

R Programming: Zero to Pro

r02proers

2022-10-16

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.

Contributing to the Book

To improve the book, we would like to ask for your help.

Submitting Questions or Feedback

If you have any questions or feedback (including typos or grammar issues) about any materials in the book, we greatly appreciate if you can write to us at r02pro007@gmail.com¹. We will also add your name in the Acknowledgment section to show our gratitude.

Contributing Exercises

We warmly welcome everyone to help the book via contributing exercises for each section. Please make sure the exercise you are contributing only needs the knowledge up to the specific section number (it of course can use any knowledge of all sections up to the particular section) you would like to put the exercise into.

Please submit the proposed exercise(s) via the Google Form².

Thanks a lot for your contribution.

¹<mailto:r02pro007@gmail.com>

²https://docs.google.com/forms/d/e/1FAIpQLSckpcPZvRSr2vju-J2w8Nh0x_oIyYtd8uuX0xqq_ThfRWMltw/viewform

Acknowledgement

This book is the product of numerous collaborative efforts. Among all people, we would like to thank Dr. Rebecca Betensky, Dr. Rumi Chunara, and Dr. Hai Shu, for the initial planning of the new courses GPH-GU 2183: Introduction to Statistical Programming in R and GPH-GU 2183: Intermediate Statistical Programming in R. In addition, we are most grateful to the students in the course GPH-GU 2183: Introduction to Statistical Programming in R in Fall 2021 at New York University. The list of people that made contributions include Xiaofeng Yang (@well9801³), Neethu Grace Johnson (@neethujohnson01⁴), Yifan Lai (@Yifan-Lai⁵), Ruiming Yu (@OmakaseMaster⁶), Zhihao Chen (@edwardzchen⁷), Fan Bi (@fanbithededenne⁸), Rebecca Yu (@rjy2107⁹), Ting Lu (@tinglusince¹⁰), Yang Yang (@YoArtemis¹¹), Jian Li (@kegemor¹²), Grace Lin (@gracexlyhl¹³), Xin Chen (@zoexinchen¹⁴), Wanyu Hua (@hiWanyu¹⁵)

³<https://github.com/well9801>

⁴<https://github.com/neethujohnson01>

⁵<https://github.com/Yifan-Lai>

⁶<https://github.com/OmakaseMaster>

⁷<https://github.com/edwardzchen>

⁸<https://github.com/fanbithededenne>

⁹<https://github.com/rjy2107>

¹⁰<https://github.com/tinglusince>

¹¹<https://github.com/YoArtemis>

¹²<https://github.com/kegemor>

¹³<https://github.com/gracexlyhl>

¹⁴<https://github.com/zoexinchen>

¹⁵<https://github.com/hiWanyu>

Building Environment

The R session information when compiling this book is shown below.

```
sessionInfo()
#> R version 4.0.5 (2021-03-31)
#> Platform: x86_64-apple-darwin17.0 (64-bit)
#> Running under: macOS Big Sur 10.16
#>
#> Matrix products: default
#> BLAS: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRblas.dylib
#> LAPACK: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRlapack.dylib
#>
#> locale:
#> [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
#>
#> attached base packages:
#> [1] stats graphics grDevices utils datasets methods base
#>
#> other attached packages:
#> [1] dplyr_1.0.5
#>
#> loaded via a namespace (and not attached):
#> [1] rstudioapi_0.13 knitr_1.32 magrittr_2.0.1 hms_1.0.0
#> [5] tidyselect_1.1.0 R6_2.5.0 rlang_0.4.10 fansi_0.4.2
#> [9] stringr_1.4.0 tools_4.0.5 xfun_0.22 utf8_1.2.1
#> [13] DBI_1.1.1 htmltools_0.5.1.1 ellipsis_0.3.2 assertthat_0.2.1
#> [17] yaml_2.2.1 digest_0.6.27 tibble_3.1.4 lifecycle_1.0.0
#> [21] crayon_1.4.1 bookdown_0.22 readr_1.4.0 purrr_0.3.4
#> [25] vctrs_0.3.8 glue_1.4.2 evaluate_0.14 rmarkdown_2.7
#> [29] stringi_1.5.3 compiler_4.0.5 pillar_1.6.3 generics_0.1.0
#> [33] pkgconfig_2.0.3
```

In the book, package names are in **bold text** (e.g., **r02pro**), and inline codes and functions are formatted in a typewriter font (e.g. `1 + 2`, `r02pro()`). We always add paren-

theses after a function name (e.g. `r02pro()`).

Introduction

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

0.1 Installation of R, RStudio and R Packages

0.1.1 Download and Install

As the 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 operating 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. Without further ado, let's start 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 below.

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

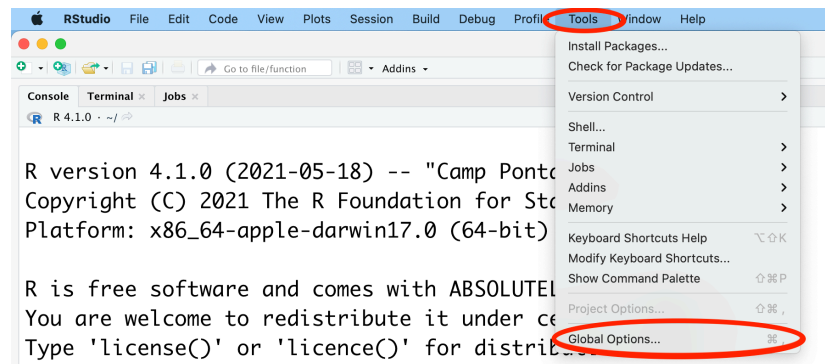


Figure 1: Global Options

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

Once done customizing the appearance, you need to click on *Apply* to save the adjustments.

