Learning R for Clinical Trial Data

Maya Gans

2020-02-03

Contents

Chapter 1

Preface

This book is geared towards people who are bravely taking the step towards learning R but have yet to even download R on their local machines.

1.1 The book will walk you through:

- Getting R Set up on your computer
- Understanding the RStudio IDE
- Using the RStudio IDE
- A gentle introduction to R
- Importing and viewing data
- Manipulating data
- Visualizing data
- Use the generated tables and plots to create a report

Chapter 2

Getting Set Up with R

2.1 Setup instructions

R and RStudio are separate downloads and installations. R is the underlying statistical computing environment, but using R alone is no fun. RStudio is a graphical integrated development environment (IDE) that makes using R much easier and more interactive. You need to install R before you install RStudio. After installing both programs, you will need to install the `tidyverse` and `haven` packages from within RStudio. Follow the instructions below for your operating system, and then follow the instructions to install `tidyverse` and `haven`.

2.1.1 Windows

If you already have R and RStudio installed Open RStudio, and click on "Help" > "Check for updates". If a new version is available, quit RStudio, and download the latest version for RStudio. To check which version of R you are using, start RStudio and the first thing that appears in the console indicates the version of R you are running. Alternatively, you can type sessionInfo(), which will also display which version of R you are running. Go on the CRAN website and check whether a more recent version is available. If so, please download and install it. You can check here for more information on how to remove old versions from your system if you wish to do so.

If you don't have R and RStudio installed:

- Download R from the CRAN website.
- Run the .exe file that was just downloaded
- Go to the RStudio download page
- Under Installers select RStudio x.yy.zzz - Windows Vista/7/8/10 (where x, y, and z represent version numbers)
- Double click the file to install it

Once it's installed, open RStudio to make sure it works and you don't get any error messages.

2.1.2 macOS

If you already have R and RStudio installed Open RStudio, and click on "Help" > "Check for updates". If a new version is available, quit RStudio, and download the latest version for RStudio.

To check the version of R you are using, start RStudio and the first thing that appears on the terminal indicates the version of R you are running. Alternatively, you can type sessionInfo(), which will also display which version of R you are running. Go on the CRAN website and check whether a more recent version is available. If so, please download and install it.

If you don't have R and RStudio installed:

- Download R from the CRAN website.
- Select the .pkg file for the latest R version
- Double click on the downloaded file to install R
- It is also a good idea to install XQuartz (needed by some packages)
- Go to the RStudio download page
- Under Installers select RStudio x.yy.zzz - Mac OS X 10.6+ (64-bit) (where x, y, and z represent version numbers)
- Double click the file to install RStudio

Once it's installed, open RStudio to make sure it works and you don't get any error messages.

2.2 Linux

Follow the instructions for your distribution from CRAN, they provide information to get the most recent version of R for common distributions. For most distributions, you could use your package manager (e.g., for Debian/Ubuntu run sudo apt-get install r-base, and for Fedora sudo yum install R), but we don't recommend this approach as the versions provided by this are usually out of date. In any case, make sure you have at least R 3.3.1.

- Go to the RStudio download page
- Under Installers select the version that matches your distribution, and install it with your preferred method (e.g., with Debian/Ubuntu sudo dpkg -i rstudio-x.yy.zzz-amd64.deb at the terminal).
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

2.3 For everyone

After installing R and RStudio, you need to install the `tidyverse` and `haven` packages.

After starting RStudio, at the console type:

```
install.packages(c("tidyverse", "haven"))
```

