Introduction to R and RStudio

Annique Claringbould

Contents

Goals . . . 1
References . . . 1
Downloading and installing R and RStudio . . . 2
Start using R . . . 2
Install and load R packages . . . 3
Finding help . . . 3

Basic concepts **4**
Variables and operators . . . 4
Data types . . . 5
Predefined functions . . . 6
Using scripts . . . 6
Data structures . . . 7

Opening and saving files **12**
Setting working directory . . . 12
Importing text files . . . 12
Importing files from other formats . . . 13
Saving files . . . 13
Session info . . . 14

Goals

With over two million users worldwide, R is becoming the leading software package for statistics and data science. It is freely available and has many utilities and possibilities for basic and advanced statistical analysis, creating graphs, data handling and writing software. As such, R is a very convenient software package that allows you to create reproducible scripts for all of the steps from experimental data to final analyses for your paper, which cannot be done with statistical packages like SPSS, SAS, or Stata.

However, for many researchers it is not directly clear how to use R, because of the R language and the way of reasoning. We would therefore like to familiarize you with R.

This vignette will teach you the basics of R through computer exercises. The following topics will be covered: the R language, R variables (objects), reading and writing data files, loading packages, and some basic manipulations.

Have fun!

References

This vignette is a modified version of this vignette.

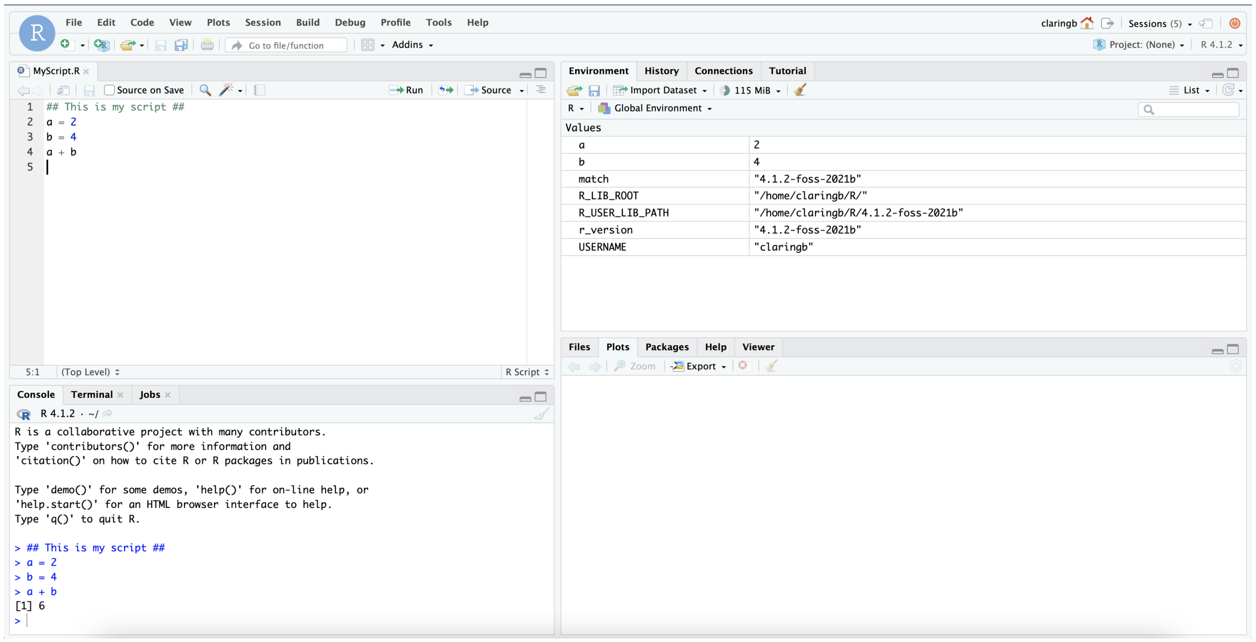
Downloading and installing R and RStudio

`R` and `RStudio` are already installed on the computers in the rooms that are used for this course. However if you want to use R for your research or private work, do the following:

