R & RStudio: the basics

Martial Foegel

2024-10-14

Laboratoire de Linguistique Formelle

R

R is an open source language and environment (*i.e.* a fully planned and coherent system) that allows for data manipulation, statistical computing and data visualization. For computing intensive tasks, it uses under the hood, C, C++ and Fortran.

Its uses is augmented by the large number of packages, containing code, data and documentation in a standardized format centralized around a software repository such as the such as the CRAN (the Comprehensive R Archive Network).

"R Core Team (2020). — European Environment Agency" (n.d.)

RStudio

Is an Integrated Development Environment (IDE) of R. It exist under the form of a Desktop application or as a server. It is a product of Posit PBC, an open source data software company.

RStudio Team (2020)

A word about Mac shortcuts

During this presentation I will show a few shortcuts. I am currently using a PC with an AZERTY keyboard so my shortcuts will reflect that. Those shortcuts shouldn't change for QWERTY/QWERTZ keyboards. For Apple keyboards on the other hand they may not match but usually whenever you see **Ctrl** you can replace it with **Cmd** on your keyboard and whenever you see **Alt** you can replace it with **Option**.

Creating a R Project

Before anything else, create a folder that will serve as a place to work from for today's practical. You can call it something like *"R_RStudio_the_basics"*.

> ⚠ Warning
>
> Avoid the uses of blank space or special characters like "&" when creating your folder (and any subsequent folders and files) as those can sometimes cause issues.

Now let's open up RStudio. We will immediately go to ***Files*** > ***New Project*** and we are going to create a project from an existing directory. Now just find the folder we've created beforehand and click on ***Create Project***.

You should end up with something looking like this :

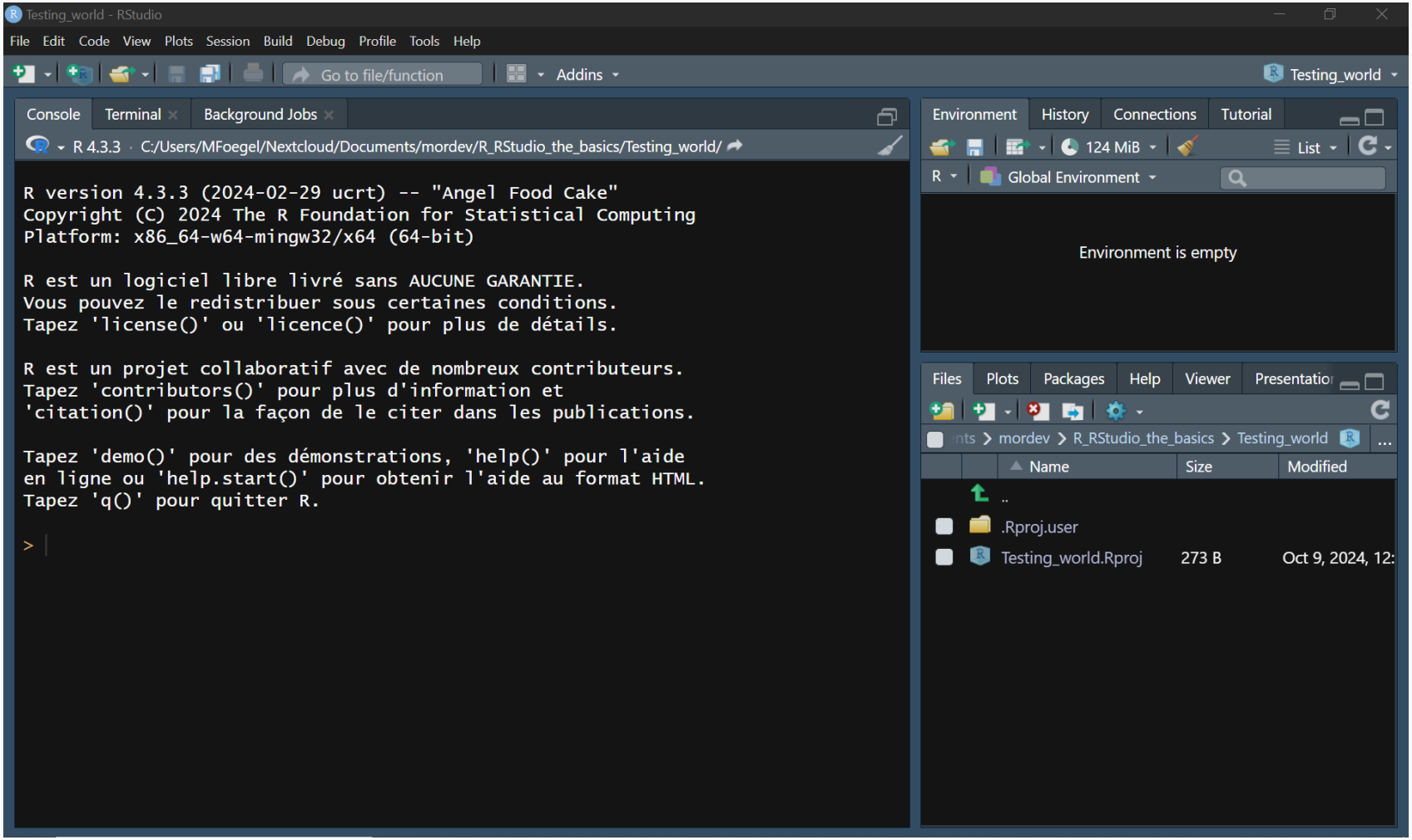


Figure 1: Example of a RStudio window

The RStudio IDE works with panes which you can rearrange. Right now you should have 3 of them with different tabs with the most important ones being:

- The **Console** and the **Terminal** on the left
- The **Environment** in the top right (empty for now)
- The **Files**, **Plots**, **Packages** and **Help** on the bottom right

