

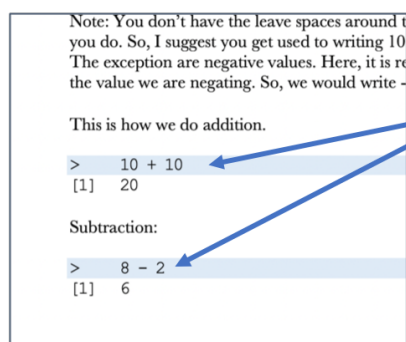
FASS512: First steps in R

Professor Patrick Rebuschat, p.rebuschat@lancaster.ac.uk

This week, we will do our first steps in R. Please work through the following handout at your own pace.

Three important things to remember:

1. As you complete the handout, please don't just read the commands, **please type every single one of them**. This is really important: Learning to program is like practicing a conversation in a new language. You will improve gradually, but only if you practice.



Every time you see these shaded lines, please **type the commands** either in the console or the script editor, as appropriate.

2. If you're stuck with something, please write down your questions (to share later in class) and **try to solve the problem**. Please ask your group members for support and, conversely, if another student is stuck, please try to help them out, too. This way, we develop a supportive learning environment that benefits everybody. In addition, get used to the Help pages in RStudio and start finding solutions online (discussion forums, online textbooks, etc.). This is really important, too. You will only really know how to do quantitative research and statistical analyses when you are doing your own research and dealing with your own data. At that point, you need to be sufficiently autonomous to solve problems, otherwise you will end up making very slow progress in your PhD.
3. Finally, if you don't complete the handout in class, **please complete the handout at home**. This is important as **we will assume that you know the material covered in this handout**. And again, the more you practice the better, so completing these handouts at home is important.

References for this handout

Many of the examples and data files from our class come from these excellent textbooks:

- Andrews, M. (2021). *Doing data science in R*. Sage.
- Crawley, M. J. (2013). *The R book*. Wiley.
- Fogarty, B. J. (2019). *Quantitative social science data with R*. Sage.
- Winter, B. (2019). *Statistics for linguists. An introduction using R*. Routledge.

Are you ready? Then let's start on the next page! 🐼

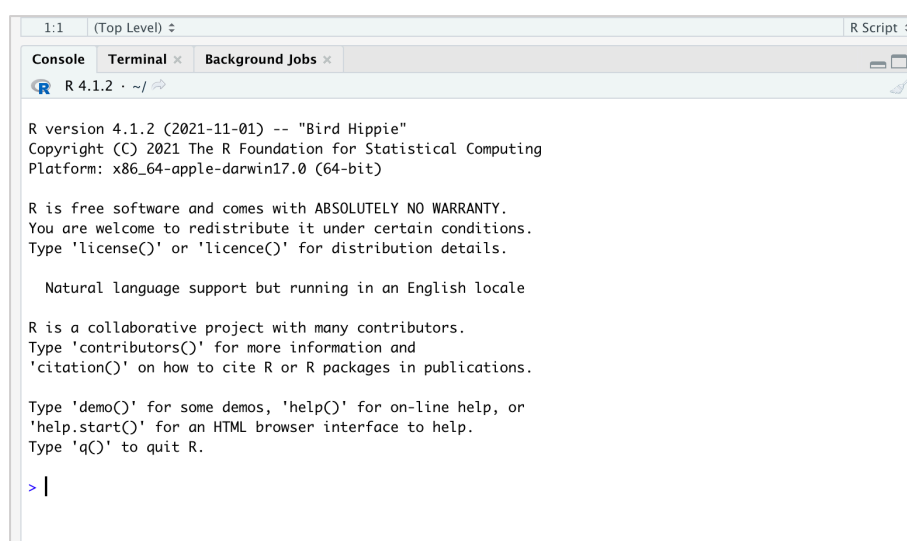
Step 1: *Using the R console*

R is a command-based system. This means: You type commands (such as the ones highlighted below), R translates the commands into machine instructions, which your computer then executes.

You can type the R commands directly into the console. Or, as you will see later, you can also type commands into the script editor and run the sequence of commands as a batch.

In the next section, we will try out writing commands in the Console pane.

Commands are typed at the command prompt `>`. We then press Return (Mac) / Enter (Windows), and our command is executed. The output of the command, if there is any output, will be displayed on the next line(s).



```

1:1 (Top Level) R Script
Console Terminal Background Jobs
R 4.1.2 ~

R version 4.1.2 (2021-11-01) -- "Bird Hippie"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

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

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |

```

Step 2: *Using R as a calculator*

Most textbooks recommend familiarizing yourself with R and RStudio by first using R as calculator. Let's try this out.