1. Go to http://cran.r-project.org/ to download R. You will be asked to click on the operating system that you are using. If you select, 'Windows' you will be confronted with three choices; 'base', 'contrib', and 'Rtools'. You only need the 'base'. Later from within R you can install packages, which allow you to enjoy all the functionalities of R. Once you click on 'base' you will be confronted with a page showing (in larger font) a link to 'Download R x.x.x for Windows' where the series of x indicate the current version (e.g. 4.1.2). When you click on the download link, you will be prompted to save the file somewhere on your computer. Save it to your preferred location. Note that this is only the installation file; there is no need to keep it after you have installed R.
2. Install R by double clicking on the installation file. The default options/settings as specified during installation will be fine. The program can very easily be customized after installation.
3. Now you could open R by double clicking on the desktop icon or through the menu. You will then be confronted with a window in a window. This is the graphical user interface or console window, which is the core of the program and acts as both input and output display. However, it is easier to learn to work with R using RStudio, as it is more user-friendly than just R.
4. To download R-Studio go to http://www.rstudio.com/products/rstudio/download/. Select from the Installers the operating system that you are using. You can either directly select 'Run' or select 'Save' when you want to save the installation file. Follow the instructions on the screen for installation of R-Studio.

Start using R

Today you will be made familiar with the program RStudio and with the very basics of the R language.



```
## This is my script ##
a = 2
b = 4
a + b
```

```
Values
a                  2
b                  4
match              "4.1.2-foss-2021b"
R_LIB_ROOT         "/home/claringb/R/"
R_USER_LIB_PATH    "/home/claringb/R/4.1.2-foss-2021b"
r_version          "4.1.2-foss-2021b"
USERNAME           "claringb"
```

```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> ## This is my script ##
> a = 2
> b = 4
> a + b
[1] 6
>
```

The pane on the left bottom is the **Console**. Here you can type in commands and let R do things for you. With the arrows ↑ and ↓ you can scroll back and forth to previous commands that were executed. On the top right, in the pane with tabs **Environment** and **History** you have an overview of all the variables, objects, datasets, and functions that are present in the working space (tab **Environment**) or you can recall the latest commands (tab **History**). A third pane on the bottom right is the output window and shows the tabs **Files**, **Plots**, **Packages**, **Help**, and **Viewer**. The tab **Files** is a file browser; the tab **Plots** will display all graphs that you will make during a session; the tab **Packages** shows all packages that have been opened; the tab **Help** provides information on the packages, functions, and datasets; the tab **Viewer** allows you to preview local web content. For example, I can see this document in **Viewer**. A fourth pane appears on the top left if you view a dataset or if you use a script.

If you have already worked in RStudio, it will have saved your last project (working space). Upon reopening RStudio, it will automatically open the last working space, scripts and history of commands from your last session. If you don't like this, you can change this in 'Tools' > 'Global options' from the menu.

There are some key terms you will need to become familiar with. First, R is an object oriented system. Anything can be an object; a single value (for example 9), a series of values (called a vector), a named variable (also called a vector), a matrix (made up of rows and columns), an array (more than two dimensions), a data frame, or a list (a larger group of objects). For now, it is not important to know each of these data types exactly, but it is good to be aware that data can be stored in different formats.

R is an open source program. This means that everyone can add new functions. The addition of new functions is possible through installation and loading of packages. A package provides a mechanism for loading optional code, data and documentation as needed. The R installation already contains ~30 packages by default. We will learn how to install and load an external package in this module.

Install and load R packages

The most common way to install packages is by typing the following into the console: `install.packages("package name")` Keep in mind that the quotes around the package name are required, and the package name is case sensitive.

Once you have installed a package, you need to load it before you can use it: `library("package name")`

You can only make use of the package and its functions/variables after installation and loading.

For instance, try installing R package `praise` and running it the following way:

```r
install.packages("praise")
library(praise)
praise()
```

Finding help

When you are new to learning R, you will (of course!) not yet know how to do things. You can search for help in multiple ways. If you already know the topic you would like to know more about, you can search for help within R.

For instance if you want to get some help on creating a matrix (one of the data types in R) you can type the following commands in the console: `help(matrix)`, `help("matrix")` or `?matrix`. The help page for the function matrix will then appear in the **Help** tab of the output pane. This help page provides information on how to use the matrix function, which arguments it needs and on the bottom of the page it also shows some examples.

