Writing functions in R

Ákos Bede-Fazekas

2022.01.27

Preparation

Please

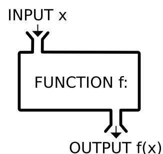
- download the code from **tinyurl.com/r-functions-code**
- open the code in RStudio (or in your preferred code editor)
- and run the code (with Ctrl+Enter) that is shown in the presentation

Also, if you want, please

- download this presentation from **tinyurl.com/r-functions-pres**

What is a function?

A function is a **block of code** which only runs when it is **called**.
You *can* pass data ("**parameters**") into a function.
A function *can* return data as a result ("**return value**").



Then,

- what is a block of code?
- how to call it?
- how to pass data?
- how to return data?

Block of code

Any part of the R code that is surrounded by curly brackets.

```
writeLines("Hello!")
writeLines("This is out of the block.")
{
  writeLines("This is inside the block.")
  writeLines("The lines within the block are run at once.")
}
writeLines("Now we are again out of the block.")
```

Indentation: core/body of the block are typically indented by 2 or 4 spaces or a tab.

Block of code

```
writeLines("Hello!")
```

```
>  Hello!
```

```
writeLines("This is out of the block.")
```

```
>  This is out of the block.
```

```
{
  writeLines("This is inside the block.")
  writeLines("The lines within the block are run at once.")
}
```

```
>  This is inside the block.
>  The lines within the block are run at once.
```

```
writeLines("Now we are again out of the block.")
```

```
>  Now we are again out of the block.
```

Why to use blocks?

Blocks can embrace several lines of code that are connected to each other. Typically used in

- conditional statements (`if`, `else`)
- loops (`for`, `while`)
- error handling (`tryCatch`)
- functions (`function`)

But, in theory, blocks can also be used anywhere else in the script.

- rarely used
- can be collapsed in the code editor

Collapse blocks

Open (lines 6-9):

```
 3
 4  writeLines("Hello!")
 5  writeLines("This is out of the block.")
 6  {
 7    writeLines("This is inside the block.")
 8    writeLines("The lines within the block are run at once.")
 9  }
10  writeLines("Now we are again out of the block.")
11
```

Collapsed (line 6):

```
 3
 4  writeLines("Hello!")
 5  writeLines("This is out of the block.")
 6  {⟺}
10  writeLines("Now we are again out of the block.")
11
```

Blocks vs. functions

A **block** is run when R reaches it during reading the code.
A **function** is not run when R reaches it but is "defined"/"declared" (i.e. created). Can be run later, when "called".

A **block** hasn't got name → we cannot call it.
A **function** has got name → we can call it by its name.

Defining a function

Defining a function = adding name to a block.

```
writeLines("Hello!")
writeLines("This is out of the block.")
named_block <- function() {
  writeLines("This is inside the block.")
  writeLines("The lines within the block are run at once.")
}
writeLines("Now we are again out of the block.")
```

Two differences:

- `named_block <- function()` in the code
- output is not written to the screen (since the block was not run)

Defining a function

```
writeLines("Hello!")
```

```
>  Hello!
```

```
writeLines("This is out of the block.")
```

```
>  This is out of the block.
```

```
named_block <- function() {
  writeLines("This is inside the block.")
  writeLines("The lines within the block are run at once.")
}
writeLines("Now we are again out of the block.")
```

```
>  Now we are again out of the block.
```

Defining a function

```
named_block <- function() {
  writeLines("This is inside the block.")
  writeLines("The lines within the block are run at once.")
}
```

- `named_block`: it is the name of the block/function (we'll use this name later to call the function).
- `function()`: it is the syntax of function definition. The parentheses are needed (these will contain the parameters).
- `<-`: value assignment similar to creating a variable (e.g. `x <- 5`)

Functions vs. variables

```
my_variable <- 42
my_function <- function() {
  writeLines("This is the body of the function.")
}
```

