A tutorial for getting started with R and RStudio

Version 1.0.0

Konstantinos I. Bougioukas

07/10/2021

Contents

Learning Objectives

- Describe the purpose and use of each pane in the RStudio IDE
- Locate buttons, options in the RStudio IDE
- Work with R projects
- Understand the concept of functions and packages
- Use R as a calculator
- Use mathematical functions and relational operators
- Understand special values (NA, Inf, NaN)
- Understand the concept of objects in R
- Set legal names to objects
- Work with assignment operators

1 R and RStudio basics

1.1 Installing R and RStudio

We first need to download and install both R and RStudio (Desktop version) on our computer. First, we install R, then we will need to install RStudio.

1. We must do this first: Download and install R

- If you are a Windows user: Click on "Download R for Windows", then click on "base", then click on the Download link.
- If you are macOS user: Click on "Download R for (Mac) OS X", then under "Latest release:" click on R-X.X.X.pkg, where R-X.X.X is the version number.

2. We must do this second: Download and install RStudio

- Scroll down to "Installers for Supported Platforms" near the bottom of the page.
- Click on the download link corresponding to your computer's operating system.

1.2 What are R and RStudio?

We will use R via RStudio. First time users often confuse the two. At its simplest R is like a car's engine while RStudio is like a car's dashboard (Figure 1).

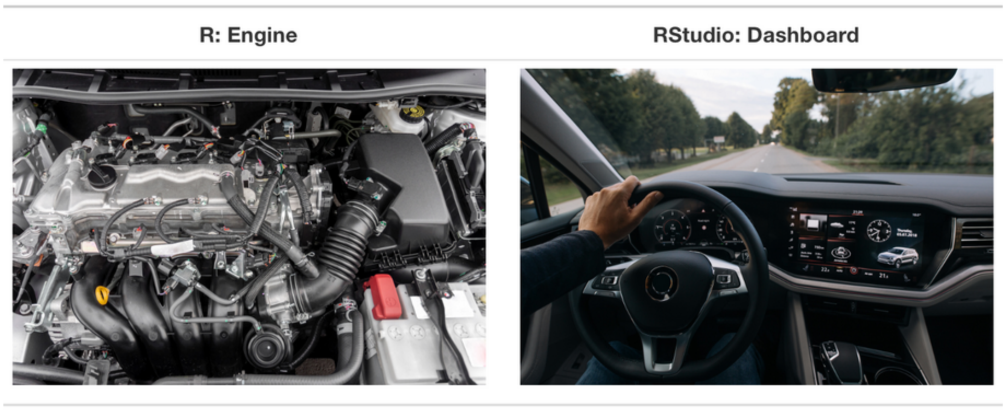


Figure 1: Analogy of difference between R and RStudio

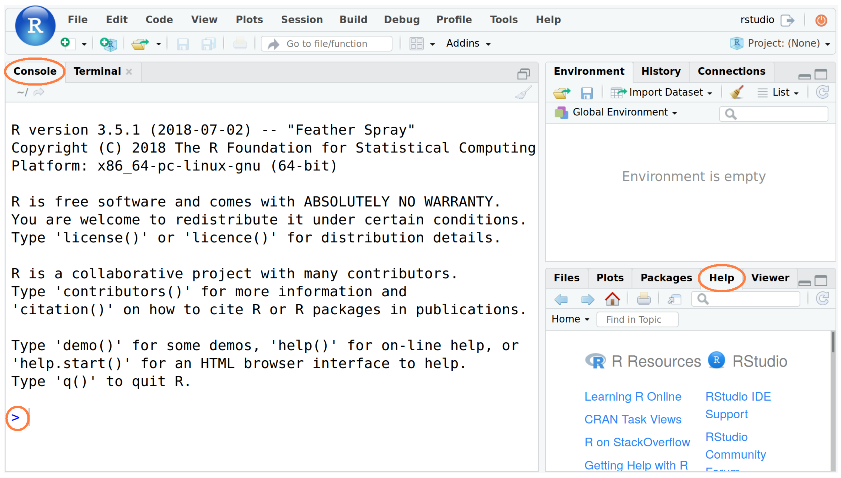
More precisely, R is a programming language and open-source software environment for statistical analysis, graphics representation and reporting while RStudio is an integrated development environment (IDE) that provides an interface by adding many convenient features and tools. So just as the way of having access to a speedometer, rear-view mirrors, and a navigation system makes driving much easier, using RStudio's interface makes using R much easier as well. After we install R and RStudio on our computer, we'll have two new programs (also called applications) we can open. We'll always work in **RStudio** and not R. Figure 2 shows what icon we should be clicking on our computer.

R: Do not open this	RStudio: Open this

Figure 2: Icons of R versus RStudio

1.3 Starting R & RStudio

R starts automatically when we open RStudio (see Figure 3) and we are greeted by three panes:



```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

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.

>
```

Figure 3: RStudio Screenshot with Console on the left and Help tab in the bottom right.

The three main panes that divide the screen are: (a) the Console pane, (b) the Environment pane, and (c) the the Files/Plots/Packages/Help/Viewer pane. Over the course of this textbook, we'll come to learn what purpose each of these panes serve.

Note The Console pane starts with information about the version number, license and contributors, and provides some guidance on how to get help. The last line is a standard command prompt **>** that indicates R is ready and expecting instructions to do something.

Let's type 14 + 16 at the R command prompt and press Enter:

```
14 + 16
```

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

So what happened?

Well, R gave us a response (output) to our input (14 + 16). That response came after we pressed the Enter key. It was `[1] 30`. It's clear that 30 is the answer to the 14 + 16. However, what does the [1] mean? At this point we can pretty much ignore it, but technically it refers to the index of the first item on each line. (Sometimes R prints out many lines as a result. The number inside the brackets helps you figure out where in the sequence we are per line).