For example, if you type `10 + 10` in the command prompt and press Return (Mac) / Return (Mac) / Enter (Windows), the result will be displayed underneath `[1] 20`.

Please try out the following commands. In the task below, just type the commands in the shaded area (), then press Return (Mac) / Enter (Windows).

Note: You don't have to leave spaces around the operators (+, -, *, /, etc.), but the lines are more readable if you do. So it's better if you get used to writing `10 + 10` rather than `10+10`, even though the output is the same. The exception are negative values. Here, it is recommended not to leave a space between the minus sign - and the value we are negating. So, we would write `-3`, not `- 3`, even though it's the same.

Please try out all of the commands on your computer now.

This is how we do addition.

```
> 10 + 10
```

```
[1] 20
```

Subtraction:

```
> 8 - 2
```

```
[1] 6
```

Multiplication:

```
> 10 * 14
```

```
[1] 140
```

Division:

```
> 112 / -8
```

```
[1] -14
```

We can also use exponents, i.e. raising a number to the power of another number, e.g., 2^8 , as in the example below:

```
> 2 ^ 8
```

```
[1] 256
```

Square root:

This is done by means of a function called `sqrt()`. We will discuss functions later. For now, just add a numerical value in the parentheses.

```
> sqrt(4)
```

```
[1] 2
```

You can also combine `+`, `-`, `*`, `/`, `^` operators in your commands. By default, the precedence order of operations will be `^` followed by `*` or `/`, followed by `+` or `-`, just like in a calculator.

```
> 2 + 3 - 4 / 2
```

```
[1] 3
```

```
> 3 + 9 / 3 * 2 ^ 8
```

```
[1] 771
```

But: You can use brackets () to overcome the default order to operations:

Observe the effect of bracketing on the precedence order of operations in the two examples below.

Example 1a:

```
> 2 + 3 - 4 / 2
```

```
[1] 3
```

Example 1b:

```
> (2 + 3 - 4) / 2
```

```
[1] 0.5
```

Example 2a:

```
> 3 + 9 / 3 * 2 ^ 8
```

```
[1] 771
```

Example 2b:

```
> (3 + 9) / 3 * 2 ^ 8
```

```
[1] 1024
```

History of commands

In the Console pane, have you noticed that pressing the up and down arrows does not allow you to go through the different lines (as would happen in a text file, e.g. Word)? Instead, when you're at the command prompt `>`, pressing the up and down arrows allows you to move through the history of executed commands. This can save you a lot of time if you want to re-run one of the previous commands.

In the Environment, History, etc. pane, you can use the History tab to see your entire command history. If you click on any line in the History tab, it will re-run the command. Again, very helpful as it saves you a lot of typing.

Let's try this out.

First, use the up and down arrows to re-execute the following command:

```
> 2 + 3 - 4 / 2
```

```
[1] 3
```

Then, use the History pane to re-execute this command:

```
> sqrt(4)
```

```
[1] 2
```

Incomplete commands, and escaping.

What happens if you try to execute an incomplete command such as the one below?

Let's try this out.

```
> 10+
```

```
+
```

As you will notice, there is something missing (another number for the addition). Rather than giving us the result of the addition, R will just display a plus sign +. This means that the command is incomplete.

And: If you keep hitting Return (Mac) / Enter (Windows), you will just get more plus signs...

```
> 10+
```

```
+
```

```
+
```

```
+
```

```
+
```

How do to deal with an incomplete command?

You can either add the missing value, here 10, as in the following case:

```
> 10+
```

```
+
```

```
+
```

```
+
```

```
+
```

```
+ 10
```

```
[1] 20
```

Or you can press ESC without entering the missing value to complete the command.

To abort a command, press the ESC key.

Step 3: *Variables*

The next important step is to learn how to create variables and assign values to variables.

In general, the assignment rule is:

```
name <- expression
```

1. *Expression*. An expression is any R code that returns some value. This can be a single number, the result of a calculation, or a complex statistical analysis.
2. *Assignment operator*. The `<-` is called the assignment operator. This assigns everything that is to its right (the expression) to the variable on its left. Rather than typing `<` and the minus sign, you can also simply press the shortcut Option+- (Mac) or Alt+- (Windows).
3. *Name*. The name is simply the name of the variable.