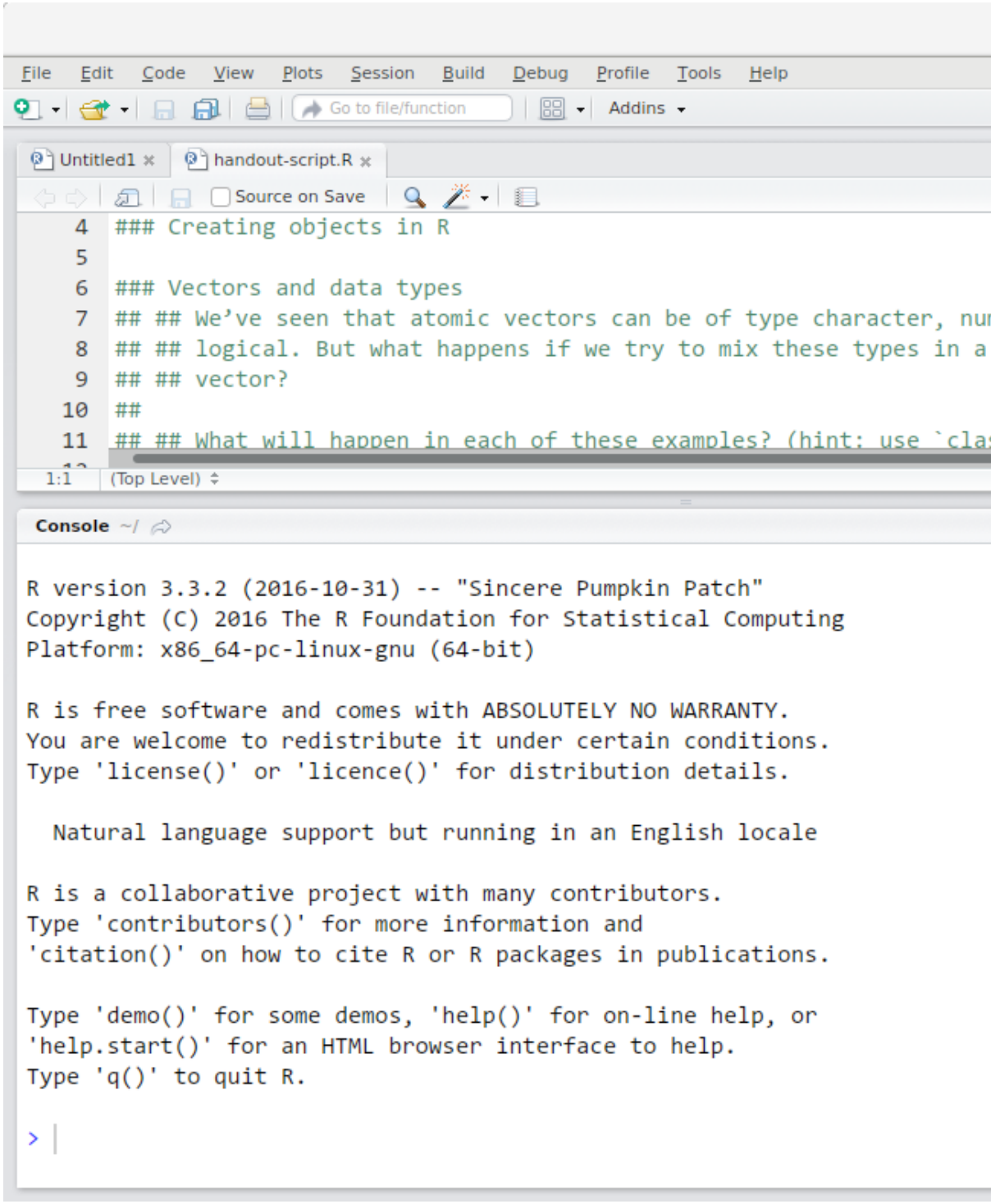
You can also do this by going to Tools -> Install Packages and typing the names of the packages separated by a comma.

Chapter 3

RStudio IDE

The RStudio IDE open-source product is free under the Affero General Public License (AGPL) v3. The RStudio IDE is also available with a commercial license and priority email support from RStudio, Inc.

We will use RStudio IDE to write code, navigate the files on our computer, inspect the variables we are going to create, and visualize the plots we will generate. RStudio can also be used for other things (e.g., version control, developing packages, writing Shiny apps) that will not be covered in this book.


File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Untitled1
handout-script.R
Source on Save
4 ### Creating objects in R
5
6 ### Vectors and data types
7 ## ## We've seen that atomic vectors can be of type character, nu
8 ## ## logical. But what happens if we try to mix these types in a
9 ## ## vector?
10 ##
11 ## ## What will happen in each of these examples? (hint: use `cla
1:1
(Top Level)
Console ~/
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 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.
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.
>

RStudio is divided into 4 “Panes”:

1) The Source for your scripts and documents (top-left, in the default layout),
2) Your Environment/History (top-right)
3) Your Files/Plots/Packages/Help/Viewer (bottom-right)
4) The R Console (bottom-left).

