

R Programming: Zero to Pro

Yang Feng and Jianan Zhu

2021-09-29

Contents

Preface

This book is for anyone who is interested in learning R and Data Science. It is designed for people with zero background in programming.

We also have a companion R package named **r02pro**, containing the data sets used as well as interactive exercises for each part.

Introduction

This chapter begins with the installation of R, RStudio, and R Packages in Section ??, and shows how to use R as a fancy calculator in Section ??.

0.1 Installation of R, RStudio and R Packages

0.1.1 Download and Install

As a first step, you need to download R and RStudio, whose links are as follows. For both software, you need to choose the version that corresponds to your operation system.

Download R: <https://cloud.r-project.org/>

Download RStudio: <https://rstudio.com/products/rstudio/download/#download>

RStudio is an *Integrated Development Environment* for R, which is powerful yet easy to use. Throughout this book, you will use RStudio instead of R to learn R programming. Next, let's get started with a quick tour of RStudio.

0.1.2 RStudio Interface

After opening RStudio for the first time, you may find that the font and button size is a bit small. Let's see how to customize the appearance.

a. Customize appearance

On the RStudio menu bar, you can click *Tools*, and then click on *Global Options* as shown in the following figure.

Then, you will see a window pops up like Figure ??. After clicking on *Appearance*, you can see several drop-down menus including *Zoom* and *Editor font size*, among other choices shown.

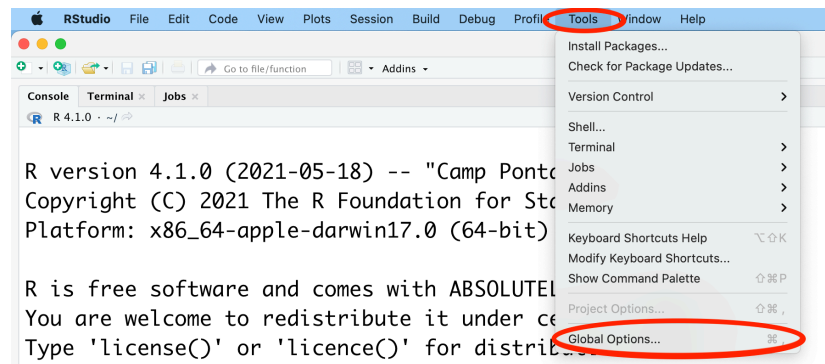


Figure 1: Global Options

- *Zoom* controls the overall scale for all elements in RStudio interface, including the sizes of menu, buttons, as well as the fonts.
- *Editor font size* controls the size of the font only in the code editor.

After adjusting the appearance, you need to click on *Apply* to save our settings.

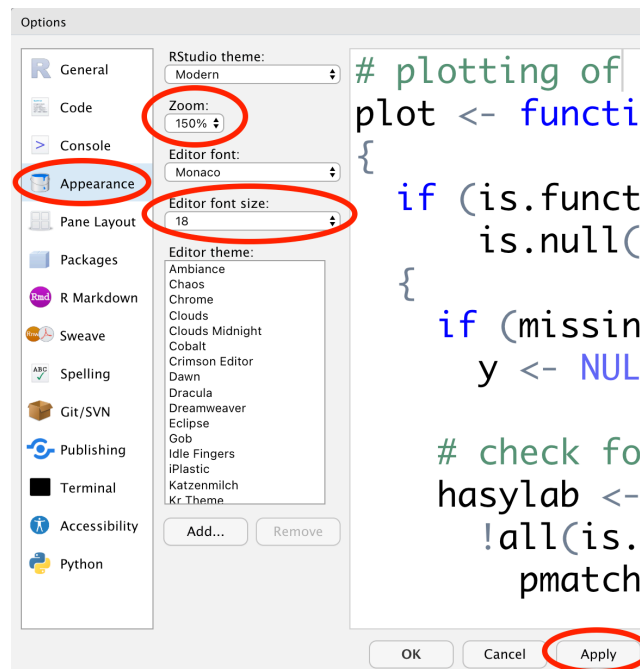


Figure 2: Zoom and Editor font size

Here, we change the *Zoom* to 150% and set the *Editor font size* to 18.

b. Four panels of RStudio

Now, the RStudio interface is clearer with bigger font size. Although RStudio has four panels, not all of them are visible to us at the beginning (Figure ??).