1.4 R scripts

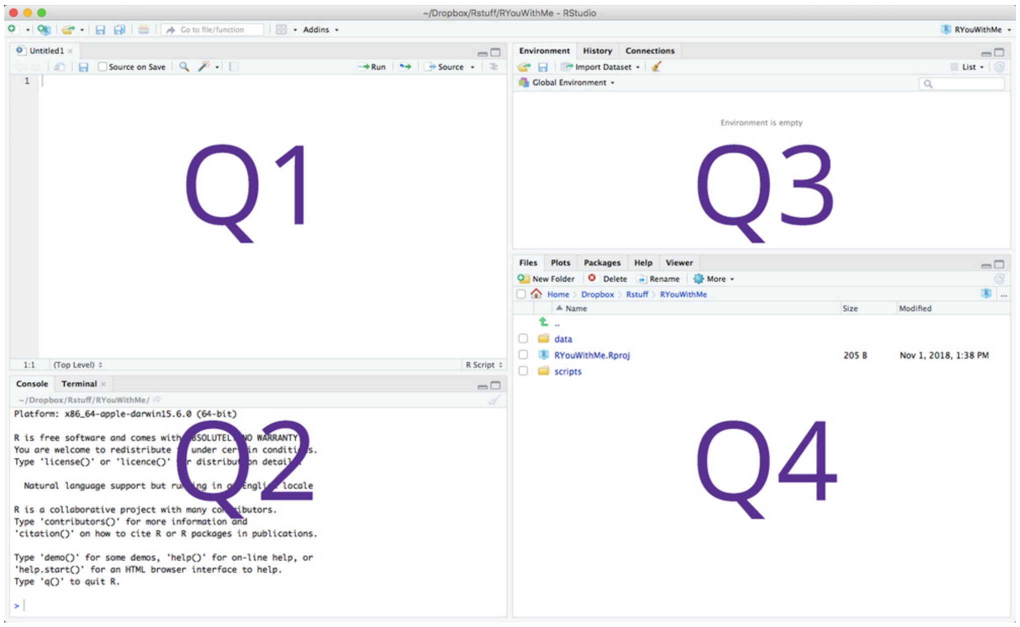
Usually, we do our programming by writing our commands in script files. An R script (with the ".R" extension) is simply a text file in which our R commands are stored in a logical sequence, and then they can be run.

There are many advantages writing our R code as an R script file:

- We can save and reuse our code
- We can document our work (however, we can only include one-line comments and these must be prefixed with the hash symbol, #)

- We can share our work with others
- We can move beyond writing one line of code at a time

If we go to **File > New File > R Script** in the RStudio menu, we should see another pane (Q1) opened on the left above the interactive console (see Figure 4). This is where we can write a length script with lots of lines of code, and save the file for future use.



```
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

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.

>
```

Figure 4: RStudio Screenshot with four panes.

Therefore, the RStudio window is divided into **four** panes (quadrants) that contain:

- Q1 - code editor (script)
- Q2 - console
- Q3 - environment (workspace), history
- Q4 - files, plots, packages, help, viewer (for local web content)

Note The four panes might be in a different order that those in Figure 4. If we'd like, we can change the order of the windows under RStudio preferences (**Tools > Global Options > Pane layout**). We can also change their shape by either clicking the minimize or maximize buttons on the top right of each pane, or by clicking and dragging the middle of the borders of the windows.

We can execute our code line by line by putting the cursor on the line or selecting a chunk of lines (by highlighting the text) and pressing the **run** button in the source window.

Note We can also run our selected code using the keywboard shortcut "Ctrl + Enter".

1.5 Errors, warnings, and messages in R

Let's type the following at the R command prompt and press Enter:

```
hello
```

we get the following error:
Error: object 'hello' not found

Note Keep in mind that R is an object oriented programming language, meaning that **everything** in R is an object.

One thing that intimidates new R and RStudio users is how it reports errors, warnings, and messages. R reports errors, warnings, and messages in a glaring red font, which makes it seem like it is scolding us. However, seeing red text in the console is not always bad.

R will show red text in the console pane in three different situations:

- **Errors:** When the red text is a legitimate error, it will be prefaced with "Error…" and try to explain what went wrong. Generally when there's an error, the code will not run.
- **Warnings:** When the red text is a warning, it will be prefaced with "Warning:" and R will try to explain why there's a warning. Generally our code will still work, but with some caveats.
- **Messages:** When the red text doesn't start with either "Error" or "Warning", it's just a friendly message.

Now, let's type the following:

```
1 + 2 -
+
```

Note If an R command is not complete then R will show a plus sign (+) prompt on second and subsequent lines until the command syntax is correct. To break out this, press the escape key (ESC).

1.6 R Help resources

Before asking others for help, it's generally a good idea for us to try to help ourself. It is strongly recommended to learn how to use R's useful and extensive built-in help system which is an essential part of finding solutions to our R programming problems.

We can use the `help()` function or `?` help operator which provide access to the R documentation pages for a specific term. For example, if we want information for the `mean` we type the following commands (which give the same result):

```
help(mean)
```

or

```
?mean
```

Note In console to recall a previously typed commands use the up arrow key (↑). To go between previously typed commands use the up and down arrow (↓) keys. To modify or correct a command use the left (←) and right arrow (→) keys.

Two question marks (`??`) will search R documentation for a phrase or term. So for example, let's say you want to search documentation for `linear regression` analysis. Keep in mind if your phrase is more than one word long, you must put it in quotation marks.

```
??"linear regression"
```

To do a keyword search use the `apropos()` command with the keyword in double quotes (`"keyword"`) or single quote (`'keyword'`). For example:

```
apropos("mean")
```

```
##  [1] ".colMeans"      ".rowMeans"
##  [3] "colMeans"       "cummean"
##  [5] "kmeans"         "mean"
##  [7] "mean.Date"      "mean.default"
##  [9] "mean.difftime"  "mean.POSIXct"
## [11] "mean.POSIXlt"   "mean_cl_boot"
## [13] "mean_cl_normal" "mean_sdl"
## [15] "mean_se"        "rowMeans"
## [17] "weighted.mean"
```

Use the `example()` command to run the examples at the end of the help for a function:

```
example(mean)
```

```
##
## mean> x <- c(0:10, 50)
##
## mean> xm <- mean(x)
##
## mean> c(xm, mean(x, trim = 0.10))
## [1] 8.75 5.50
```

RStudio also provides search box in the **"Help"** tab to make our life easier (see Figure 3).