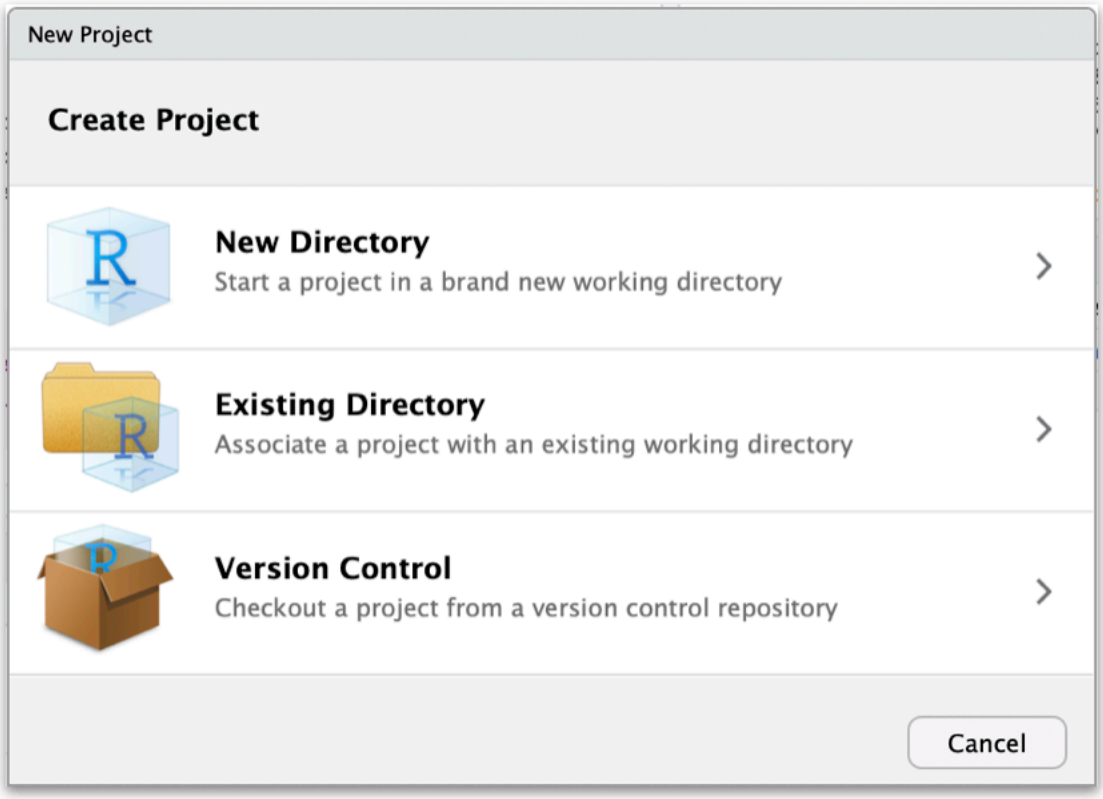
The placement of these panes and their content can be customized (see menu, Tools -> Global Options -> Pane Layout).

One of the advantages of using RStudio is that all the information you need to write code is available in a single window. Additionally, with many short-cuts, autocompletion, and highlighting for the major file types you use while developing in R, RStudio will make typing easier and less error-prone.

3.1 Getting set up

It is good practice to keep a set of related data, analyses, and text self-contained in a single folder, called the working directory. All of the scripts within this folder can then use relative paths to files that indicate where inside the project a file is located (as opposed to absolute paths, which point to where a file is on a specific computer). Working this way makes it a lot easier to move your project around on your computer and share it with others without worrying about whether or not the underlying scripts will still work.

RStudio provides a helpful set of tools to do this through its “Projects” interface, which not only creates a working directory for you, but also remembers its location (allowing you to quickly navigate to it) and optionally preserves custom settings and open files to make it easier to resume work after a break. Go through the steps for creating an “R Project” for this tutorial below.