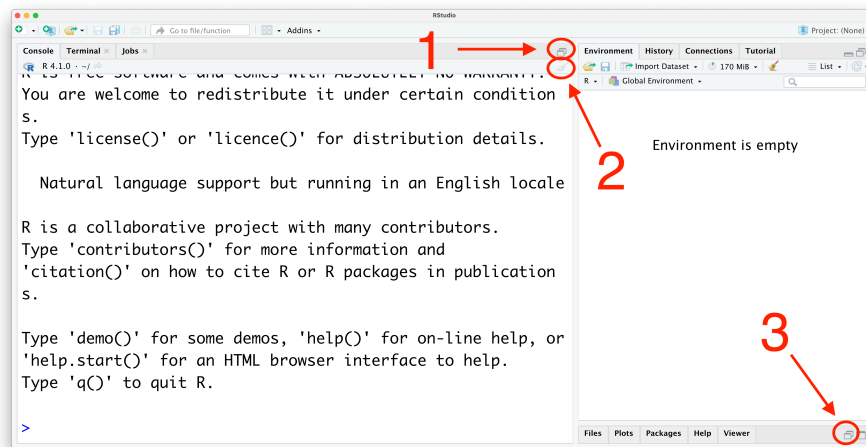


Figure 3: Unfold panels

In Figure ??, we have labeled three useful buttons as 1, 2, and 3. By clicking buttons 1 and 3, you can reveal the two hidden panels.¹ By clicking button 2, we can clear the content in the bottom left panel as shown in the following figure.

Now, let's take a close look at all four panels, which are labeled as 1-4 in Figure ?. You can change the size of each panel by dragging the two blue slides up or down and the green slide left or right.

- Panels 1 and 2 are located to the left of the green line, and are collectively called the **Code Area**. We will introduce them next.
- Panels 3 and 4 are located to the right of the green line, and are collectively called **R Support Area**. We will introduce these two panels in later sections. **Add the section numbers when available**

c. Console

Now, let's introduce the panel 2 in Figure ??, which is usually called the **Console**.

By clicking the mouse on the line after the > symbol, you can see a blinking cursor, indicating that R is ready to accept codes. Let's type 1 + 2 and press Return (on Mac) or Enter (on Windows).

¹Note that you may see different panels hidden when you open RStudio for the first time, depending on the RStudio version. However, you can always reveal the hidden panels by clicking the corresponding buttons like Buttons 1 and 3 in Figure ?.

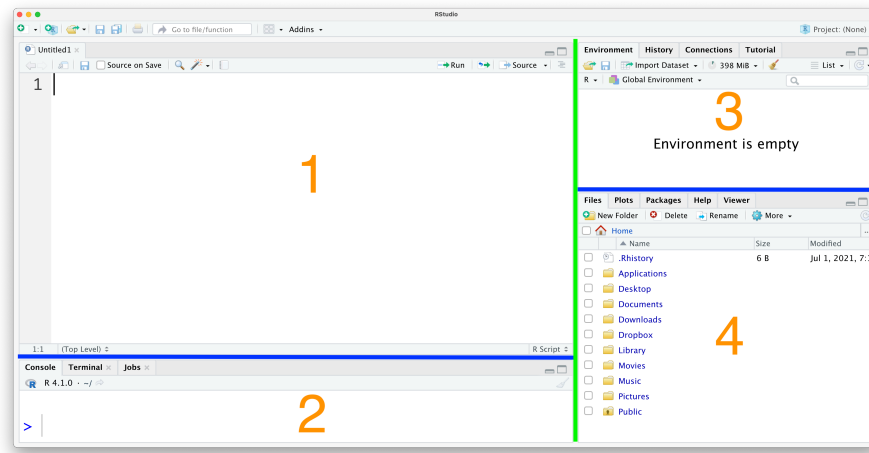


Figure 4: Four panels



It is a good habit to add spaces around an operator to increase readability of the code.