Like Google, but for R is the RSeek.Org. This is not included in R but is a great search engine built specifically for R-related queries.

Additionally, there are a lot of **on-line** resources that can help (e.g., R-bloggers. However, we must understand that blindly copying and pasting could be harmful and further it won't help us to learn and develop.

Note When we search on-line we should use [R] in our search term (e.g. [R] summary statistics by group) which gives an exact match of R keyword. Often there is more than one solution to our problem. It is good to investigate the different options.

1.7 Quitting R & RStudio

When we quit RStudio we will be asked whether to save workspace with two options:

- "Yes" – Our current R workspace (containing the work that we have done) will be restored next time we open RStudio.
- "No" – We will start with a fresh R session next time we open RStudio. For now select "*No*" to prevent errors being carried over from previous sessions.

2 Working with Projects

2.1 Create a new Project

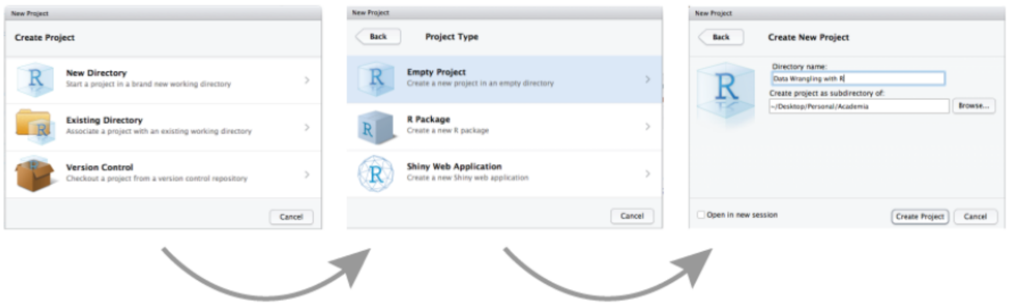
Keeping all the files associated with a project organized together – input data, R scripts, documents and figures – is such a wise and common practice that RStudio has built-in support for this via its projects.

RStudio projects are associated with R working directories. We can create an RStudio project:

- In a new directory
- In an existing directory where you already have R code and data
- By cloning a version control (Git or Subversion) repository

Let's make one to use for the rest of this introduction course. We do this: `File > New Project...`. The directory name we choose here will be the `project name`. We call it whatever we want (e.g. MSc_medstats).

RStudio projects are associated with R working directories (Figure 5):



Directory name: Data Wrangling with R

Create project as subdirectory of: ~/Desktop/Personal/Academia

Figure 5: Analogy of R versus R packages

Note

Think carefully about which **"subdirectory"** we put the project in. If we don't store it somewhere sensible, it will be hard to find it in the future!

2.2 Project folder structure

The proposed `Project` should contain the following subfolders (Figure 6):

- data: data files of any kind, such as `.csv`, `.xlsx`, `.txt`, etc.
- documents: documents of any formats, such as `.docx`, `.pdf`, `.tex`, `.Rmd`, etc.
- figures: plots, diagrams, and other figures
- scripts: all your R scripts and codes (`.R` extension)

Additionaly, there are two important files in the project folder that are created automatically by Rstudio: **`name_of_project.Rproj`**, and **`.Rhistory`**:

- `name_of_project.Rproj`: contains options and meta-data of the project (encoding, the number of spaces used for indentation, whether or not to restore a workspace with launch, etc) and can also be used as a shortcut for opening the project directly from the filesystem.

- `.Rhistory`: contains a history of code executed

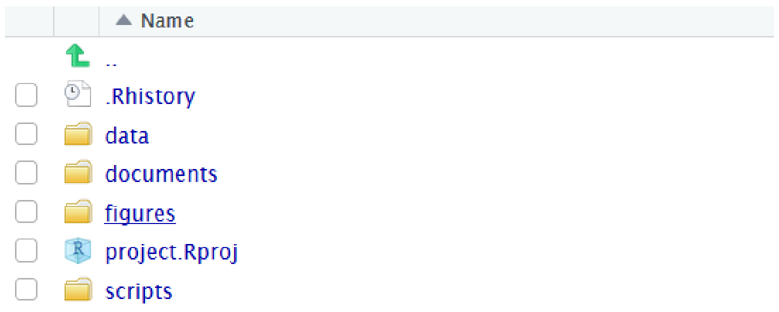


Figure 6: Project folder structure

3 R Functions

3.1 Characteristics of R Functions

We have already used some R functions searching for help (i.e. `help()`, `example()`, `apropos()`). Most of the computations in R involves using functions.

> A function essentially has a **name** and a list of **arguments** separated by a comma.

Let's have look at an example:

```r
seq(from = 5, to = 8, by = 0.5)
```

```
## [1] 5.0 5.5 6.0 6.5 7.0 7.5 8.0
```

The function name is `seq` and it has three explicitly named arguments `from`, `to` and `by`. The arguments `from` and `to` are the start and end values of a sequence that we want to create, and `by` is the increment of the sequence.

The above result can also be obtained without naming the arguments as follows:

```r
seq(5, 8, 0.5)
```

```
## [1] 5.0 5.5 6.0 6.5 7.0 7.5 8.0
```

And what about this?

```r
seq(5, 8)
```

```
## [1] 5 6 7 8
```

This result demonstrates something about how R resolves function arguments.

> If we don't use argument names, R will match the arguments in the order that they appeared (positional matching).

Here, it is assumed that we want a sequence from = 5 that goes to = 8. Since we didn't specify step size, the default value of `by` in the function definition is used, which ends up being 1 in this case.

Moreover, the `seq()` function has other arguments that we could use which are documented in the help page running `?seq`. For example, we could use the argument `length.out` (instead of `by`) to fix the length of the sequence as follows:

```r
seq(from = 5, to = 8, length.out = 16)
```

```
##  [1] 5.0 5.2 5.4 5.6 5.8 6.0 6.2 6.4 6.6 6.8
## [11] 7.0 7.2 7.4 7.6 7.8 8.0
```

Note Some arguments in a function may be **optional**. We can use args() for displaying the argument names and corresponding default values of a function.

