Basic techniques for creating functions

What is a function?

- We work with pre-specified functions all the time (stored in R packages, e.g. `fread()` from the `data.table` package).
- For example, the function `sd()` is basically a function to avoid typing `sqrt(var(x))` all the time.

```
sqrt(var(c(2,4,5,6,2,3,4)))
OUTPUT [1] 1.3850513
sd(c(2,4,5,6,2,3,4))
OUTPUT [1] 1.3850513
```

- However, we don't have to rely only on pre-specified functions. We can write our own functions to perform specific tasks.

What is a function?

- We work with pre-specified functions all the time (stored in R packages, e.g. `fread()` from the `data.table` package).
- For example, the function `sd()` is basically a function to avoid typing `sqrt(var(x))` all the time.

```
sqrt(var(c(2,4,5,6,2,3,4)))
OUTPUT [1] 1.3850513
sd(c(2,4,5,6,2,3,4))
OUTPUT [1] 1.3850513
```

- However, we don't have to rely on pre-specified functions, we can write our own functions to perform specific tasks.

Why should you write a function? (1/2)

Assume you want to calculate your return on investment for 3 different projects. You would start writing the R code as follows:

```
ROI_1 = (sum(data1$Revenue)- sum(data1$Cost))/
         (sum(data1$Cost))

ROI_2 = (sum(data2$Revenue)- sum(data2$Cost))/
         (sum(data2$Cost))

ROI_3 = (sum(data3$Revenue)- sum(data3$Cost))/
         (sum(data3$Cost)
```

Now imagine doing this for 100 projects …

Are you sure this has been done correctly. Or might there be a copy-paste-mistake?

Why should you write a function? (2/2)

You can make your life easier by writing and using a simple function for this operation:

```
ROI <- function(x){
      ...
      ...
}
```

1 Define the function (we learn today how to replace the "…" with the necessary code

```
ROI(data1)
ROI(data2)
ROI(data3)
```

2 Apply the function on the datasets

Thus, functions have two main advantages:

1. Reduce mistakes from copy and pasting.
2. By modularizing your code, functions facilitate code re-use and simplified maintenance/re-factoring.

Function components

Every function has three parts:

1. Arguments
2. Body
3. Environment

```
myFun <- function(arg1, arg2=1){
          ...
        }
```

Function arguments 1

Function body: put your code here 2

Name of your function 3

The environment of a function is where it is defined and where it calls the arguments from 4

Function components

Every function has three parts:

1. Arguments
2. Body
3. Environment

The environment of a function is where it is defined and where it calls the arguments from 4

Function arguments 1

```
myFun <- function(arg1, arg2=1){
          ...
         }
```

Name of your function 3

Function body: put your code here 2

Function components

Every function has three parts:

1. Arguments
2. Body
3. Environment

```
myFun <- function(arg1, arg2=1){
    ...
  }
```

1 **Function arguments**

2 **Function body:** put your code here

3 Name of your function

4 The environment of a function is where it is defined and where it calls the arguments from

Function components

Every function has three parts:

1. Arguments
2. Body
3. Environment

```
myFun <- function(arg1, arg2=1){
    ...
  }
```

Function arguments 1

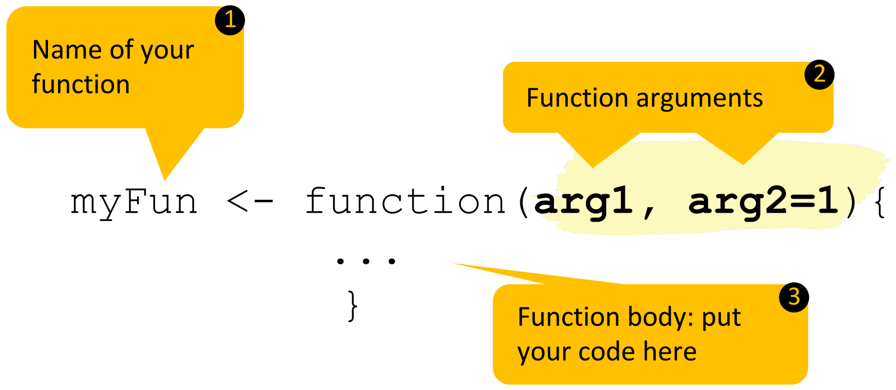
Function body: put your code here 2

Name of your function 3

The environment of a function is where it is defined and where it calls the arguments from 4