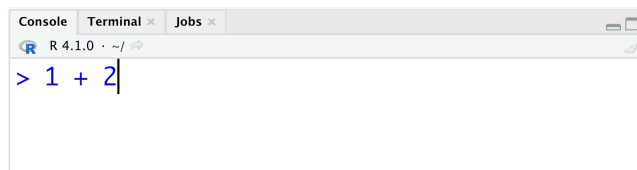


Figure 5: Writing code in the console

Hooray! You have successfully ran our first piece of R code and gotten the correct answer 3. Note that the blinking cursor now appears on the next line, ready to accept a new line of code.

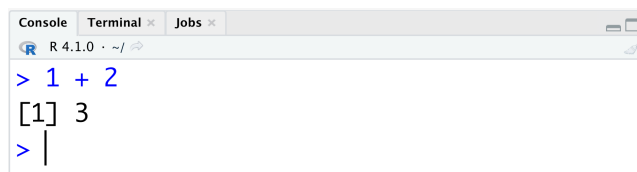


Figure 6: R code(2)

Although the console may work well for some quick calculations, you need to resort to

the panel 1 in Figure ?? (usually called the **Editor**) to save our work and run multiple lines of code at the same time.

d. Save R codes as scripts

The **Editor** panel is the go-to place to write complicated R codes, which you can save as R scripts for repeated use in the future.

Firstly, we will introduce how to run codes in scripts. Let's go to the editor and type `1 + 2`. To run this line of code, you can click the *Run* button. The keyboard shortcut of running this line of code is `Cmd+Return` on Mac or `Ctrl+Enter` on Windows.

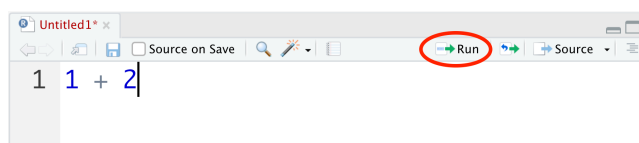


Figure 7: script

RStudio will then send the line of code to the console and execute the code.

After finishing writing codes in the editor, you can save them as a script. To do that, you can click the *Save* button as shown in the Figure ?. The keyboard shortcut of saving files is `Cmd+S` on Mac or `Ctrl+S` on Windows.

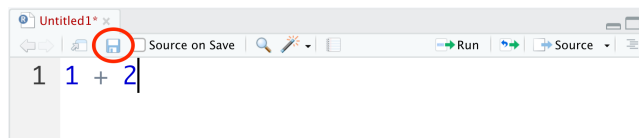


Figure 8: Save (I)

Then you would see a pop-up file dialog box, asking you for a file name and location to save it to. Let's call it `lesson1.1` here.

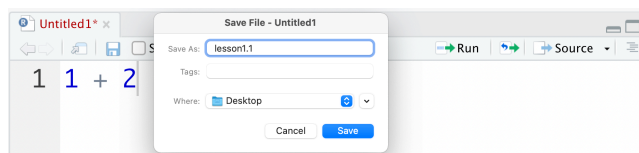


Figure 9: Save (II)

After saving files successfully, you can confirm the name of the R script on the top.

Lastly, if you want to create a new R script, we can click the `+` button on the menu, then select *R Script*. Note that there are quite a few other options including *R Markdown*, which will be introduced in Section ?. Then you will see a new file created.

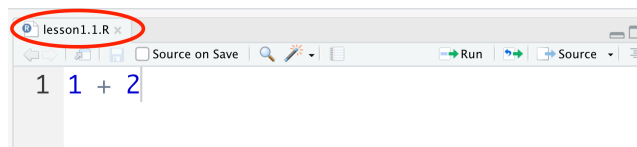


Figure 10: Save (III)

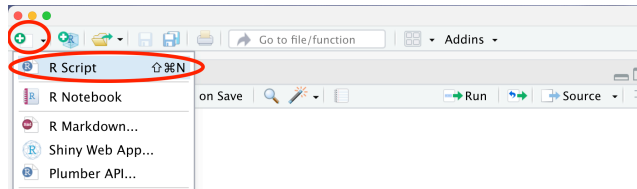


Figure 11: create a new script

0.1.3 Install and load R packages

Now, you have had a basic understanding of RStudio, it is time to introduce **R packages**, which greatly extend the capabilities of base R. There are a large number of publicly available R packages. As of July 2021, there are more than 17K R packages on Comprehensive R Archive Network (CRAN), with many others located in Bioconductor, GitHub, and other repositories.