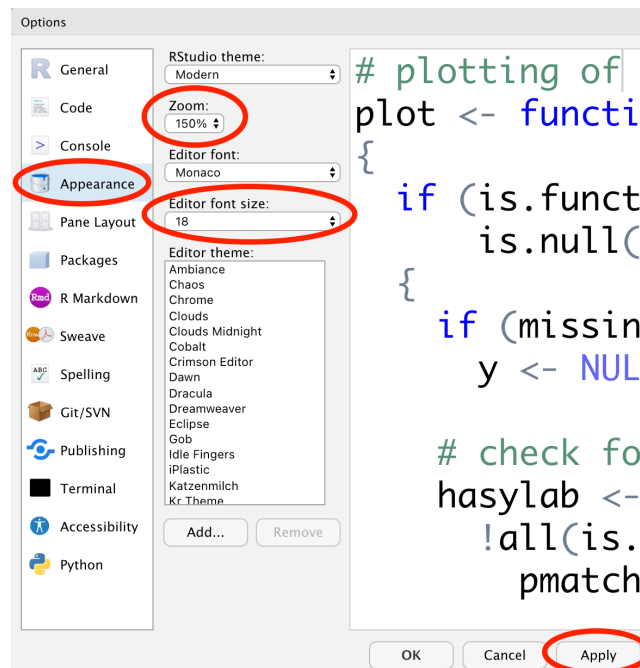


Figure 2: Zoom and Editor font size

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

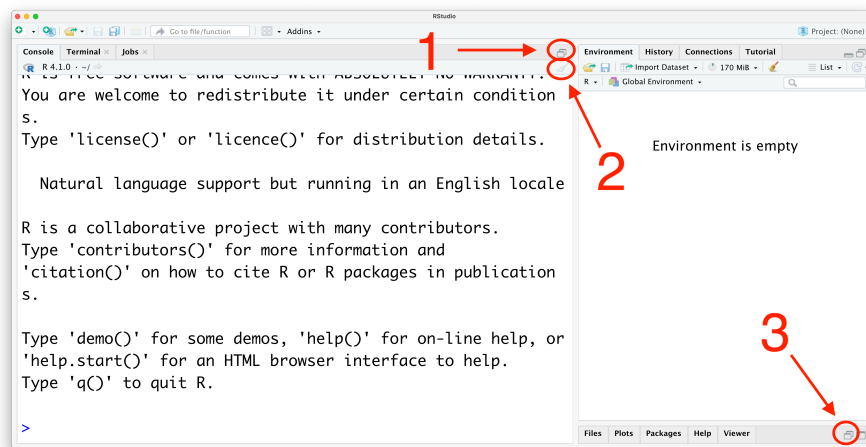


Figure 3: Unfold panels

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.



Note 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 ??.

By clicking button 2, we can clear the content in the bottom left panel (Panel 2 in Figure ??) 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*.

- Located to the left of the green line, Panels 1 and 2 together compose the **Code Area**. We will introduce them in the following parts of this section.
- Located to the right of the green line, Panels 3 and 4 together make up the **R Support Area**. We will introduce these two panels in later sections.

c. Console

Firstly, we will introduce panel 2 in Figure ??, which is usually called the **Console**. The console window is the place for you to type in codes (i.e. the things you want R to do) and you will get the results immediately once you run the codes.

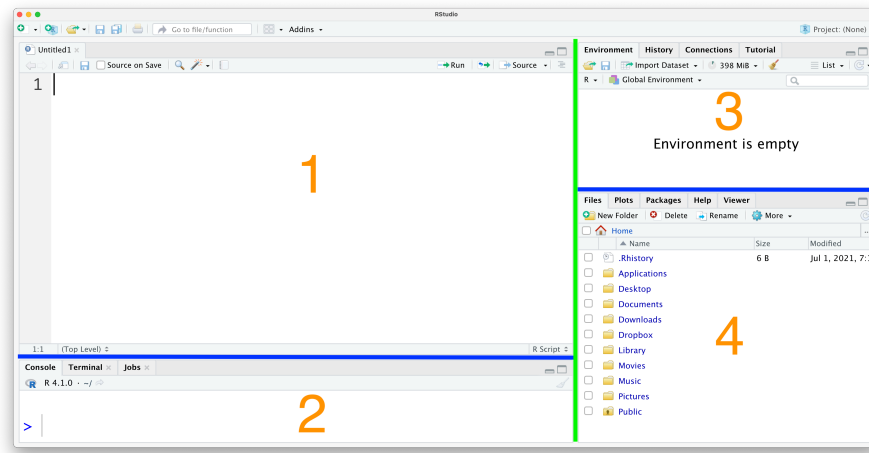


Figure 4: Four panels

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).



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

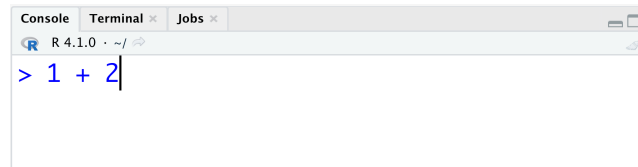


Figure 5: Writing code in the console

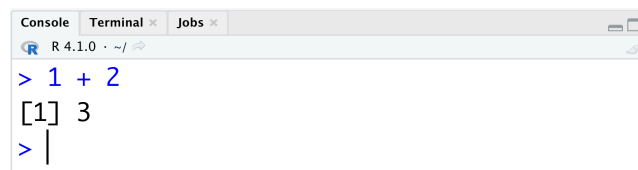


Figure 6: R code(2)

Hooray! You have successfully run the 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.



The curious you may found that there is a `[1]` showing before the result 3. In fact, the `[1]` is an index indicator, showing the next element has an index of 1 in this particular object. We will revisit this point when we introduce vectors in the beginning of Chapter ?? .

Although the console may work well for some quick calculations, you need to resort to panel 1 in Figure ?? (known as the **Editor**) to save our work and run multiple lines of code at once.

d. Editor

The **Editor** panel is the go-to place to write complicated R codes, which you can save as R files for repeated use in the future. Several kinds of files are available in RStudio. In particular, **R script**, **R Markdown**, and **R Notebook** are the three most common file formats. In order to let you get started better, we will start with R script since this is the simplest file format in R. In Chapter 12 and , we will introduce R Markdown and R Notebook in detail.

In the editor panel, you may notice that RStudio has created a file by default (Figure ??). The default file RStudio provided is **R script**.