Let's see for example the `log()` function:

```r
args(log)
```

```
## function (x, base = exp(1))
## NULL
```

In the `log()` function `base` comes with a default value exp(1). This means that `base` is an optional argument.

R will use our value for `base` if we supply one:

```r
log(15, base = 10)  # R uses our value 10
```

```
## [1] 1.176091
```

otherwise R will use the default vaue which here is `exp(1)`:

```
log(15)  # R uses the default value of `exp(1)`
```

```
## [1] 2.70805
```

In contrast, `x` is a required argument because it is not come with a default value. If we don't supply `x` argument the `log()` function will fail:

```
log(base=10)
```

Error: argument x is missing, with no default

Note Not all functions have (or require) arguments.

For example:

```
date()
```

```
## [1] "Thu Oct 07 11:50:40 2021"
```

Note Functions "live" in Base R and in R packages!

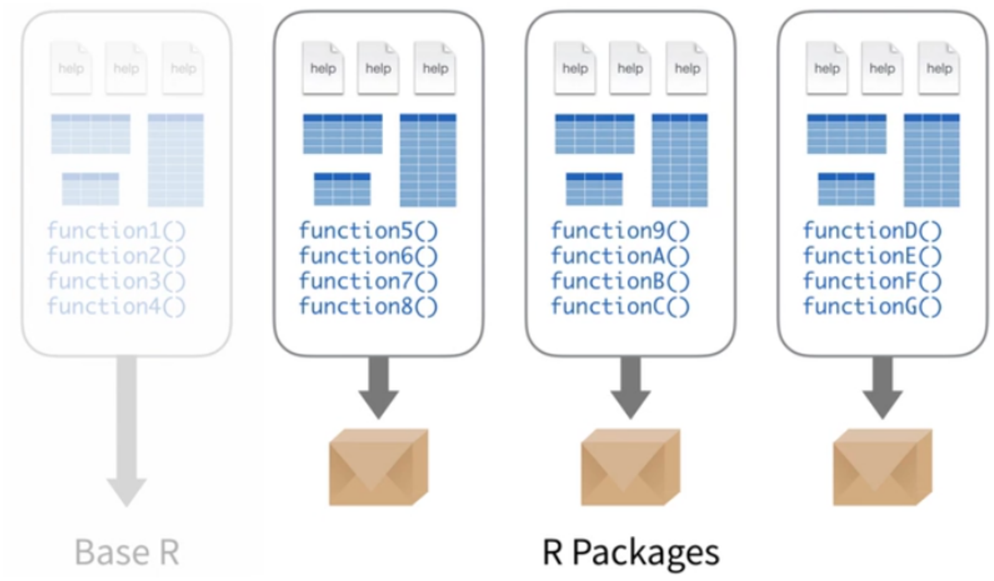


Figure 7: Functions live inside the R packages that can be downloaded from the internet

3.2 Custom Functions

We can create our own functions (using the `function()`) which is a very powerful way to extend R. Writing our own functions is outside the scope of this guide. As we get more and more familiar with R it is very likely that we will eventually need to learn about them but for now we don't need to.

For example a function that convert Celsius to Kelvin is:

```r
celsius_to_kelvin <- function(temp_C) {
  temp_K <- temp_C + 273.15
  return(temp_K)
}
# freezing point of water in Kelvin
celsius_to_kelvin(0)
```

```
## [1] 273.15
```

4 R packages

4.1 What are R packages? A good analogy

R packages are another interesting point in R.

> R packages extend the functionality of R by providing **additional functions, data, and documentation**. They are written by a world-wide community of R users and can be downloaded from the internet.

For example, among the many packages we will use in this course are the `ggplot2` package for data visualization, the `dplyr` package for data wrangling.

A good analogy for R packages is that they are like apps we can download onto a mobile phone (Figure 8):

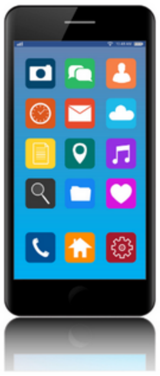
R: A new phone	R Packages: Apps you can download
	

Figure 8: Analogy of R versus R packages

So `R` **is like a new mobile phone**: while it has a certain amount of features when we use it for the **first time**, it doesn't have everything. `R` **packages are like the apps** we can download onto our phone from Apple's App Store or Android's Google Play.

Let's continue this analogy by considering the **Instagram** app for editing and sharing pictures. Say we have purchased a new phone and we would like to share a photo we have just taken with friends and family on Instagram. We need to:

1. *Install the app*: Since our phone is new and does not include the Instagram app, we need to download the app from either the App Store or Google Play. We do this once and we're set for the time being. We might need to do this again in the future when there is an update to the app.
2. *Open the app*: After we've installed Instagram, we need to open the app.

Once Instagram is open on our phone, we can then proceed to share our photo with our friends and family. The process is very similar for using an R package. We need to:

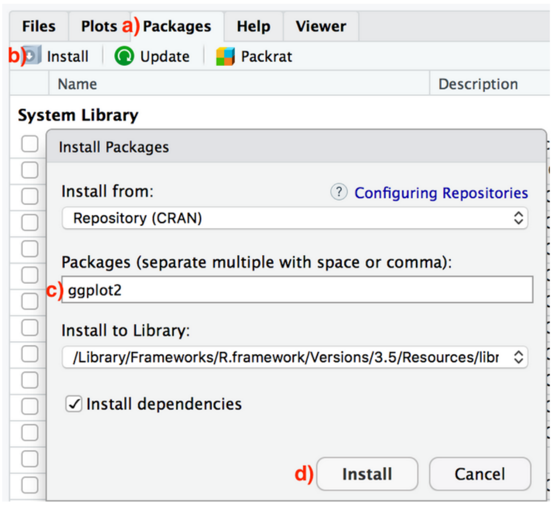
1. ***Install the package***: This is like installing an app on our phone. Most packages are not installed by default when we install R and RStudio. Thus if we want to use a package for the first time, we need to install it first. Once we've installed a package, we likely won't install it again unless we want to update it to a newer version.
2. ***Load the package***: Loading a package is like opening an app on our phone. Packages are not loaded by default when we start RStudio on our computer; we need to "load" each package we want to use every time we start RStudio.

