

# Programming in R

---

Dries Debeer & Benjamin Becker

14. September 2023

FGME Post-Conference Workshop

# Introduction

---

## Who are we?

### Dries Debeer

Statistical Consultant at Ghent University (FPPW)

scDIFtest, permimp, eatATA, mstDIF

dries.debeer@ugent.be

### Benjamin Becker

Researcher at IQB (Verbund Forschungsdaten)

eatGADS, eatDB, eatATA, pisaRT

b.becker@iqb.hu-berlin.de

## Who are you?

1. Position and research topics?
2. Previous knowledge and experience
  - with R?
  - with other statistical software?
  - with other programming languages?
3. Expectations for this workshop?

## Why care about R coding?

1. Increase efficiency!
  - Save time and nerves
  - Avoid errors and bugs
  - High transfer effect to other projects (with data analysis)
2. Successful collaborations (including with your future self!)
3. Code as deliverable (i.e., part of research paper)

# Agenda

- Loops & iteration
- Functions (part I)
- Functions (part II)
- Functionals & split-apply-combine
- Good programming practices

## Quick Recap

---

# Basic *Objects* in R

Vectors	
logical	TRUE, FALSE, NA
integer	1L, 142, -5, ..., NA
double	1.0, 1.25784, pi, ..., NA NaN, -Inf, Inf
character	"1", "Some other string", ..., NA

Vectors can have multiple elements **of the same type** →  
`length()` starting from 0



## Basic *Objects* in R

More basic object types	
list	a <b>list</b> of vectors list(), as.list()
matrix	a <b>vector</b> with a "dim" argument: two dimensions matrix(), as.matrix() matrix algebra
array	a <b>vector</b> with with a "dim" argument
data.frame	a <b>list</b> with vectors of equal length data.frame(), as.data.frame()

## data.frame vs matrix

A data.frame is a list of vectors → columns can be of different types.

Try it!

```
iris_dat <- head(iris)
is.list(iris_dat)
dim(iris_dat)
length(iris_dat)
iris_dat[1:2,]
```

## data.frame vs matrix

A matrix/array is a vector with dimensions → all elements/columns are of the same type.

Try it!

```
iris_mat <- as.matrix(iris_dat)
dim(iris_mat)
length(iris_mat)
iris_mat[1:2,]
```

# Loops & Iteration

---

# Loops & iteration

R has specific tools (functions) that help organize the flow of computations.

You can repeat a similar computation multiple times typically with changing options (“iteration”). The most commonly used tools are:

- loops
  - `for`
  - `while`
  - `repeat`
- `apply` - family

# Loops & Iteration - for

for statements have the basic form

```
for (element in vector) {  
    computation  
}
```

For each element in the vector, the computation is executed.  
Often, the computation depends on the element in that iteration.

# Loops & Iteration - for

```
# iterate over a numeric vector
for (index in 1:3){
  cat(" computation -")
}

> computation - computation - computation -

# iterate over a character vector
for (name in c("Alice", "Bob", "Casey")){
  if(name == "Bob") cat(" This was Bob -")
  else cat(" Not Bob -")
}

> Not Bob - This was Bob - Not Bob -
```

# Loops & Iteration - while

while statements have the basic form

```
while (condition){  
    computation  
}
```

As long as the condition is TRUE, the computation is executed. Often, the computation depends on something that is related to the condition.



# Loops & Iteration - repeat

repeat statements have the basic form

```
repeat {  
  computation  
}
```

Without a `break` the computation is repeated infinite times

## Loops & Iteration - next break

- `next` starts next iteration
- `break` ends iteration (of the innermost loop)

# Loops & Iteration - next break

- next starts next iteration
- break ends iteration (of the innermost loop)

```
for(index in 1:8) {  
  if (index %in% c(3, 5)) next  
  if (index > 6) break  
  print(index)  
}
```

```
> [1] 1  
> [1] 2  
> [1] 4  
> [1] 6
```

# Loops & Iteration - nested loops

Nested loops (over the rows and columns of a matrix)

```
matrix <- matrix(NA, nrow = 2, ncol = 3)
for (rowNr in 1:2){
  for (colNr in 1:3){
    matrix[rowNr, colNr] <- rowNr * 10 + colNr
  }
}
matrix
```

```
>      [,1] [,2] [,3]
> [1,]   11   12   13
> [2,]   21   22   23
```