A function is similar to a variable:

- it has name
- a value (i.e., the block/“body of the function” + the parameters if any) is assigned to the name
- after the assignment we can get its content as many times as we want
- we can copy and delete it

Getting the content, copy and delete

We can get the content (i.e., the value of the variable or the body+parameters of the function) by typing its name.

```
my_variable
```

```
>  [1] 42
```

```
my_function
```

```
>  function() {
>    writeLines("This is the body of the function.")
>  }
```

Getting the content, copy and delete

We can copy the function similarly to copying a variable.

```
my_variable2 <- my_variable
my_function2 <- my_function
```

We can delete it with the `rm()` function.

```
rm(my_variable2)
rm(my_function2)
my_variable2 # error
```

```
> Error in eval(expr, envir, enclos): object 'my_variable2'
not found
```

```
my_function2 # error
```

```
> Error in eval(expr, envir, enclos): object 'my_function2'
not found
```

Calling a function

Getting the function body is not what we want!
We need to execute the code inside the function body by “calling the function”.

Call a function by typing its name followed by parentheses.

```
my_function()
```

```
>  This is the body of the function.
```

Functions in R

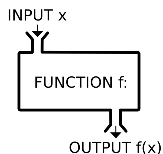
There are several functions in R:

- built-in functions installed with R
  - e.g. `mean()`,
  - `lm()`,
  - `cbind()`,
  - `paste()` etc.
- functions installed with packages
  - e.g. `rda()`,
  - `lmer()`,
  - `ggplot()` etc.
- functions that we defined
  - e.g. `my_function()`

There are not so much differences between these categories.

What is a function?

A function is a **block of code** which only runs when it is **called**.
You *can* pass data ("**parameters**") into a function.
A function *can* return data as a result ("**return value**").



Then,

- ~~what is a block of code?~~
- ~~how to call it?~~
- how to pass data?
- how to return data?

Return value - 3 categories

Some functions return one or more values. Some don't.
And some do but do it invisibly.

Without return value

Functions that doesn't return a value:

- e.g. `writeLines()`
- `plot()`
- `save()`
- `str()`
- `set.seed()`
- `my_function()` etc.

Without return value

Functions that doesn't return a value

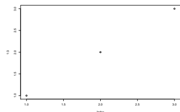
- are called "procedures" in the programming terminology,
- their result is `NULL`,
- they do something in the background ("side effect")

Possible side effects:

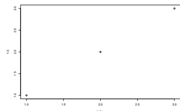
- writing to the console
- plotting to the screen
- opening/closing/writing to files (or any other connections)
- modifying something in the background (variables, options etc.)

Without return value

```
plot(1:3)
```



```
returned_value <- plot(1:3)
```



```
returned_value
```

```
>  NULL
```

Without return value

```r
a <- 35
save(a, file = "out_file.RData")
```

```r
returned_value <- save(a, file = "out_file.RData")
returned_value
```

```
>  NULL
```

Invisible return value

Functions returning their result invisibly:

- e.g. `print()`
- `par()`
- `options()`
- `attach()`
- `load()` etc.

Invisible return value

Functions returning their result invisibly

- do something in the background (“side effect”),
- and also produce some not-too-important results
- that can be captured by assignment,
- but will be lost otherwise

Examples:

- `load()`:
  - loads variables from a file (side effect)
  - and invisibly returns the name of the loaded variables
- `par()`:
  - changes the graphical parameters (side effect)
  - and invisibly returns the previous parameters (for backuping)

Invisible return value

CASE $\#1$: not captured $\rightarrow$ the returned value will be lost:

```
load("out_file.RData")
```

But what is/are the variable(s) that was/were loaded??

Invisible return value

```
rm(a)
variables_before_load <- ls()
load("out_file.RData")
variables_after_load <- ls()
setdiff(variables_after_load, variables_before_load)
```

```
>  [1] "a"                    "variables_before_load"
```

Huhh, a bit tedious...