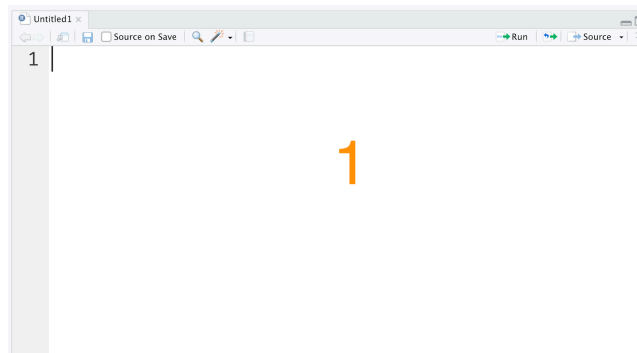


Figure 7: R script

Next, 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 select this line of code and click the *Run* button. The keyboard shortcut of running this line of code is `Cmd+Return` on Mac or `Ctrl+Enter` on Windows. RStudio will then send the line of code to the console and execute the code.

You can also run multiple lines of code by selecting the lines and clicking the *Run* button or using the keyboard shortcut. (Figure ??)

Here, three lines of codes are selected. After running these three lines of code together, you can see that the console executes each line of code and you will get the corresponding answer one by one. Therefore, you can write any number of lines of codes in the script, and you can get the answer of each line in the console.

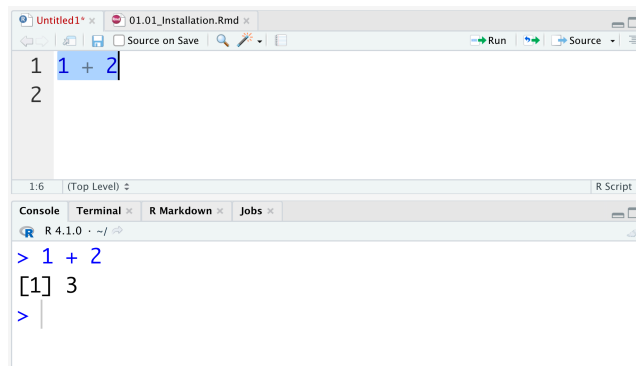


Figure 8: Run codes in script (I)

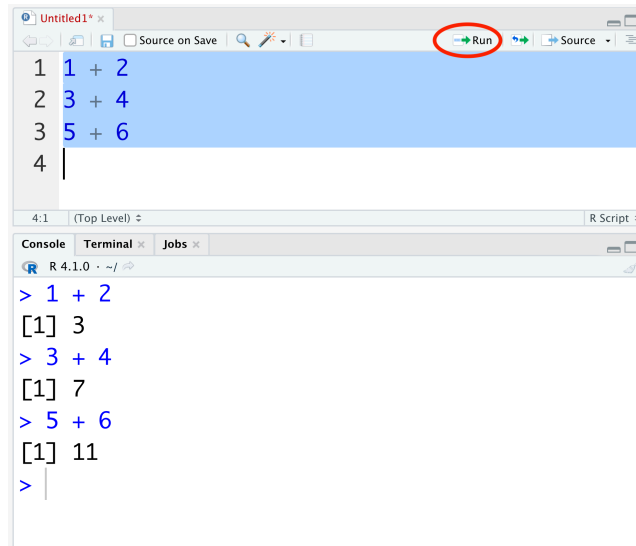
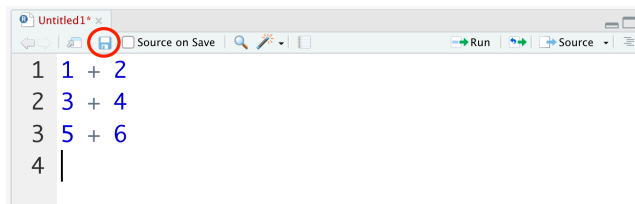
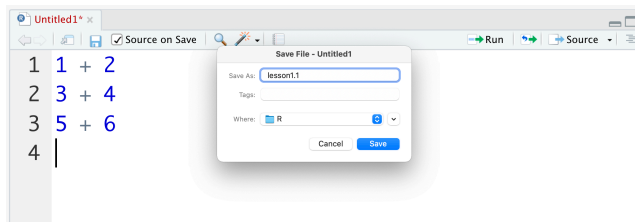


Figure 9: Run codes in script (II)

After finishing writing codes in the editor, there may be hundreds or more lines of codes in the script. Now, you may wonder if you need to write these codes again when you want to use the same codes next time. The answer is absolutely NO!!! One of the most important features of R files is that R files can be saved for future use. So do R scripts! To do that, you can click the *Save* button as shown in Figure ???. The keyboard shortcut of saving files is `Cmd+S` on Mac or `Ctrl+S` on Windows.

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.

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

**Figure 10:** Save (I)**Figure 11:** Save (II)

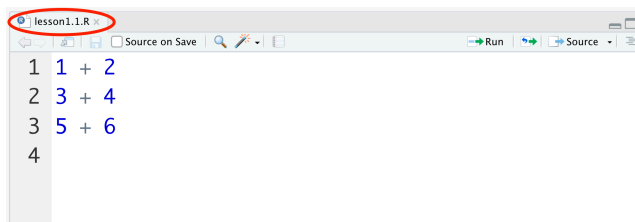
Then if you close this script and open it again, you would directly see the previous three lines of codes without writing them again.

Lastly, if you want to create a new R script, you 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 Chapter ??.

Consequently, you will see a new file created.

0.1.3 Install and load R packages

Now, you have had a basic understanding of RStudio, it is time to meet **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.

**Figure 12:** Save (III)

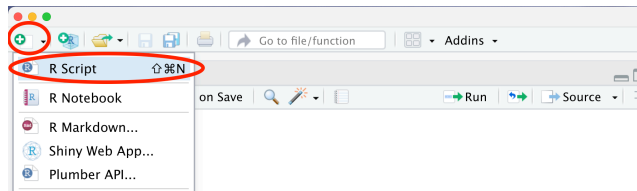


Figure 13: create a new script (I)

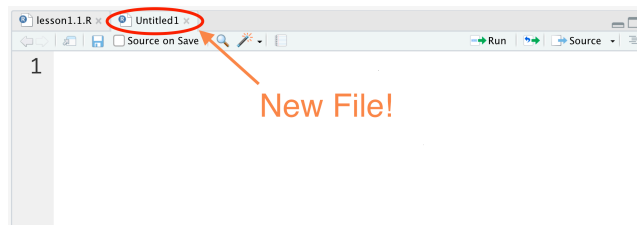


Figure 14: create a new script (II)

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 accordingly. 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 return a plus in 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.