# Iteration - Good practice

## Programming advice

Use `seq()`, `seq_len()`, or `seq_along()`, not `1:length(x)`.

```
x <- numeric()
for (index in 1:length(x)){
  print(index)
}

> [1] 1
> [1] 0

for (index in seq_along(x)){
  print(index)
}
```

# Loops & Iteration - Good practice

## Programming advice

Don't grow, replace.

```
x <- letters
result1 <- numeric()           # grow
result2 <- numeric(length(x)) # replace
for (index in seq_along(x)){
  result1 <- c(result1, paste(index, x[index])) # grow
  result2[index] <- paste(index, x[index])      # replace
}
```

# Loops & Iteration - apply

`apply` lets you iterate over rows or columns of a matrix or `data.frame`. You can *apply* a function to all rows/columns

```
apply(matrix,  
      MARGIN = 1,      # 1 = iterate over rows  
                        # 2 = iterate over columns  
      FUN = function)  # function to apply to rows/columns
```

# apply

- for objects with dimensions (matrix, array, data.frame)
- apply over (a) chosen dimension(s)

```
my_matrix <- matrix(1:6, nrow = 2)
apply(my_matrix, 1, max)      # apply per row

> [1] 5 6

apply(my_matrix, 2, max)      # apply per column

> [1] 2 4 6
```



# apply

```
my_array <- array(1, dim = c(2, 3, 4))
apply(my_array, c(1, 2), sum) # per row and column

>      [,1] [,2] [,3]
> [1,]    4    4    4
> [2,]    4    4    4

apply(my_array, 3, sum)      # per "third dimension"

> [1] 6 6 6 6
```

# Exercises



# Functions I

---

*“To understand computations in R, two slogans are helpful: Everything that exists is an object. Everything that happens is a function call.”*

— John Chambers, Extending R (2016)

# Function Calls

Computing in R happens through function calls. A function is applied to one or more objects, and returns an object after the computation.



**Figure 1:** A function call.

The typical use is:

```
function(object1, argument = object2)
```

# Function Calls

- Computations that seem not to be done using function calls are actually also function calls. Try `<-` (a, 5)` or `>` (5, 2)`
- most functions that seem not to return an object, return it invisibly. Check `(a <- 5)`.

# Building Blocks

Functions are the building blocks of R code. Writing functions allows you to organize and optimize the computations that you want to do.

Functions should:

- have a clear purpose
- be well documented
- be portable

## **Central stepping stone for R users:**

Move from solely using functions written by others to writing your own functions.



# Function definition

- Name
- Arguments/Formals (input)
- Body (what happens inside, R-code with the computations)
- Output

# Function definition

```
countNA <- function(x) {  
  out <- sum(is.na(x))  
  out  
}
```

# Name  
# Arguments/Formals  
# Body  
# Output

# Function Names

Every function needs a (meaningful) name!

- Usually a **verb** (what does the function do?)
- Avoid existing names
- Better longer than unclear
- CamelCase vs snake\_case

# Function Names

## Good

- `computeAIC()`
- `removeNAs()`
- `drop_NA_rows()`
- `factor_to_dummies()`

## Bad

- `myFun()`
- `foo()`
- `statistics()`
- `data_preparation()`

# Arguments

Most functions take one or multiple inputs.

These are usually:

- One or two data arguments
- Additional Options

## Functions with zero arguments

- `getwd()`
- `Sys.time()`
- ...

## Functions with one argument

- `dim()`
- `names()`
- ...

## Functions with multiple arguments

- `mean()`
- `median()`
- `lm()`
- ...



# Arguments

## Programming advice

Less arguments = better!

# Arguments

Often arguments have to be objects of a specific type.

```
sum(c("a", "b", "c")) # gives an error
```

```
> Error in sum(c("a", "b", "c")): ungültiger 'type' (character)  
des Argumentes
```

The documentation typically gives (or should give) information about what objects the arguments should be. Check `?sum`

# Default arguments

What happens if the user omits an argument?

```
countNA <- function(x, percent) {  
  out <- sum(is.na(x))  
  if(percent) out/length(x)  
  out  
}  
x <- c(1, 5, NA, 3)  
countNA(x = x)
```

```
> Error in countNA(x = x): Argument "percent" fehlt (ohne  
Standardwert)
```

# Default arguments

Default arguments are made for such instances!

```
countNA <- function(x, percent = FALSE) {  
  out <- sum(is.na(x))  
  if(percent) out/length(x)  
  out  
}  
x <- c(1, 5, NA, 3)  
countNA(x = x)  
  
> [1] 1
```

# Default arguments

Additional arguments give (the user) flexibility. Default arguments keep the function easy to use.