Invisible return value

CASE #2: captured → the returned value will be assigned to a variable:

```
loaded_variables <- load("out_file.RData")
loaded_variables
```

```
>  [1] "a"
```

Of course, the side effect (i.e., loading the variables from the file) is done in both cases.

Visible return value

Functions returning one or more value(s) visibly:

- e.g. `mean()`
- `sum()`
- `lm()`
- `paste()` etc.

Most of the functions in R return some values visibly.
Multiple values are returned in vectors, lists or more complex objects (e.g. `lm()`).

Visible return value

Functions returning one or more value(s) visibly

- are called for their output
- they might (but rarely) have side effects
- the returned value should be captured
- otherwise it will be (printed on the console and then) lost
- printing works only in interactive mode!

```
sum(3, 4, 3)
```

```
>  [1] 10
```

```
summed_values <- sum(3, 4, 3)
summed_values * 2
```

```
>  [1] 20
```

Returning value

Our own function can be a function that

- does not return any value (we have seen this)
- returns a value invisibly
- returns a value visibly

Returning value - explicit statement

Explicit statement of returning a value:

- `invisible()`: returns a value invisibly
- `return()`: returns a value visibly

Example for `invisible()`:

```
random_text <- function() {
  text_beginning <- "Writing functions in R is "
  text_ends <- c("a great challange", "funny", "hard for
me", "really interesting")
  selected_end <- sample(x = text_ends, size = 1)
  text <- paste0(text_beginning, selected_end, ".")
  writeLines(text)
  invisible(selected_end)
}
```

Returning value - explicit statement

```
set.seed(20220127)
random_text()
```

```
>  Writing functions in R is really interesting.
```

```
random_text()
```

```
>  Writing functions in R is funny.
```

```
captured_end <- random_text()
```

```
>  Writing functions in R is a great challange.
```

```
captured_end
```

```
>  [1] "a great challange"
```

Returning value - explicit statement

Example for `return()`:

```
random_favorite <- function() {
  my_favorite_numbers <- c(57, 3, 0, 217)
  randomly_selected <- sample(x = my_favorite_numbers, size
= 1)
  return(randomly_selected)
}
random_favorite()
```

```
>  [1] 0
```

```
random_favorite()
```

```
>  [1] 3
```

```
random_favorite()
```

```
>  [1] 57
```

Returning value - explicit statement

The body can contain multiple `return()`s and/or `invisible()`s (typically within conditional statements).

`return()` and `invisible()` are not necessarily at the end of the function body.

Execution of the body stops when `return()` or `invisible()` is reached. Further lines of code will never executed.

Returning value - explicit statement

```
sunny_day <- function() {
  is_sunny <- sample(x = c(TRUE, FALSE), size = 1)
  if (is_sunny) {
    writeLines("This is a sunny day.")
    return(is_sunny)
    writeLines("Use sun lotion on the beach.")
  } else {
    writeLines("This is a cloudy day.")
    return(is_sunny)
    writeLines("Don't forget your umbrella.")
  }
}
```

Returning value - explicit statement

```r
set.seed(20220127)
sunny_day()
```

```
>  This is a cloudy day.
>  [1] FALSE
```

```r
sunny_day()
```

```
>  This is a cloudy day.
>  [1] FALSE
```

```r
sunny_day()
```

```
>  This is a sunny day.
>  [1] TRUE
```

Returning value - without explicit statement

If the end of a function body is reached without calling `return()` or `invisible()` explicitly, the value of the last evaluated expression is returned.
Hence, `return(x)` can be preplaced by `x` at the end of the function body.

```
random_even <- function() {
  even_numbers <- seq(from = 2, to = 20, by = 2)
  randomly_selected <- sample(x = even_numbers, size = 1)
  randomly_selected
}
```

