .dmg file to run the installer. Simply follow the steps of the installer, and you should be prepared to use RStudio.

When you open the RStudio program (either by searching for it, or double-clicking on a desktop icon), you'll see the following interface:



An RStudio session usually involves 4 sections (“panes”), though you can customize this layout if you wish:

- **Script:** The top-left pane is a simple text editor for writing your R code. While it is not as robust as a text editing program like Atom, it will colourise code, “auto-complete” text, and allows you to easily execute your code. Note that this pane is hidden if there are no open scripts; select `File > New File > R Script` from the menu to create a new script file.

In order to execute (run) the code you write, you have two options:

1. You can execute a section of your script by selecting (highlighting) the desired code and

pressing the “Run” button (keyboard shortcut: ctrl and enter). If no lines are selected, this will run the line currently containing the cursor. This is the most common way to execute code in R.

2. You can execute an entire script by using the Source command to treat the current file as the “source” of code. Press the “Source” button (hover the mouse over it for keyboard shortcuts) to do so. If you check the “Source on save” option, your entire script will be executed every time you save the file (this may or may not be appropriate, depending on the complexity of your script and its output).
- **Console:** The bottom-left pane is a console for entering R commands. This is identical to an interactive session you’d run on the command-line, in which you can type and execute one line of code at a time. The console will also show the printed results from executing the code you execute from the Script pane. **NB.** just like with the command-line, you can use **the up arrow key** to easily access previously executed lines of code
 - **Environment:** The top-right pane displays information about the current R environment—specifically, information that you have stored inside of variables (see below). In the above example, the value 201 is stored in a variable called x. You’ll often create dozens of variables within a script, and the Environment pane helps you keep track of which values you have stored in what variables. This is incredibly useful for debugging!
 - **Plots, packages, help, etc.:** The bottom right pane contains multiple tabs for accessing various information about your program. When you create visualisations, those plots will render in this quadrant. You can also see what packages you’ve loaded or look up information about files. *Most importantly*, this is also where you can access the official documentation for the R language. If you ever have a question about how something in R works, this is a good place to start!

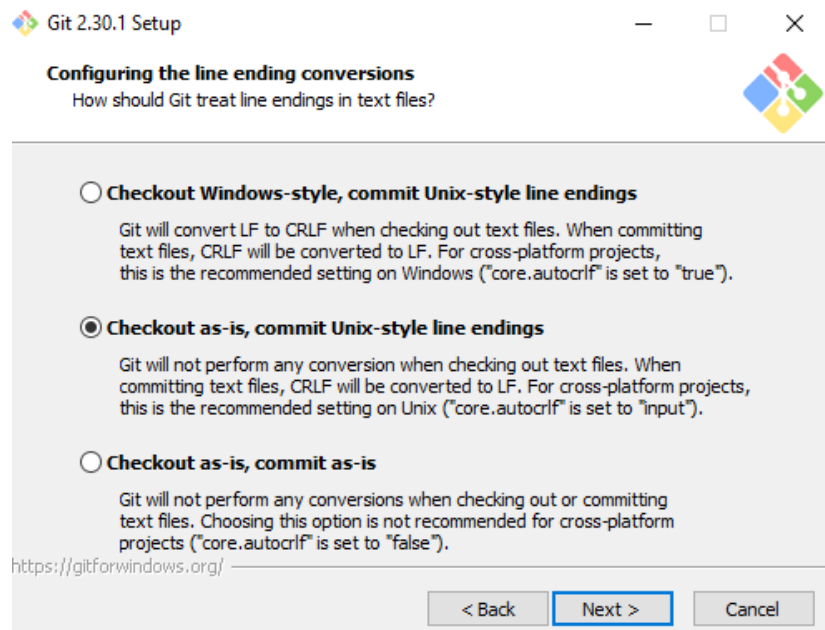
Note: you can use the small spaces between the quadrants to adjust the size of each area to your liking. You can also use menu options to reorganize the panes if you wish.