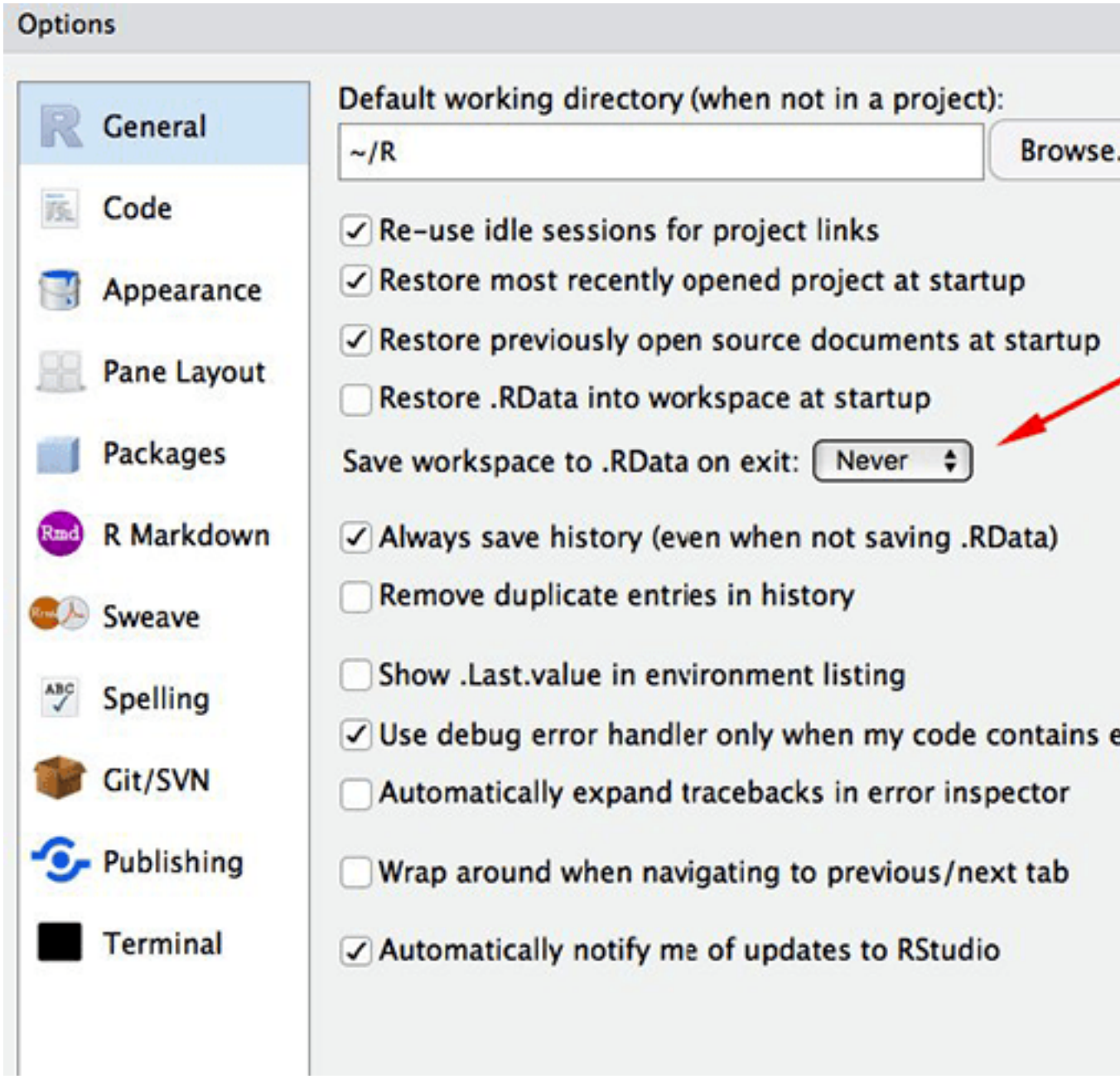
3.2 Start RStudio.

- Under the File menu, click on New Project.
- Choose New Directory, then New Project.
- Enter a name for this new folder (or "directory"), and choose a convenient location for it.
- Click on Create Project.

3.3 Optional Preferences

RStudio's default preferences generally work well, but saving a workspace to .RData can be cumbersome, especially if you are working with larger datasets. To turn that off, go to Tools –> 'Global Options' and select the 'Never' option for 'Save workspace to .RData' on exit.' This step is optional, but if you love

something it's sometimes best to let it go.



3.4 Organizing your working directory

Using a consistent folder structure across your projects will help keep things organized, and will also make it easy to find/file things in the future. This can be especially helpful when you have multiple projects. In general, you may create directories (folders) for scripts, data, and documents.

- data_raw/
- data/

Use these folders to store raw data and intermediate datasets you may create for the need of a particular analysis. For the sake of transparency and provenance, you should always keep a copy of your raw data accessible and do as much of your data cleanup and preprocessing programmatically (i.e., with scripts, rather than manually) as possible.

- report.Rmd

We will be using an RMarkdown file to create our report. This allows for inline coding with plot and table outputs. We are going to keep the report in the root of our working directory because we are only going to use one file and it will make things easier. Outside of this demonstraition you'd most likely create a folder of reports and title them accordingly.