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

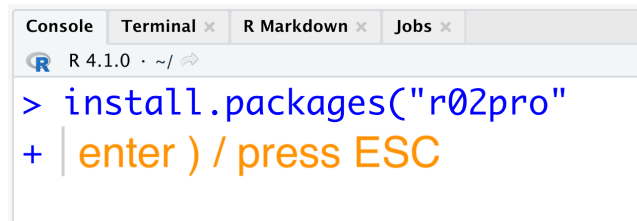


Figure 15: Miss the right parenthesis

```
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. When installing packages without quotation marks, you will see an error message, showing *object not found*.

```
(r02pro)
```

0.1.4 Exercises

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

0.2 Use R as a Fancy Calculator

After learning how to run codes in R, we will introduce how to use R as a fancy calculator.

0.2.1 Add comments using “#”

Before we get started, the first item we will cover is adding comments for codes. In R, you can use the hash character # at any position of a given line to initiate a comment,

and anything after # 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
```

By running this line of code (either in the console or in the editor!), you will get a value of 5.5, which is the answer of $6 - 1 / 2$. As demonstrated, R will not run syntax after the hash character #. Commands and strings after the # are notations or explanations that can make codes easier to understand. Here, the comment informs you the operation order of previous code: 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 necessary comments in our codes to help you learn R.

0.2.2 Basic calculation

Now let's start to use R as a calculator! In the previous section we introduced operations such as addition, subtraction, multiplication, division, as well as the combination of multiple basic operations. Additionally, 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

0.2.3 Get help in R

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



Sometimes, you may have no idea how a particular function works. Fortunately, R provides a detailed documentation for each function.

a. Ask for help

First, we will introduce how to ask for help in R, and below are three common ways to seek for more information.

- Use a 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 16: Ask for help

b. Documentation for functions

After using the above operations to ask for help in R, you can get the documentation of the function in the help window. The documentation consists of different parts, let's take the `sign()` function as an example (Figure ???),

This documentation contains the following parts:

- *Description*: A text-format introduction of the function of interest. The introduction describes the function's mechanism, the acceptable input and output types, and some notes of the function.
- *Usage*: The way the function looks like.
- *Arguments*: A detailed description of the input.
- *Details*: A detailed description of the function, including the background, some complicated usage, and special cases of the function.
- *See Also*: Some functions related or similar to this function.
- *Examples*: Sample codes and their corresponding answers. You can simply copy codes in the *Examples* part and run them in the editor or in the console. Note that all words after `#` are comments and will be ignored by R.

Here, from the documentation of the `sign()` function, you will know that the `sign()` function can return the signs of numbers, which means it will return 0 for zero, return 1 for positive numbers, and return -1 for negative numbers.

The documentation of different functions can contain different parts, we will give you the introduction of other functions in the following sections.

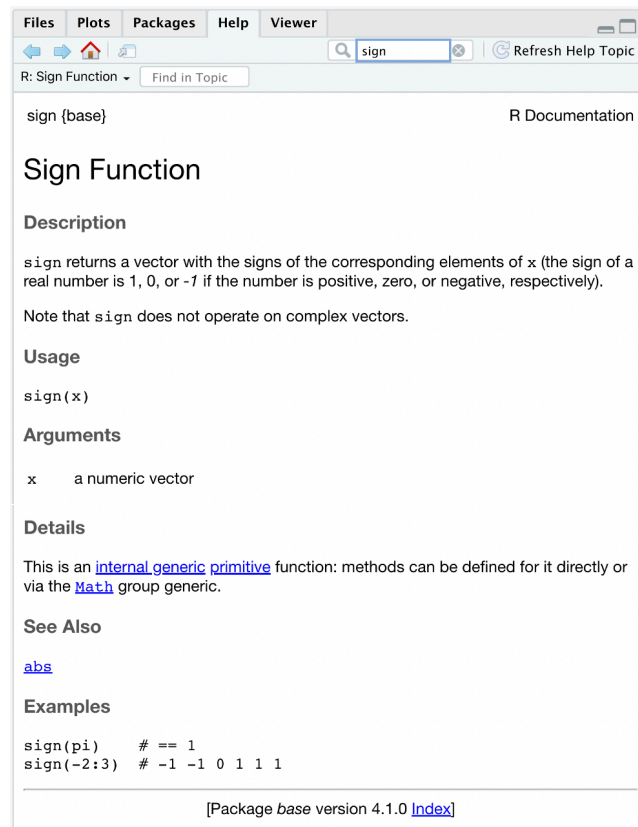


Figure 17: Documentation for function

0.2.4 Approximation

Next, let's move on to the approximation in R. When computing $7 / 3$, the answer is not a whole number as 7 is not divisible by 3. Approximation will come in handy under such circumstances. Let's take $7 / 3$ as the example.

a. Get the integer part and the remainder

Code	Name
<code>7%/%3</code>	integer division
<code>7%%3</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)
#> [1] 2
ceiling(7 / 3)
#> [1] 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)
#> [1] 2
round(7 / 3, digits = 3)
#> [1] 2.333

```

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`.

0.2.5 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 logarithmic 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 shortcuts `log10()`, `log2()`, and `log()`, respectively. Let's see an example of `log10()`, the

logarithm function with base 10. Here, we have added a comment to help you have a better understanding of `log10()`.

```
10^6
#> [1] 1e+06
log10(1e6) #log10(x) = log(x, 10)
#> [1] 6
```

Next, let's see `log2()`, the logarithm function with base 2. There is also a comment for `log2()` here.

```
2^10
#> [1] 1024
log2(1024) #log2(x) = log(x, 2)
#> [1] 10
```

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 . You can see the general form of `log()` in the comment.