Returning value - without explicit statement

```r
random_even()
```

```
>  [1] 20
```

```r
random_even()
```

```
>  [1] 10
```

```r
random_even()
```

```
>  [1] 2
```

Without returning value

Our function does not return any value (i.e., returns `NULL`) if

1. `return()` or `invisible()` is called explicitly without an argument or
2. - execution of the body of the function is finished
   - without explicit statement of returning a value
   - and the execution of last line of code does not result in a value (i.e., `NULL`)

Without returning value

```r
do_nothing <- function() {
  x <- 3
  invisible()
  return(x) # never reached
}
do_nothing()
```

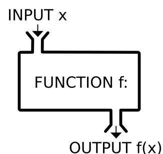
```
>  [1] 3
```

```r
result <- do_nothing()
result
```

```
>  [1] 3
```

What is a function?

A function is a **block of code** which only runs when it is **called**.
You *can* pass data ("**parameters**") into a function.
A function *can* return data as a result ("**return value**").



Then,

- ~~what is a block of code?~~
- ~~how to call it?~~
- how to pass data?
- ~~how to return data?~~

Parameters

Parameters are the inputs of the function.
There are two types of parameters:

- named parameters
- unnamed parameters (...)

We'll learn only about named parameters. Anyway, unnamed parameters are really important features in R, but needs advenced programming skills to understand their correct usage.

Parameters

Named parameters have two types:

- those that has default values (i.e., optional parameters), and
- those that hasn't (i.e., mandatory parameters)

Parameters

Let's see an example: `rnorm()`.
This function produces some random values from a normal (Gaussian) distribution.

Check its help page.

```
?rnorm
```

Parameters

The Normal Distribution

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

`x, q`
: vector of quantiles.

`p`
: vector of probabilities.

`n`
: number of observations. If `length(n) > 1`, the length is taken to be the number required.

`mean`
: vector of means.

`sd`
: vector of standard deviations.

Parameters

```
rnorm(n, mean = 0, sd = 1)
```

According to its help page, `rnorm()` has 3 named parameters:

- `n`: number of the needed random values
- `mean`: the mean of the normal distribution from which the random numbers should be generated
- `sd`: the standard deviation of the normal distribution from which the random numbers should be generated

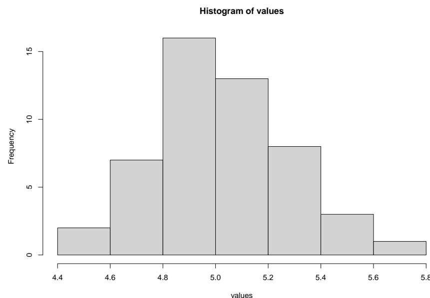
`n` is mandatory, while `mean` and `sd` have default values, so these are optional parameters.

Calling a function with parameters

```
rnorm(n = 10, mean = -50, sd = 6)
```

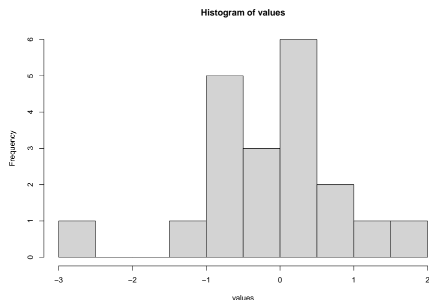
```
>    [1] -44.08701 -57.75795 -55.40136 -41.02584 -45.44844
>    [6] -53.59784 -48.16727 -48.43761 -53.23154 -47.36366
```

```
values <- rnorm(n = 50, mean = 5, sd = 0.2)
hist(values)
```



Calling a function with parameters

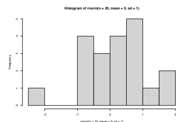
```
values <- rnorm(n = 20, mean = 0, sd = 1)
hist(values)
```



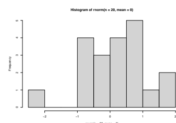
Since `mean` and `sd` have default values, these can be removed from the call.