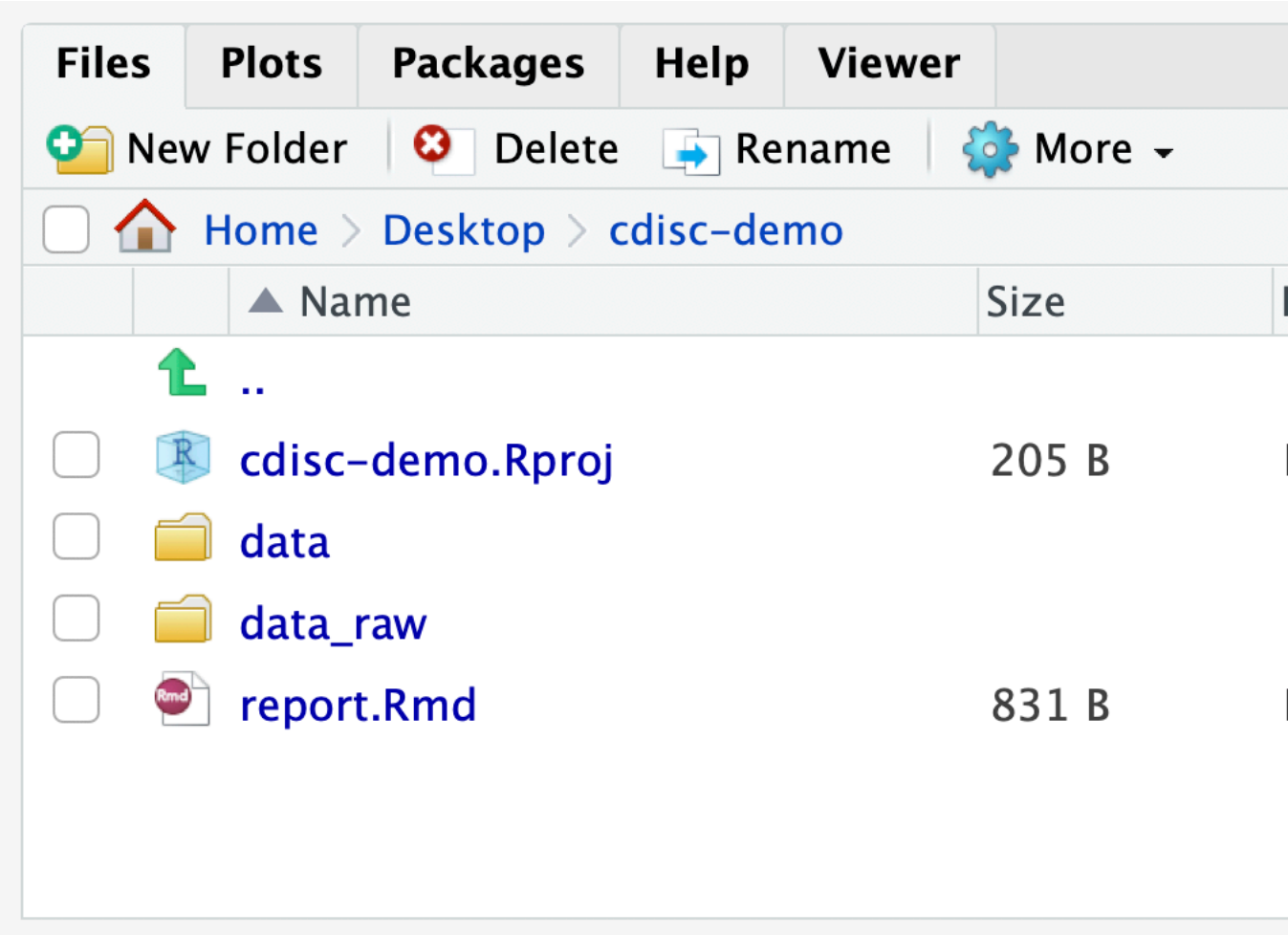
- Additional (sub)directories depending on your project needs (like scripts and functions)

We will need a `data_raw/` folder for our demo project to store our raw `sas7bdat` files, and we will use `data/` for when we learn how to export data as CSV files, and a `report.Rmd` file for our generated report containing figures and tables.

Under the Files tab on the right of the screen, click on New Folder and create a folder named data_raw within your newly created working directory (e.g., ~/cdisc-demo/). (Alternatively, type dir.create("data_raw") at your R console.)

Repeat these operations to create a data folder. Under the Files tab you can click New File then RMarkdown and create the `report.Rmd`.

Your working directory should now look like this:


Files
Plots
Packages
Help
Viewer
New Folder
Delete
Rename
More
Home
Desktop
cdisc-demo
Name
Size
..
cdisc-demo.Rproj
205 B
data
data_raw
report.Rmd
831 B

Chapter 4

Interacting with R

The basis of programming is that we write down instructions for the computer to follow, and then we tell the computer to follow those instructions. We write, or code, instructions in R because it is a common language that both the computer and we can understand. We call the instructions commands and we tell the computer to follow the instructions by executing (also called running) those commands.

There are two main ways of interacting with R: by using the console or by using script files (plain text files that contain your code). The console pane (in RStudio, the bottom left panel) is the place where commands written in the R language can be typed and executed immediately by the computer. It is also where the results will be shown for commands that have been executed. You can type commands directly into the console and press Enter to execute those commands, but they will be forgotten when you close the session.

Because we want our code and workflow to be reproducible, it is better to type the commands we want in the script editor, and save the script. This way, there is a complete record of what we did, and anyone (including our future selves!) can easily replicate the results on their computer.

RStudio allows you to execute commands directly from the script editor by using the `Ctrl + Enter` shortcut (on Macs, `Cmd + Return` will work, too). The command on the current line in the script (indicated by the cursor) or all of the commands in the currently selected text will be sent to the console and executed when you press Ctrl + Enter. You can find other keyboard shortcuts in this RStudio cheatsheet about the RStudio IDE.