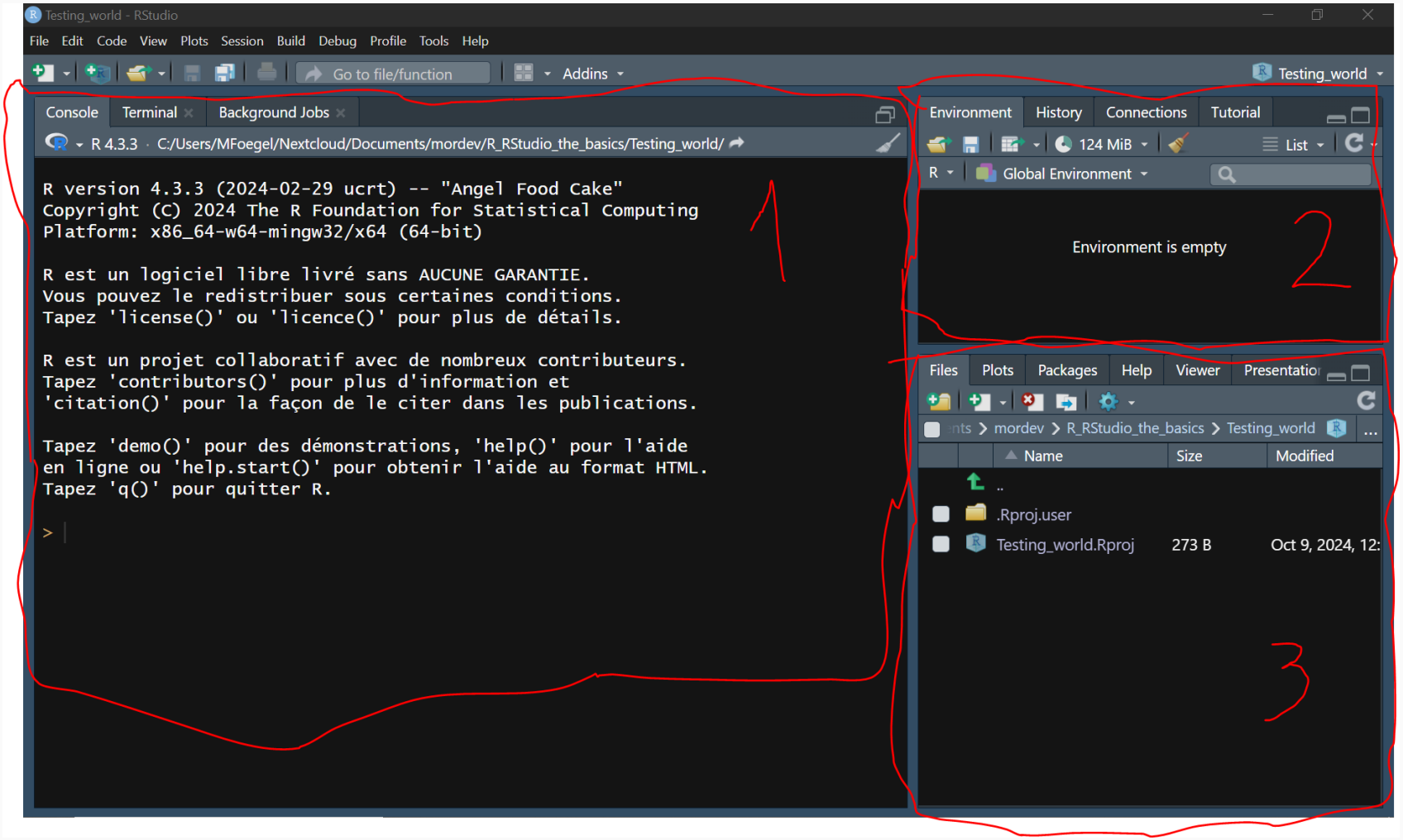


Figure 2: Example of panes

Another thing to note is that the project directory is set as the working directory. As you can see underlined below, we are currently in the folder that we've created at the beginning, and we can see the files of that folder in the bottom right panel in the **Files** tab. If you want you can check the working directory by tapping `getwd()` in the console.

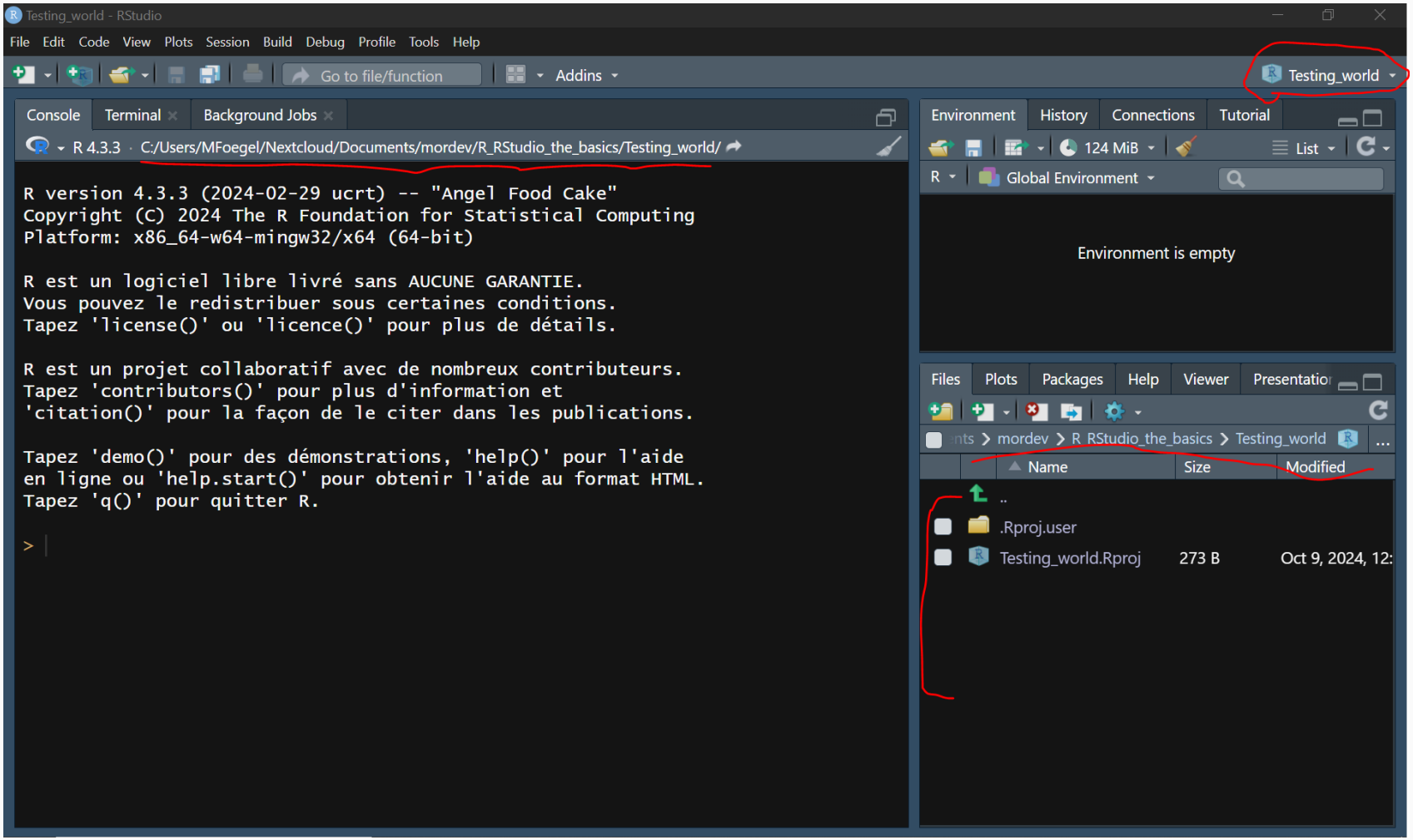


Figure 3: Information on the working directory

Making a R Project

Creating a R Project created a project file (*.RProj* extension) into the folder. This can be used as a shortcut for opening your project and starting a new R session. The project will create a few files as you go: a *.RData* file that will contain all R objects saved into your environment when you close your project and restore them on reopening, a *.Rhistory* file with the project history, and a few other. Also as we've seen, the current working directory will be set to the project directory.

! Important

Create a different R Project for each "real life" project you have (experiment, article, etc). Either directly into your specific project folder or into a dedicated sub-folder.

Making a R Script

We can write code directly into the console. Let's do a classic and type `print("Hello World")` into the console.

If we want to write something into a script that we can save, we have to create the corresponding file. Go to ***File*** $>$ ***New File*** $>$ ***R Script***. You should have a new pane that arrived on the top left.

Operators

There are a lot of operators in R. Let's try a few ones out in our new script !