Let's now show how to perform these two steps for the `ggplot2` package for data visualization.

4.2 Package installation

There are two ways to install an R package: an easy way and a more advanced way. Let's install the `ggplot2` package the easy way first as shown in Figure 9. In the Q4 - files, plots, packages, help pane of RStudio:

a) Click on the "Packages" tab.
b) Click on "Install" next to Update.
c) Type the name of the package under "Packages (separate multiple with space or comma):" In this case, type `ggplot2`.
d) Click "Install."



Install from: Repository (CRAN)

Packages (separate multiple with space or comma): ggplot2

Install to Library: /Library/Frameworks/R.framework/Versions/3.5/Resources/libr

☑ Install dependencies

Figure 9: Installing packages in R the easy way

An alternative but slightly less convenient way to install a package is by typing `install.packages("ggplot2")` in the console pane of RStudio and pressing Return/Enter on our keyboard. Note we must include the quotation marks around the name of the package.

Much like an app on our phone, **we only have to install a package once**. However, if we want to update a previously installed package to a newer version, we need to reinstall it by repeating the earlier steps.

4.3 Package loading

Recall that after we've installed a package, we need to **load it**. In other words, we need to "open it." We do this by using the `library()` command (note that the the quotation marks are not necessary when we are loading a package). For example, to load the `ggplot2` package, run the following code in the console pane. What do we mean by "run the following code"? Either type or copy & paste the following code into the console pane and then hit the Enter key.

```
library(ggplot2)
```

If after running the earlier code, a blinking cursor returns next to the > “prompt” sign in console, it means we were successful and the `ggplot2` package is now loaded and ready to use. If however, we get a red “error message” that reads...

Error in library(ggplot2) : there is no package called ‘ggplot2’

... it means that we didn’t successfully install it.

There is one way in R that can use a `function` without using `library()`. To do this, we can simply use the notation `package::function` .

For example:

```
ggplot2::ggsave()
```

The above notation tells R to use the `ggsave()` function from `ggplot2` without load the `ggplot2` package.

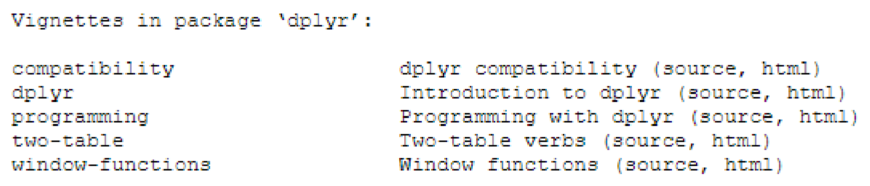
4.4 Getting help on packages with `vignette()`

Many packages come with **“vignettes”**: tutorials and extended example documentation. Without any arguments, `vignette()` will list all vignettes for all installed packages; `vignette(package="package-name")` will list all available vignettes for `package-name`, and `vignette("vignette-name")` will open the specified vignette.

If a package doesn’t have any vignettes, we can usually find help by typing `help("package-name")`, or `?package-name`.

For example, let’s list the vignettes available in `dplyr` package. We will be using the `dplyr` package later in this textbook, to manipulate tables of data. we might want to take a look at these now, or later when we cover `dplyr`. Figure 10 shows that there are several vignettes included in the package.

```
vignette(package="dplyr")
```



```
Vignettes in package 'dplyr':

compatibility               dplyr compatibility (source, html)
dplyr                       Introduction to dplyr (source, html)
programming                 Programming with dplyr (source, html)
two-table                   Two-table verbs (source, html)
window-functions            Window functions (source, html)
```

Figure 10: Vignettes in package 'dplyr'

The `dplyr` vignette looks like it might be useful later. We can view this with:

```
vignette(package="dplyr", "dplyr")
```