2.3 Git/Bash

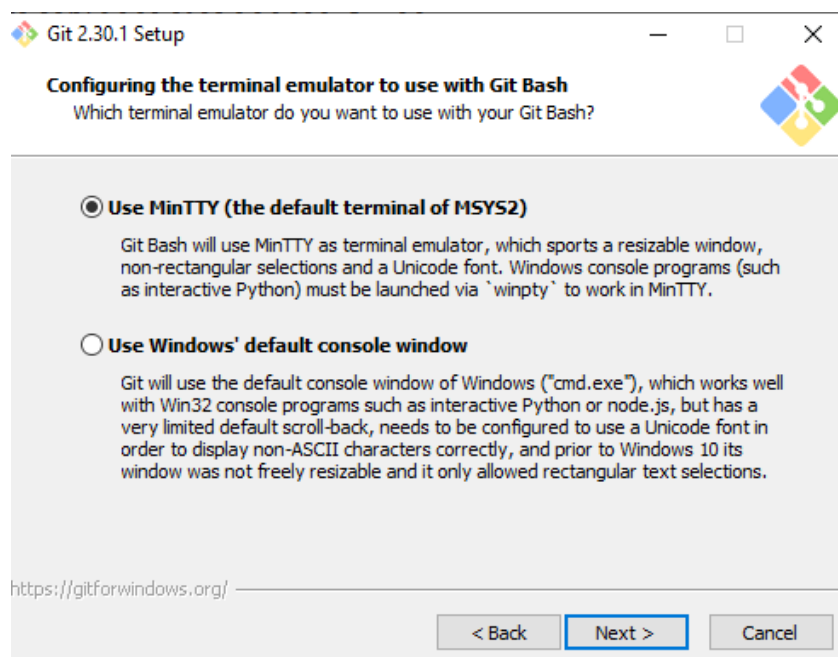
Git is a version control system that provides a set of commands that allow you to manage changes to written code, particularly when collaborating with other programmers. To start, you will need to download and install the software. If you are on a Mac, git should already be installed.



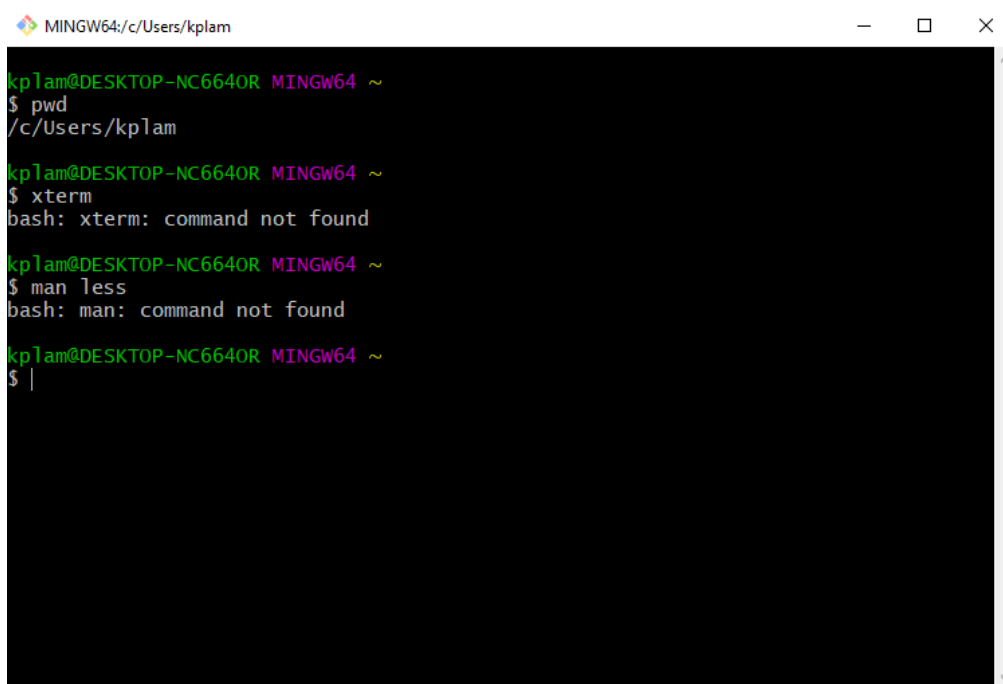
If you are using a Windows machine, this will also install a program called Git Bash, which provides a text-based interface for executing commands on your computer. Here, set up Git with the following two options:



and (for reason which will become clear later) with a bash emulator as follows:



All is well, open the `Git bash` program either by searching for it, or double-clicking on a desktop icon, you'll see the following interface:

A screenshot of a MINGW64 terminal window. The title bar at the top reads "MINGW64:/c/Users/kplam". The terminal shows a series of commands and their outputs: the user enters 'pwd' and receives '/c/Users/kplam'; they enter 'xterm' and receive 'bash: xterm: command not found'; they enter 'man less' and receive 'bash: man: command not found'; and finally, they enter a single character '|'.

```
MINGW64:/c/Users/kplam
kplam@DESKTOP-NC6640R MINGW64 ~
$ pwd
/c/Users/kplam
kplam@DESKTOP-NC6640R MINGW64 ~
$ xterm
bash: xterm: command not found
kplam@DESKTOP-NC6640R MINGW64 ~
$ man less
bash: man: command not found
kplam@DESKTOP-NC6640R MINGW64 ~
$ |
```

3. Example

R is an extraordinarily powerful open-source software program built for working with data. It is one of the most popular data science tools due to its ability to efficiently perform statistical analysis, implement machine learning algorithms, and create data visualisations. R is the primary programming language used throughout this module, and understanding its foundational operations is key to being able to perform more complex tasks.

To test your installations, open RStudio to begin using R. As good practice, start by noting the purposes of your code and the main goal. Here's an example, which you can copy, paste and edit into your new script:

```
# CSC-20053 Workshop 1 - R Demo
# Learning R with RStudio on how to import public datasets and explore them
# March 8, 2021
```

The next few lines of code *usually* load the packages you will need for your analysis¹. For example, you might load a package for formatting data, or for making maps. For now, let's postpone this until later.

The next lines of code should define your *working directory*. This is a folder on your computer where R will look for data, save your plots, etc. To make your workflow easier, it is good practice to save everything related to one project in the same place, as it will save you a lot of time typing up computer paths or hunting for files that got saved R-knows-where. For instance, you could save your script and all the data for this workshop in a folder called "DRAFT". Note that It is good practice to avoid spaces in file names as it can sometimes confuse R. For bigger projects, however, consider having a root folder with the name of the project.

¹ A package contains R commands which can be loaded into R to provide extra functionality.

```
getwd()
setwd("C:/Users/kplam/Documents/RS")
getwd()
```

Practice is the best way to learn any new language, so let's jump straight in and do some of our own statistical analysis using a publicly available dataset which I have obtained (from the [NBA Gateway](#)) and saved and attached it for this workshop as `NBAbio.csv`. In RStudio, you can either click on the `Import dataset`² button and navigate to where you have saved your file, or use the `read.csv()` command as we will be doing in this workshop. See the alternative means in footnote (2) below.

```
nbabio <- read.csv("NBAbio.csv")
```

In passing, it should be noted that R works best with `.csv` (comma separated values) files. If you entered your data in Excel, you would need to click on `Save as` and select `csv` as the file extension. When entering data in Excel, **DO NOT** put any spaces in your row names, as they will confuse R later. Some computers save `.csv` files with semicolons `;`, not commas `,` as the separators. If this is the case, use **`read.csv2` instead of `read.csv`**, or alternatively use the argument `"sep"` (for separator) in the `read.csv` function, as shown here: `r.csv("your-file-path", sep = ";")`. This done, remember to **save your script once in a while**. If you have not saved it already, save it now in the same directory as the rest of the practical/workshop file, and give it a meaningful name. ← Checkpoint 1.

A really important step is to check that your data was imported without any mistakes. It's good practice to always run this code and check the output in the console - do you see any missing values, do the numbers/names make sense? If you go straight into analysis, you risk later finding out that R didn't read your data correctly and having to re-do it, or worse, analysing wrong data without noticing. To preview more than just the few first lines, you can also click on the object in your Environment panel, and it will show up as a spreadsheet in a new tab next to your open script. Large files may not display entirely, so keep in mind you could be missing rows or columns.

```
head(nbabio)           # Displays the first few rows
tail(nbabio)           # Displays the last rows
str(nbabio)
```

Here, you will notice the `taxonGroup` variable shows as a `character` variable, but it should be a factor (*categorical* variable), so we'll force it to be one. When you want to access just one column of a *data frame*, you append the variable name to the object name with a dollar `$` sign. This syntax let you see, modify, and/or reassign this variable. Next, enter the following R commands:

```
head(nbabio$taxonGroup) # Displays the first few rows of this column only
class(nbabio$taxonGroup) # Tells you what type of variable we're dealing with: it is
character now but we want it to be a factor
nbabio$taxonGroup <- as.factor(nbabio$taxonGroup)
```

Next, copy and paste the followings into your script:

² If you use `Import`, a window will pop up previewing your data. Make sure that next to `Heading` you have selected `Yes` (this tells R to treat the first row of your data as the column names) and click `Import`. In the console, you will see the code for your import, which includes the file path.


```
dim(nbabio)           # Displays number of rows and columns
summary(nbabio)       # Gives you a summary of the data
summary(nbabio$taxonGroup) # Gives you a summary of that particular
                        #variable (column) in your dataset
```

In that last line of code, the `as.factor()` function turns whatever values you put inside into a factor (here, we specified we wanted to transform the character values in the `taxonGroup` column from the `nbabio` object). However, if you were to run just the bit of code on the right side of the arrow, it would work that one time, but would not modify the data stored in the object. By assigning with the arrow the output of the function to the variable, the original `nbabio$taxonGroup` in fact gets overwritten: the transformation is stored in the object. Try again to run `class(nbabio$taxonGroup)` - what do you notice?

Next, enter the following line and observe the (error message!) output:

```
Beetle <- filter(nbabio, taxonGroup == "Beetle")
```

To fix the error, install the 'dplyr' package and use it as follows.

```
install.packages("dplyr")
library(dplyr)
```

Briefly, the first line install the named package, which you only need to install packages once, so in this case you can type directly in the console box, rather than saving the line in your script and re-installing the package every time. Once installed, you just need to load the packages using `library(package-name)`. For this workshop, we will be using the `dplyr` package to provide extra commands for formatting and manipulating data. As the module progresses, you will learn more about the powerful features of `dplyr` in later work. ← Checkpoint 2

In passing, the `nbabio` object has occurrence records of various species collected by NBA Gateway from 2000 to 2016. To explore this dataset, we will create a graph showing how many species were recorded in each *taxonomic* group. You could calculate species abundance in *Excel* spreadsheet (a 'formidable' task!), but that has several disadvantages, especially when working with large datasets like this; e.g. you have no record of what you clicked on, how you sorted the data and what you copied and/or deleted - mistakes can slip by without you noticing. In R, on the other hand, you have your script, so you can go back and check all the steps in your analysis.

This done, we first need to split `nbabio` into multiple objects, each containing rows for only one taxonomic group. We do this with the useful `filter()` function from the `dplyr` package, as follows:

```
Beetle <- filter(nbabio, taxonGroup == "Beetle")
Bird <- filter(nbabio, taxonGroup == "Bird")
Butterfly <- filter(nbabio, taxonGroup == "Butterfly")
Dragonfly <- filter(nbabio, taxonGroup == "Dragonfly")
Flowering.Plants <- filter(nbabio, taxonGroup == "Flowering.Plants")
Fungus <- filter(nbabio, taxonGroup == "Fungus")
Hymenopteran <- filter(nbabio, taxonGroup == "Hymenopteran")
Lichen <- filter(nbabio, taxonGroup == "Lichen")
Liverwort <- filter(nbabio, taxonGroup == "Liverwort")
Mammal <- filter(nbabio, taxonGroup == "Mammal")
Mollusc <- filter(nbabio, taxonGroup == "Mollusc")
```


Once you have created objects for each taxon, we can calculate species abundance, *i.e.* the number of different species in each group. Using the R commands below, we will nest two functions together: `unique()`, which identifies different species, and `length()`, which counts them. You can try them separately in the console and see what they return. ← Checkpoint 3

```
a <- length(unique(Beetle$taxonName))
b <- length(unique(Bird$taxonName))
c <- length(unique(Butterfly$taxonName))
d <- length(unique(Dragonfly$taxonName))
e <- length(unique(Flowering.Plants$taxonName))
f <- length(unique(Fungus$taxonName))
g <- length(unique(Hymenopteran$taxonName))
h <- length(unique(Lichen$taxonName))
i <- length(unique(Liverwort$taxonName))
j <- length(unique(Mammal$taxonName))
k <- length(unique(Mollusc$taxonName))
```

It should be noted that the above code is quite repetitive and using a lot of copying and pasting! That's not particularly efficient - in future, however, you will learn how to use more of `dplyr`'s functions and achieve the same result with way less code! You will be able to do everything you just did in ONE single line - promise(!) ← Checkpoint 4