Arithmetic operators

Among Arithmetic operators we have:

+, −, / and *	for addition, subtraction, division and multiplication
^	for the exponent
%%	for the modulus (the remainder of a division)
%*%	for matrix multiplication

Arithmetic operators

Among Arithmetic operators we have:

+, −, / and *	for addition, subtraction, division and multiplication
^	for the exponent
%%	for the modulus (the remainder of a division)
%*%	for matrix multiplication

Type out a simple 1+1in your script and do either **Ctrl** + **Enter** on that line or click on the **Run** button on the top right of the console panel just to make sure we are not in a binary number system !

Relational operators

`<`	strictly less than
`>`	strictly more than
`<=`	less than or equal to
`>=`	more than or equal to
`==`	equal to
`!=`	not equal to

Nothing riveting here, I know, unless... ?

Relational operators

`<`	strictly less than
`>`	strictly more than
`<=`	less than or equal to
`>=`	more than or equal to
`==`	equal to
`!=`	not equal to

Nothing riveting here, I know, unless... ?

Go to a new line on your script and type `1==TRUE` and run the line. What happens in this case ?

Basic object types

The most common object types in R are:

numeric	any number with or without decimals positive or negative will be of the type number
integer	a special case of the numeric type that needs to be specified with the function `as.integer()`
complex	you can create complex number using the function `complex()`
character	a data type for storing text by using `" "` or `' '` around text
logical	can only take 3 types of value : `TRUE`, `FALSE` or `NA` (for missing value)

Let's try them out. By calling the function `class()` around a numerical value, you can check that they are in fact a numeric type. Do the same with `"Hello World"` and `TRUE`.

Now let's do some weird ones. What happens if you do `class(NaN)`?

Now let's do some weird ones. What happens if you do `class(NaN)`?

It's a numeric class since `NaN` stands for "Not a Number"

Now let's do some weird ones. What happens if you do `class(NaN)`?

It's a numeric class since `NaN` stands for "Not a Number"

And what happens when you `class(as.numeric(TRUE))` ?

Now let's do some weird ones. What happens if you do `class(NaN)`?

It's a numeric class since `NaN` stands for "Not a Number"

And what happens when you `class(as.numeric(TRUE))` ?

`TRUE` is automatically translated to `1` in numeric using the function `as.numeric()` (hence the result on the previous slide). On the other side `FALSE` will be translated to `0`.

Comments

We've only written a few lines in our script but we've already encountered some stuff that may seem weird to you. If you want to remember what that was all about, I would encourage you to start commenting your code using # in front of some text (for example before our last line in the script you can write `#TRUE is automatically translated to 1`).

Ctrl + **Maj** + **C** can be used as a shortcut to comment one or multiple lines.

Comments

We've only written a few lines in our script but we've already encountered some stuff that may seem weird to you. If you want to remember what that was all about, I would encourage you to start commenting your code using # in front of some text (for example before our last line in the script you can write `#TRUE is automatically translated to 1`).

Ctrl + **Maj** + **C** can be used as a shortcut to comment one or multiple lines.

If you want to add sections to your code that can appear in your outline (at the bottom of your script or by clicking the top right button of your script panel), you can use `# your title here with 4 ----` . Adding one more #in the front will create a subsection.

Ctrl + **Maj** + **R** can be used as a shortcut to add a section

Assignment operators

For now nothing has been saved in our environment (in the top-right pane). If we want to save an object in the environment we have to assign it to a variable.

`<-`	Leftward assignment

Alt + - can be used as a shortcut for the assignment operator `<-`

Let's try it, type `a <- 42` in a new line of script. You should see it appear in your top-right panel and you can check that it is a numeric value by either using `class(a)` or `is.numeric(a)`.

Assignment operators

For now nothing has been saved in our environment (in the top-right pane). If we want to save an object in the environment we have to assign it to a variable.

`<-`	Leftward assignment

Alt + - can be used as a shortcut for the assignment operator `<-`

Let's try it, type `a <- 42` in a new line of script. You should see it appear in your top-right panel and you can check that it is a numeric value by either using `class(a)` or `is.numeric(a)`.

Technically you can use rightward assignment in R ->but you won't see many people use it. You can also use =for leftward assignment, but the use of this sign for assignment is problematic in some cases and is best avoided.

About your workspace

“Have you tried turning it off and on again ?” - the IT Crowd

About your workspace

"Have you tried turning it off and on again ?" - the IT Crowd

If you save your script now by doing ***File*** $>$ ***Save*** or **Ctrl** $+$ **S** under the name "operators_and_object_types" for example, you should see a new file appear on the **Files** pane of the bottom right panel. If you were to close the RStudio project now, it would ask you if you want to save the workspace image. If you say yes, a *.RData* file should appear in your folder (you should also now see a *.Rhistory* file). Thanks to the *.RData* file, if you were now to reopen the project, your environment should be restored, and the variable *a* should be present in the top-right panel.

About your workspace

"Have you tried turning it off and on again ?" - the IT Crowd

If you save your script now by doing ***File*** $>$ ***Save*** or **Ctrl** + **S** under the name "operators_and_object_types" for example, you should see a new file appear on the **Files** pane of the bottom right panel. If you were to close the RStudio project now, it would ask you if you want to save the workspace image. If you say yes, a *.RData* file should appear in your folder (you should also now see a *.Rhistory* file). Thanks to the *.RData* file, if you were now to reopen the project, your environment should be restored, and the variable *a* should be present in the top-right panel.

You can deactivate this feature at any point. Also, if you want to make sure your environment is saved by saving it manually, you can use the function `save.image()` which will save your current working environment. Alternatively, you can use the function `save()` to save some specific object from your environment.

Vectors

In R, you can create an empty vector using the function `vector()` or combine different element separated by `,` inside the function `c()` which combines arguments.

On two separate lines write `x <- vector(length = 2)` and `y <- c(a, 42)` and run those lines. Once again they should now appear in your environment. You can see that `a` has been replaced by its value in the vector `y`.

Vectors

In R, you can create an empty vector using the function `vector()` or combine different element separated by `,` inside the function `c()` which combines arguments.

On two separate lines write `x <- vector(length = 2)` and `y <- c(a, 42)` and run those lines. Once again they should now appear in your environment. You can see that `a` has been replaced by its value in the vector `y`.

`x` is a logical vector, but we can overwrite that. Try attributing the values `42` and `43` to the vector by using `a` twice.

Vectors

In R, you can create an empty vector using the function `vector()` or combine different element separated by `,` inside the function `c()` which combines arguments.

On two separate lines write `x <- vector(length = 2)` and `y <- c(a, 42)` and run those lines. Once again they should now appear in your environment. You can see that `a` has been replaced by its value in the vector `y`.

`x` is a logical vector, but we can overwrite that. Try attributing the values `42` and `43` to the vector by using `a` twice.

If you did `x <- c(a, a+1)`, `x` should now be a numerical vector of length 2.

Vectors

In R, you can create an empty vector using the function `vector()` or combine different element separated by `,` inside the function `c()` which combines arguments.

On two separate lines write `x <- vector(length = 2)` and `y <- c(a, 42)` and run those lines. Once again they should now appear in your environment. You can see that `a` has been replaced by its value in the vector `y`.

`x` is a logical vector, but we can overwrite that. Try attributing the values 42 and 43 to the vector by using `a` twice.

If you did `x <- c(a, a+1)`, `x` should now be a numerical vector of length 2.