Please run the following command in the Console.

```
> x <- 4 * 8
```

```
>
```

It appears that nothing happened; after all, there seems to be no output in the command prompt. However, something did happen after we executed the command `x <- 4 * 8`.

You will see this when you execute the following command.

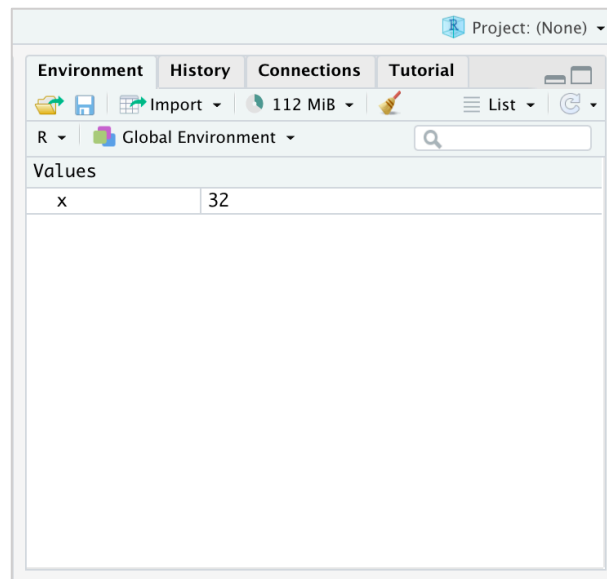
```
> x
```

```
[1] 32
```

As you see, the previous command `x <- 4 * 8` assigned the multiplication `4 * 8` to the variable `x`.

By typing the name of the variable `x` in the command line and pressing Return (Mac) / Enter (Windows), the value of the variable will be displayed, in this case 32 (being the product of `4 * 8`).

There is another way to confirm that you created a variable. If you check the Environment tab in the Environment, History, Connections, etc. pane at the top right of your RStudio window, you will now see that new variable listed.



Once you have created a variable, you can use it for other calculations just like you would with any other number.

```
> x * 36
```

```
[1] 1152
```

```
> x / 2 + (2 * 2 ^ 6)
```

```
[1] 144
```

We can also assign any of these values to new variables.

```
> y <- x * 28
```

We can now check whether the output of this new variable, which should be 896, as y equals 32 (the value of x) times 28.

```
> y
```

```
[1] 896
```

Naming your variables

The name has to follow certain naming conventions. The variable name can consist of letters (upper or lower case), numbers, dots, and underscores. However, it must begin with a letter or a dot that is not followed by a number.

The following would be fine, for example.

```
> y1357
```

```
> .Rxwy
```

```
> x_y_z
> abc_321
```

But these would not. Can you identify why?

```
> _x
> .1px
> a_b_c
> x-y-z
```

Even though you can use nonsense sequences such as the ones above, it is good practice to select variable names that are meaningful, short, without dots (use underscore `_` instead), and ideally in lowercase characters. For example:

```
> age <- 56
> income <- 101034
> is_married <- TRUE
> years_married <- 27
```

Step 4: *Data structures (1): Vectors*

Data structures are variables that refer to collections of values. There are different types of data structures (e.g., lists, matrices, arrays), but we will focus on two particularly important ones: vectors and data frames.

Vectors are one-dimensional sequences of values. They are very simple but fundamental data structures, as you will see below when we talk about data frames. For this reason, it's worth learning about vectors, how to create and manipulate them. Below we will cover numeric and character vectors.

How to create a vector

You can create vectors by using the `c()` function. The letter `c` stands for combine. All the items within brackets, separated by commas, will be assigned to the variable on the left of the assignment operator.

If you type the following command, you will create a vector with seven elements. This is called a numeric vector as the elements within the brackets are numbers.

```
> numbers <- c(2, 3, 5, 7, 11, 13, 17)
```


We can now use this vector to perform all kinds of operations. Try out the following, for example.

```
> numbers + 1
```

```
[1] 3 4 6 8 12 14 18
```

```
> numbers / 2
```

```
[1] 1.0 1.5 2.5 3.5 5.5 6.5 8.5
```

```
> numbers ^ 2
```

```
[1] 4 9 25 49 121 169 289
```

Note how the operations are applied to *each number* in the vector.