To install an R package, you need to use a built-in R **function**, which is `install.packages()`. A **function** takes in **arguments** (inputs) and performs a specific task. After the function name, we always need to put **a pair of parentheses** with the arguments inside.

While there are many built-in R functions, R packages usually contain many useful functions as well, and we can also write our own functions, which will be introduced in Chapter ??.

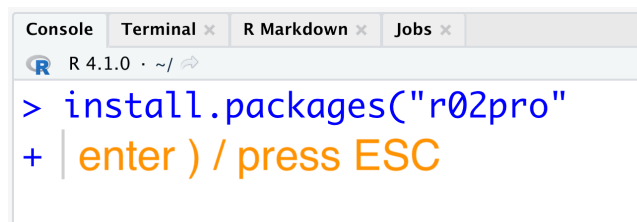
With `install.packages()`, the argument is the package name with a pair of quotation marks around it. The task it performs is installing the specific package into R. Here, you will install the companion package for this book, named `r02pro`, a.k.a. *R Zero to Pro*. The `r02pro` package contains several data sets that will be used throughout the book, and interactive exercises for each subsection.

```
install.packages("r02pro")
```



If you miss the right parenthesis, R will show a plus on the next line (as shown in Figure ??), waiting for more input to complete the command. If this happens, you can either enter the right parenthesis, or press ESC to escape this

command. When you see a blinking cursor after the `>` symbol, you can write new codes again.



```
Console Terminal x R Markdown x Jobs x
R 4.1.0 · ~/
> install.packages("r02pro"
+ | enter ) / press ESC
```

Figure 12: Miss the right parenthesis

After a package is installed, you still need to load it into R before using it. To load a package, we use the `library()` function with the package name as its argument. Here, the quotation marks are not necessary.

```
library(r02pro)
```

Note that once a package is installed, you don't need to install it again on the same machine. However, when starting a new R session, you would need to load the package again.



Quotation marks are necessary for installing R packages, but are not necessary for loading packages. If we install packages without quotation marks. We will see an error message, showing *object not found*.

```
(r02pro)
```

0.1.4 Exercises

1. Which of the following code using to install packages into R will cause an error?
 - `install.packages("r02pro")`
 - `install.packages(r02pro)`
2. Write R code to load the package **r02pro**
3. Write R code to calculate $2 + 3$.

0.2 Use R as a Fancy Calculator

While R is super powerful, it is, first of all, a very fancy calculator.

0.2.1 Add comments using “#”

The first item we will cover is about adding comments. In R, you can add comments using the pound sign #. In each line, anything after # are comments, which will be ignored by R. Let's see an example,

```
6 - 1 / 2 #first calculate 1/2=0.5, then 6-0.5=5.5
#> [1] 5.5
```

Just looking at the resulting value 5.5, you may not know the detail of the calculation process. The comment informs you the operation order: the division is calculated before the subtraction.

In general, adding comments to codes is a very good practice, as it greatly increases readability and make collaboration easier. We will also add many comments in our codes to help you learn R.

0.2.2 Basic calculation

Now let's start to use R as a calculator! You can use R to do addition, subtraction, multiplication×division, and combine multiple basic operations. You can also calculate the square root, absolute value and the sign of a number.

Operation	Explanation
1 + 2	addition
1 - 2	subtraction
2 * 4	multiplication
2 / 4	division
6 - 1 / 2	multiple operations
sqrt(100)	square root
abs(-3)	absolute value
sign(-3)	sign

While the first seven operations in the table look intuitive, you may be wondering, what does the `sign()` function mean here? Is it a stop sign?



Sometimes, you may have no idea how a particular function works. Fortunately, R provides a detailed documentation for each function. There are three ways to ask for help in R.

- Use question mark followed by the function name, e.g. `?sign`
- Use help function, e.g. `help(sign)`
- Use the help window in RStudio, as shown in Figure ???. The help window is the panel 4 of Figure ?? in Section ?. Then type in the function name in the box to the right of the magnifying glass and press return.