To get access to n^{th} element of a vector, n being an integer, use `[n]` after said vector. Contrary to most programming language, a vector starts at 1 and not 0, *i.e.* to the get the first element of the vector `x` use `x[1]`.

> **i** If you want to create a numeric vector with increments of 1 or -1, you can use `:` between two numbers.

Logical operators

Now that we've introduced vectors, let's talk about the last main type of operators, logical ones. They are very classic:

`!`	logical NOT
`&`	element-wise logical AND
`&&`	logical AND
`|`	element-wise logical OR
`||`	logical OR

Logical operators

Now that we've introduced vectors, let's talk about the last main type of operators, logical ones. They are very classic:

`!`	logical NOT
`&`	element-wise logical AND
`&&`	logical AND
`|`	element-wise logical OR
`||`	logical OR

Try to check if both n^{th} elements of the vectors `x` and/or `y` are equal to 42.

Logical operators

Now that we've introduced vectors, let's talk about the last main type of operators, logical ones. They are very classic:

`!`	logical NOT
`&`	element-wise logical AND
`&&`	logical AND
`|`	element-wise logical OR
`||`	logical OR

Try to check if both n^{th} elements of the vectors `x` and/or `y` are equal to `42`.

```
x == 42 | y == 42
```

```
x == 42 & y == 42
```

One more miscellaneous operator

`%in%`	Find if an element belongs to a vector

Use this operator to find if `42` is inside the vector `x`.

One more miscellaneous operator

`%in%`	Find if an element belongs to a vector

Use this operator to find if `42` is inside the vector `x`.

```
42 %in% x
```

Plots

Now that we have vectors, we can actually plot them together. Try the following:

```
plot(x, y)
```

A graphic should now show up in your bottom right pane in the **Plots** tab. You can see two points on this plot which corresponds to the coordinates of the first and second elements of your vectors.

Matrices

If we want to do more than one dimensional object, we can use matrices. Here you can create a matrix using the function `cbind()` (column bind) on our two vectors. Don't forget to give it a name !

Matrices

If we want to do more than one dimensional object, we can use matrices. Here you can create a matrix using the function `cbind()` (column bind) on our two vectors. Don't forget to give it a name !

```
ex_matrix <- cbind(x, y)
```

If you now type `ex_matrix` in your script you should see your full matrix with named columns, since this one was created from named elements.

Matrices

If we want to do more than one dimensional object, we can use matrices. Here you can create a matrix using the function `cbind()` (column bind) on our two vectors. Don't forget to give it a name !

```
ex_matrix <- cbind(x, y)
```

If you now type `ex_matrix` in your script you should see your full matrix with named columns, since this one was created from named elements.

To get access to the i^{th} row and j^{th} column you can use `[i,j]` after your matrix. Can you use this to pick out the 43 out of your matrix ?

Matrices

If we want to do more than one dimensional object, we can use matrices. Here you can create a matrix using the function `cbind()` (column bind) on our two vectors. Don't forget to give it a name !

```
ex_matrix <- cbind(x, y)
```

If you now type `ex_matrix` in your script you should see your full matrix with named columns, since this one was created from named elements.

To get access to the i^{th} row and j^{th} column you can use `[i,j]` after your matrix. Can you use this to pick out the 43 out of your matrix ?

```
ex_matrix[2,1]
```

Matrices

If we want to do more than one dimensional object, we can use matrices. Here you can create a matrix using the function `cbind()` (column bind) on our two vectors. Don't forget to give it a name !

```
ex_matrix <- cbind(x, y)
```

If you now type `ex_matrix` in your script you should see your full matrix with named columns, since this one was created from named elements.

To get access to the i^{th} row and j^{th} column you can use `[i,j]` after your matrix. Can you use this to pick out the 43 out of your matrix ?

```
ex_matrix[2,1]
```

> **i** We could have created a matrix using row binding of the function `rbind()`, or with the function `matrix()` (you would need to clarify the number of rows or columns in this case).

Arrays

What if we wanted to do more than 2 dimensional object ? Then we can use arrays. The following code will create a 3-dimensional array:

```
ex_array <- array(c(ex_matrix, ex_matrix), dim = c(2,2,2))
```

Arrays

What if we wanted to do more than 2 dimensional object ? Then we can use arrays. The following code will create a 3-dimensional array:

```
ex_array <- array(c(ex_matrix, ex_matrix), dim = c(2,2,2))
```

> You can use `()` around the line of code that assign a value to a variable to show the result of the assignment without needing to use a second line, *e.g.* `(ex_array <- array(c(ex_matrix, ex_matrix), dim = c(2,2,2)))`

Dataframes

Check what happens if you try to add a new column called `Participant = c("A", "B")` to our matrix using `cbind()`.

Dataframes

Check what happens if you try to add a new column called `Participant = c("A", "B")` to our matrix using `cbind()`.

```
cbind(ex_matrix, c("A", "B"))
```

You can see that every element of the matrix is between quotation mark `" "`. And that's because matrix can only ever have one type of basic data type. You can check the type of the individual column using the function `str()` or looking at the object in your **Environment** tab in the top right panel if you assigned it to a named object.

Dataframes

Check what happens if you try to add a new column called `Participant = c("A", "B")` to our matrix using `cbind()`.

```
cbind(ex_matrix, c("A", "B"))
```

You can see that every element of the matrix is between quotation mark `" "`. And that's because matrix can only ever have one type of basic data type. You can check the type of the individual column using the function `str()` or looking at the object in your **Environment** tab in the top right panel if you assigned it to a named object.

That's where dataframes come in handy. They allow for different column to have different types. Try to add `Participant = c("A", "B")` just like before but this time using `cbind.data.frame()` and check the resulting object using `str()`. You should now see both numeric and character column types.

Dataframes

Check what happens if you try to add a new column called `Participant = c("A", "B")` to our matrix using `cbind()`.

```
cbind(ex_matrix, c("A", "B"))
```

You can see that every element of the matrix is between quotation mark `" "`. And that's because matrix can only ever have one type of basic data type. You can check the type of the individual column using the function `str()` or looking at the object in your **Environment** tab in the top right panel if you assigned it to a named object.

That's where dataframes come in handy. They allow for different column to have different types. Try to add `Participant = c("A", "B")` just like before but this time using `cbind.data.frame()` and check the resulting object using `str()`. You should now see both numeric and character column types.

```
ex_dataframe <- cbind.data.frame(ex_matrix, Participant = c("A", "B"))
```

Dataframes are the most common way to store datasets, with each row being a datapoint or observation, and each column a variable.

Factors, a special type of vector

Let us first duplicate the rows of our dataframe using `ex_dataframe_2 <- rbind.data.frame(ex_dataframe, ex_dataframe)`. If we want to check out our 3^{rd} column, we can either use `[,3]` after our dataframe or use `$` and use the variable/column name. Now try to show the participant column in your console.

Factors, a special type of vector

Let us first duplicate the rows of our dataframe using `ex_dataframe_2 <- rbind.data.frame(ex_dataframe, ex_dataframe)`. If we want to check out our 3^{rd} column, we can either use `[,3]` after our dataframe or use `$` and use the variable/column name. Now try to show the participant column in your console.

What we see is a vector of characters. However, when we want to do an analysis on our data, we may want to consider `Participant` as a categorical variable, as R wouldn't now what to do with a bunch of characters. In order to do that we will use a type of object called factors. Try to change the column type of `Participant` to a factor.