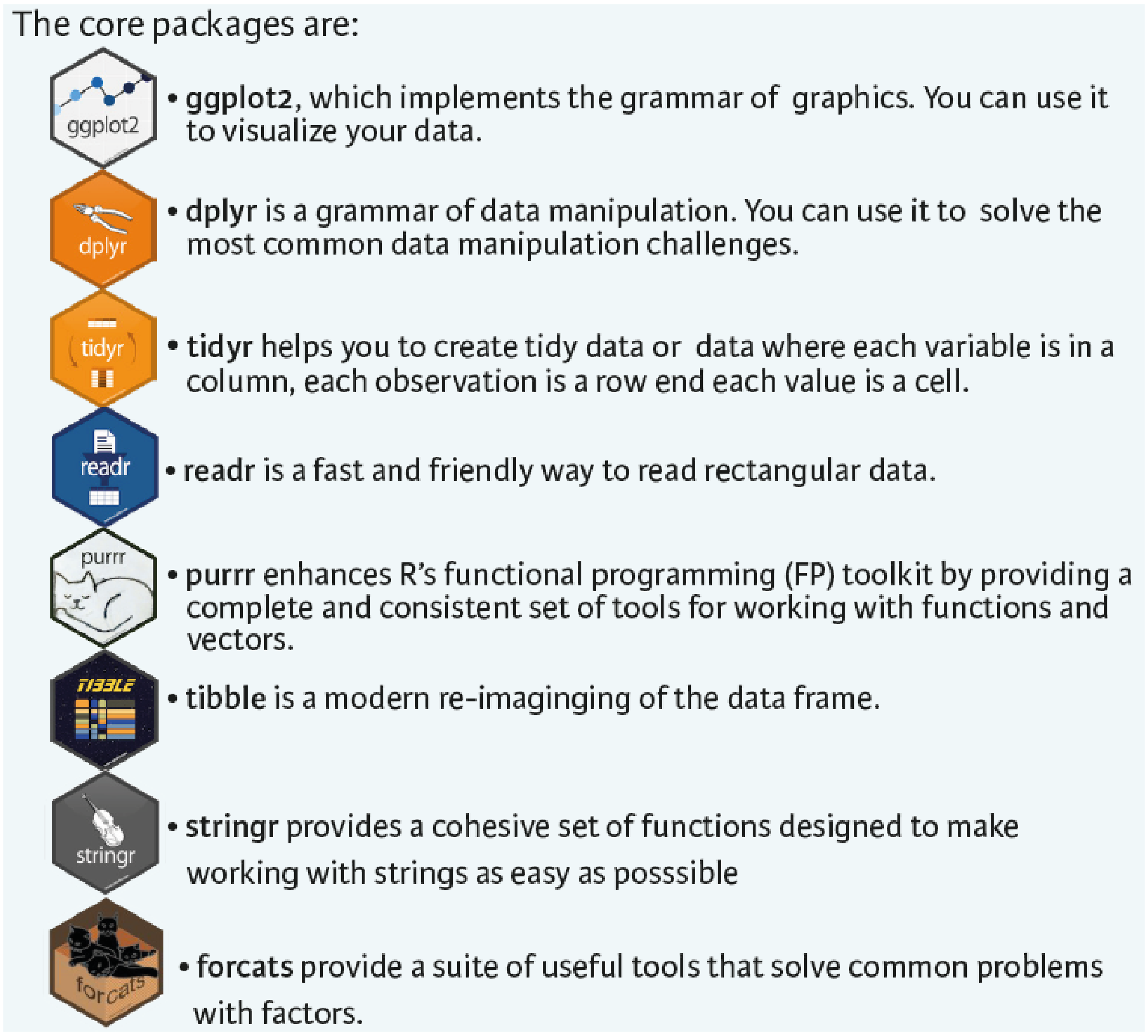
4.5 The `{tidyverse}` package

In this tutorial we will use the `tidyverse` package. The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

The command `install.packages("tidyverse")` will **install** the complete tidyverse. The `tidyverse` package provides a shortcut for downloading the following packages:

```
##  [1] "broom"         "cli"
##  [3] "crayon"        "dbplyr"
##  [5] "dplyr"         "dtplyr"
##  [7] "forcats"       "googledrive"
##  [9] "googlesheets4" "ggplot2"
## [11] "haven"         "hms"
## [13] "httr"          "jsonlite"
## [15] "lubridate"     "magrittr"
## [17] "modelr"        "pillar"
## [19] "purrr"         "readr"
## [21] "readxl"        "reprex"
## [23] "rlang"          "rstudioapi"
## [25] "rvest"          "stringr"
## [27] "tibble"         "tidyr"
## [29] "xml2"           "tidyverse"
```

When we **load** the tidyverse package with the command `library(tidyverse)`, R will load the **core tidyverse** and make it available in our current R session (Figure 11):



The core packages are:

- **ggplot2**, which implements the grammar of graphics. You can use it to visualize your data.
- **dplyr** is a grammar of data manipulation. You can use it to solve the most common data manipulation challenges.
- **tidyr** helps you to create tidy data or data where each variable is in a column, each observation is a row end each value is a cell.
- **readr** is a fast and friendly way to read rectangular data.
- **purrr** enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors.
- **tibble** is a modern re-imaginging of the data frame.
- **stringr** provides a cohesive set of functions designed to make working with strings as easy as posssible
- **forcats** provide a suite of useful tools that solve common problems with factors.

Figure 11: Core tidyverse packages

Note The "tidyverse" packages out of the core list have more specialised usage and are not loaded automatically with library(tidyverse), **so we'll need to load each one with its own call to library()**.

4.6 The `{here}` package

We've talked about what `Projects` are and why we should use them, but what really makes `Projects` in `RStudio` shine is the use of the `here()` function from the {here} package. What `here()` does is eliminate the need for us to do what's called "hard-coding" our file path.

> The place where the file lives on our computer is called the `path`. We can think of the path as directions to the file. There are two kinds of paths: `absolute paths` and `relative paths`.

For example, suppose Alice and Bob are working on a project together and want to read in R studio of their computers a dataset file named `covid19.csv` data. They could do this using either a `relative` or an `absolute` path. We show what both choices below:

1. **Reading data using an `absolute` path**

Alice's file is stored at `C:/alice/project/data/covid19.csv`

and the command in R script should be:

```
covid19 <- read_csv("C:/alice/project/data/covid19.csv")
```

while Bob's is stored at `C:/bob/project/data/covid19.csv`

and the command in R script should be:

```r
covid19 <- read_csv("C:/bob/project/data/covid19.csv")
```

Even though Alice and Bob stored their files in the same place on their computers (in their `C` disk), the absolute paths are different due to their different usernames. If Bob has code that loads the `covid19.csv` data using an absolute path, the code won't work on Alice's computer.

2. **Reading data using a `relative` path**

The command in R script should be:

```r
covid19 <- read_csv(here("data", "covid19.csv"))
```

What `here()` does is tell R that the file structure starts at the Project-level, and so every subsequent call starts at the Project-level, and allows us to navigate throughout each of the folders and files within a given Project.

The relative path from inside the project folder (data/covid19.csv) is the same on both computers; any code that uses relative paths will work on both!

5 Calculations with R

5.1 Using R as a calculator

5.1.1 Arithmetic calculations

The simplest thing we could do with R is do arithmetic with scalar data. For example:

```
1 + 100
```

```
## [1] 101
```

R printed out the answer, with a preceding `[1]`. Don't worry about this for now, we'll explain that later. For now think of it as indicating output.

Note A scalar data is the most basic data structure that holds only a single atomic value at a time. Using scalars, more complex data types can be constructed (such as vectors and matrices).

5.1.2 Order of operations (arithmetic operators)

The basic arithmetic operators are presented in Figure 12:

Arithmetic Operators
+
-
*
/
^

Figure 12: Arithmetic operators

Remember when using R as a calculator, the order of operations is the same as we would have learned back in school.