In practice, however, one of the most useful ways of learning R is by Googling what you want to achieve and looking for others that have posted similar questions on websites like https://stackexchange.com/, https://stackoverflow.com/, or R blogs. Even if you use R every day, you will still end up searching for particular commands or packages very regularly.

Some useful website about R: Quick-R: http://www.statmethods.net/ CRAN introduction to R: https://cran.r-project.org/doc/manuals/R-intro.pdf CRAN R reference card: https://cran.r-project.org/doc/contrib/Short-refcard.pdf

Basic concepts

Variables and operators

First create a new object called 'x'. Our object `x` will initially be something as simple as an individual value, say 5. To communicate this in R, type one of the following commands in the console and hit the enter key:

```
x<-5
x=5
```

Notice how both = and <- refer to the same operation. The operation `=` and `<-` perform "is assignment"; in other words, we have assigned 5 to x. These two commands are the most used ones to assign a value to a variable.

Commands can contain spaces, R will internally delete them:

```
x <- 5
x = 5
```

It is up to you to decide upon your own style, i.e. whether you prefer to write spaces or not.

To see what x is, type in the console:

```
print(x)
```

Alternatively, you can look in the right upper pane **Environment** under **Values**.

Commands can be combined in one line using the separator ';'. E.g. the command `x <- 5; print(x)` will assign the value 5 to x (first command) and print it (second command).

If you want to see the outcome of the assignment immediately, you can put the command in between brackets: `(x <- 5)`

Arithmetic operators

Now we can perform simple arithmetic or complex algebra using our assigned value for x. For example:

```
x+3 #addition
x-3 #subtraction
x/6 #division
x*6 #multiplication
x^2 #exponentiation
x**2 #exponentiation: x**2 == x^2
x^1/2
x^(1/2)
```

You can assign the outcome to a new object by e.g.:

```
y <- x+3
```

and with `y` or `print(y)` tell R to show you what the value of the new variable is. Or you can do the assignment and let R tell you the answer in one command by typing `(y <- x+3)`.

Logical operators

In addition to arithmetic operators you can also apply logical operators to the objects:

```
x < 5 #smaller than
x <= 5 #smaller than or equal to
x > 5 #larger than
x >= 5 #larger than or equal to
x == 5 #equal to
x != 5 #not equal to
! (x==5) #not
x>3 & x<10 #and
x==5 | x==6 #or
```

Like before it is also possible to assign the outcome to a new variable. This variable would then be of the logical format and can only take values of `TRUE` or `FALSE`. For example:

```
y <- (x==5)
y
```

You can also directly assign a value of `TRUE` or `FALSE` to a variable:

```
y <- FALSE
#or
y <- F
```

Instead of `TRUE` or `FALSE`, R also recognises the first letters `T` and `F`, respectively.

Data types

We have already discussed a couple of data types that R can handle: numbers and logical values (`TRUE` and `FALSE`).

The following data types are recognised by R:

Data.type	Description	Examples
Logical	A Boolean variable	`TRUE`, `FALSE`
Numeric	Any real number	`1.07518361`, `9`
Integer	A whole number (indicated by `L`)	`156L`, `-2L`, `1L`
Complex	A complex number with real and imaginary parts	`i+2`
Character	A string, used to save words or phrases	`'B'`, `'Hello world!'`

You can store strings in a variable, just like numbers, but you need to use quotation marks:

```
p <- "Hello"
q <- "Maria"
```

In general, R is very good at recognising the data type, and it will adjust to what it thinks is most appropriate. If you want to check the data type of a variable, you can use the command `class()`. For example:

```
class(1)
class("A")
x <- paste(c(1,2),"A")
class(x)
```

You can convert a variable into a numerical data type using `as.numeric(x)` and into a character using `as.character(x)`.