Factors, a special type of vector

Let us first duplicate the rows of our dataframe using `ex_dataframe_2 <- rbind.data.frame(ex_dataframe, ex_dataframe)`. If we want to check out our 3^{rd} column, we can either use `[,3]` after our dataframe or use `$` and use the variable/column name. Now try to show the participant column in your console.

What we see is a vector of characters. However, when we want to do an analysis on our data, we may want to consider `Participant` as a categorical variable, as R wouldn't now what to do with a bunch of characters. In order to do that we will use a type of object called factors. Try to change the column type of `Participant` to a factor.

```
ex_dataframe_2$Participant <- as.factor(ex_dataframe_2$Participant)
```

Now if you check `ex_dataframe_2$Participant`, you will see the same vector but this time without the quotation mark and with a new parameter, levels !

This is due to the fact a factor contains *n* underlying numerical values which are associated with a character label (here `A` and `B`), the levels. No element of said vectors can take a value outside the associated levels, but a level can be assigned to a factor without it being present in the data.

Lists

What if... and stay with me here... we wanted to save our matrix, our array and our dataframe, or even just a into one single object ? Well that's where the list shines. You can use the function `list()`, with all your elements inside separated by `,` to do that. Try it !

Lists

What if... and stay with me here... we wanted to save our matrix, our array and our dataframe, or even just a into one single object ? Well that's where the list shines. You can use the function `list()`, with all your elements inside separated by `,` to do that. Try it !

```
ex_list <- list(ex_matrix, ex_dataframe, ex_array, "a" = a)
```

> Use `"name_of_your_element" = your_element` to add a named element.

Lists

What if... and stay with me here... we wanted to save our matrix, our array and our dataframe, or even just a into one single object ? Well that's where the list shines. You can use the function `list()`, with all your elements inside separated by `,` to do that. Try it !

```
ex_list <- list(ex_matrix, ex_dataframe, ex_array, "a" = a)
```

> Use `"name_of_your_element" = your_element` to add a named element.

To access the n^{th} elements of your list you have to use `[[n]]` after your list.

Making a R Markdown file

Let's switch it up and use a RMarkdown file now. Go to ***File*** > ***New File*** > ***R Markdown***. Give it a title, which will be your document's title, leave the other options by default click on **OK**. You should be on a brand new file and a new tab should appear on the top left pane. It should look like the following:

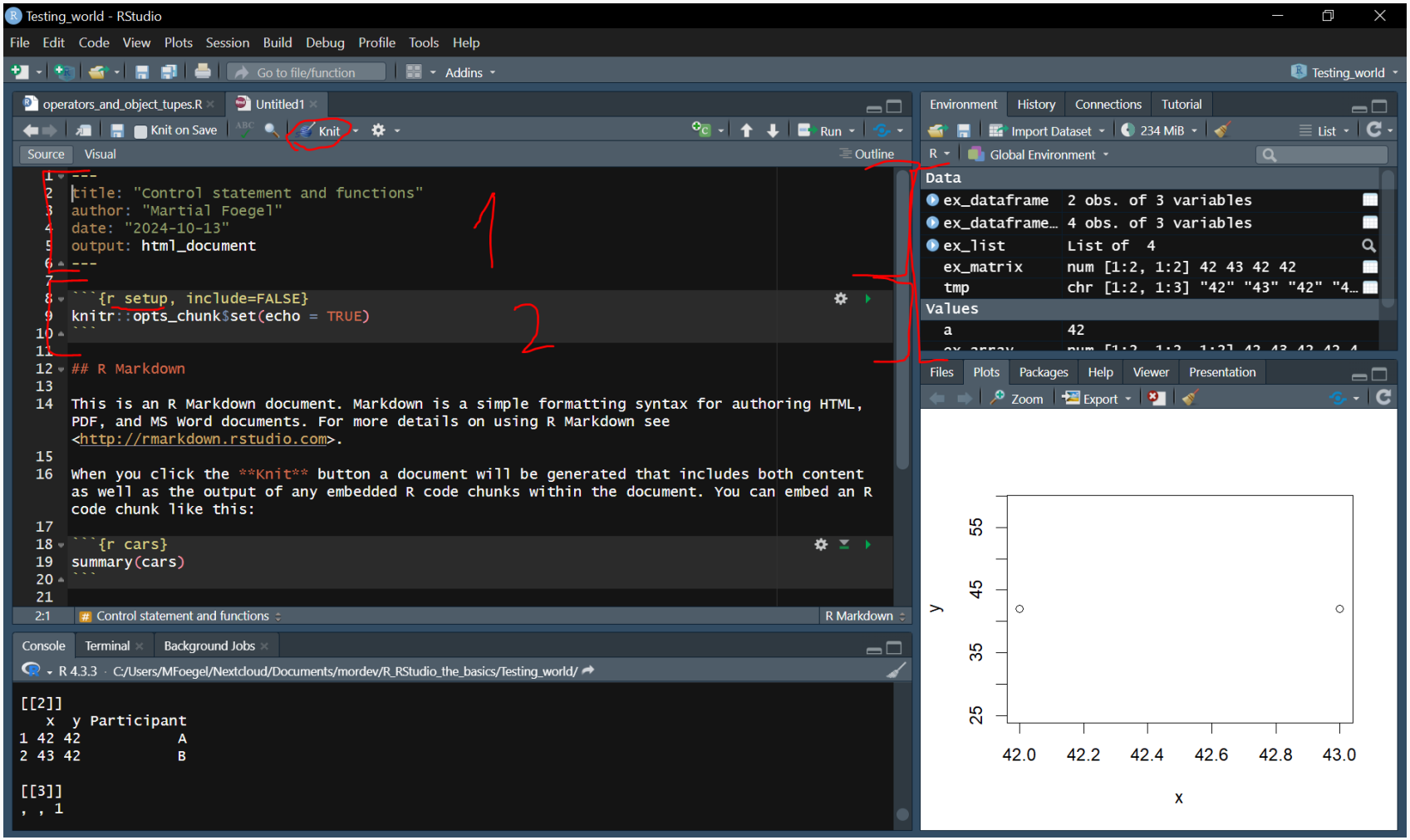


Figure 4: Example of an RMarkdown file in RStudio

You should see that we've kept the same environment as before (since we're inside the same project). You should also see in the section 1 the YAML header which set some overall settings when knitting the document. What is knitting ? Well click on the button **Knit** and you should see your document rendered as a html output, just as specified by the YAML header.