Calling a function with parameters

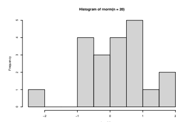
```
set.seed(1); hist(rnorm(n = 20, mean = 0, sd = 1))
```




```
set.seed(1); hist(rnorm(n = 20, mean = 0))
```




```
set.seed(1); hist(rnorm(n = 20))
```



Calling a function with parameters

But `n` hasn't got default value!

```
hist(rnorm()) # argument "n" is missing, with no default
```

```
> Error in rnorm(): argument "n" is missing, with no
default
```

Calling a function with parameters

Parameters have not only names but order.
This is why naming the parameters is necessary only if you want to give the parameters in different order.

These are the same:

```
set.seed(1); rnorm(n = 5, mean = 15, sd = 3)
```

```
>  [1] 13.12064 15.55093 12.49311 19.78584 15.98852
```

```
set.seed(1); rnorm(mean = 15, n = 5, sd = 3)
```

```
>  [1] 13.12064 15.55093 12.49311 19.78584 15.98852
```

```
set.seed(1); rnorm(5, 15, 3)
```

```
>  [1] 13.12064 15.55093 12.49311 19.78584 15.98852
```

Calling a function with parameters

R first matches the named parameters (15 $\rightarrow$ `mean`). Then tries to match the unnamed parameters (5, 3) based on the order of the lacking parameters (`n`, `sd`).

```
set.seed(1); rnorm(n = 5, mean = 15, sd = 3)
```

```
>  [1] 13.12064 15.55093 12.49311 19.78584 15.98852
```

```
set.seed(1); rnorm(mean = 15, 5, 3)
```

```
>  [1] 13.12064 15.55093 12.49311 19.78584 15.98852
```

Calling a function with parameters

Of course, optional parameters are optional even if not named!
E.g., `sd` is missing:

```
set.seed(1); rnorm(n = 5, mean = 15)
```

```
>  [1] 14.37355 15.18364 14.16437 16.59528 15.32951
```

```
set.seed(1); rnorm(mean = 15, 5)
```

```
>  [1] 14.37355 15.18364 14.16437 16.59528 15.32951
```

Calling a function with parameters

If you want to skip a parameter (e.g. `mean`):

- you must use the names
- or add extra commas

```
set.seed(1); rnorm(n = 5, sd = 3)
```

```
> [1] -1.8793614 0.5509300 -2.5068858 4.7858424 0.9885233
```

```
set.seed(1); rnorm(5, , 3)
```

```
> [1] -1.8793614 0.5509300 -2.5068858 4.7858424 0.9885233
```

Calling a function with parameters

A suggestion from the wise Uncle Ákos:

Almost **always use parameter names**.
The only exception is the function that has one mandatory parameter (e.g. `print(x)`, `str(object)`, `set.seed(seed)`).

Calling a function with parameters

Some R users prefer calling functions without parameter names, like this:

```
seq(3, 12, 3)
```

```
>  [1]  3  6  9 12
```

I recommend the following solution, since it is

- much more readible
- harder to do mistakes

```
seq(from = 3, to = 12, by = 3)
```

```
>  [1]  3  6  9 12
```

Arguments vs. parameters

The terms "argument" and "parameter" are usually thought to be interchangeable.
To be honest, these differ.

Parameter is the **p**arking place for the **a**rgument (the **a**uto).

Arguments vs. parameters

So calling

```
seq(from = 3, to = 12, by = 3)
```

```
>  [1]  3  6  9 12
```

means that **parameters** `from`, `to` and `by` get the **arguments** (i.e., the values) 3, 12 and 3 for further processing inside the body of `seq()`.

Anyway, hardly anyone cares the difference between parameters and arguments, so it's not a problem if you interchange them.
I do too. . .