Predefined functions

There are many predefined functions in R base library that will be useful when programming in R. Here are some examples of numerical functions with their description written as a comment, preceeded by `#`. These functions will only work if `x` is numeric. In documentation about R and in this tutorial, you can recognise that someone is discussing a predefined function when there is a combination of letters followed by `()`. The brackets indicate that you should add something in between the brackets to run the function. For example:

```
abs(x) #absolute value of x
sqrt(x) #square root of x
round(x, digits=n) #round x to n decimal points, try with round(2.1246, digits=2)
log(x) #natural logarithm of x
log10(x) #common logarithm of x
exp(x) #e^x
```

Like before, the outcomes of these numerical functions can directly be assigned to a variable:

```
y <- sqrt(x)
```

Of course, other functions work with non-numerical data types, like characters. For example, you can combine two words using the `paste()` function:

```
paste("Hello", "Maria", sep=" ")
```

```
## [1] "Hello Maria"
```

Using scripts

So far we have typed commands in the console window. In practice it is most useful to use scripts, which are text files (usually with the extension .R) in which the commands are stored. These text files can be saved and hence the use of scripts makes it easier to repeat an analysis that you did before or to adapt it. Also for quality control issues, it is advised to use scripts since you can quickly see what analyses you have done. Finally, use of scripts will make life easier for successors or other colleagues, if you move jobs.

You can open a new script in RStudio using the menu. Click on 'File' > 'New File' > 'R Script'. A new pane will then be opened in the upper left side of the window.

In this window you can type in commands like you did in the console window.

A convenient tool in scripting is to add comments to the script to explain the command(s). This can be done by putting a `#` in front of the comment. You will see that R recognises the comment, because the formatting is different from regular code. Comments do not have to be at the beginning of a line, they can also be put after the command. For example:

```
#assigning value to x
x <- 225

y <- sqrt(x) #calculating the square root of x
```

It is good practice to add comments. While you are working on a script, it is usually clear to you what each command means. However when you have to repeat or adapt the analyses after some time because you need to revise it for your paper, or when a colleague wants to use the same script, it will often turn out to be rather difficult to figure out what you have done. Therefore we recommend to make a habit out of adding comments while you create a script. It may not be necessary to comment on each line.

Once you have created a script, you can rerun your analysis and import variables, data, and functions from this script into your current workspace.

Exercises

Exercise 1:

- Create a new object called 'x' and assign it a value of 13.4
- Divide x by 2 and assign the result to a variable called 'y'
- What is the value of 'y'?

Exercise 2:

- Perform log10-transformation on 'x' and add y
- Assign the result to a variable called 'z'
- What is the value of 'z'?

Exercise 3:

- Write a command that gives as output 'The value of z is . . . '
- In this command, round the value of z to three decimal digits *Hint*: Use the functions `paste()` and 'round()"

Data structures

There are five basic data structures that are available in R:

Data.structure	Description
Vector	A collection of values that all have the same data type. The elements of a vector are all numbers, giving a numeric vector, or all character values, giving a character vector. A vector can be used to represent a single variable in a data set.
Factor	A collection of values that all come from a fixed set of possible values. A factor is similar to a vector, except that the values within a factor are limited to a fixed set of possible values. A factor can be used to represent a categorical variable in a data set, e.g., sex, case-control, etc.
Matrix	A two-dimensional collection of values that all have the same type. The values are arranged in rows and columns.
Data frame	A collection of vectors that all have the same length. This is like a matrix, except that each column can contain a different data type. A data frame can be used to represent an entire data set.
List	A collection of data structures. The components of a list can be simply vectors–similar to a data frame, but with each column allowed to have a different length. However, a list can also be a much more complicated structure. This is a very flexible data structure. Lists can be used to store any combination of data values together.

Vectors

As mentioned earlier, R can handle a lot of data structures. So far we used objects that contained a single value. You can create vectors, which are collections of values:

```
age <- c(18, 21, 22, 43, 23, 54, 60, 20, 33, 72)
```

The `c()` function is used to concatenate values and yields a one-dimensional object. Type `print(age)` to see the result of this command. With the function `length()`, you can find out the length of the elements in a vector.

```
length(age)
```

```
## [1] 10
```

Elements can be extracted from a vector by using the index between square brackets:`age[1]` is the first element of the vector and is equal to 18.