```
exp(1)
#> [1] 2.718282
exp(3)
#> [1] 20.08554
log(exp(3)) #log(x) = log(x, exp(1))
#> [1] 3
```

0.2.6 Trigonometric function

R also provides the common trigonometric functions.

```
cos(pi)
#> [1] -1
acos(-1)
#> [1] 3.141593
```

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)
#> [1] 1
asin(1)
#> [1] 1.570796
```

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)
#> [1] 1
atan(1)
#> [1] 0.7853982
```

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

0.2.7 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 $x = 724$.

0.3 Object Assignment

In the last section, you have seen the power of R as a fancy calculator. However, in order to do more complicated and interesting tasks, it is super helpful to store intermediate results for future use.

Let's take a look at a concrete example. Say if you want to do the following three calculations, all 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
```

Using R as a fancy calculator (Section ??), you need to type the expression `exp(3) / log(20,3) * 7` three times, which is a bit cumbersome. In this section, you will learn how to do **object assignment**, which can avoid the need for typing the same expression more than once.

0.3.1 What is an R Object?

Before we get into the details, let's first introduce **object**, which is perhaps the most fundamental thing in R. 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 the expression `exp(3) / log(20,3) * 7` is also an object.

If you run 5, you will get one element of value 5 from the output. Similarly, if you run `1 + 2`, you will get one element of value 3 from the output. You can try to run `exp(3) / log(20,3) * 7` by yourself. In these three examples, you can see that there is only one **element** in each object.

However, an object can contain more than one elements, and each element has its own **value**, which is possibly different from that of another element. Naturally, different objects can contain different values.

0.3.2 Assignment Operation with `<-`

With the importance of objects in mind, let's learn how to do **object assignments** in R. To do object assignments, you need to assign **value(s)** to a **name** via the **assignment operator**, which will create a new object with the name you specified. Once the object assignment operation is done, you can simply use the name in subsequent calculations without redundancy. Let's start with a simple example,

```
x_num <- 5
```

The assignment operation has three components. From left to right

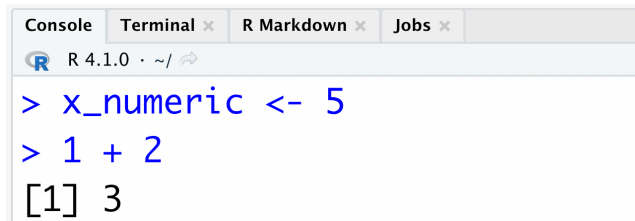
- the first component `x_num` is the **object name** of a new object, which has certain naming rules that will be discussed shortly in Section ??.
- The second component is the **assignment operator** `<-`, which is a combination of the less than sign `<` immediately followed by the minus sign `-`, with **no space** in between.
- The final component is the object name's assigned **value(s)**, 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(s) 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 situation when we ran `1 + 2` which gives us the answer 3 (as shown in Figure ??). You may be wondering, did we successfully make our first assignment operation?

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



```

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

```

Figure 18: No output

```

x_num
#> [1] 5

```

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

Note that R object names are **case-sensitive**. For example, you have defined `x_num`, but if you type `X_num`, the console will return an error message as follow.

```

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

```

In addition, you can assign **value(s)** of an expression to a name. Let's try to simplify the three expressions we showed at the beginning of this section. It is easy to observe that the three expressions share a common term $\exp(3) / \log(20,3) * 7$. Let's assign the common term to a name.

```

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

```

Now you have successfully created an object `y_num` with value 51.561191. Using the named object `y_num`, you can simplify the three calculations as follows.

```

y_num + 3
y_num - 3
y_num / 3

```



Note that in the object assignment process, it is not the expression itself but rather the value(s) of the expression, that is assigned to a name. So you will not get the expression $\exp(3) / \log(20,3) * 7$ by running `y_num`.

You can also try the following examples by yourself.

```
z_num1 <- floor(7 / 3)
z_num1
z_num2 <- 7%%3
z_num2
```

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

0.3.3 Object naming rule

As you now see, the assignment operation in R is very straightforward. In general, R is very flexible in the name you can give to an object. However, there are three important rules you need to follow.

a. Must start with a letter or . (period)

In addition, if starting with period, the second character can't be a number.

b. Can only contain letters, numbers, _ (underscore), and . (period)

One recommended naming style is to use lowercase letters and numbers, and use underscore to separate words within a name. So you can use relatively longer names that is more readable. For example, `this_is_name_6` and `super_rich_88` are great names.

c. Can not use special keywords as names.

For example, `TRUE <- 12` is not permitted as `TRUE` is a special keyword in R. You can see from the following that this assignment operation leads to an error message.

```
TRUE <- 12
#> Error in TRUE <- 12: invalid (do_set) left-hand side to assignment
```

Some commonly used reserved keywords that cannot be used as names are listed as below.

TRUE	else
FALSE	for
NA	while
Inf	break
NaN	next
function	repeat
if	return

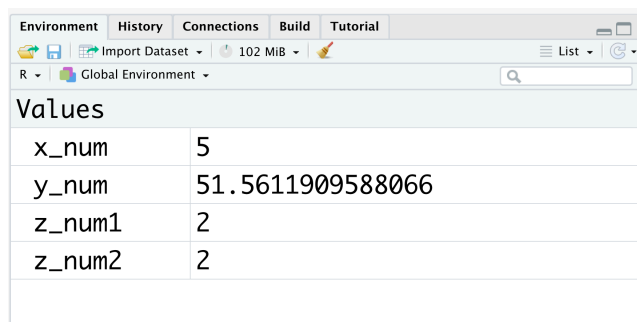
To get a complete list of reserved words, you can run the following code.

```
?Reserved
```

0.3.4 Review objects in environment

At this point, we've introduced the rules of creating objects in R. Now, you can also confirm the success of object assignments by inspecting the **Environment**, located in the top right panel (**panel3 in Figure ?? in Section ??**).