Now that we have species abundance for each taxon, we can combine all those values in a vector. A vector is another type of R object that stores values. As opposed to a data frame, which has two dimensions (rows and columns), a vector only has one. When you call a column of a data frame like we did earlier with `nbabio$taxonGroup`, you are essentially producing a vector - but you can also create them from scratch. We do this using the `c()` function, where `c` stands for *concatenate*, or *chain* (if that makes it easier to remember). We can also add labels with the `names()` function, so that the values are not coming out of the blue, as follows:

```
heterog <- c(a,b,c,d,e,f,g,h,i,j,k) # NB. Create new heterog (= herterogenity) object
names(heterog) <- c("Beetle",
                    "Bird",
                    "Butterfly",
                    "Dragonfly",
                    "Flowering_Plants",
                    "Fungus",
                    "Hymenopteran",
                    "Lichen",
                    "Liverwort",
                    "Mammal",
                    "Mollusc")
```

The followings should be noted:

- The spaces in front of and behind `<-` and after, are added to make it easier to read the code.
- All the labels have been indented on a new line - otherwise the line of code gets very long and hard to read.
- Take care to check that you are matching your vector values and labels correctly, as you would not want to label the number of beetles as lichen species(!) As mentioned earlier, the good thing about keeping a script is that we can go back and check that we have indeed assigned the number of beetle species to `a`. Even better practice would have been to give more meaningful names to our objects, such as `beetle_species`, etc.

Finally, we can now visualise species' abundance with the `barplot()` function. Plots appear in the bottom right window in RStudio.

```
barplot(heterog)
barplot(heterog,xlab="Taxonomy", ylab="#Species")
taxon <- c("Beetle",
          "Bird",
          "Butterfly",
          "Dragonfly",
          "Flowering.Plants",
          "Fungus",
          "Hymenopteran",
          "Lichen",
          "Liverwort",
          "Mammal",
          "Mollusc")
```

There are a few things not quite right that we should fix - there are no axis titles, not all column labels are visible, and the value for plant species ($n = 521$) exceeds the highest value on the y axis, so we need to extend it. The great thing about R is that you do NOT need to come up with all the code on your own - you can use the `help()` function and see what arguments you need to add in. This is left as an exercise.

Before we end this practical session, it is instructive to summarise what we have achieved so far. So far we have created *vectors*, i.e. a series of values, each with a label. This object type is suitable when dealing with just one set of values. Often, however, you will have more than one variable and have multiple data types - e.g. some continuous, some categorical. In those cases, we use data frame objects, which are *tables* of values, each of which has a two-dimensional structure with rows and columns, with each column can have a different data type. With this in mind, we will use the `data.frame()` function, but first we will create an object that contains the names of all the taxa (one column) and another object with all the values for the species richness of each taxon (another column).

```
# Creating an object called "taxon" that contains all the taxon names
taxon <- c("Beetle",
          "Bird",
          "Butterfly",
          "Dragonfly",
          "Flowering.Plants",
          "Fungus",
          "Hymenopteran",
          "Lichen",
          "Liverwort",
          "Mammal",
          "Mollusc")

# Turning this object into a factor, i.e. a categorical variable
taxon_fr <- factor(taxon)
# Combining all the values for the number of species in an object called abundance
abundance <- c(a,b,c,d,e,f,g,h,i,j,k)
#
# Creating the data frame from the two vectors
BioDiversity2016 <- data.frame(taxon_fr, abundance)
```

The last line of code create a data frame with our species abundance data, and then save it using `write.csv()`. ← **Check point 5 (last!)**

As before, you can view this new object with `barplot`, noting that this is one of the final goals (visualisation) of this practical workshop.

```
barplot(BioDiversity2016$abundance, names.arg=c("Beetle",
                                                "Bird",
                                                "Butterfly",
                                                "Dragonfly",
                                                "Flowering.Plants",
                                                "Fungus",
                                                "Hymenopteran",
                                                "Lichen",
                                                "Liverwort",
                                                "Mammal",
                                                "Mollusc"),
        xlab="Taxa", ylab="Number of species", ylim=c(0,625))
```

3.1 An exercise

Well done! If you are completely new to R, do **NOT** worry if you don't grasp quite everything just yet. Go over the individual parts of this workshop where you found difficult to ensure that you get up to speed with certain concepts.

As a practice, you are given a new `.csv` (*Exercise_101*) which contain (fictional) values of some quantities (with units) measured on four different species of sea animals. Can you produce a bar plot of the mean value for each species and save it to your computer? What could the function for calculating the mean be? Can you also guess **why** the chosen species **what** these quantities represent? **Hints:** Think simple.