Try ?lm

## Programming advice

- Think about which arguments to include, and which should (not) have defaults
- Choose sensible defaults

# Single return object

Pure functions return a single object.

- (Standard) The last evaluated object
- Object defined by return()



**Figure 2:** A pure function.

# Return object

`return()` stops the computation, and returns the object.

```
return_early <- function(x, early) {  
  x2 <- x*2  
  if(early) (return(x2))  
  out <- x + x2 # not executed  
  out  
}  
return_early(2, early = TRUE)  
  
> [1] 4  
  
return_early(2, early = FALSE)  
  
> [1] 6
```

# Return object

Multiple return objects can be combined in a list!

```
do_this <- function(vector, other_vector) {  
  # many computations  
  return(list(output1 = this,  
              output2 = that))  
}
```

# Name  
# Arguments  
# Body  
# Output



# Return Object

The return object as a list with multiple objects.

```
get_info <- function(x){  
  mean_x <- mean(x)  
  median_x <- median(x)  
  n_obs_x <- length(x)  
  range_x <- range(x)  
  return(list(mean = mean_x, median = median_x,  
              n_obs = n_obs_x, range = range_x))  
}
```

```
str(get_info(airquality$Wind))
```

```
> List of 4  
> $ mean : num 9.96  
> $ median: num 9.7  
> $ n_obs : int 153  
> $ range : num [1:2] 1.7 20.7
```

- `browser()`
- `traceback()`
- `options(error = recover)`
- `options(warn = 2)`

# browser()

Inspecting a function interactively

```
some_function <- function(x, y) {  
  z <- x + y  
  browser()  
  z  
}  
some_function(x = 1, y = 5)
```

# browser()

```
> some_function <- function(x, y) {  
+   z <- x + y  
+   browser()  
+   z  
+ }  
> some_function(x = 1, y = 5)  
Called from: some_function(x = 1, y = 5)  
Browse[1]> |
```

# browser()

Navigating within a browser:

ls() Show existing objects in the current environment

c Exit the browser and continue execution

Q Exit the browser, return to top level

where Show call stack

Being able to chose an environment from the call stack:

```
# on
options(error = recover)

# off
options(error = NULL)
```

Being able to choose an environment from a call stack:

```
Error in pretty_table(x, x_label = x_label) : length(x) > 1 is not TRUE
Enter a frame number, or 0 to exit

1: by(mtcars, mtcars$carb, function(sub_dat) {
  pretty_statistics(sub_dat$cy1, x_label = "Cy1")
2: by.data.frame(mtcars, mtcars$carb, function(sub_dat) {
  pretty_statistics(sub_dat$cy1, x_label = "Cy1")
3: structure(eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = simplify))), data), call =
4: eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = simplify))), data)
5: eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = simplify))), data)
6: tapply(seq_len(32), list(mtcars$carb), c(4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4,
7: tapply(X = ans[index], FUN = FUN, ...)
8: FUN(X[[i]], ...)
9: FUN(data[x, , drop = FALSE], ...)
10: #2: pretty_statistics(sub_dat$cy1, x_label = "Cy1")
11: #3: pretty_table(x, x_label = x_label)
12: #2: stopifnot(length(x) > 1)

Selection: |
```

Turning warnings into errors

```
# on
options(warn = 2)

# off
options(warn = 1)
```



# Exercises



# Functions II

---

## Why write functions?

- They make code ...
  - shorter (less repetition)
  - easier to read and understand
- They help avoid copy-paste errors
- They make it easier to change your code
- They increase transfer
  - other use cases
  - other projects
  - other persons
- They keep your work space clean

## Writing a function:

```
RMSE <- get_RMSE(predictions, observations)
```

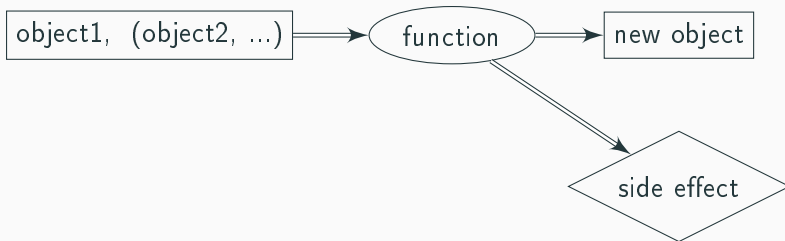
## Not writing a function:

```
diff <- observations - predictions  
sq_diff <- diff^2  
m_sq_diff <- mean(diff)  
RMSE <- sqrt(m_sq_diff)
```

# Side Effects

Functions can have “side effects”:

- console output
- plots
- write/save on drive
- ...



