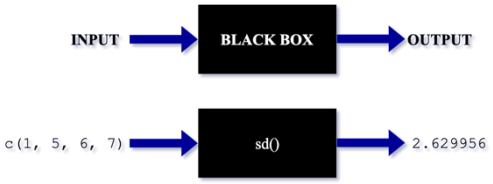
3 Functions

Functions are an extremely important concept in almost every programming language; R is not different. After learning what a function is and how you can use one, you'll take full control by writing your own functions.

Introduction to Function

In this chapter we'll have a closer look at very powerful concept in R. Functions. Not surprisingly, you've already used functions before. Remember the time you created a list? You used list() function. Or the time you wanted to display the contents of a variable? You used the print() function. But what are function and how do they work? You can think of a function as some kind of black box. You give an input to the black box the black box processes this input and it returns some output. Let's have a look at this black-box principle with a specific example. The R sd() function calculates the standard deviation of a vector. Our black box in this case is the sd() function. If you give the sd() function a vector containing 1, 5, 6 and 7 as an input the number 2 point 63, the standard deviation of these 4 values, will be your output.



How can you use the function sd() in R? You already know how! Simply type sd followed by parentheses. Inside the parentheses, you specify the so called function arguments. These are the inputs to tour functions. In our case, we have a single argument, the vector containing four values. We could just as well assign the input vector to a variable, say the variable values, and then call sd on values. In both cases, the value 2 point 63 gets printed to the console. That's because we did not assign the result on the function to a variable. If you want to reuse the result of the function, simply use the assignment operator as you did before so many times. Let's assign the output of our function to a variable my_sd. If we now print my_sd to the console, we see that it contains 2 point 63.

Call function in R

```
> sd(c(1, 5, 6, 7))
[1] 2.629956

> values <- c(1, 5, 6, 7)
> sd(values)
[1] 2.629956

> my_sd <- sd(values)
> my_sd
[1] 2.629956
```

Here I assumed that everybody knows how to use the sd() function. For the sd() you can guess that you have to input a vector, but there are many functions out there for which the usage is less straightforward. For information on what a function does and how it should be used, you can look up the documentation pf the R function using the help function. For example, for the sd function, we type help(sd), or question mark sd. These are equivalent. These commands will guide you to RDocumentation. Function documentation presents a lot of information. If we have a look at the "Usage" section, we see that the sd function actually takes two arguments, x and na (dot) rm. A strange thing here is that na (dot) rm is followed by an equals sign and FALSE, while x is not.

Function documentation

```
help(sd)
?help

sd(x, na.rm = FALSE)
```

sd {stats} R Documentation

Standard Deviation

Description

This function computes the standard deviation of the values in x. If na.rm is TRUE then missing values are removed before computation proceeds.

Usage

```
sd(x, na.rm = FALSE)
```

Arguments

x	a numeric vector or an R object but not a factor coercible to numeric by as.double(x).
na.rm	logical. Should missing values be removed?

Details

Like var this uses denominator *n - 1*.

The standard deviation of a length-one or zero-length vector is NA.

See Also

var for its square, and mad, the most robust alternative.

Examples

```
sd(1:2) ^ 2
```

[Package *stats* version 3.6.1 Index]

Well, this is a bummer. Asking for help on the sd function only gave us more questions. First off, the first argument is called x, but we didn't specify it anywhere when calling sd on the values variable. How did R know what we meant? Second, what's up with this = FALSE for the na (dot) rm argument? And finally, how come sd(values) worked fine although sd seems to need two arguments? Do not despair, all of these question will be solved in a moment! When you call an R function, R has to match your input values to the function's arguments. To put it differently, R has to know that by values you mean the argument x of the sd() function. This is because R matched the values to the x argument by position. Values is the first element inside the parentheses, so R knows that you mean the first argument of the sd() function, which is x.