If you exercised previous examples, you can find the newly assigned objects `x_num` and `y_num` (possibly also `z_num1` and `z_num2`) in your *Environment* Viewer. You may also notice that the name `TRUE`, which we tried but failed to assign the value 12 to, doesn't appear in the *Environment* (as shown in Figure ??). You can check all the **named objects** and their values in this area.



Values	
<code>x_num</code>	5
<code>y_num</code>	51.5611909588066
<code>z_num1</code>	2
<code>z_num2</code>	2

Figure 19: The environment (I)

From the picture above, you can see that the value of `x_num` is 5. In this case, let's try to assign the new value 6 to `x_num` and see what will happen next.

```
x_num <- 6
x_num #check its value
#> [1] 6
```

Now you can see that the value of `x_num` has changed from 5 to 6. Generally, when assigning a new value to an object, R will update the object's value, and the previous value will no longer be stored. You can verify the result by inspecting the Environment tab, where only the new value of the object will be displayed.

So it is helpful to monitor the environment from time to time to make sure everything looks fine. Notice that objects without names will *not* be shown in the environment.

You can also see the list of all the named objects (just names without values) using the built-in R function `ls()`.

```
ls()
#> [1] "key_mat" "Keys"    "x_num"   "y_num"
```

All the objects shown in the environment or on the list are stored in the memory, so

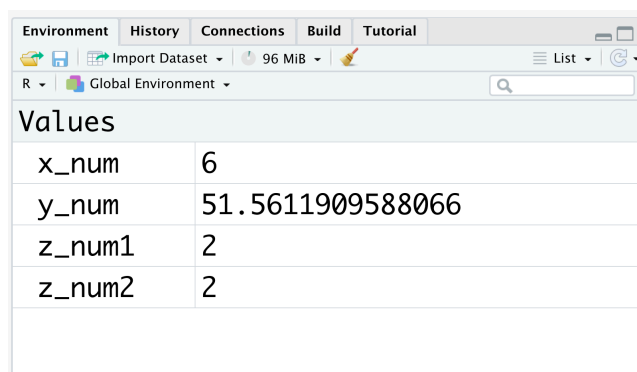


Figure 20: The environment (II)

they are available for us in subsequent codes. It is a good habit to do object assignments if you want to retrieve their values at a later time.

0.3.5 Object types

So far in this section, you have learned how to do object assignments. The values you assigned are all numbers, i.e. of numeric type. Actually, an object may contain more than one values. Also, an object may contain values other than the numeric type, such like character and logical ones. Depending on the **composition of values**, the object belongs to one particular type.

Type	Section
Atomic Vector	\@ref(r-objects)
Matrix	\@ref(matrix)
Array	\@ref(array)
Data Frame	\@ref(dataframe)
Tibble	\@ref(tibble)
List	\@ref(list)

We will focus on atomic vectors in Chapter ?? and discuss other object types in Chapter ??.

While some of the object types look more intuitive than others, you have nothing to worry about since we have the next two chapters devoted to the details of R objects. Objects are the building blocks of R programming and it will be time well spent mastering every object type.

0.3.6 Exercises

1. Write the R code to assign the value 20 to the name num_1.

2. Which of the following is a valid object name in R?

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

3. Write the R code to get the list of all objects in the environment.

R Objects (I): Atomic Vectors

In this chapter, we focus on the most fundamental R object type: **atomic vectors**. We will introduce different types of atomic vectors, creating vectors with patterns, applying different functions and operations on vectors, comparing and extracting vectors. We will also discuss data and times, factors, and how R represents unexpected results.

0.4 Introduction to Numeric Vectors

We will start off this chapter by learning **numeric vectors**. Numeric vectors are perhaps the most commonly used member of the **atomic vector** family, where all elements are of the same type.

0.4.1 Creation and class

A **numeric vector** is an atomic vector containing only numbers. For example, 6 is a numeric vector with one element of value 6.

By assigning the value 6 to the name `x1`, you can create a new numeric vector `x1` with value 6. As a result, you can refer to `x1` in subsequent calculations. For any vector, you can use the `length()` function to check the number of elements it contains.

```
6                                #a numeric vector
#> [1] 6
x1 <- 6                          #x1 is also a numeric vector
x1                                #check the value of x1
#> [1] 6
length(6)                        #length of 6
#> [1] 1
length(x1)                       #length of x1
#> [1] 1
```

Given the output, you can see that 6 is a numeric vector with length 1 and you have successfully created the numeric vector `x1` of length 1.

Moving on, you may wonder, can a numeric vector contain more than one value? The answer is a big YES! In R, you can use the `c()` function (`c` is short for combine) to store several elements into one single numeric vector.

```
c(1, 3, 3, 5, 5)           #use c() to combine elements into a numeric vector of length 5
#> [1] 1 3 3 5 5
y1 <- c(1, 3, 3, 5, 5)     #y1 is a numeric vector of length 5
y1                         #check the value of y1
#> [1] 1 3 3 5 5
length(y1)                #length of y1
#> [1] 5
```

In this example, firstly you have created a length-5 object using the `c()` function with arguments being the five elements separated by **comma**. Since all elements are numbers, this object is still a numeric vector. Then after assigning the values to the name `y1`, you will get a new numeric vector `y1` with 5 elements. Among the five elements, although some of them have the same value, R still recognizes and stores them separately. This is the reason why the length of `y1` is 5 instead of 3.

Similar to `x1`, you can verify the contents of `y1` and check the length of it via the `length()` function.