**Figure 3:** A function with side effect.

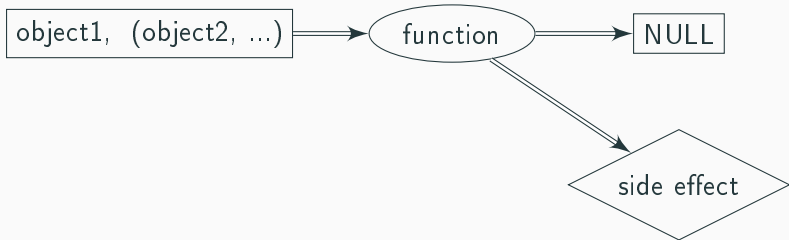
Console output: ?cat and ?print

```
print_info <- function(x){  
  info <- get_info(x)  
  cat("There are ", info$n_obs,  
      " observed values. \nThe mean is ",  
      round(info$mean, 2), ". \nThe median is ",  
      round(info$median, 2), ". \n", sep = "")  
}  
print_info(airquality$Wind)
```

```
> There are 153 observed values.  
> The mean is 9.96.  
> The median is 9.7.
```

## Programming advice

- Write pure functions (no-side effects)
- Write separate functions for side effects
- Plotting functions should return NULL or the plot as an object



**Figure 4:** A side effect function.

# Error, warning, & message

Error: computation is interrupted without return object!

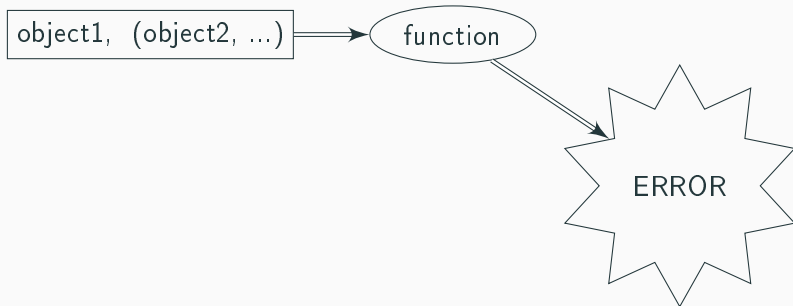
?stop

```
get_log_xtox <- function(x) {  
  if(!is.numeric(x)) stop("This does not work!")  
  x_x <- x^x  
  return(log(x_x))  
}  
get_log_xtox("a")  
  
> Error in get_log_xtox("a"): This does not work!
```



## Error, warning, & message

Error: computation is interrupted without return object!



**Figure 5:** Computation with Error.

## Error, warning, & message

?stopifnot is an abbreviation for if(!test) stop():

```
get_log_xtox <- function(x) {  
  stopifnot(is.numeric(x))  
  x_x <- x^x  
  return(log(x_x))  
}  
get_log_xtox("a")
```

```
> Error in get_log_xtox("a"):  is.numeric(x) ist nicht TRUE
```

# Error, warning, & message

Message: To inform the user about something.

?message

```
get_log_xtox <- function(x) {  
  x_x <- x^x  
  message("Thank you for using this function!")  
  return(log(x_x))  
}  
get_log_xtox(2)  
  
> Thank you for using this function!  
  
> [1] 1.386294
```

# Error, warning, & message

Warning: Warn the user that something may be fishy.