At some point in your analysis you may want to check the content of a variable or the structure of an object, without necessarily keeping a record of it in your script. You can type these commands and execute them directly in the console. RStudio provides the Ctrl + 1 and Ctrl + 2 shortcuts allow you to jump between

the script and the console panes.

If R is ready to accept commands, the R console shows a > prompt. If it receives a command (by typing, copy-pasting or sent from the script editor using Ctrl + Enter), R will try to execute it, and when ready, will show the results and come back with a new > prompt to wait for new commands.

If R is still waiting for you to enter more data because it isn't complete yet, the console will show a + prompt. It means that you haven't finished entering a complete command. This is because you have not 'closed' a parenthesis or quotation, i.e. you don't have the same number of left-parentheses as right-parentheses, or the same number of opening and closing quotation marks. When this happens, and you thought you finished typing your command, click inside the console window and press Esc; this will cancel the incomplete command and return you to the > prompt.

4.1 Seeking help

Use the built-in RStudio help interface to search for more information on R functions RStudio help interface. This panel by default can be found at the lower right hand panel of RStudio. As seen in the screenshot, by typing the word "Mean", RStudio tries to also give a number of suggestions that you might be interested in. The description is then shown in the display window.

If you need help with a specific function, let's say `barplot()`, you can type: `?barplot`

If you just need to remind yourself of the names of the arguments, you can use: `args(lm)`

If you are looking for a function to do a particular task, you can use the help.search() function, which is called by the double question mark `??`. However, this only looks through the installed packages for help pages with a match to your search request

If you can't find what you are looking for, you can use the rdocumentation.org website that searches through the help files across all packages available.

Finally, a generic Google or internet search `R <task>` will often either send you to the appropriate package documentation or a helpful forum where someone else has already asked your question.

Chapter 5

Introduction to R

5.1 Learning Objectives

- Define the following terms as they relate to R: object, assign, call, function, arguments, options.
- Assign values to objects in R.
- Learn how to name objects
- Use comments to inform script.
- Solve simple arithmetic operations in R.
- Call functions and use arguments to change their default options.
- Inspect the content of vectors and manipulate their content.
- Subset and extract values from vectors.
- Analyze vectors with missing data.

5.2 Creating objects in R

You can get output from R simply by typing math in the console:

```
3 + 5
```

```
## [1] 8
```

However, to do useful and interesting things, we need to assign values to objects. To create an object, we need to give it a name followed by the assignment operator <-, and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of `x` is `3`. The arrow can be read as 3 **goes into** `x`. For historical reasons, you can also use = for assignments, but not

in every context. Because of the slight differences in syntax, it is good practice to always use `<-` for assignments.

In RStudio, typing `Alt + -` (push `Alt` at the same time as the `-` key) will write `<-` in a single keystroke in a PC, while typing `Option + -` (push `Option` at the same time as the `-` key) does the same in a Mac.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long. They **cannot** start with a number (`2x` is not valid, but `x2` is). R is **case sensitive** (e.g., `weight_kg` is different from `Weight_kg`).

There are some names that cannot be used because they are the names of fundamental functions in R (e.g., `if`, `else`, `for`). In general, even if it's allowed, it's best to not use other function names (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`).

If in doubt, check the help to see if the name is already in use. It's also best to avoid dots (.) within an object name as in my.dataset. There are many functions in R with dots in their names for historical reasons, but because dots have a special meaning in R (for methods) and other programming languages, it's best to avoid them.

It is also recommended to use nouns for object names, and verbs for function names. It's important to be consistent in the styling of your code (where you put spaces, how you name objects, etc.). Using a consistent coding style makes your code clearer to read for your future self and your collaborators. In R, three popular style guides are Google's, Jean Fan's and the tidyverse's. You can install the `lintr` package to automatically check for issues in the styling of your code.

5.2.1 Objects vs. variables

What are known as objects in R are known as variables in many other programming languages. Depending on the context, object and variable can have drastically different meanings. However, in this lesson, the two words are used synonymously.

When assigning a value to an object, R does not print anything. You can force R to print the value by using parentheses or by typing the object name:

```
weight_kg <- 55  # doesn't print anything
```

```
(weight_kg <- 55)  # but putting parenthesis around the call prints the value of `weigh
```

```
## [1] 55
```

```
weight_kg # and so does typing the name of the object
```

```
## [1] 55
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (weight in pounds is 2.2 times the weight in kg):

```
2.2 * weight_kg
```

```
## [1] 121
```

We can also change an object's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