When you assign several values to a name, the order of the values will not change after assignment. If you create two numeric vectors containing the same numbers but in different orders, the two vectors will be two different ones maintaining the specified orders. For example,

```
y2 <- (1, 3, 5, 7, 9)
y2
#> [1] 1 3 5 7 9
y3 <- (9, 7, 5, 3, 1)
y3
#> [1] 9 7 5 3 1
```

In addition to using numbers inside the `c()` function, you can also use numeric vectors as the arguments to create a longer vector. The new, longer vector will combine the input numeric vectors in the given order.

```
c(x1, y1)                 #use c() to combine several numeric vectors into one numeric vector
#> [1] 6 1 3 3 5 5
z1 <- c(x1, y1)
z1
#> [1] 6 1 3 3 5 5
length(z1)
#> [1] 6
```

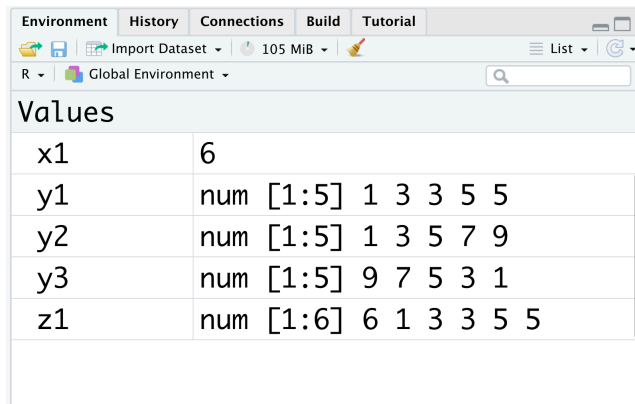
Since `x1` contains 1 numeric value and `y1` contains 5 numeric values, `z1` is a numeric vector of length 6 after combination.

For any vector, you can use the function `class()` to check its **class**. A class can be thought of as a “type,” providing a description about the vector and determining what functions can be applied to it.

```
class(x1)
#> [1] "numeric"
class(y1)
#> [1] "numeric"
class(z1)
#> [1] "numeric"
```

From the results, you can see that `x1`, `y1`, and `z1` are all numeric, which is the reason why they are called *numeric vectors*.

As introduced in Section ??, you can check the named objects via the environment panel as shown in Figure ??.



The screenshot shows the RStudio Environment panel with tabs for Environment, History, Connections, Build, and Tutorial. The Environment tab is active, displaying a table of objects. The table has two columns: the first column lists object names (x1, y1, y2, y3, z1) and the second column shows their corresponding values. The values are displayed as numeric vectors with their type (num) and length in brackets.

Values	
x1	6
y1	num [1:5] 1 3 3 5 5
y2	num [1:5] 1 3 5 7 9
y3	num [1:5] 9 7 5 3 1
z1	num [1:6] 6 1 3 3 5 5

Figure 21: The environment (I)

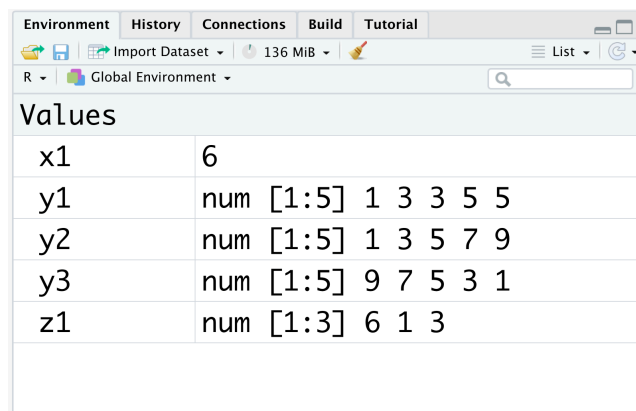
We can see that the environment panel has two columns, with the first column showing the list of object names and the second column showing the corresponding information for each object. The information includes the vector type (here *num* is short for numeric), the vector length, and the value(s) of the vector. Note that if the vector is of length 1 (for example `x1`), the environment will not show the type or the length.

In the last section, we have introduced how to change the value of an object by reassigning it. Similarly, you can also assign a new value, or new values, to `x1`. Notice that the values you are going to assign can have different length with the previous values. Let's see an example,

```
z1 <- c(6, 1, 3)
z1      #check the value of z1
#> [1] 6 1 3
```

Now, you can see that `z1` contains 3 numeric values, so `z1` is a numeric vector of length 3.

As expected, you can also view the newly assigned values of `z1` in the environment panel, as shown in Figure ??.



Values	
x1	6
y1	num [1:5] 1 3 3 5 5
y2	num [1:5] 1 3 5 7 9
y3	num [1:5] 9 7 5 3 1
z1	num [1:3] 6 1 3

Figure 22: The environment (II)

Finally, you can use the `vector(mode, length)` function to create a vector of certain mode and length.

```
vector("numeric", 4)
```

0.4.2 Operations and recycling rule

Since numeric vectors are purely made of numbers, you can do **arithmetic operations** between them, just like the fancy calculator in Section ??. If two or more vectors are of the **same length**, the operation is done **element-wisely**. In other words, R will perform the operation between elements in the same index of different vectors. First, let's create another vector `x2` of length 1 and compute the sum of `x1` and `x2`. Also recall that we've previously created a length-1 numeric vector `x1` with value 6.

```
x1
#> [1] 6
x2 <- 3
x1 + x2
#> [1] 9
```

Then obviously you will get 9! If you assign this operation to a name, you will create a new numeric vector with the *result* of the operation as the value.

```
s1 <- x1 + x2
s1
#> [1] 9
```

Here, `s1` is a length-1 numeric vector with value 9.

Similarly, you can create another vector `y2` of the same length as vector `y1`. Then, you can do operations between `y1` and `y2`.

```
y1
#> [1] 1 3 3 5 5
y2 <- c(2, 4, 1, 3, 2)
y1 * y2
#> [1] 2 12 3 15 10
```

The result is yet another length-5 vector. To check the calculation was indeed done element-wisely, you can verify that the value of the first element is $1 * 2 = 2$, and value of the second element is $3 * 4 = 12$, etc.

You can also store the result of multiplication for future use by assigning it to a name.

```
s2 <- y1 * y2
s2
#> [1] 2 12 3 15 10
```

To have the calculation done element-wisely, R requires two or more vectors to have the same length. However, there is an important **recycling** rule in R, which is quite useful and enables us to write simpler code. Specifically, if one vector is shorter than the other vector, R will **recycle** (repeat) the shorter vector until it matches in length with the longer one so that element-wise calculations can be done conveniently. This recycling is most often used for an operation between a **length>1** vector and a **length=1** vector. Let's see an example.

```
y1 + x1
#> [1] 7 9 9 11 11
```

From the result, you can see that `x1` is recycled five times to match in length with `y1`, becoming a length-5 numeric vector with five sixes. Subsequently, each element in `y1` is added by 6.