Adding parameters to our function

Adding parameters to our function is really simple:

- just give the name of the parameters within the parentheses (separated by commas)
- and give a default value if you want (use the = sign)
- just like in the help pages

```
increase <- function(x, by = 1) {
  return(x + by)
}
```

Adding parameters to our function

Calling the function is as learnt previously:

```r
increase(x = 5)
```

```
>  [1] 6
```

```r
increase(5)
```

```
>  [1] 6
```

```r
increase(x = 7, by = 4)
```

```
>  [1] 11
```

```r
increase(7, 4)
```

```
>  [1] 11
```

```r
increase(by = 4, x = 7)
```

```
>  [1] 11
```

Adding parameters to our function

Let's get the first (or the one from the *n*th position) odd or even number from a numeric vector called `numbers`.

```
get_odd_even <- function(numbers, odd = TRUE, position = 1)
{
  odd_numbers <- numbers[numbers %% 2 == 1]
  even_numbers <- numbers[numbers %% 2 == 0]
  preferred_numbers <- if (odd) odd_numbers else
even_numbers
  if (length(preferred_numbers) < position) {
    return(NA)
  } else {
    return(preferred_numbers[position])
  }
}
```

Adding parameters to our function

```r
get_odd_even(numbers = 1:10)
```

```
>  [1] 1
```

```r
get_odd_even(numbers = 1:10, odd = FALSE)
```

```
>  [1] 2
```

```r
get_odd_even(numbers = 1:10, position = 4)
```

```
>  [1] 7
```

```r
get_odd_even(numbers = 1:10, odd = FALSE, position = 4)
```

```
>  [1] 8
```

Scope and lifetime

Within the function body, you can

- use the parameters as if they were variables,
- create new variables.

But these variables are temporary ones!

Scope and lifetime

Parameters or new variables: `numbers`, `odd`, `position`, `odd_numbers`, `even_numbers` and `preferred_numbers`.

```
get_odd_even <- function(numbers, odd = TRUE, position = 1)
{
  odd_numbers <- numbers[numbers %% 2 == 1]
  even_numbers <- numbers[numbers %% 2 == 0]
  preferred_numbers <- if (odd) odd_numbers else
even_numbers
  if (length(preferred_numbers) < position) {
    return(NA)
  } else {
    return(preferred_numbers[position])
  }
}
```

Scope and lifetime

These temporary variables have a limited lifetime.
They are born when the function is **called** (not when defined!), and died when the execution of the function is finished.

Although we defined the function (and variable `odd_numbers` inside its body and `position` as its parameter), this will fail:

```
odd_numbers # error
```

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

```
position # error
```

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

`odd_numbers` and `position` have not been born yet, since the function is not called.

Scope and lifetime

```
get_odd_even(numbers = 1:10, odd = FALSE, position = 4)
```

```
>  [1] 8
```

```
odd_numbers # error
```

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

```
position # error
```

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

When the function is finished (it returns a value), the temporary variables die.

Scope and lifetime

The temporary variables have a local scope. They are available inside the function body and might mask (but not delete/modify) global variables with the same name.

```
odd_numbers <- c(7, 9, 11)
get_odd_even(numbers = 1:10, odd = TRUE, position = 2)
```

```
>  [1] 3
```

```
odd_numbers
```

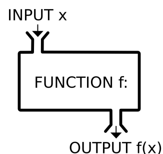
```
>  [1]  7  9 11
```

Temporarily, two `odd_numbers` existed in parallel:

- a global one containing 7, 9 and 11
- a local one containing 1, 3, 5, 7 and 9

What is a function?

A function is a **block of code** which only runs when it is **called**.
You *can* pass data ("**parameters**") into a function.
A function *can* return data as a result ("**return value**").



Then,

- ~~what is a block of code?~~
- ~~how to call it?~~
- ~~how to pass data?~~
- ~~how to return data?~~

Why should I write a function?

Why should I write a function?
The answer is simple: to **avoid repetition**.