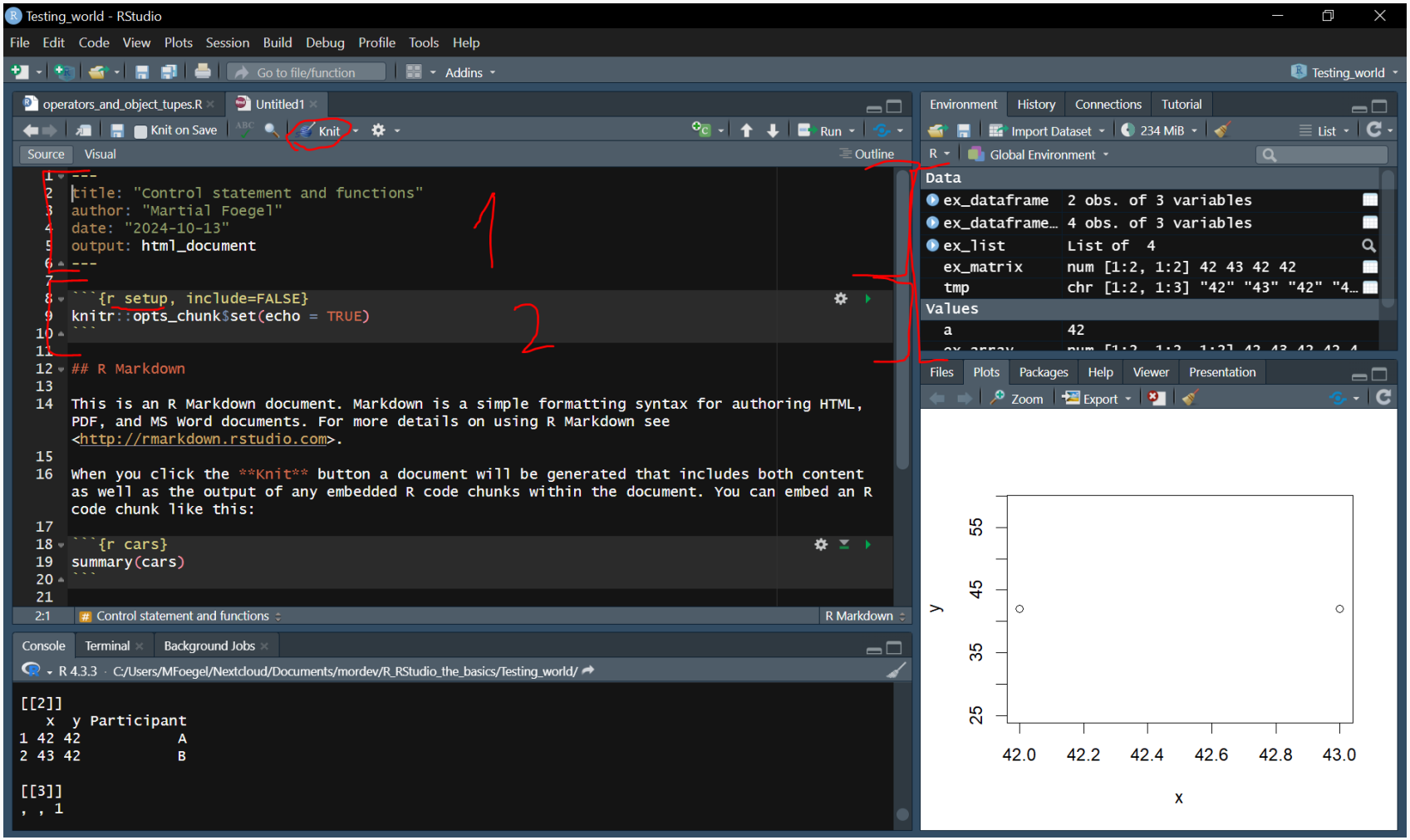
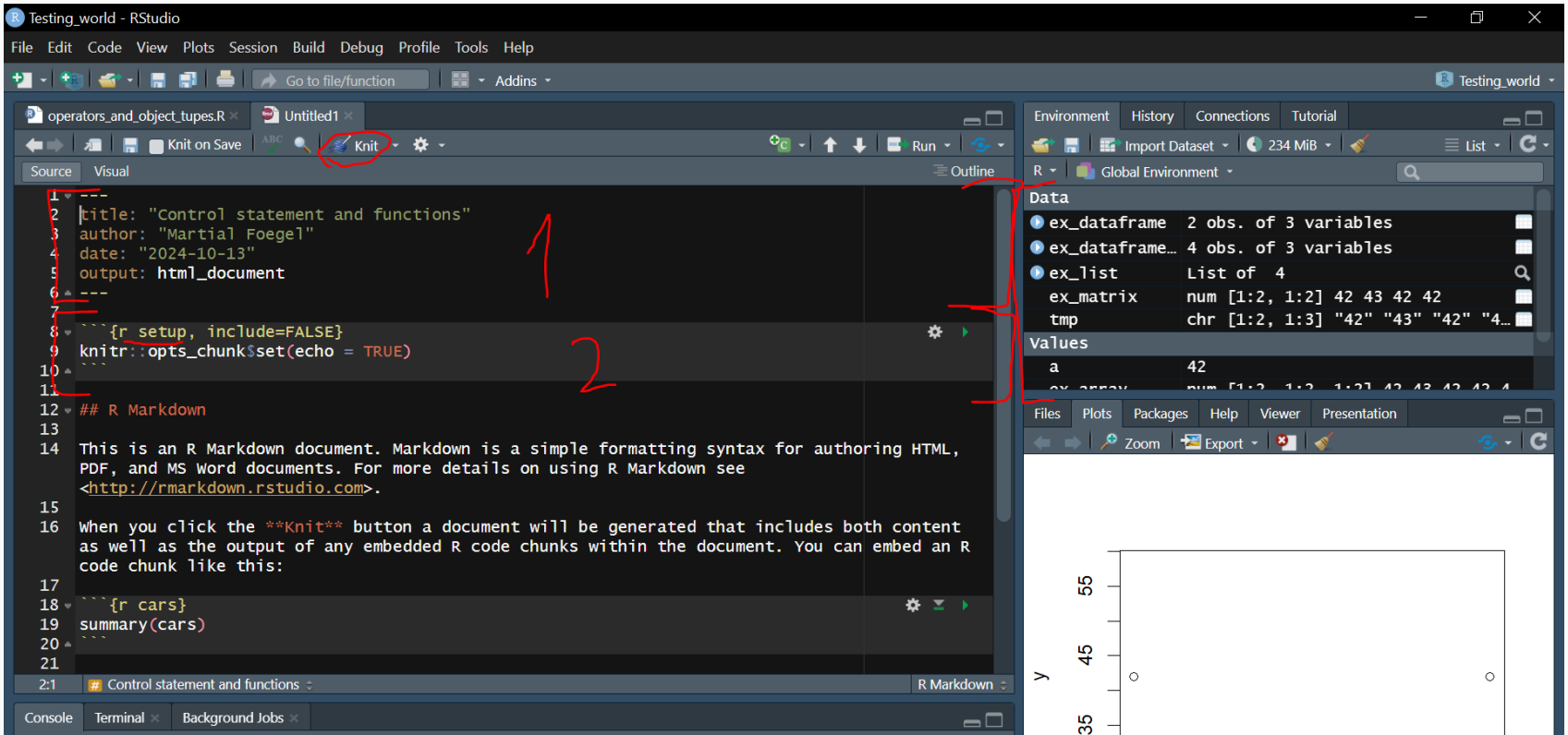


Figure 5: Example of an RMarkdown file in RStudio

Below the YAML header you should see in section 2 a "chunk", which is the name of this particular setup that allows code to be run inside. The rest of the document is just like any text editor, and you can see already subsections being made with the use of two # in front of some text. To run a chunk you have to press the little green arrow on the right of the chunk. The first chunk is special as it is a "setup chunk", meaning whenever you reopen this file, whichever chunk you end up running, this one will be run first.

> You can use the setup chunk to make sure you load the packages necessary for your code.



Running a R script in a RMarkdown file

Let's clean our environment. If you click on the little broom of the top right panel, you will get rid of the current environment.