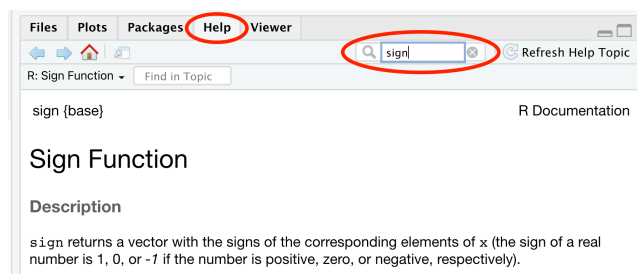


Figure 13: Ask for help

0.2.3 Approximation

After learning about doing basic calculations, let's move on to do approximation in R. When you do division, for example, when computing $7 / 3$, the answer is not a whole number since 7 is not divisible by 3. Under these circumstances, approximation operators are very handy to use. Let's take $7 / 3$ as the example.

a. Get the integer part and the remainder

Code	Name
<code>7 % / 3 = 2</code>	integer division
<code>7 % % 3 = 1</code>	modulus

We all know that $7 = 3 * 2 + 1$. So the *integer division* will pick up the integer part, which is 2 here; and the *modulus* will get the remainder, which is 1.

b. Get the nearby integer

```
floor(7 / 3)
ceiling(7 / 3)
```

Since $2 \leq 7/3 \leq 3$, you can use the `floor` function to find the *largest integer* $\leq 7/3$, which is 2; and the `ceiling` function gives the *smallest integer* $\geq 7/3$, which is 3.

c. Round to the nearest number

```
round(7 / 3)
round(7 / 3, digits = 3)
```

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

0.2.4 Power & logarithm

You can also use R to do *power* and *logarithmic* operations.

Generally, you can use `^` to do power operations. For example, 10^5 will give us 10 to the power of 5. Here, 10 is the *base* value, and 5 is the *exponent*. The result is 100000, but it is shown as `1e+05` in R. That's because R uses the so-called *scientific notation*.



scientific notation: a common way to express numbers which are too large 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 . Let's see some examples,

$$1 \times 10^5 = 1\text{e}+05 \quad (1)$$

$$2 \times 10^4 = 2\text{e}+04 \quad (2)$$

$$1.2 \times 10^{-3} = 1.2\text{e}-03 \quad (3)$$

In mathematics, the *logarithmic operations* are inverse to the power operations. If $b^y = x$ and you only know b and x , you can do logarithm operations to solve y using the general form $y = \log(x, b)$, which is called the logarithm of x with base b .

In R, logarithm functions with base value of 10, 2, or the natural number e have short-cuts `log10()`, `log2()`, and `log()`, respectively. Let's see an example of `log10()`, the logarithm function with base 10.

```
10^6
log10(1e6) #log10(x) = log(x, 10)
```

Next, let's see `log2()`, the logarithm function with base 2.


```
2^10
log2(1024) #log2(x) = log(x, 2)
```

Before moving on to the natural logarithm, note that the natural number e needs to be written as `exp(1)` in R. When you want to do power operations on e , you can simply change the argument in the function `exp()`, for example, `exp(3)` is e to the power of 3. Here, `log()` without specifying the base argument represents the logarithm function with base e .

```
exp(1)
exp(3)
log(exp(3)) #log(x) = log(x, exp(1))
```

0.2.5 Trigonometric function

R also provides the common trigonometric functions.

```
cos(pi)
acos(-1)
```

Here, `acos()` is the inverse function of `cos()`. If we set $\cos(a) = b$, then we will get $\text{acos}(b) = a$.

```
sin(pi/2)
asin(1)
```

Similarly, `asin()` is the inverse function of `sin()`. If we set $\sin(a) = b$, then we will get $\text{asin}(b) = a$.

```
tan(pi/4)
atan(1)
```

Also, `atan()` is the inverse function of `tan()`. If we set $\tan(a) = b$, then we will get $\text{atan}(b) = a$.

0.2.6 Exercises

1. Write R code to compute $\sqrt{5} \times 5$.

2. Write R code to get help on the function `floor`.
3. Write R code to compute the square of π and round it to 4 digits after the decimal point.
4. Write R code to compute the logarithm of 1 billion with base 1000.
5. Write R code to verify $\sin^2(x) + \cos^2(x) = 1$, for $x = 724$.