From highest to lowest precedence:

- Parentheses: `( )`

- Exponents: `^` or `**`
- Divide: `/`
- Multiply: `*`
- Add: `+`
- Subtract: `-`

Therefore:

```r
3 + 5 * 2
```

```
## [1] 13
```

5.1.3 Parentheses

Use parentheses to group operations in order to force the order of evaluation if it differs from the default, or to make clear what we intend.

```r
(3 + 5) * 2
```

```
## [1] 16
```

This can get unwieldy when not needed, but clarifies our intentions. Remember that others may later read our code.

```r
(3 + (5 * (2 ^ 2))) # hard to read
3 + 5 * 2 ^ 2       # clear, if we remember the rules
3 + 5 * (2 ^ 2)     # if we forget some rules, this might help
```

Note The text after each line of code is called a **"comment"**. Anything that follows after the hash symbol **"#"** is ignored by R when it executes code. It is considered good practice to comment your code when working in an .R script.

5.2 Mathematical functions

R has many built in mathematical functions. To call a function, we simply **type its name, followed by open and closing parentheses**. Remember that anything we type inside the parentheses are called the function's **arguments**.

Examples

```r
sin(pi/2)  # trigonometry functions, angles in radians
```

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

```r
log(100)  # natural logarithm
```

```
## [1] 4.60517
```

```r
log10(100) # base-10 logarithm
```

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

```r
exp(0.5) # e^(1/2)
```

```
## [1] 1.648721
```

```r
sqrt(9)    # squared root
```

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

```r
abs(-9)    # absolute value
```

```
## [1] 9
```

Hint The **round()** function is often very useful.

For example:

```r
round(7 / 3)                       # rounding the number the nearest integer
```

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

```r
round(7 / 3, digits = 3)           # rounding the number to two decimal places
```

```
## [1] 2.333
```

The round function follows the **rounding principle**. By default, we will get the nearest integer to 7 / 3, which is 2. If we want to control the approximation accuracy, we can add a digits argument to specify how many digits we want after the decimal point. Here you will get 2.333 after adding digits = 3.

Note that if the first digit that is dropped is exactly 5, R uses a rule that's common in programming languages: Always round to the nearest **even** number. For example:

```r
round(1.5)
```

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

```r
round(2.5)
```

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

```r
round(4.5)
```

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

```r
round(5.5)
```

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

5.3 Relational (comparison) operators

The basic relational operators are presented in Figure 13:

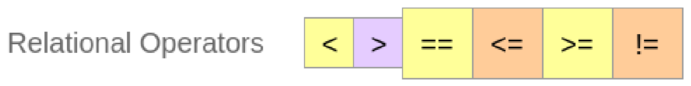


Figure 13: Relational operators

We can also evaluate the relationship between two arithmetic values using the `relational operators` (Table 2.1):

Table 1: Relational (comparison) operators

symbol	read as
<	less than
>	greater than
==	equal to
<=	less than or equal to
>=	greater than or equal to
!=	not equal to

Examples

```r
2 < 1  # less than
```

```
## [1] FALSE
```

```r
1 > 0  # greater than
```

```
## [1] TRUE
```

```r
1 == 1  # equal to (double equal sign for equality)
```

```
## [1] TRUE
```

```r
1 <= 1  # less than or equal to
```

```
## [1] TRUE
```

```r
-9 >= -3 # greater than or equal to
```

```
## [1] FALSE
```

```r
1 != 2  # not equal to (inequality)
```

```
## [1] TRUE
```

When dealing with real numbers (floating point number), we must be very careful when checking equality. For example, we know from trigonometry that:

$$\cos(\frac{\pi}{2}) = \cos(\frac{3\pi}{2}) = 0$$

However, if we compare the two numbers we get:

```
cos(pi/2) == cos(3*pi/2)
```

```
## [1] FALSE
```

In this instance, we can test for 'near equality' applying the `all.equal()` function which tests for equality with a difference tolerance of 1.5e-8.

```
all.equal(cos(pi/2), cos(3*pi/2))
```

```
## [1] TRUE
```

If the difference is greater than the tolerance level 1.5e-8 the function will return the mean relative difference.

Try to test this:

```
all.equal(4.0000007, 4.0000008)
```

Note: Comparing Numbers We should never use **==** to compare two numbers unless they are integers (whole numbers) or fractions whose denominator is a power of 2 (2^n, where n is integer). Computers may only represent decimal numbers with a certain degree of precision (rounded to typically 53 binary digits accuracy which allows representing decimal numbers to 15 or 16 significant digits), so two numbers which look the same when printed out by R, may actually have different underlying representations and therefore be different by a small margin of error (called Machine numeric tolerance).

Note: Scientific notation is a special way of expressing numbers that are too big or too small to be conveniently written in decimal form. Generally, it expresses numbers in forms of $m \times 10^n$ and R uses the **e** notation. Note that the e notation has nothing to do with the natural number e .

Examples

- 0.0055 is written 5.5×10^{-3}
  because 0.0055 = 5.5 × 0.001 = 5.5 × 10^{-3} or 5.5e-3
- 0.000000015 is written 1.5×10^{-8}
  because 0.000000015 = 1.5 × 0.00000001 = 1.5 × 10^{-8} or 1.5e-8
- 5500 is written 5.5×10^{3}
  because 5500 = 5.5 × 1000 = 5.5 × 10^{3} or 5.5e3
- 150000000 is written 1.5×10^{8}
  because 150000000 = 1.5 × 100000000 = 1.5 × 10^{8} or 1.5e8

5.4 Special values

When dealing with numeric data, integers, real or complex numbers are not enough to manage the real and the mathematical world.

5.4.1 Missing values (NA)

For example, in the real world, missing values may occur when recording sensor information. R uses a special numeric value `NA` standing for `Not available`. Mathematical operations using `NA` produces NA:

```
cos(NA)
```

```
## [1] NA
```

5.4.2 Infinitive: `-Inf` or `Inf`

There is also a special number `Inf` which represents infinity. Fortunately, R has special numbers for this.