Create a new chunk either with ***Code*** $>$ ***Insert chunk*** or with the shortcut **Ctrl** $+$ **Alt** $+$ **I**. Inside this chunck use the function `source()` with the name of your R script file and run the chunk. You will see your console running code and shortly after your environment will file up again !

Running a R script in a RMarkdown file

Let's clean our environment. If you click on the little broom of the top right panel, you will get rid of the current environment.

Create a new chunk either with ***Code*** $>$ ***Insert chunk*** or with the shortcut **Ctrl** $+$ **Alt** $+$ **I**. Inside this chunck use the function `source()` with the name of your R script file and run the chunk. You will see your console running code and shortly after your environment will file up again !

```
source(file = "operators_and_object_types.R")
```

Control structures

To control the flow of the execution of code, R has control structures.

If-else

Test a condition and act on it. A simple `if` statement would go like :

```
if(<condition is met>){
  #do something
}
```

You can try a very simple one by writing in a chunk `if(a == 42){print("Yay")}` and running it. It should print out `"Yay"`.

If-else

Test a condition and act on it. A simple `if` statement would go like :

```
if(<condition is met>){
  #do something
}
```

You can try a very simple one by writing in a chunk `if(a == 42){print("Yay")}` and running it. It should print out `"Yay"`.

For multiple condition use multiple `if` statements.

```
if(<condition 1 is met>){
  #do something
}
if(<condition 2 is met>){
  #do something else
}
```

I you want to apply something only whenever the previous condition is not met, use `else`.

```
if(<condition 1 is met>){
  #do something
} else if(<condition 2 is met>){
  #do something if the condition 1 is not met
} else {
  #do something if the condition 1 and 2 are not met
}
```

For

Executes a loop a fixed number of time. It involves an iterator variables taking successive values from a vector. The basic `for` loop looks like

```
for(an_iterator_variable in a_vector){
  #do something
}
```

Simple examples here are `for(i in 1:5){print(i)}` or `for(i in x){print(i)}`.

For

Executes a loop a fixed number of time. It involves an iterator variables taking successive values from a vector. The basic `for` loop looks like

```
for(an_iterator_variable in a_vector){
  #do something
}
```

Simple examples here are `for(i in 1:5){print(i)}` or `for(i in x){print(i)}`.

With what we've learned until now, you should have enough knowledge to try your hand at a simple FizzBuzz task. For number going from 1 to 100:

1. If the number is a multiple of 3, you need to print "Fizz" instead of that number.
2. If the number is a multiple of 5, you need to print "Buzz" instead of that number.
3. If the number is a multiple of both 3 and 5, you need to print "FizzBuzz" instead of that number.

There are multiple ways to do that, but let me remind you that you can use %% for the modulus and that the order of `if` statements may be important here.

There are multiple ways to achieve this task, and here is one of them:

```
for (i in 1:100){
  if(i %% 15 == 0) {print("FizzBuzz")}
  else if(i %% 3 == 0) {print("Fizz")}
  else if(i %% 5 == 0) {print("Buzz")}
  else{print(i)}
}
```

While and Repeat

The `for` loop is the most common in R, but there are other ways to do loops:

- `While` repeats a loop as long as a specific condition is met:

```
while (<condition is met>){
  #do something
}
```

- `Repeat` will repeat a loop infinitely unless you `break` from it.

```
repeat{
  #do something
}
```

While and Repeat

The `for` loop is the most common in R, but there are other ways to do loops:

- `While` repeats a loop as long as a specific condition is met:

```
while (<condition is met>){
  #do something
}
```

- `Repeat` will repeat a loop infinitely unless you `break` from it.

```
repeat{
  #do something
}
```

> ⚠ Warning
>
> Be careful with those loops as they are easier to end in infinite loop. If you end up in one, know that you can end the current running code by pressing the **STOP** button on the bottom left pane or by pressing the red square on the in the chunk.

Next and Break

`next` is used to skip an iteration of a loop and `break` to exit a loop.

Using a `for` loop and a `1:10` vector, as well as your new-found knowledge of `next` and `break`, can you print values from 1 to 8 skipping 5 ?

Next and Break

`next` is used to skip an iteration of a loop and `break` to exit a loop.

Using a `for` loop and a `1:10` vector, as well as your new-found knowledge of `next` and `break`, can you print values from 1 to 8 skipping 5 ?

```
for(i in 1:10){
  if(i == 5){next}
  print(i)
  if(i == 8){break}
}
```

Functions

A function takes as inputs some arguments, run some code and then produce an output.

Basic R functions

We have already use multiple functions as part of this practical (`getwd()`, `print()`, etc). For example the `grep()` function needs two argument, a pattern and a vector, and will put out the indices of the matches. If in a new chunk you write `ex_vector <- c("Hello", "World")`, and then `grep("e", ex_vector)`, you will end up with `1`, as only the first element of the vector as an `"e"` in it.

Basic R functions

We have already use multiple functions as part of this practical (`getwd()`, `print()`, etc). For example the `grep()` function needs two argument, a pattern and a vector, and will put out the indices of the matches. If in a new chunk you write `ex_vector <- c("Hello", "World")`, and then `grep("e", ex_vector)`, you will end up with `1`, as only the first element of the vector as an `"e"` in it.

> **i** With `grep()` and all of its associated function, "regular expressions" can be used.

Basic R functions

We have already use multiple functions as part of this practical (`getwd()`, `print()`, etc). For example the `grep()` function needs two argument, a pattern and a vector, and will put out the indices of the matches. If in a new chunk you write `ex_vector <- c("Hello", "World")`, and then `grep("e", ex_vector)`, you will end up with `1`, as only the first element of the vector as an `"e"` in it.

> **i** With `grep()` and all of its associated function, "regular expressions" can be used.

But what if you wanted `grep()` to return the matching argument rather than its indices ?

If you write `?grep`, your bottom right panel should turn to the **Help** page for that function (you can also directly use the search function of that tab). In all of this you will find an optional parameter called *value*. Can you see now what needs to be changed in our previous function ?

If you write `?grep`, your bottom right panel should turn to the **Help** page for that function (you can also directly use the search function of that tab). In all of this you will find an optional parameter called *value*. Can you see now what needs to be changed in our previous function ?

```
grep("e", ex_vector, value = T)
```

If you write `?grep`, your bottom right panel should turn to the **Help** page for that function (you can also directly use the search function of that tab). In all of this you will find an optional parameter called *value*. Can you see now what needs to be changed in our previous function ?

```
grep("e", ex_vector, value = T)
```

> **Note**
>
> If you want to see how the function is actually written out, most of the time you can type the function without parenthesis and the actual inner working of the function will be revealed to you.

The pipe |>

Since R 4.1, a new tool has made it's way in R, the pipe ! Rather than using a lot of parenthesis that you have to read from the inside out, you can use one or multiple pipes to allow a more natural left-to-right way of reading. For example:

```
round(mean(x))
```

The pipe |>

Since R 4.1, a new tool has made it's way in R, the pipe ! Rather than using a lot of parenthesis that you have to read from the inside out, you can use one or multiple pipes to allow a more natural left-to-right way of reading. For example:

```
round(mean(x))
```

Can be replaced with

```
x |> mean() |> round()
```

The corresponding shortcut is **Ctrl** + **Shift** + **M**

The pipe |>

Since R 4.1, a new tool has made it's way in R, the pipe ! Rather than using a lot of parenthesis that you have to read from the inside out, you can use one or multiple pipes to allow a more natural left-to-right way of reading. For example:

```
round(mean(x))
```

Can be replaced with

```
x |> mean() |> round()
```

The corresponding shortcut is **Ctrl** + **Shift** + **M**

Tip

If your style of coding in R is more traditional, you may end up with lots of parentheses. In that case I would strongly encourage you to use the "rainbow parentheses" (***Tools*** > ***Global Options*** > ***Code*** > ***Display*** > ***Use rainbow parentheses***).