For any vector (number sequence), we can also refer to individual numbers that form the vector. We do this by means of indexing operations `[]`. For example, to get the first element of the vector, we can type the following. This will display the first element in the vector.

```
> numbers [c(1)]
```

```
[1] 2
```

But you can also extract more than one element from the vector, and even specify the order. For example:

```
> numbers [c(4, 1, 2)]
```

```
[1] 7 2 3
```

This will retrieve the fourth element in the vector (the number 7), the first element (2) and the second (3).

Or you can refer to consecutive set of elements in the vector, such as the fourth to the seventh elements. For this, simply type:

```
> numbers [c(4 : 7)]
```

```
[1] 7 11 13 17
```

We can also retrieve all elements with the exception of one. For this, we use the minus sign to exclude the vector element that we want to exclude, as in the following example.

```
> numbers [-1]
```

```
[1] 3 5 7 11 13 17
```

Finally, we can also exclude a sequence of elements by adding the minus sign before the `c()` function. This will exclude from the output all elements that are specified within the brackets.

```
> numbers [-c(1, 3, 5, 7)]
```

```
[1]  3  7 13
```

The `numbers` vector is a sequence of numbers. We can find out the type of vector by using the function `class()`. This will confirm that `numbers` is a numeric vector.

```
> class(numbers)
```

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

We can also create vectors with elements that are character strings. Here, each element needs to be surrounded by quotation marks (single or double), as in the example below.

```
> colleges <- c('bowland', 'cartmel', 'county', 'furness', 'fylde',  
  'graduate', 'grizedale', 'lonsdale', 'pendle')
```

```
> colleges
```

```
[1] "bowland"  "cartmel"  "county"   "furness"  "fylde"
```

```
[6] "graduate" "grizedale" "lonsdale" "pendle"
```

This type of vector is character, as you can verify by typing the following.

```
> class(colleges)
```

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

You can index character vectors, just like numeric vectors.

```
> colleges[3]
```

```
[1] "county"
```

```
> colleges[-4]
```

```
[1] "bowland"  "cartmel"  "county"   "fylde"    "graduate"
```

```
[6] "grizedale" "lonsdale" "pendle"
```

```
> colleges[2:5]
```

```
[1] "cartmel" "county"  "furness" "fylde"
```

But you cannot perform arithmetic functions on character vectors, of course.

```
> colleges * 2
```

```
Error in colleges * 2 : non-numeric argument to binary operator
```

Coercing vectors

In vectors, all of the elements must be of the same type. For example, you cannot have a vector that has both numbers and character strings as elements. If you try to create a vector with both numbers and character strings, then some of your elements will be coerced into other types.

If you type the following, you will see that the attempt to create a mixed vector with character strings (bowland, cartmel, county, fylde) and numbers (1, 2, 5, 7, 8) converted the numbers into character strings, as evidenced by the quotation marks. You cannot perform calculations on these numbers as R interprets them as text strings.

```
> c('bowland', 'cartmel', 'county', 'furness', 'fylde', 1, 2, 5, 7, 8)
```

```
[1] "bowland" "cartmel" "county" "furness" "fylde" "1"
```

```
[7] "2" "5" "7" "8"
```

Finally, you can also combine vectors using the `c()` function. For example, we can create a new vector called `new_numbers` by combining the original numbers vectors and adding the squares and cubes of numbers.

```
> new_numbers <- c(numbers, numbers ^ 2, numbers ^ 3)
```

```
> new_numbers
```

```
[1] 2 3 5 7 11 13 17 4 9 25 49 121
```

```
[13] 169 289 8 27 125 343 1331 2197 4913
```

We have now produced a new vector `new_numbers` with 21 elements. These don't fit all in a line and so are wrapped over two lines. The first row displays elements 1 to 12, and the second row begins with element 13. Now you can also see the meaning of the `[1]` on the output. The `[1]` is just the index of the first element of the vector shown on the corresponding line. `[1]` refers to the first element in our vector (the number 2), and `[13]` refers to the 13th element in our vector (169).

Naming vectors

The elements of a vector can be named, too. Each element in our vector can have a distinct label, which can be useful.

```
> ages <- c(bob = 27, bill = 34, charles = 76)
```

```
> ages
```

```

bob    bill  charles
27      34    76

```

We can now access the values of the vector by the label or by the index as before.

```
> ages['bill']
```

```

bill
34

```

```
> ages[2]
```

```

bill
34

```

We can also add names to existing vectors by using the `names()` function. In the following example, we first assign new values to the vector `ages`. This will delete the previous numbers and labels. We then assign names to this vector using the `names()` function.

```
> ages <- c(23, 54, 8)
```

```
> names(ages) <- c("michaela", "jane", "jacques")
```

```
> ages
```

```

michaela    jane  jacques
23         54      8

```

Missing values

Last but not least. Sometimes, we have missing values in our data. In R, missing values are denoted by `NA`. This is not treated as a character string but as a special symbol.