Function components: Arguments

- Arguments are the variables passed to the function.
- You can specify as many arguments as necessary.
- Define defaults with `=`, e.g. `arg2=1`



```
myFun <- function(arg1, arg2=1){
          ...
         }
```

Function components: Body

- The body contains the code to be executed by the function.
- The last evaluated value is returned. However, it is best practice to define the return value explicitly using `return()`.

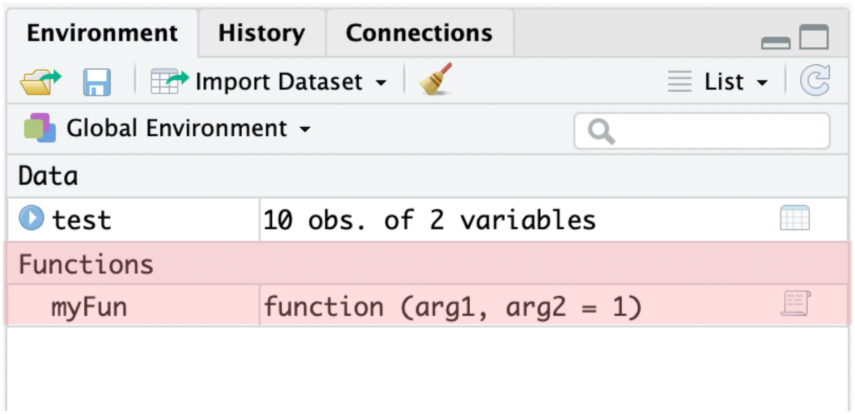
```
myFun <- function(arg1, arg2=1){
          ...
          put your code here
          ...
          return(value)
          }
```

Function components: Environment

- Defines, where the function stores and also (in the first place) looks for variables.
- Objects created in the function are stored in a function-specific, local environment and cannot be accessed unless explicitly returned.
- Best practice is to only use the arguments passed to the function and not variables in the "global" environment.

```
myFun  <- function(arg1,
arg2=1){
                ...
               return(value)

               }
```



Data

test	10 obs. of 2 variables	

Functions

myFun	function (arg1, arg2 = 1)	

A simple example: Creating a function to add two variables

We define a simple function that adds two variables:

Arguments [1]

Default parameter, i.e. `y` takes the value `1` if not specified otherwise in the function call [2]

```
add <- function(x, y= 1){
                    result <- x + y
                    return(result)
                    }
```

Body [3]

Function call: [4]
x=4
y by default 1

```
add(4)
OUTPUT: [1] 5
```

Function call: [5]
x=4
y=5

```
add(4,5)
OUTPUT: [1] 9
```

A simple example: Creating a function to add two variables

We define a simple function that adds two variables:

Arguments 1

Default parameter, i.e. `y` takes the value `1` if not specified otherwise in the function call 2

```
add <- function(x, y= 1){
                        result <- x + y
                        return(result)
                        }
```

Body 3

Function call: 4
x=4
y by default 1

```
add(4)
OUTPUT: [1] 5
```

Function call: 5
x=4
y=5

```
add(4,5)
OUTPUT: [1] 9
```

A simple example:
Creating a function to add two variables

We define a simple function that adds two variables:

Arguments ❶

Default parameter, i.e. `y` takes the value `1` if not specified otherwise in the function call ❷

```
add <- function(x, y= 1){
                result <- x + y
                return(result)
                }
```

Body ❸

Function call: ❹
x=4
y by default 1

```
add(4)
OUTPUT: [1] 5
```

Function call: ❺
x=4
y=5

```
add(4,5)
OUTPUT: [1] 9
```