You can also change an item in a vector by referring to the index number:

```
age[2] <- 30
age
```

```
##  [1] 18 30 22 43 23 54 60 20 33 72
```

You can get part of the vector by using the `:` operator:

```
age[1:3]
```

```
## [1] 18 30 22
```

There are many handy functions that can help you create or manipulate vectors:

```
c(3:6) #Creates a range of numbers: (3,4,5,6)
c(10:1) #Creates a range of numbers in reversed order
seq(0,10,0.5) #Generates a range of numbers between 1 and 10, with steps of 0.5
rep(5,10) #Repetition of 10 times the number 5
sort(c(3,6,7,1)) #Sorts a vector
```

You can also make calculations directly on all items in a vector:

```
x <- c(2,5,3,6,9)
x*2
```

```
## [1]  4 10  6 12 18
```

```
log(x, base=2)
```

```
## [1] 1.000000 2.321928 1.584963 2.584963 3.169925
```

You can use the logical operators within a vector to obtain the elements that meet a specific condition:

```
age[age>30]
```

```
## [1] 43 54 60 33 72
```

```
age[age>20 & age<50]
```

```
## [1] 30 22 43 23 33
```

You can add two vectors, as long as they are of the same length:

```
x <- c(2,5,3,6,9)
y <- c(10,8,4,5,6)
x+y
```

```
## [1] 12 13  7 11 15
```

Two vectors can be joined by the concatenate command:

```
c(x,y)
```

```
##  [1]  2  5  3  6  9 10  8  4  5  6
```

Descriptive statistics

The base library of R includes many functions to calculate descriptive statistics:

```
mean(x, na.rm=TRUE) #mean of object x, na.rm indicates if you want to exclude missing values
sd(x) #standard deviation of object x
median(x) #median of object x
quantile(x, probs) #quantiles of numeric vector x. probs is the numeric vector of the percentiles
min(x) #minimum of object x
max(x) #maximum of object x
range(x) #range of object x, equivalent to min(x) and max(x)
scale(x, center=TRUE, scale=TRUE) #column center or standardize a matrix
```

Factors

A factor is a vector object used to specify a discrete classification (grouping) of variables. For instance, for a sample of individuals we can create the following vector:

```
s <- c("male", "female", "male", "male", "female", "female", "female")
```

By default this is a vector of character strings. It can be converted into a factor with two levels (male or female) with the command

```
sex <- factor(s)
```

Try typing:

```
print(s)
print(sex)
```

and check out the differences.

Matrix

R can also deal with multi-dimensional objects. For instance, you can create two-dimensional matrices in R with the command `matrix()`. With the options `ncol=`and/or `nrow=` you can specify the number of columns or rows, respectively.

A matrix can be constructed in many ways. For instance like this:

```
values <- matrix(c(2,1,6,1,5,7), nrow = 3, ncol = 2)
values
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    1    5
## [3,]    6    7
```

As you can see, a matrix is created from a vector of numbers, containing 3 rows (`nrow`) and 2 columns (`ncol`). You can use `dim()` to retrieve the dimensions of an R object, e.g., matrix or data frame.

To extract a single value from a matrix, we need to specify the column and the row number, in the format `matrix[row,col]`:

```
values[3,1]
```

```
## [1] 6
```

[1] 6

If you leave out the column, you can get the entire row and vice versa:

```
values[3,]
```

```
## [1] 6 7
```

```
values[,1]
```

```
## [1] 2 1 6
```

Other useful functions to apply on matrices are: `transpose()`, to switch rows and columns, and `rownames()` and `colnames()` to set and retrieve names for your rows and columns, respectively.

Data frame

For matrices, all elements of the object need to have the same format. If you want to include columns of different data types, you can use a *data frame*. Data frames can be created from scratch by combining vectors:

```
sex <- c("male", "female", "male")
weight <- c(80, 55, 70)
height <- c(1.90, 1.62, 1.78)
mydata <- data.frame(sex, weight, height)
print(mydata)
```

```
##      sex weight height
## 1   male     80   1.90
## 2 female     55   1.62
## 3   male     70   1.78
```

You can see that this is a data frame by running the `class()` function again:

```
class(mydata)
```

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

The columns of a data frame can be extracted by the following command `mydata$columnname`:

```
mydata$height
```

```
## [1] 1.90 1.62 1.78
```