You can insert a placeholder for a missing value into a numeric or character vector by simply typing `NA` in the list of elements.

```
> a <- c(1, 5, 7, NA, 11, 14)
```

```
> a
```

```
[1] 1  5  7 NA 11 14
```

```
> b <- c('michaela', 'bill', NA, 'jane')
```

```
> b
```

```
[1] "michaela" "bill"      NA          "jane"
```

Step 5: *Data structures (2): Data frames*

Data frames are probably the most important data structure in R. They are the default form for representing data sets for statistical analyses.

Data frames have a certain number of columns, and each column has the same number of rows. Each column is a vector, and so data frames are essentially collections of equal-length vectors. (This is also why we spent so much time on vectors above...)

There are two ways of creating data frames. We can create a data frame in R by importing a data file, usually in .csv or .xlsx format. (We will discuss how to import files next week.)

Or we can use the `data.frame()` function to create a data frame from scratch, as in the following example.

```
> data_df <- data.frame(name = c('bill', 'jane', 'jacques'), age =
  c(23, 54, 8))
> data_df

  name age
1  bill  23
2  jane  54
3 jacques   8
```

As you can see, we have now created a data frame with two columns (name, age) and three rows (displaying the name and age of each person, Bill, Jane and Jacques). The columns are our variables and the rows are our observations of these variables.

We can refer to specific elements of the data frame by using indices. In the example below, there are two elements within the index `[]`, one for the rows, one for the columns. These are separated by a comma `[,]`.

For example, to refer to the element that is in the second row, first column, you would type the following.

```
> data_df[2, 1]

[1] "jane"
```

You can also retrieve multiple elements. One way of doing this is by leaving one of the indices blank.

If you leave the first index blank (e.g., `[, 1]`), then you are telling R that you want the information from all the rows. In the example below, you retrieve the information that is in all the rows that are in column 1.

```
> data_df[ , 1]
```

```
[1] "bill"    "jane"    "jacques"
```

In contrast, if you leave the second index blank (`[2,]`), you will retrieve the information found in the second row across the two columns.

```
> data_df[2 , ]
```

```
      name age
2 jane   54
```

We can also be more specific. For example, you can refer to the first and third row of the second column, we would type:

```
> data_df[c(1, 3), 2]
```

```
[1] 23  8
```

On the other hand, we can also refer to a column by name. To do this, we need to use the `$` notation.

```
> data_df$name
```

```
[1] "bill"    "jane"    "jacques"
```

```
> data_df$age
```

```
[1] 23 54  8
```

(Did you notice that RStudio will automatically suggest the variable names after you typed the `$`? The code completion feature in RStudio makes writing and executing code much easier!)



Step 6: Functions

While data structures hold data in R, functions are used to do things with the data. You have already encountered a few functions above, namely `sqrt()`, `c()`, and `data.frame()`.

Across all R packages and R's standard library, there are tens of thousands of functions available for you to use. However, most of our analyses in this course will require only a relatively small number of functions.

Below is a general introduction to functions; we will cover functions in more detail as we progress through this course.

Functions tend to have the following structure:

```
function(argument 1, argument 2, argument 3, ...)
```

We can think of functions as actions and arguments as the inputs, i.e. something the functions act on. Most functions require at least one argument. If they have more than one argument, these are separated by commas as in the example above.

For example, see what happens when you type the following two commands.

```
> sqrt()  
  
Error in sqrt() : 0 arguments passed to 'sqrt' which requires 1
```

```
> sqrt(4)
```

```
[1] 2
```

The function `sqrt()` requires an argument; if you fail to supply an argument you will get an error message (`Error in sqrt()...`)

Here are another few functions that are helpful for our analyses.