This allows us to represent entities like:

```
1/0
```

```
## [1] Inf
```

The `Inf` can also be used in ordinary calculations:

```
exp(-Inf)
```

```
## [1] 0
```

5.4.3 Not A Number (NaN)

The value NaN represents an undefined value and it is usually the product of some arithmetic operations. For example:

```
Inf/Inf
```

```
## [1] NaN
```

```
0/0
```

```
## [1] NaN
```

```r
cos(Inf)
```

```
## Warning in cos(Inf): NaNs produced
```

```
## [1] NaN
```

6 R Objects

6.1 What are the objects in R

R works with objects (it is an object-oriented programming language). All the things that we manipulate or encounter in R such as numbers, data structures, functions, the results from a function (e.g., graphs) are types of objects. Some objects come with the packages in R. Other objects are user-created. User-created objects have names that are assigned by the user. R has a workspace known as the global environment that can be used to store objects.

Objects in R can have many properties associated with them, called attributes. These properties explain what an object represents and how it should be interpreted by R. Two of the most important attributes of an R object are the `class` and the `dimension` of the object.

Attributes of an object (if any) can be accessed using the `attributes()` function. Not all R objects contain attributes, in which case the `attributes()` function returns `NULL`.

For example, the famous `iris` data set is a `data.frame` that contains 150 rows and 5 columns:

```
class(iris); dim(iris)
```

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

```
## [1] 150   5
```

```
attributes(iris)
```

```
## $names
## [1] "Sepal.Length" "Sepal.Width"
## [3] "Petal.Length" "Petal.Width"
## [5] "Species"
##
```

```
## $class
## [1] "data.frame"
##
## $row.names
##   [1]   1   2   3   4   5   6   7   8   9  10
##  [11]  11  12  13  14  15  16  17  18  19  20
##  [21]  21  22  23  24  25  26  27  28  29  30
##  [31]  31  32  33  34  35  36  37  38  39  40
##  [41]  41  42  43  44  45  46  47  48  49  50
##  [51]  51  52  53  54  55  56  57  58  59  60
##  [61]  61  62  63  64  65  66  67  68  69  70
##  [71]  71  72  73  74  75  76  77  78  79  80
##  [81]  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
## [101] 101 102 103 104 105 106 107 108 109 110
## [111] 111 112 113 114 115 116 117 118 119 120
## [121] 121 122 123 124 125 126 127 128 129 130
## [131] 131 132 133 134 135 136 137 138 139 140
## [141] 141 142 143 144 145 146 147 148 149 150
```

Note R commands are usually separated by a new line but they can also be separated by a semicolon (`;`).

6.2 Named storage of objects

6.2.1 Assignment operator (<-)

In R we can store things in objects using the leftward assignment operator (<-) which is an arrow that points to the left, created with the `less-than (<)` sign and the hyphen (-) sign (keyboard shortcut: `Alt` + - for Windows and `Option` + - for Mac).

For example, suppose we would like to store the number 1/40 for future use. We will assign this value to an object called `x`. To this, we type:

```r
x <- 1/40
```

Notice that assignment does not print a value. Instead, R stores it for later in the **object** `x`. Call object `x` now and see that it contains the **value** `0.025`:

```r
x
```

```
## [1] 0.025
```

If we look for the `Environment` tab in one of the panes of RStudio, we will see that `x` and its value have appeared.

Our object `x` can be used in place of a number in any calculation that expects a number:

```r
log(x)
```

```
## [1] -3.688879
```

It is important the space before and after comparison operators and assignments. For example, suppose we want to code the expression `x smaller than -1/50` :

- With spaces

```r
x < -1/50  # with spaces
```

```
## [1] FALSE
```

The result is the logical `FALSE` because the value x (equals to 1/40) is higher than -1/50.

- Without spaces

```r
x<-1/50 # without spaces
x
```

```
## [1] 0.02
```

If we omit the spaces we end up with the assignment operator and we have `x <- 1/50` wich equals to 0.02.

6.2.2 Other types of assignment

Note It is also possible to use the **“="** or **“->"** rightward operator for assignment (but these are much less common among R users).

For example:

```r
x = 1/40
x
```

```
## [1] 0.025
```

or

```r
1/40 -> x
x
```

```
## [1] 0.025
```

It is a good idea to **be consistent** with the operator we use.

Note Surrounding the assignment with parentheses results in both assignment and “print to screen" to happen.

For example:

```r
(x <- 1/40)
```

```
## [1] 0.025
```

6.3 Reassigning an object

Notice also that objects can be **reassigned**. For example, recall the `x` object:

```r
x
```

```
## [1] 0.025
```

then type the following:

```r
x <- 100
x
```

```
## [1] 100
```

`x` used to contain the value 0.025 and now it has the value 100.

Moreover, assignment values can contain the object being assigned to:

```r
x <- x + 1 #notice how RStudio updates its description of x on the Envirinment tab
x
```

```
## [1] 101
```

The right hand side of the assignment can be any valid R expression and it is fully evaluated **before** the assignment takes place.

6.4 Legal object names

Object names must start with a letter and **can** contain letters, numbers, underscores (`_`) and periods (.). They **cannot** start with a number or underscore, nor contain spaces at all. Moreover, they can not contain "reserved" words.

Different people use different conventions for long object names, these include:

- periods.between.words
- underscores_between_words
- camelCaseToSeparateWords

What we use is up to us, but we must **be consistent**. We might ask help:

```
??make.names
??clean_names
```

Note R is **case-sensitive** – it treats capital letters differently from lower-case letters.

```
Y <- 50
Y
```

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

but...

```
y
```

Error: object 'y' not found

6.5 We are not limited to store numbers in objects

In objects we can also store other data types. For example, we can store characters:

```r
sentence <- "the cat sat on the mat"
```

Note that we need to put strings of characters inside **quotes**.

But the type of data that is stored in an object affects what we can do with it:

```r
sentence + 1
```

```
Error in sentence + 1: non-numeric argument to binary operator
```

7 Activities

Activity 1

(a) Write R code to assign the value `20` to the name `num_1`.

(b) Which of the following is a valid object name in R?

- 2.True
- else
- I_am_not_a_valid_name
- I_am_a_Pretty#_name

Activity 2

(a) Think about what is the value of 1^{NA} in R. Try to run it in R. Does it agree with your thoughts?

(b) Think about what is the value of 0^{NA} in R. Try to run it in R. Does it agree with your thoughts?