However, it doesn't have to be this way. It would be perfectly equivalent to match the arguments by name, by specifically saying that we want the x argument to be values. We can do this by using equal sign. The result is exactly the same. Now what's up with this na (dot) rm argument? The documentation shows that na (dot) rm is a logical value, indicating whether or not missing values should be removed. Let's experiment with this first, by adding an NA to the values vector and calling the sd() function once more with the values argument. The result is simply NA, as the sd function did not remove the missing values before calculating the standard deviation. This is because by default, the na (dot) rm is FALSE, causing sd to not remove the missing values.

```
na.rm argument
> values <- c(1, 5, 6, NA)
> sd(values)
[1] NA
```

That's exactly what the Usage section of sd's documentation tells us: na (dot) rm is FALSE indicates that by default NA's will not be removed. So, if you do not specify the na (dot) rm argument, na (dot) rm will be set to FALSE. For the case where the values vector contains a missing value, an NA, we'll want to set the na (dot) rm to TRUE. The sd function will then remove missing values before calculating the actual standard deviation. We can do this by letting R match the arguments by position. R knows that we want to set the x argument to values and the na (dot) rm argument to TRUE because of the order in which we set the function's input. Matching by name is also possible. We explicitly say that the na (dot) rm argument must be TRUE. Notice from this last expression that R knows how to handle a mix of matching by position and by name: the first argument was matched by position, while the second one was matched by name.

```
na.rm argument
> sd(values, TRUE)
[1] 2.645751
```

```
na.rm argument
> sd(values, na.rm = TRUE)
[1] 2.645751
```

This also solves our third question: sd(values) does not throw any errors although we didn't define the na (dot) rm argument: R sees that we haven't specified it, so it takes the default value. However, if we had decided to leave the x argument unspecified, for example by simply calling sd() without arguments. We will get an error: argument x is missing, with no default. Remember from the Usage section of the documentation that x did not have a default value, while na (dot) rm did. This tells us that function arguments for which no default is specified, have to be specified by the user of the function, otherwise an error is likely to occur. Before wrapping up this introduction of functions, I want to point you to a very useful function, the args() function. This is a function ti learn about the arguments of a functions without having to read through the entire documentation. For the sd() function, we can use args(sd). The output tells us that the first argument, x, has no default arguments, while na (dot) rm, the second argument, is FALSE by default.

```
> sd()
Error in is.vector(x) : argument "x" is missing, with no default
```

```
> args(sd)
function(x, na.rm = FALSE)
NULL
```

Functions may be a daunting concept at first, but knowing all about them is important to get a good understanding of R in general. R functions are used literally all the time. Let us recap on three key ideas. First of all, functions work like a type of black box: you give some values as an input, the function processes this input and generates an output. Next, R matches function arguments by position or by name, and finally, some function arguments can have a default value, which can be overridden. If you do not specify the value of an argument that has no default, typically an error will occur.

Writing Functions

You now know about different ways to use R functions, but this is not the end. You can also write your own R functions. You might wonder when you would want to this. Well, this is mostly a question of experience, but there are some guidelines. A function typically serves a particular need or solves a particular problem, without having to care about how the function does this. Remember the 'black-box principle' I mentioned before? If you're writing your own functions, you are writing your own black box that takes inputs and generates an output. How the black box goes about solving its task is not important once you've written the function. You want to be able to use this function just as if it was a standard R function, such as mean(), sd() or list().

You don't know how these work under the hood either, do you? Writing your own functions is super simple as soon as you know the syntax. Let's define a function that calculates the triple of its input, called triple. In a black-box manner, it would look like this: a numeric goes in, the triple function does its magic, and the triple of the numeric comes out. How does this look in R code? You use the function construct for this. This function recipe reads itself as: create a new function, my_fun, that takes arg1 and arg2 as arguments and performs the code in the body on these arguments, eventually generating an output. On our case, we want our function to be called triple, so let's go ahead and take that step. Next, we also know that our triple function will have a single input, a number. We replace the arg1, arg2 part by a single argument, named x. We're almost there. What do we want the body, the function's actual code, to be? To calculate the triple of x, we simply put 3*x.


```
The triple() function
my_fun <- function(arg1, arg2) {
  body
}
```

```
The triple() function
triple <- function(arg1, arg2) {
  body
}
```

```
The triple() function
triple <- function(x) {
  3 * x
}
```