We can also use the `$` to make a new column in a dataframe:

```
mydata$age <- c(25, 30, 56)
print(mydata)
```

```
##      sex weight height age
## 1   male     80   1.90  25
## 2 female     55   1.62  30
## 3   male     70   1.78  56
```

However, sometimes we want to make a new column based on existing data. For example, we might want to calculate the body mass index of the people in the data frame. BMI = weight/(height*height). We can do this using base R functionality as follows:

```
mydata$bmi <- mydata$weight/(mydata$height^2)
print(mydata)
```

```
##      sex weight height age      bmi
## 1   male     80   1.90   25 22.16066
## 2 female     55   1.62   30 20.95717
## 3   male     70   1.78   56 22.09317
```

However, most modern R programming uses the pipe operator, `%>%`, in combination with the `dplyr` package.

```
#install the package
install.packages('dplyr')
```

```
#load dplyr
library('dplyr')

#make a new column
mydata %>%
    mutate(BMI = weight/height^2)
```

```
##      sex weight height age      bmi      BMI
## 1   male     80   1.90  25 22.16066 22.16066
## 2 female     55   1.62  30 20.95717 20.95717
## 3   male     70   1.78  56 22.09317 22.09317
```

The pipe operator makes code very readable. It can be used multiple times in a row, where each line indicates a new operation. For example, here we first make the BMI column, then group the data by sex, calculate the average BMI per group and finally round the number:

```
#make a new column
mydata %>%
    mutate(BMI = weight/height^2) %>%
    group_by(sex) %>%
    summarise(avg_BMI = mean(BMI)) %>%
    mutate(avg_BMI = round(avg_BMI, digits = 1))
```

```
## # A tibble: 2 x 2
##   sex    avg_BMI
##   <chr>    <dbl>
## 1 female    21
## 2 male      22.1
```

Lists

Another class of data is a list. This is an object consisting of an ordered collection of objects known as its components. There is no particular need for the components to be of the same mode or type. For example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, a function, and so on. An example of a list is

```
mylist <- list(name="Fred",
               wife=c("Catherine"),
               parents=c("Patrick", "Corney"),
               no.children=3,
               child.ages=c(4,7,9))
```

This list consists of five components, two of them are character vectors of length 1, one character vector of length 2, one numeric vector of length 1, and one numeric vector of length 3. The various components of list can be extracted from the list using double square brackets. Like before, either indices or names can be used or the $ notation for extracting a component.

Try the following and see what happens:

```r
mylist[[1]]
```

```
## [1] "Fred"
```

```r
mylist[["wife"]]
```

```
## [1] "Catherine"
```

```r
mylist$no.children
```

```
## [1] 3
```

```r
mylist[["child.ages"]][1]
```

```
## [1] 4
```

Opening and saving files

Now we will describe in more detail how to open files within RStudio and how to save the output.

Setting working directory

You will often start your script by telling R what folder you are working in. You can use the command `setwd()` to set the location of your working directory. It is convenient to save your script, input and output in the same folder.

Type `getwd()` to see what folder you are currently in. If it is *not* `/g/teachinglab/data/atacseq/material/vignettes/Intro` use this to change the directory:

```r
setwd("/g/teachinglab/data/atacseq/material/vignettes/") #Don't forget the quotes
```

You can also change your current working directory by clicking on **Session** > **Set Working Directory** > **Choose Directory...**, but that way you will not record the folder in your script.

You can use the command `dir()` to see all the files in the directory that you are currently working in.

Importing text files

We often use R to do analyses on data that is saved in a text file (with .txt or .csv extension). To import that data into R, you can use multiple commands. The most widely used commands in base R are `read.table()` and `read.csv()`.

When R reads character data from a file, it will automatically convert them to factors. If you don't want that, you must add the option `as.is=TRUE` or `stringsAsFactors=FALSE` to the `read.table()` function.

Exercises

Exercise 4:

Read in the test file myfirstfile.txt:

```r
read.table('IntroRFiles/myfirstfile.txt')
```

`read.table()` is pretty good at recognising the separator between values automatically, but you can set it manually as well. Run the same command again, but now add `sep = "."`. What happens now?

Now, run the initial command, but this time add `header = TRUE` to the command. What's the difference?