R Objects (I): Vectors

In the last chapter, we have seen the power of R as a fancy calculator. However, in order to do more complicated and interesting tasks, we may need to store intermediate results for future use.

Let's take a look at a concrete example. Say if you want to do the following calculations involving $\exp(3) / \log(20,3) * 7$.

```
(exp(3) / log(20,3) * 7) + 3 #addition  
(exp(3) / log(20,3) * 7) - 3 #subtraction  
(exp(3) / log(20,3) * 7) / 3 #division
```

You need to type the expression three times, which is a bit cumbersome. In this chapter, we will introduce how to assign the value of the $\exp(3) / \log(20,3) * 7$ to a name for future use. Then, for any operation involving $\exp(3) / \log(20,3) * 7$, you can just use the corresponding name instead.

0.3 Object Assignment

0.3.1 Object

Firstly, we will introduce the most important thing in R, which is called **object**. In principle, **everything that exists in R is an object**. For example, the number 5 is an object, the expression $1 + 2$ is an object, and $\text{floor}(7 / 3)$ is also an object.

In the above examples, there is only one **element** in the object. However, the object can contain more than one elements, and each element has its own **value**. Notice that different elements can have the same value. Values can be of three types including numerical, character, and logical.

For example, $1 + 2$ is an object with one element of value 3.

0.3.2 Assignment Operation with <-

Knowing the importance of objects, let's introduce how to do **object assignments** in R. To do object assignments, you assign **value(s)** to a **name** via the assignment operator, which will create a new object with a name. You can use the new named object once it is created in subsequent calculations without redundancy. Let's start with a simple example,

```
x_numeric <- 5
```

The assignment operation has three components. From left to right

- the first component `x_numeric` is the **object name** of a new object, which has certain naming rules which we will discuss shortly in Section ??.
- The second component is the **assignment operator** `<-`, which is a combination of the less than sign `<` immediately followed by the minus sign `-`.
- The final component is the **value(s)** to be assigned to the name, which is 5 here.



There is no space between `<` and `-` in the assignment operator `<-`. Note that although `=` may also appear to be working as the assignment operator, it is not recommended as `=` is usually reserved for specifying the value of arguments in a function call, which will be introduced in Section ??.

After running the code above, you will see no output in the console, unlike the case when we ran `1 + 2` which gives us the answer 3 (as shown in the Figure ??). You may be wondering, did we successfully make our first assignment operation?

```
Console Terminal x R Markdown x Jobs x
R 4.1.0 · ~/
> x_numeric <- 5
> 1 + 2
[1] 3
```

Figure 14: No output

To verify it, you can run the code with just the object name to check its value. (For named objects, you can get their values by running codes with their object names.)

```
x_numeric  
#> [1] 5
```

Great! You get the value 5, indicating that you have successfully assigned the value 5 to the name `x_numeric`, and you have created a new object `x_numeric`. You can use `x_numeric` instead of 5 to do the subsequent calculations because `x_numeric` and 5 have the same value.

You can also assign value(s) of any R expression (including operations and functions) to a name. Sometimes, the value(s) of an R expression may be unintuitive. To get the value(s) of an R expression, R will get the result of it firstly.

In this case, R will first calculate the result of $\exp(3) / \log(20, 3) * 7$ and assign the value of the result to a name. Let's see the following example.

```
y_numeric <- exp(3) / log(20, 3) * 7  
y_numeric  
#> [1] 51.56119
```

Now you have successfully created an object `y_numeric` with value 51.56119. Using the named object `y_numeric`, you can do the same three calculations introduced at the beginning of this chapter as follows.

```
y_numeric + 3  
y_numeric - 3  
y_numeric / 3
```

You can also try the following examples by yourself.

```
a <- floor(7 / 3)  
a  
b <- 7 %/% 3  
b
```

Clearly, using the object assignment, we can greatly simplify our code and avoid redundancy.

Note that R object names are **case-sensitive**. For example, we have defined `x_numeric`, but if you type `X_numeric`, you will get an error message as follow.

```
X_numeric  
#> Error in eval(expr, envir, enclos): object 'X_numeric' not found
```