Making your own function

You can create your own function using the following template:

```
function_name <- function(arg1, arg2, ... ) {
  # Code here
}
```

Try to create a function that will take one vector as an input. If that vector is of type character, print it, if it is of type a numerical, output the average of the vector.

> **Tip**
>
> You should name your function using by describing what it does. You should also use the function `warning()` inside your function if you want to clarify to the user what he may be doing wrong.

```r
print_or_avg <- function(a_vector){
  if(is.character(a_vector)){print(a_vector)}
  else if(is.numeric(a_vector)){mean(a_vector)}
  else{warning("You need a character or a numerical vector")}
}

print_or_avg(a_vector = c("Hello", "World"))
(result <- print_or_avg(x))
print_or_avg(TRUE)
```

```r
print_or_avg <- function(a_vector){
  if(is.character(a_vector)){print(a_vector)}
  else if(is.numeric(a_vector)){mean(a_vector)}
  else{warning("You need a character or a numerical vector")}
}

print_or_avg(a_vector = c("Hello", "World"))
(result <- print_or_avg(x))
print_or_avg(TRUE)
```

By default the function will return the last object created. If you want to return a specific object, use `return()`. If you want your function to return multiple objects, remember that you can return a list containing multiple objects !

If you want your argument to have a default value, use `arg = default_value` when defining the function.

We're nearing the end of this presentation, so this next task is a bit harder. I'm going to give you one of function that I put at the top of all my scripts. Can you guess what it's purpose is ?

```
ipak <- function(pkg, load_pkg = T){
  new.pkg <- pkg[!(pkg %in% installed.packages()[, "Package"])]
  if (length(new.pkg)){
    install.packages(new.pkg, dependencies = TRUE)}
  if (load_pkg){
    sapply(pkg, require, character.only = TRUE)}
}
```

```r
# ipak function: install and load multiple R packages.
# check to see if packages are installed. Install them if they are not,
# then load them into the R session if load_pkg is set to true.

ipak <- function(pkg, load_pkg = T){
  new.pkg <- pkg[!(pkg %in% installed.packages()[, "Package"])]
  if (length(new.pkg)){
    install.packages(new.pkg, dependencies = TRUE)}
  if (load_pkg){
    sapply(pkg, require, character.only = TRUE)}
}

# usage
packages_to_load <- c("data.table", "dplyr", "brms", "ggplot2")
packages_not_to_load <- c("readxl", "job", "stringr")

ipak(packages_to_load)
ipak(packages_not_to_load, load_pkg = F)
```

Packages

"If you ever find something that should have its own function in R, but for some reason doesn't, then there surely is somewhere on the web a package that contains this function" - someone, probably !

Packages

"If you ever find something that should have its own function in R, but for some reason doesn't, then there surely is somewhere on the web a package that contains this function" - someone, probably !

You can find them in the **Packages** tab of the bottom right panel, with a handy **Install** button for any that are not already installed on your computer but are on the CRAN (The Comprehensive R Archive Network).

Here are a few libraries that are among the most common and useful:

`knitr`	used for report creation
`tidyverse`	collection of packages designed for data manipulation
`ggplot2`	one of the package of the tidyverse used for some beautiful graphs
`dplyr`	one of the package of the tidyverse which provides a common set of verbs as functions for data manipulation
`tidyr`	one of the package of the tidyverse for tidying your data
`shiny`	for creating a web interface
`lme4`	for linear mixed effect analysis
`brms`	for Bayesian analysis
`jobs`	for running code in the background of your session and still be able to use RStudio

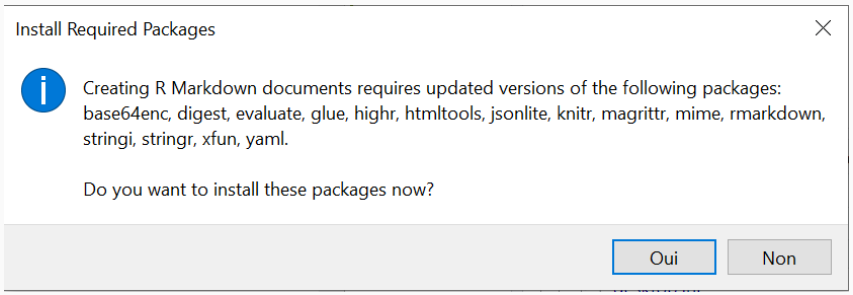
About active memory usage

You may have noticed, has we've gone through this presentation, that whenever you've written down a three letter combination in some chunk or script, that R proposed a few option to auto-complete it. This is because whenever you use RStudio, all of the basic R functions and all the elements in the environment are loaded in your active memory. It's great, but each package that you load and object that you created will be added to this active memory. So keep an eye on the little circle on **Environment** tab of your top-right panel, and maybe once in a while **Free the Unused R Memory**.

Making a quarto presentation

Quarto is the new all-in-one solution proposed by POSIT to create presentations, articles, websites, etc. It's the evolution of the R Markdown format that requires a lot of different packages working together.

What is needed to create a R Markdown file:



What is needed to create a Quarto document:

```
The knitr package is not available in this R installation.
Install with install.packages("knitr")

The rmarkdown package is not available in this R installation.
Install with install.packages("rmarkdown")
```

Apart from being less dependent on packages, the main advantage of Quarto compared to Markdown is the ease to use citations by using Zotero: a simple `@` and you can just find any article from your Zotero library and the bibliography will be automatically added at the end of your document !

If you want to try to replicate this presentation from RStudio, you can ! Go to the ***LLF Forge***, and go to ***R_RStudio_the_basics*** and download the full folder called ***Full_presentation***. Once downloaded, you will find a R Project called *R_RStudio_the_basics.Rproj*. Open it and then also open *Presentation_R_RStudio.qmd*. If you now click **Render** in the top left panel of Rstudio, should see after just a bit a HTML document open in your web browser !

Good practices

In no particular order:

- work using a R Project for each "real life" project
- comment your code often, even in a RMarkdown file
- use sections to separate out your code
- factorize your code whenever you end up repeating something more than twice
- don't use too many packages at once:
  - constantly installing and loading packages uses a lot of energy;
  - you risk having duplicated function names and not know which function you are actually using !
- Know the basic shortcuts, and if you need a reminder go to ***Help*** $>$ ***Keyboard Shortcuts Help*** (there is a shortcut for this too !)
- Use cheat sheets for quick and easy references for packages, as well as for RMarkdown or the RStudio IDE (***Help*** $>$ ***Cheat Sheets***).

References i

"R Core Team (2020). — European Environment Agency." n.d. https://www.eea.europa.eu/data-and-maps/indicators/oxygen-consuming-substances-in-rivers/r-development-core-team-2006.

RStudio Team. 2020. *RStudio: Integrated Development Environment for r*. Boston, MA: RStudio, PBC. http://www.rstudio.com/.