Exercise 5:

Of course you do not just want to display the data in the console: you want to be able to work with it. You can assign the data to a variable. It is customary in R to call the variable with data in it `dm` (for data matrix) or `df` (for data frame), but this is completely up to you.

Read in myfirstfile.txt and assign it to `df`:

```
df <- read.table('IntroRFiles/myfirstfile.txt', header = T)
```

Other commands to import text files are implemented in packages: `fread()` in 'data.table', `read.table.ffdf()` from 'ff', and `read.big.matrix()` from 'bigmemory'. These functions are often useful for larger datasets. A good resource regarding these possibilities can be found here

Importing files from other formats

Other data formats can also be opened in R. Most extensions will have a specific function or package for opening and reading the data. For example, in this course you will be using RDS files and h5 files. The following lines have the package installation commented out, because you will not need these right now, but the command is there if you would like to use these in the future.

```
seuratObj = readRDS(file) #open an RDS file, this is base R functionality

#Install package Seurat for h5
#install.packages("Seurat")
library(Seurat)
countsFile = Read10X_h5(file)
```

It is also possible to read Excel tables (.xlsx) with the `readxl` package; GFF files, used to describe genes and their features, with the package `rtracklayer` from the website Bioconductor; and JSON files with the `jsonlite` package.

```
#Install package readxl for Excel files
#install.packages("readxl")
library(readxl)
read_excel(file, sheet = 1) #open Excel .xls or .xlsx file, if you have multiple sheets you can indicat

#Install package rtracklayer (via the website bioconductor) for GFF files, often used to describe genes
#if (!requireNamespace("BiocManager", quietly = TRUE))
#    install.packages("BiocManager")
#BiocManager::install("rtracklayer")
library(rtracklayer)
readGFF(file) #open GFF file

#Install package jsonlite for JSON files
#install.packages("jsonlite")
library(jsonlite)
fromJSON(file)
```

Saving files

After you have done your analyses, you will write the output to a file to keep it for later. Most of the time, we save files as text, csv or RDS files again. Similar to opening, there are functions for that. A simple text file can be saved with `write.table()`, this will write the data frame as a text file in your current working directory with a space as a separator between the columns. As with `read.table()`, the function has various arguments which can be used to alter these settings.

Exercises

Exercise 6: Change your working directory to `/mnt/data`

```
setwd("/mnt/data") #Don't forget the quotes
```

Save `df` as a new file called 'mysecondfile.txt':

```
write.table(df, 'mysecondfile.txt', quote = FALSE)
```

What happens if you add `col.names = FALSE` to the command? What about changing `quote = FALSE` to `quote = TRUE`?

You are now done with the Rstudio intro! Here's one last command to run:

```
praise()
```

Session info

It is good practice to print the so-called session info at the end of an R script, which prints all loaded libraries, their versions etc. This can be helpful for reproducibility and recapitulating which package versions have been used to produce the results obtained above.

```
sessionInfo()
```

```
## R version 4.1.2 (2021-11-01)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: CentOS Linux 7 (Core)
##
## Matrix products: default
## BLAS/LAPACK: /g/easybuild/x86_64/CentOS/7/haswell/software/FlexiBLAS/3.0.4-GCC-11.2.0/lib64/libflexi
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] dplyr_1.0.8
##
## loaded via a namespace (and not attached):
##  [1] rstudioapi_0.13  knitr_1.36       magrittr_2.0.3   tidyselect_1.1.2
##  [5] R6_2.5.1         rlang_1.0.2      fastmap_1.1.0    fansi_1.0.3
##  [9] stringr_1.4.0    highr_0.9        tools_4.1.2      xfun_0.30
## [13] utf8_1.2.2       DBI_1.1.1        cli_3.3.0        htmltools_0.5.2
## [17] ellipsis_0.3.2   assertthat_0.2.1 yaml_2.3.5       digest_0.6.29
## [21] tibble_3.1.6     lifecycle_1.0.1  crayon_1.5.1     purrr_0.3.4
## [25] vctrs_0.4.1      glue_1.6.2       evaluate_0.14    rmarkdown_2.11
## [29] stringi_1.7.6    compiler_4.1.2   pillar_1.7.0     generics_0.1.2
## [33] pkgconfig_2.0.3
```