?warning

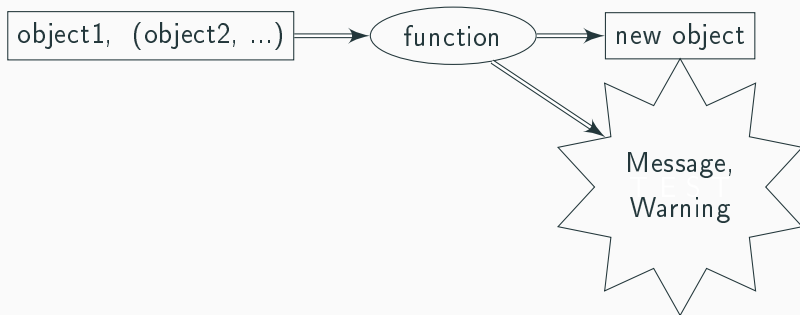
```
get_log_xtox <- function(x) {  
  if(x < 0 && (x %% 2 == 0))  
    warning("Not sure you can trust the result.",  
           call. = FALSE)  
  x_x <- x^x  
  return(log(x_x))  
}  
get_log_xtox(-2)
```

```
> Warning: Not sure you can trust the result.
```

```
> [1] -1.386294
```

## Error, warning, & message

Message & warning: computation is NOT interrupted!



**Figure 6:** A message or warning.

## Programming advice

- Choose carefully when something warrants a message, warning or error
- Write clear and helpful warnings, errors, messages

## Where does a function find objects?

R uses specific rules to find objects, which lead to the following:

```
a <- 55
add_a <- function(x){
  return(x + a)
}
add_a(5)

> [1] 60
```

When a function is called, the computations in the body are run line by line. When R cannot find an object inside the function, it looks outside the function.

# Where does a function find objects?

## Name masking!

Objects inside the function mask objects outside the function with the same name.

```
a <- 55
add_a <- function(x){
  a <- 5
  return(x + a)
}
add_a(5)

> [1] 10
```



## ... dot-dot-dot

R has a special argument (in the definition of the function):

... (dot-dot-dot)

Useful when you don't know how many arguments there will be.

**Examples:**

- ?sum
- ?save
- ?cbind
- ?paste
- ...

## ... dot-dot-dot

A function that checks for multiple objects if they are character vectors. (A wrapper around `(?is.character)`)

```
is_character <- function(...){  
  input <- list(...)  
  out <- logical(length(input))  
  for(ell_nr in seq_along(input)){  
    out[ell_nr] <- is.character(input[[ell_nr]])  
  }  
  names(out) <- names(input)  
  out  
}  
is_character(a = "Awesome", b = 5, new = "YES")  
  
>      a      b    new  
> TRUE FALSE TRUE
```

## ... dot-dot-dot

... can take *any* number of additional arguments.

Useful for passing arguments to other functions like:

- apply-family
- plot-family
- ...

## ... dot-dot-dot

apply example:

```
get_quantiles <- function(x, ...){  
  if(is.null(dim(x))) return(quantile(x, ...))  
  apply(x, 2, quantile, ...)  
}  
get_quantiles(airquality, na.rm = TRUE,  
              probs = c(.2, .8))
```

```
>      Ozone Solar.R  Wind Temp Month Day  
> 20%    14      92  6.90   69   5.4   7  
> 80%    73     266 12.96   86   8.0  25
```

### WARNING!

Watch out with spelling mistakes, arguments can get lost!

```
get_quantiles <- function(x, ...){  
  if(is.null(dim(x))) return(quantile(x, ...))  
  apply(x, 2, quantile, ...)  
}  
get_quantiles(airquality, na.rm = TRUE,  
              probs = c(.2, .8))
```

```
>      Ozone Solar.R Wind Temp Month Day  
> 0%      1.00      7.00  1.7   56     5   1  
> 25%     18.00    115.75  7.4   72     6   8  
> 50%     31.50    205.00  9.7   79     7  16  
> 75%     63.25    258.75 11.5   85     8  23  
> 100%    168.00    334.00 20.7   97     9  31
```

# Writing Functions

Before creating a function

- What should my function do?
- Which input objects (Arguments)?
- Which additional options (Arguments)?
- What should the output object be?

After creating a function

- Test it
- Add input validation
- Document

# Exercises



# Functionals

---



# Higher Order Functions

Higher order functions are functions that either **take functions as input** or **return functions as output**.

# Functionals

As defined by Hadley Wickham: A **functional** is a function that takes another function as an input. Common argument names are `FUN` or `f`.

## Examples

- `apply-family`
- `Reduce, Filter`
- `nlm`
- `optimize`
- ...