We can count the number of elements in a given vector by using the `length()` function.

```
> length(new_numbers)
```

```
[1] 21
```

We can calculate sum, mean, median, and standard deviation as follows. (More on this in future sessions when we discuss exploratory data analysis.)

```
> sum(new_numbers)
```

```
[1] 9668
```

```
> mean(new_numbers)
```

```
[1] 460.381
```

```
> median(new_numbers)
```

```
[1] 25
```

```
> sd(new_numbers)
```

```
[1] 1152.162
```

Nested functions

Functions can also be nested, as in the following example.

```
> round(sqrt(mean(new_numbers)))
```

```
[1] 21
```

Here, we use three functions, each nested within the other. In the example, we first calculated the mean of the vector `new_numbers` (460.381), then the square root of this value and finally rounded it. We could have done the same calculation in three steps, as in the example below, but nesting the functions in a single command allows us to be more efficient.

Compare the following the following three commands to the one immediately above, `round(sqrt(mean(new_numbers)))`. As you will, you reach the same goal, but in one case you do it in one step, in the other case in three steps. There are many ways of doing things in R.

```
> mean(new_numbers)
```

```
[1] 460.381
```

```
> sqrt(460.381)
```

```
[1] 21.45649
```

```
> round(21.45649)
```

```
[1] 21
```

Optional arguments

In some cases, functions can also take an additional argument. A good example is the `mean()` function, which can an additional argument called `trim`. If we add the `trim` argument to the `mean` function, the command will first remove a certain proportion of the extreme values of the vector and then calculate the mean. This is very useful when we are dealing with outliers in our data, for example. (We will talk more about outliers in future sessions.)

In order to trim observations, we need to specify a value between 0 to 0.5. This will trim the highest and the lowest values before calculating the mean. Naturally, a trim value of 0 means you're not trimming anything, so the value assigned to trim should be greater than 0. For example, a value of 0.1 means you're trimming the 10% highest and lowest observations, 0.2 means you're trimming the 20% highest and lowest, and so forth.

```
> mean(new_numbers)
```



```
[1] 460.381
```

```
> mean(new_numbers, trim=0.0)
```

```
[1] 460.381
```

```
> mean(new_numbers, trim=0.1)
```

```
[1] 150.1765
```

```
> mean(new_numbers, trim=0.2)
```

```
[1] 66.92308
```

```
> mean(new_numbers, trim=0.3)
```

```
[1] 44.11111
```

```
> mean(new_numbers, trim=0.4)
```

```
[1] 26.2
```

```
> mean(new_numbers, trim=0.5)
```

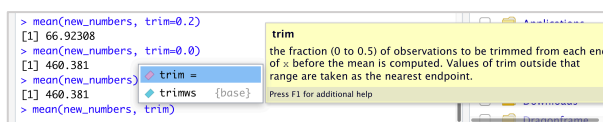
```
[1] 25
```

Function help pages

RStudio has very helpful pages for the available functions. This is useful when you're not sure if a function requires an argument, or if you're in doubt about the use of arguments such as `trim`.

On the one hand, you will see that support readily appears as you are typing in the function.

As you can see in the screenshot below, the moment we type `> mean(new_numbers, trim...` the following windows popped up to provide guidance on `trim`.



You can access the help page for a given function in different ways.

The most efficient one is by typing a question mark and the function name in the command line. If you now press Return (Mac) / Enter (Windows), this command will also open the help page for the function.

```
> ?mean()
```

Alternatively, you can use the `help()` function in the command line, as below.

```
> help('mean')
```

Last but not least, you can also go to the search line of the Help tab in the Files, Plots, Packages, Help pane (bottom right of the screen) and type the name of the function there (mean). There is a useful shortcut to search R Help, namely Ctrl+Option+F1 (Mac) and Ctrl+Alt+F1 (Windows).

In each of the cases above, RStudio will open the help page for the function in question in the Help tab.

The screenshot shows the RStudio environment with the following components:

- Top Panel:** Includes the menu bar (File, Edit, Session, View, Help) and the toolbar with icons for file operations, running code, and environment management.
- Source Editor:** Contains R code for generating random numbers and calculating their mean and standard deviation. The code includes comments in Chinese and uses functions like `set.seed`, `runif`, `mean`, `sd`, and `sqrt`.
- Console:** Displays the output of the code, showing the generated random numbers, their mean, standard deviation, and the results of the `mean` and `sd` functions. The output is in Chinese.
- Environment:** Shows the current environment with variables `new_numbers` and `mean_new_numbers`.
- Plots:** The plot pane is empty, showing only the axes.
- Files:** The file pane shows the current project directory.
- Help:** The help pane is empty, showing only the search bar.
- View:** The view pane is empty, showing only the search bar.
- Presentation:** The presentation pane is empty, showing only the search bar.