```
## [1] 126.5
```

This means that assigning a value to one object does not change the values of other objects For example, let's store the animal's weight in pounds in a new object, weight_lb:

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

5.2.2 Comments

The comment character in R is #, anything to the right of a # in a script will be ignored by R. It is useful to leave notes and explanations in your scripts. RStudio makes it easy to comment or uncomment a paragraph: after selecting the lines you want to comment, press at the same time on your keyboard `Ctrl + Shift + C`. If you only want to comment out one line, you can put the cursor at any location of that line (i.e. no need to select the whole line), then press `Ctrl + Shift + C`.

5.2.3 Functions and their arguments

Functions are "canned scripts" that automate more complicated sets of commands including operations assignments, etc. Many functions are predefined, or can be made available by importing R packages (more on that later). A function usually takes one or more inputs called arguments. Functions often (but not always) return a value. A typical example would be the function sqrt(). The input (the argument) must be a number, and the return value (in fact, the output) is the square root of that number. Executing a function ('running it') is called calling the function. An example of a function call is:

```
b <- sqrt(a)
```

Here, the value of `a` is given to the `sqrt()` function, the `sqrt()` function calculates the square root, and returns the value which is then assigned to the object `b`. This function is very simple, because it takes just one argument.

The return 'value' of a function need not be numerical (like that of `sqrt()`), and it also does not need to be a single item: it can be a set of things, or even a dataset. We'll see that when we read data files into R.

Arguments can be anything, not only numbers or filenames, but also other objects. Exactly what each argument means differs per function, and must be looked up in the documentation. Some functions take arguments which may either be specified by the user, or, if left out, take on a default value: these are called options. Options are typically used to alter the way the function operates, such as whether it ignores 'bad values', or what symbol to use in a plot. However, if you want something specific, you can specify a value of your choice which will be used instead of the default.

Let's try a function that can take multiple arguments: `round()`.

```
round(3.14159)
```

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

Here, we've called `round()` with just one argument, `3.14159`, and it has returned the value `3`. That's because the default is to round to the nearest whole number. If we want more digits we can see how to do that by getting information about the round function. We can use `args(round)` to find what arguments it takes, or look at the help for this function using `?round`.

```
args(round)
```

```
## function (x, digits = 0)
## NULL
```

```
?round
```

We see that if we want a different number of digits, we can type `digits = 2` or however many we want.

```
round(3.14159, digits = 2)
```

```
## [1] 3.14
```

If you provide the arguments in the exact same order as they are defined you don't have to name them:

```
round(3.14159, 2)
```

```
## [1] 3.14
```

And if you do name the arguments, you can switch their order:

```
round(digits = 2, x = 3.14159)
```

```
## [1] 3.14
```

It's good practice to put the non-optional arguments (like the number you're rounding) first in your function call, and to then specify the names of all optional arguments. If you don't, someone reading your code might have to look up the definition of a function with unfamiliar arguments to understand what you're doing.

5.3 Vectors and data types

A vector is the most common and basic data type in R, and is pretty much the workhorse of R. A vector is composed by a series of values, which can be either numbers or characters. We can assign a series of values to a vector using the `c()` function. For example we can create a vector of animal weights and assign it to a new object `weight_g`:

```
weight_g <- c(50, 60, 65, 82)
weight_g
```

```
## [1] 50 60 65 82
```

A vector can also contain characters:

```
animals <- c("mouse", "rat", "dog")
animals
```

```
## [1] "mouse" "rat"   "dog"
```

The quotes around "mouse", "rat", etc. are essential here. Without the quotes R will assume objects have been created called `mouse`, `rat` and `dog`. As these objects don't exist in R's memory, there will be an error message.

There are many functions that allow you to inspect the content of a vector. `length()` tells you how many elements are in a particular vector:

```
length(weight_g)
```

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

```
length(animals)
```

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

An important feature of a vector, is that all of the elements are the same type of data. The function `class()` indicates the class (the type of element) of an object:

```
class(weight_g)
```

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

```
class(animals)
```

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

The function `str()` provides an overview of the structure of an object and its elements. It is a useful function when working with large and complex objects:

```
str(weight_g)
```

```
##  num [1:4] 50 60 65 82
```

```
str(animals)
```

```
##  chr [1:3] "mouse" "rat" "dog"
```

You can use the `c()` function to add other elements to your vector:

```
weight_g <- c(weight_g, 90) # add to the end of the vector
weight_g
```

```
## [1] 50 60 65 82 90
```

```
weight_g <- c(30, weight_g) # add to the beginning of the vector
weight_g
```