If we execute this function definition, a new object gets defined in our workspace, triple. Now, let's go ahead and calculate the triple of 6. Just like we did before with R's built-in functions, we use standard parentheses. If we call triple(6), R figures out that the x argument corresponds to the value 6. Next, the function's body is executed, calculating 3 times 6. The result is 18. How does R know that it has to return this value? That's because the last expression evaluated in aa R function becomes the return value.

```
ls()
[1] "triple"
```

```
triple(6)
[1] 18"
```

You can also explicitly specify the return value, by using the return statement. Let's change the function body to use a intermediary value y. Inside the function, we assign to y the triple of x and next, we return this value y. If we source this function,

```
triple <- function(x) {
  y <- 3 * x
  return(y)
}
```

and call the triple function on the variable 6, we get the exact same result, 18. Using a return at the end of your function body is not always useful, but there are other cases where the return statement will come in handy. We'll learn about them in a bit. Let's try something different now. Suppose we want to write a function, called math_magic. that takes two numbers as inputs, and calculates the sum of the product and the division of both numbers. So if we put in 4 and 2 we want it to return (4 times 2) plus (4 devided by 2), which is 10.

Let's start over from our function recipe. We replace my_fun by math_magic and change the arguments of the function to have two inputs, a and b. By the way, I'm just choosing these two arguments names, but you can choose other names as well, as long as they are consistent with the function body. Finally, we modify the body. We don't need to include a return statement. Sourcing this function and calling it on the numbers 4 and 2 gives us the result we expect. Great!

```
The math_magic() function
my_fun <- function(arg1, arg2) {
  body
}
```

```
The math_magic() function
math_magic <- function(a, b) {
  a*b + a/b
}
```

```
> math_magic(4, 2)
[1] 10
```

Let's experiment some more. What happens if we call math_magic with only one argument? We get an error because the argument b is missing with no default. We could solve this by making the second argument of the math_magic function optional. We do this by adding default value, say, 1, to the argument list of the function using the equals sign. Sourcing the function definition again and calling math_magic(4) now, gives us 8. Because the b argument was not specified, R set b to 1 inside the function, so 4 times 1 plus 4 divided by 1 was computed, resulting in 8.

```
> math_magic(4)
Error in math_magic(4) : argument "b" is missing, with no default
```

```
Optional argument
math_magic <- function(a, b = 1) {
  a*b + a/b
}
```

```
> math_magic(4)
[1] 8
```

Let's now call the math_magic function with the numbers 4 and 0. Thr result is Inf, R's way of saying infinity. That's because R divided 4 by 0 in the second part of the calculation, which leads to infinity. Suppose we want to guard our function against this misuse of the math_magic function, by having the function return 0 when the b argument is 0. We can simply extend our function with an if-test with a return statement inside. If we now call the math_magic function with the second argument equal to 0, the condition for the if-test is true and we simply return zero. The return statement, similar to the break statement in a for and while loop, returns 0 and the rest of the function body is ignored. The a times b plus divided by b part of the function is never reached in this case. Using the return statement, which proves to be quite useful here, we can halt the execution virtually anywhere we want. I guess those were my 2 cents on writing functions.

```
> math_magic(4, 0)
[1] Inf
```

```
Use return()
math_magic <- function(a, b = 1) {
  if(b == 0){
    return(0)
  }
  a*b + a/b
}
```

```
> math_magic(4, 0)
[1] 0
```

R Packages

Because you've learned so much in this chapter by now, let us do a brief recap. First, you've learned about the different ways to use R's built-in functions. Next, you took full control by actually creating your own R functions. You can use these newly defined functions just as we use R's built-in functions like mean, list and sample, just to name a few. But, wait? How come these functions are 'built in'? These function are not in my workspace, so how on earth does R know where to find these functions mean, list and sample? Well. All of these built-in functions are part of R packages that are loaded in your R session. R packages are bundles of code, data documentation, and tests that are easy to share with others. For example, the mean, list and sample functions are all part of the base package, which contains the basic functionality to use R. Another example package, specifically for data visualization, is the ggvis package. It contains function such as layer_points, scale_nominal and add_axis.