The apply-family *applies* a function repeatedly. This can be seen as an abstraction of a for loop, with the following advantages:

- requires less code to write
- can be easier to read / understand
- does not store intermediate results
- no need to replace / grow

The members of the apply-family in Base R are:

- `lapply` vector / list  $\rightarrow$  list
- `sapply` vector / list  $\rightarrow$  vector (matrix)
- `apply` matrix / array / data.frame  $\rightarrow$  vector (matrix)
- `tapply`, `by`
- `mapply`, `Map`
- `rapply`, `eapply`, `vapply`

A popular alternative from the tidyverse: `purrr`-package

- `map` vector / list  $\rightarrow$  list
- `map2` multiple vectors / lists  $\rightarrow$  list
- ...

Our focus: `lapply`

**Why?**

- Consistent output
- Fast
- No dependencies
- We want to understand R basics

# lapply

`lapply` has two main arguments

`X` the input list/vector

`FUN` the function that should be repeatedly applied

```
example_list <- list(vec1 = c(1, 3, 4),  
                     vec2 = c(4, 2, 10),  
                     vec3 = c(2, NA, 1))  
lapply(example_list, FUN = mean)
```

```
> $vec1  
> [1] 2.666667  
>  
> $vec2  
> [1] 5.333333  
>  
> $vec3  
> [1] NA
```

# lapply

Other arguments can be passed through lapply via '...'.  
lapply

```
example_list <- list(vec1 = c(1, 3, 4),  
                     vec2 = c(4, 2, 10),  
                     vec3 = c(2, NA, 1))  
lapply(example_list, FUN = mean, na.rm = TRUE)
```

```
> $vec1  
> [1] 2.666667  
>  
> $vec2  
> [1] 5.333333  
>  
> $vec3  
> [1] 1.5
```



# lapply

We can use our own functions as input.

```
dropNAs <- function(x) {  
  x[!is.na(x)]  
}  
lapply(example_list, FUN = dropNAs)  
  
> $vec1  
> [1] 1 3 4  
>  
> $vec2  
> [1] 4 2 10  
>  
> $vec3  
> [1] 2 1
```

Anonymous functions can be used as input.

```
lapply(example_list, FUN = function(x) x[!is.na(x)])
```

```
> $vec1
```

```
> [1] 1 3 4
```

```
>
```

```
> $vec2
```

```
> [1] 4 2 10
```

```
>
```

```
> $vec3
```

```
> [1] 2 1
```

# lapply

Data.frames are lists, too.

```
lapply(iris, FUN = class)
```

```
> $Sepal.Length
```

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

```
>
```

```
> $Sepal.Width
```

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

```
>
```

```
> $Petal.Length
```

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

```
>
```

```
> $Petal.Width
```

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

```
>
```

```
> $Species
```

```
> [1] "factor"
```

# lapply

Vectors can be used as input, but often vectorization could be used instead.

```
lapply(c(1, 2, 3), FUN = function(x) {  
  paste0("ID", x)  
})
```

```
> [[1]]  
> [1] "ID1"  
>  
> [[2]]  
> [1] "ID2"  
>  
> [[3]]  
> [1] "ID3"
```

Limitation of `lapply`:

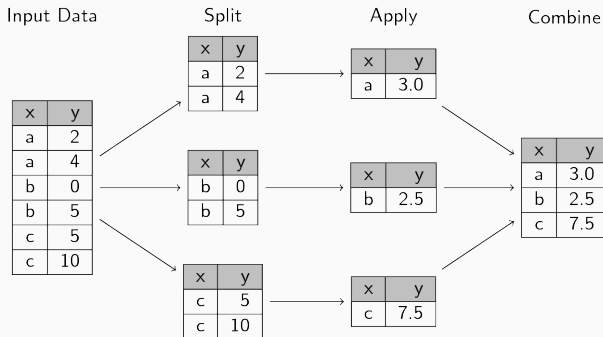
Only a single list/vector can be supplied as input. `Map` is a generalization of `lapply`! It is usually needed less often but a very powerful tool.