```
## [1] 30 50 60 65 82 90
```

In the first line, we take the original vector `weight_g`, add the value 90 to the end of it, and save the result back into `weight_g`. Then we add the value 30 to the beginning, again saving the result back into `weight_g`.

We can do this over and over again to grow a vector, or assemble a dataset. As we program, this may be useful to add results that we are collecting or calculating.

An **atomic vector** is the simplest R **data type** and is a linear vector of a single type. Above, we saw 2 of the 6 main atomic vector types that R uses: `"character"` and `"numeric"` (or `"double"`). These are the basic building blocks that all R objects are built from. The other 4 atomic vector types are:

- `"logical"` for `TRUE` and `FALSE` (the boolean data type)
- `"integer"` for integer numbers (e.g., `2L`, the `L` indicates to R that it's an integer)
- `"complex"` to represent complex numbers with real and imaginary parts (e.g., `1 + 4i`) and that's all we're going to say about them
- `"raw"` for bitstreams that we won't discuss further

You can check the type of your vector using the `typeof()` function and inputting your vector as the argument.

Vectors are one of the many data structures that R uses. Other important ones are lists (`list`), matrices (`matrix`), data frames (`data.frame`), factors (`factor`) and arrays (`array`).

5.4 Subsetting vectors

If we want to extract one or several values from a vector, we must provide one or several indices in square brackets. For instance:

```
animals <- c("mouse", "rat", "dog", "cat")
```

```
animals[2]
```

```
## [1] "rat"
```

```
animals[c(3, 2)]
```

```
## [1] "dog" "rat"
```

We can also repeat the indices to create an object with more elements than the original one:

```
more_animals <- animals[c(1, 2, 3, 2, 1, 4)]
more_animals
```

```
## [1] "mouse" "rat"   "dog"   "rat"   "mouse" "cat"
```

R indices start at 1. Programming languages like Fortran, MATLAB, Julia, and R start counting at 1, because that's what human beings typically do. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do.

5.4.1 Conditional subsetting

Another common way of subsetting is by using a logical vector. `TRUE` will select the element with the same index, while `FALSE` will not:

```
weight_g <- c(21, 34, 39, 54, 55)
weight_g[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
```

```
## [1] 21 39 54
```

Typically, these logical vectors are not typed by hand, but are the output of other functions or logical tests. For instance, if you wanted to select only the values above 50:

```
weight_g > 50   # will return logicals with TRUE for the indices that meet the condition
```

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

So we can use this to select only the values above 50

```
weight_g[weight_g > 50]
```

```
## [1] 54 55
```

You can combine multiple tests using & (both conditions are true, AND) or | (at least one of the conditions is true, OR):

```
weight_g[weight_g < 30 | weight_g > 50]
```

```
## [1] 21 54 55
```

```
weight_g[weight_g >= 30 & weight_g == 21]
```

```
## numeric(0)
```

Here, < stands for "less than", > for "greater than", >= for "greater than or equal to", and == for "equal to". The double equal sign == is a test for numerical equality between the left and right hand sides, and should not be confused with the single = sign, which performs variable assignment (similar to <-).

A common task is to search for certain strings in a vector. One could use the "or" operator | to test for equality to multiple values, but this can quickly become tedious. The function %in% allows you to test if any of the elements of a search vector are found:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[animals == "cat" | animals == "rat"] # returns both rat and cat
```

```
## [1] "rat" "cat"
```

```
animals %in% c("rat", "cat", "dog", "duck", "goat")
```

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

```
animals[animals %in% c("rat", "cat", "dog", "duck", "goat")]
```

```
## [1] "rat" "dog" "cat"
```

5.5 Missing data

As R was designed to analyze datasets, it includes the concept of missing data (which is uncommon in other programming languages). Missing data are represented in vectors as NA.

When doing operations on numbers, most functions will return NA if the data you are working with include missing values. This feature makes it harder to overlook the cases where you are dealing with missing data. You can add the argument na.rm = TRUE to calculate the result while ignoring the missing values.

```
heights <- c(2, 4, 4, NA, 6)
```

```
mean(heights)
```

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

```
max(heights)
```

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

```
mean(heights, na.rm = TRUE)
```

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

```
max(heights, na.rm = TRUE)
```

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

If your data include missing values, you may want to become familiar with the functions `is.na()`, `na.omit()`, and `complete.cases()`. See below for examples.

```
## Extract those elements which are not missing values.
heights[!is.na(heights)]
```

```
## [1] 2 4 4 6
```

```
## Returns the object with incomplete cases removed. The returned object is an atomic vector of t
na.omit(heights)
```

```
## [1] 2 4 4 6
## attr(,"na.action")
## [1] 4
## attr(,"class")
## [1] "omit"
```

```
## Extract those elements which are complete cases. The returned object is an atomic vector of ty
heights[complete.cases(heights)]
```

```
## [1] 2 4 4 6
```

Now that we have learned how to write scripts, and the basics of R's data structures, we are ready to start working with an ADSL dataset!

Chapter 6

Starting With Data

Chapter 7

Manipulating Data

Chapter 8

Visualizing Data

Chapter 9

Creating a Report