Writing a repetitive script

- is time-consuming and
- gives more occasion to introduce bugs in your scripts.

Why should I write a function?

Let's say we have a large `data.frame` containing species in columns, locations in rows and the abundance in the cells.

```
abundances_df <- data.frame(matrix(data = sample(x = 0:10,
size = 100 * 5, replace = TRUE), ncol = 5))
colnames(abundances_df) <- LETTERS[1:5]
str(abundances_df)
```

```
>  'data.frame':    100 obs. of  5 variables:
>   $ A: int  10 2 0 4 4 9 5 9 6 8 ...
>   $ B: int  2 10 0 5 5 3 8 4 0 2 ...
>   $ C: int  6 3 3 0 9 3 8 8 10 8 ...
>   $ D: int  7 8 5 0 7 8 8 10 5 7 ...
>   $ E: int  7 0 2 0 2 5 10 7 6 3 ...
```

Why should I write a function?

Let's imagine we have not only 5 but much more species (several hundred). And we want to do some repetitive tasks for each species. Huhh, seems to be a lot of work!!

Example task:

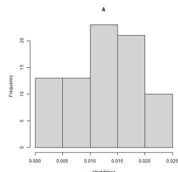
- delete the values that larger than 8 (these are measurement errors)
- calculate the relative abundance of the selected species (as the ratio of total abundance within the studied locations)
- plot a histogram
- and save the histogram as a PNG file

Why should I write a function?

Just try to solve the tasks for one selected species, called "A", out of the several hundred species:

```
abundance <- abundances_df[, "A"]
abundance[abundance > 8] <- NA
abundance <- abundance / sum(abundance, na.rm = TRUE)
png("A.png")
hist(abundance, breaks = 6, main = "A")
dev.off()
```

```
> pdf
>    2
```



Why should I write a function?

Now do this for all the other several hundred speces.

The **dull** solution: Ctrl+C, Ctrl+V, and change the species name ("A") in all rows.
And repeat it until you go crazy...

The **smart** solution: write a function, and call this function for all species

Why should I write a function?

When translating a code to function, be sure that

- every calculation is dependent on its input parameters,
- so you should change "A" to something else.

```
clean_and_plot <- function(species_name) {
  abundance <- abundances_df[, species_name]
  abundance[abundance > 8] <- NA
  abundance <- abundance / sum(abundance, na.rm = TRUE)
  png(paste0(species_name, ".png"))
  hist(abundance, breaks = 6, main = species_name)
  dev.off()
  invisible()
}
```

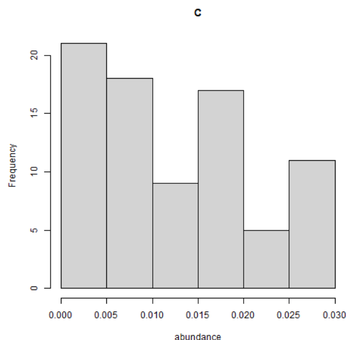
"A" is now got as input parameter (the value of species_name).

Why should I write a function?

Let's try our new function.

```
clean_and_plot(species_name = "C")
```

This is C.png:



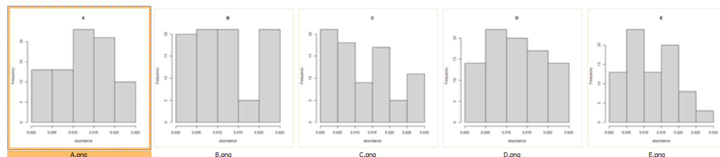
Why should I write a function?

It works. So we can call it repetitively inside a `for` loop:

```
for (selected_species in colnames(abundances_df)) {
  clean_and_plot(species_name = selected_species)
}
```

Or the same with `lapply()` (more elegant solution):

```
nulls <- lapply(X = colnames(abundances_df), FUN =
clean_and_plot)
```



The end

Thank you for your kind attention.