# Split & Apply & Combine

A common use case for the apply-family is the **Split & Apply & Combine** paradigm. Here, we want to perform the same analyses for various subgroups in our data set:

- **split** a data.frame or vector (`?split`)
- **apply** computations on each split (`?lapply`)
- **combine** the results (`?do.call`)

# Split & Apply & Combine



# Split & Apply & Combine

```
head(iris)
```

```
> Sepal.Length Sepal.Width Petal.Length Petal.Width Species
> 1           5.1           3.5           1.4           0.2 setosa
> 2           4.9           3.0           1.4           0.2 setosa
> 3           4.7           3.2           1.3           0.2 setosa
> 4           4.6           3.1           1.5           0.2 setosa
> 5           5.0           3.6           1.4           0.2 setosa
> 6           5.4           3.9           1.7           0.4 setosa
```

```
table(iris$Species)
```

```
>
>      setosa versicolor virginica
>       50       50       50
```



# Split & Apply & Combine

Splitting the data set via a single (or multiple) grouping variables

```
data_list <- split(iris, f = iris$Species)
class(data_list)

> [1] "list"

length(data_list)

> [1] 3
```

# Split & Apply & Combine

Apply the same computation to all data sets

```
out_list <- lapply(data_list, function(subdat) {  
  mod <- lm(Sepal.Length ~ Sepal.Width, data = subdat)  
  sum_mod <- summary(mod)  
  out <- c(Intercept = coef(mod)[[1]],  
    Slope = coef(mod)[[2]],  
    r2 = sum_mod$r.squared)  
  round(out, 3)  
})
```

## Split & Apply & Combine

```
out_list[["virginica"]]
```

```
> Intercept      Slope      r2  
>      3.907      0.902      0.209
```

# Split & Apply & Combine

Combine the results

```
do.call(rbind, out_list)
```

```
>           Intercept Slope    r2
> setosa         2.639 0.690 0.551
> versicolor     3.540 0.865 0.277
> virginica      3.907 0.902 0.209
```

# Exercises



# Good programming practices

---

“Write code for humans, not for machines!”

Invest time in writing readable R-code.

- It will make collaborations easier
- It will make debugging easier
- It will make your analyses more reproducible

There is a complete *tidyverse* style-guide

<https://style.tidyverse.org/>.



# Go easy on your eyes

- with spaces before and after: `- + / * = <- < == >`
- always use `<-` for assignments
- only use `=` in function calls
- use indentation (largely automatic in RStudio)
- `CamelCaseNames` vs `snake_case_names`
- be consistent!
- wrap long lines at column 70-80 (Rstudio)

# White space

```
new_var=(var1*var2/2)-5/(var3+var4)
```

```
# versus
```

```
new_var <- (var1 * var2 / 2) - 5 / (var3 + var4)
```

# Indentation

```
for(name in names){formula=as.formula(paste0("y~.",name))
fit<-lm(formula,data=my_data)
coefs[["name"]]=coef(fit)
print(name)
print(summary(fit))}
```

# versus

```
for(name in names){
  formula <- as.formula(paste0("y~.", name))
  fit <- lm(formula, data = my_data)
  coefs[["name"]] <- coef(fit)
  print(name)
  print(summary(fit))
}
```

## Wrap long lines

```
final_results <- data.frame(first_variable =  
  sqrt(results$mean_squared_error), second_variable =  
  paste0(results$condition, results$class, sep = ":"),  
  third_variable = results$bias)
```

# versus

```
final_results <- data.frame(  
  first_variable = sqrt(results$mean_squared_error),  
  second_variable = paste0(results$condition,  
                           results$class, sep = ":"),  
  third_variable = results$bias)
```

# Go easy on your mind

- use meaningful names: “self-explainable”
- always write the formal arguments in function calls (except the first)
- benefit from autocompletion (`<tab>`) => embrace longer names
- use `TRUE` and `FALSE` not `T` and `F`
- comment, comment, comment
  - NOT what (should be clear from the code)
  - but why
  - explain the reasoning, not the code

# Use meaningful names

```
V <- myFun(m1_B)
```

```
# versus
```

```
RMSE_age_gender <- get_RMSE(lm_age_gender)
```

## Programming advice

Use verbs for functions and nouns for other objects.

# Write formal arguments

Benefit from auto completion using tab

```
m1_B <- lm(outcome ~ age*gender,  
           exp1, condition_1, freq)
```

# versus

```
lm_age_gender <- lm(outcome ~ age * gender,  
                   data = exp1,  
                   subset = condition_1,  
                   weights = freq)
```

# Comment, comment, comment

```
## Start every Rscript with a comment that explains
##  what the code in the script does, why it does
##  this, and to which project it belongs.
##  Your future self will be very thankful!
##
## Mention which packages you are using in this Rscript.

## Use sections to separate chunks -----

## Maybe even subsections =====

## Recode variables so that missings are coded as "NA"
dat[dat %in% c(99, 999)] <- NA # missings coded 99 or 999
```



# Keep your code slim

Try to limit your *package-dependencies*.