Before you can use a package, you will first have to install it. The base package is automatically installed when you install R. The ggvis package on the other hand won't come with the bundled R installation. But fear not! You can easily install it from inside R, using the install (dot) packages function, which, by the way is a function of the utils package. This function goes to CRAN. CRAN is short for the Comprehensive R Archive Network, a repository where thousands of packages are available. The function downloads the package file and installs the package on your system. All of this is done with this single command in R, pretty cool right? We've now installed the ggvis package, but we can't use it yet. To do that, we'll have to actually load the package into our current R session. When R loads a package, it actually attaches it to the search list. This is a list of packages and environments that R looks through to find the variable or function you want to use. To have a look at this list, you can use the search() function. Whenever you execute code that depends on any other variable or function, R goes through all these packages one after the other to find it. Apart from all the packages that are loaded into our R session, we also see ".GlobalEnv"; this list is our own workspace, where the user-defined R objects live.

```
install.packages("ggvis")
```

```
> search()
[1] ".GlobalEnv" ... "Autoloads" "package:base"
```

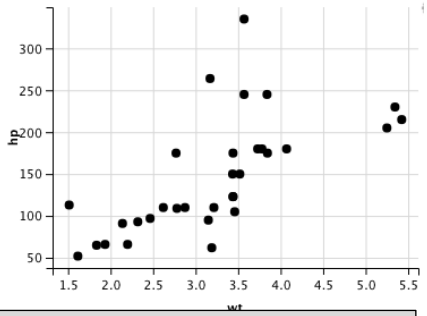
You can also see that the search() path already contains a bunch of packages. When R is started, it loads 7 packages on default, among which is the base package. Others are the utils, datasets and methods package. That's why, when you start R, you can use the mean function, or the install (dot) packages function without having to explicitly load the package. The ggvis package that we've installed earlier, however, won't be loaded automatically. If we try to access the ggvis function, for example, R will return an error, telling us that the function ggvis could not be found. That's because ggvis is not yet in the search list. To access ggvis' functionality, we'll have to load the package using the library command. This command takes the name of the package and adds the package to the search list, right after the global environment, making all the functions, data and pre-compiled code it contains available and ready to be used. We can check this by running search() once more. Indeed, ggvis is now part of the search list. If we now execute the same command, a pretty awesome graph shows up!

```
> ggvis(mtcars, ~wt, ~hp)
Error: could not find function "ggvis"
```

```
> library("ggvis")
> search()
[1] ".GlobalEnv" "package:ggvis" ... "package:base"
```

Now R could find the function in the list of attached packages and use it to create this nice plot. Before you proceed to the exercises, I want to tell you about the require function. Just like the library function, require loads packages into your R session. The only difference appears when you're trying to load a package that is not yet installed. Let's say you want to load the data (dot) table package, a package to perform data manipulation, but that this package is not yet installed.

```
> ggvis(mtcars, ~wt, ~hp)
```



If you call library(data.table). R throws an error in this case.

```
> library("data.table")
Error in library("data.table") : there is no package called 'data.table'
```

However, when you execute require(data.table), you get a warning. Also, the result of this require function will be FALSE if attaching the package failed. This is a good alternative when you want to avoid errors, for example when you're attaching packages dynamically inside functions.

```
> require("data.table")
Warning message: ...
```

```
> result <- require("data.table")
Loading required package: data.table
Warning message: ...
> result
[1] FALSE
```

So to wrap up: the install.packages() function installs packages for you, while library() and require() load them for you. And when you load packages, you're attaching them to a search list, making them available in your current R session. Before you start writing your own functions, first do a quick search for packages that do the same thing. You can simply install and load the package, have a look at the documentation and avoid having to rewrite a bunch of code that's already been written. I'm not saying that you should never write your own functions, but for common problems such as data manipulation or visualization, there are some pretty neat packages out there that will get you up and running in to time.