By now you have created several objects, and you may find that objects will not be saved in R if you don't assign their values to names, for example, the results

of `y1 + x1` is not shown in the environment.

The followings are a few additional examples you can try.

```
y1 * x2
y1 / 5
y2 - x1
```

0.4.3 Storage types (doubles and intergers)

Now, it is time to learn how numeric vectors are stored in R. To find the **internal storage type** of an R object, you can use the `typeof()` function.

```
my_num <- c(1.5, 3, 4)
typeof(my_num)           #check the internal storage type
#> [1] "double"
```

You can see that the internal storage type of `my_num` is **double**, meaning that `my_num` is stored as a **double precision** numeric value. In fact, R stores numeric vectors as double precision vectors by default. Let's see another example,

```
my_dbl <- c(3, 4)
typeof(my_dbl)           #check the internal storage type
#> [1] "double"
```

Different from `my_num` which contains a non-integer (1.5), all elements in `my_dbl` are integers. However, the storage type of `my_dbl` is still double, same as `my_num`. When all values of a numeric vector are integers (such as `my_dbl`), you can store it as an **integer vector**, which is also a numeric vector. To do this, you only need to put an "L" after each integer in the vector. Let's create an integer vector and check its storage type as well as its class.

```
my_int <- c(3L, 4L)
typeof(my_int)
#> [1] "integer"
class(my_int)
#> [1] "integer"
```

You can see that internal storage type of `my_int` is indeed of integer type, with the class of it being `integer` as well. It is also worth noting that the displaying value of `my_double` and `my_int` are the same.


```
my_double
#> Error in eval(expr, envir, enclos): object 'my_double' not found
my_int
#> [1] 3 4
```

You can also check the vector type and values in the environment. (as shown in Figure ??)

Values	
my_dbl	num [1:2] 3 4
my_int	int [1:2] 3 4
my_num	num [1:3] 1.5 3 4

Figure 23: Different storage types

From the picture above, you can see that the values of `my_int` are still 3 and 4, which are the same as those of `my_dbl`. The difference between these two vectors is that `my_int` is an integer vector since its internal storage type is integer, and such a storage type offers great *memory savings* compared to doubles.

Notice that a numeric vector's internal storage type is consistent. Say if you assign multiple numeric values to an object, even when you assign an "L" to most values, as long as the object obtains at least one decimal value, the vector will be stored in R as double. This rule also implies that any numeric vector containing at least one decimal value cannot be transformed to an integer vector.

```
my_num2 <- c(1.5, 3L, 4L)
typeof(my_num2)
#> [1] "double"
class(my_num2)
#> [1] "numeric"
```

Moreover, as you can see by running the code below, whenever you put an "L" after an decimal value, you will get the warning and the storage type will remain double.

```
my_num3 <- c(1.5L, 3L, 4L)
typeof(my_num3)
```

```
#> [1] "double"  
class(my_num3)  
#> [1] "numeric"
```

In conclusion, all the numeric vectors will be stored as `double` by default. If all values in a numeric vector are integers, you can convert this numeric vector into an integer vector, and the storage type of this vector will be `integer`, which can save memories compared to doubles. That being said, don't get confused: both double and integer vectors belong to numeric vectors.



Despite the differences between integers and doubles, you can usually ignore their differences unless you are working on a very big data set. R will automatically convert objects between integers and doubles when necessary.

0.4.4 Printing

Now, you have learned numeric vectors along with their possible storage types. In this part, let's discuss how you can customize the output digit of a number via printing. Let's start with `pi`, which is a mathematical constant you may be familiar with. `pi` is also an internal numeric vector available for use in R, meaning that it will appear in the environment panel without requiring you to assign it to a name.

```
pi  
#> [1] 3.141593
```

As you can see from the output, R prints out 7 **significant digits** by default, though in fact we need infinitely many digits to faithfully represent `pi`. To print out an object with a customized significant digit number, you can use the `print()` function that contains useful argument called `digits`, which controls the number of significant digits to be printed. Let's see the following examples.

You can try the following examples.

```
print(pi, digits = 20)           #print pi for 20 significant digits  
#> [1] 3.141592653589793116  
print(pi, digits = 4)           #print pi for 4 significant digits  
#> [1] 3.142
```



Note that the `round()` function also has an argument `digits`, which has a **different** meaning, representing the number of digits after the decimal point.

```

      (pi, digits = 4)           #print pi for 4 significant digits
#> [1] 3.142
      (pi, digits = 4)           #round pi to 4 decimal places
#> [1] 3.1416

```

You may be wondering whether happens if `digits` is larger than the number of the actual significant digits of a number. Let's try the following example.

```

print(1.2, digits = 5)
#> [1] 1.2

```

Clearly, the `print()` function will print out at most the significant digits of the number.

When you print a vector with more than one element, the same number of decimal places is printed for all elements. In this case, the `digits` parameter represents the **minimum** number of significant digits, and that at least one element will be encoded with that minimum number.

```

print(c(pi, exp(1), log(2)), digits = 4)
#> [1] 3.1416 2.7183 0.6931
print(c(pi, exp(1), log(2), exp(-5)), digits = 4)
#> [1] 3.141593 2.718282 0.693147 0.006738
print(c(20000, 1.2, 2.34), digits = 3)
#> [1] 20000.00 1.20 2.34

```

As you can imagine, the `print()` function will be very useful in creating tables that look more streamlined.

0.4.5 Exercises

Write the R code to complete the following tasks.

1. Create a numeric vector named `vec_1` with values (2, 4, 6, 8), get its length, find out its class, and get its storage type.
2. For the numeric vector `vec_2 <- c(1, 3, 7, 10)`, get the value of the 3rd element, multiple the 3rd element by 5, and verify the change.
3. Create a vector `vec_3` where each element is twice the corresponding element in `vec_1` minus half the corresponding element in `vec_2`.
4. Create an integer vector `int_1` that contains integers (2, 4, 6, 8). Check its class and storage type.
5. Print out the vector (e, e^2, e^3) with 5 significant digits.