Only load `library()` the packages that you absolutely need. If you are only using `dplyr`, it does not make sense to load the complete `tidyverse`.

**Controversial:** when possible, use the `::` operator (and consider not loading the package). `<package>::<function>`

- explicit dependencies
- less name conflicts

# Never Attach

Forget about `attach()`!

Don't use it, unless you completely understand what happens (see `?attach`).

Use `with(data.frame, expression)` instead.

```
# using with()
n <- 2e+4
data <- data.frame(x = runif(n),
                   y = runif(n),
                   z = seq_len(n))
result <- with(data, exp(x) / log(z) + 5 * sqrt(y))
```

Writing code is error prone. Incorporate tests and checks in your workflow.

- minimal examples
- write tests and checks
- helpful packages: `testthat`, `RUnit`, `testit`, ...

Computing speed can become an issue. Avoid common pitfalls:

- don't grow, but replace
- vectorize where possible
- check the computing speed

?system.time, microbenchmark or profiling tools

```
n <- 2e+4  
data <- data.frame(x = runif(n),  
                   y = runif(n),  
                   z = seq_len(n))
```

Don't grow!

```
system.time({  
  new_data <- NULL  
  
  for(row_nr in seq_len(NROW(data))){  
    new_data <- cbind(  
      data[row_nr,],  
      result = exp(data$x[row_nr]) /  
        log(data$z[row_nr]) +  
        5 * sqrt(data$y[row_nr]))  
  }  
})
```

```
>      User      System verstrichen  
>      2.13      0.06      2.19
```

Replace!

```
system.time({  
  n_rows <- dim(data)[1]  
  data$result <- rep(NA, n_rows)  
  
  for(row_nr in seq_len(n_rows)){  
    data$result[row_nr] <- exp(data$x[row_nr]) /  
      log(data$z[row_nr]) +  
      5 * sqrt(data$y[row_nr])  
  }  
})
```

```
>      User      System verstrichen  
>      0.28      0.42      0.71
```

Vectorize!

```
system.time({  
  data$result <- exp(data$x) / log(data$z) +  
    5 * sqrt(data$y)  
})
```

```
>      User      System verstrichen  
>      0      0      0
```

# Speed

Compare the speed of different implementations using:

`microbenchmark::microbenchmark`

```
get_mean1 <- function(x){  
  weight <- 1/length(x)  
  out <- 0  
  for(i in seq_along(x)){  
    out <- out + x[i] * weight  
  }  
  return(out)  
}
```

```
get_mean2 <- function(x){  
  sum(x)/length(x)  
}
```



Compare the speed of different implementations using:

```
microbenchmark::microbenchmark
```

```
x <- rnorm(500)
microbenchmark::microbenchmark(
  mean(x), get_mean1(x), get_mean2(x))
```

> Unit: nanoseconds

	expr	min	lq	mean	median	uq	max	neval
>	mean(x)	3400	3501	3689.01	3502	3602	10400	100
>	get_mean1(x)	13401	13601	38963.99	13701	13802	2536500	100
>	get_mean2(x)	601	701	9639.95	702	801	886001	100

## Programming advice

Don't worry about speed before it becomes an issue.

## Wrap Up

---

- Investing time in learning R pays off
- It's a steady learning curve
- Learn from masters
- Rewrite important code - the first attempt is usually not the best approach

# General R Advice

- Document well
- Use a consistent style
- Write functions
- Split long functions in smaller ones
- Write wrappers
- Use Iteration (don't copy paste)
- Use matrix operations and vectorized functions instead of loops
- Use git

## R Resources

- Advanced R Ed. 1 (<http://adv-r.had.co.nz/>)
- Advanced R Ed. 2 (<https://adv-r.hadley.nz/>)
- R Inferno ([https://www.burns-stat.com/pages/Tutor/R\\_inferno.pdf](https://www.burns-stat.com/pages/Tutor/R_inferno.pdf))
- R Packages (<https://r-pkgs.org/>)
- Clean Code ([https://thixalongmy.haugiang.gov.vn/media/1175/clean\\_code.pdf](https://thixalongmy.haugiang.gov.vn/media/1175/clean_code.pdf))
- The Pragmatic Programmer

Thank you for your attention!

Thank you for your attention!

Questions? Remarks?