Programming in R

Dries Debeer & Benjamin Becker

31. March and 01. April 2022

FDZ Spring Academy

Introduction

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

Introduction

Who are you?

1. Occupation, employer?
2. Previous knowledge and experience
   - with R?
   - with other statistical software?
   - with other programming languages?
3. Specific interest/motivation for this workshop?

Motivation

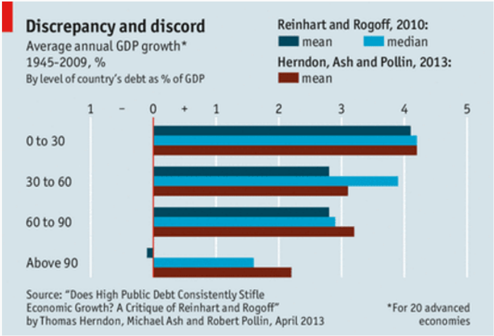
1. Increase efficiency!
    - Save time and nerves
    - Avoid errors and bugs
    - High transfer effect to all projects (with data analyses)
2. Successful collaborations (including with your future self!)
3. Code as deliverable (i.e., part of research paper)

Motivation

Two of your worst collaborators:

1. Past Self
    - the biggest mess in existence
    - did not document anything
    - uses a completely different style of writing code
    - does not reply to e-mails
2. Future Self
    - has the memory of a goldfish
    - will have zero understanding for your current brilliance

Motivation


Discrepancy and discord
Average annual GDP growth*
1945-2009, %
By level of country's debt as % of GDP
Reinhart and Rogoff, 2010:
mean
median
Herndon, Ash and Pollin, 2013:
mean
1
-
0
+
1
2
3
4
5
0 to 30
30 to 60
60 to 90
Above 90
Source: "Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff" by Thomas Herndon, Michael Ash and Robert Pollin, April 2013
*For 20 advanced economies

Motivation

The Sunday Telegraph Sunday 17 May 2020 ... 7

Coronavirus

Modelling behind lockdown was an unreliable buggy mess, claim experts

Data that predicted 500,000 could die in UK unless extreme measures were taken are impossible to replicate, say scientific teams

Science

By Hannah Boland *and* Ellie Zolfagharifard

THE Covid-19 modelling that sent Britain into lockdown, shutting the economy and leaving millions out of work has been criticised by experts.

Prof Neil Ferguson's Imperial College computer coding was derided as "totally unreliable" by leading figures, who warned it was "something you wouldn't stake your life on".

The model, credited with forcing the Government to U-turn and introduce a nationwide lockdown, is a "buggy mess, which looks more like a bowl of angel hair pasta than a finely tuned piece of programming", said David Richards, the co-founder of British data technology company WANdisco.

"In our commercial reality, we would fire anyone for developing code like this and any business that relied on it to produce software for sale would likely go bust."

The comments are likely to reignite a row over whether the UK was right to go into lockdown, with conflicting models suggesting people may have already acquired substantial herd immunity and Covid-19 may have hit Britain earlier than first thought.

Scientists have also been split on the fatality rate of Covid-19, which has resulted in vastly different models.

Up until now, significant weight has been attached to Imperial's model, which placed the fatality rate higher than others and predicted 510,000 in the UK could die without a lockdown.

It was said to have prompted a dramatic change in government policy, causing businesses, schools and restaurants to be shut immediately in March. The Bank of England has predicted that the economy could take a year to return to normal, after its worst recession in more than three centuries.

The Imperial model works by using code to simulate transport links, population size, social networks and healthcare provisions to predict how coronavirus would spread. However, questions have emerged over whether the model is accurate, after researchers released its code, which in its original form was "thousands of lines" developed over more than 13 years.

In its initial form the code was unreadable, developers claimed, with some parts looking "like they were machine translated from Fortran", an old coding language, according to John Carmack, a US developer, who helped clean the code before it was published.

Yet, the problems appear to go much deeper than messy coding. Many have claimed that it is almost impossible to reproduce the same results from the same data, using the same code.

Scientists from the University of Edinburgh said they got different results when they used different machines, and even in some cases using the same machines. "There appears to be a bug in either the creation or re-use of the network file. If we attempt two completely identical runs, only varying in that the second should use the network file produced by the first, the results are quite different," the Edinburgh researchers wrote on the Github online forum. After a discussion with a Github developer, a fix was provided.

It is said to be one of a number of bugs discovered within the system. Github developers said that the model was "stochastic" (random), and "multiple runs with different seeds should be undertaken to see average behaviour".

It has prompted questions from specialists, who say "models must be capable of passing the basic scientific test of producing the same results given the same initial set of parameters ... otherwise, there is simply no way of knowing whether they will be reliable." It comes amid a wider debate over whether the Government should have relied more heavily on numerous models before making policy decisions.

Writing for the *Telegraph* online, Prof Sir Nigel Shadbolt, the principal of Jesus College, Oxford, and chairman of the Open Data Institute – which he co-founded with World Wide Web inventor Sir Tim Berners-Lee – said: "Having a diverse variety of models, particularly those that enable policymakers to explore predictions under different assumptions, and with different interventions, is incredibly powerful."

Like the Imperial code, a rival model by Prof Sunetra Gupta at the University of Oxford works on a so-called "SIR approach" in which the population is divided into those that are susceptible, infected and recorded. However, while Prof Gupta assumed that 0.1 per cent of infected people would die, Prof Ferguson worked on 0.9 per cent. That led to a dramatic reversal in government policy from attempting to build "herd immunity" to a full-on lockdown.

Concerns over Prof Ferguson's model have been raised, with Dr Konstantin Boudnik, the VP of architecture at WANdisco, saying his track record did not inspire confidence. In the early 2000s, Prof Ferguson's models incorrectly predicted up to 136,000 mad cow disease deaths, 200 million from bird flu and 65,000 from swine flu.

"The facts from the early 2000s are just yet another confirmation that their modelling approach was flawed to the core," says Dr Boudnik. "We don't know for sure if the same model/code was used, but we clearly see their methodology wasn't rigorous then and surely hasn't improved now."

A spokesman for Imperial's Covid-19 team said: "The Government has never relied on a single disease model to inform decision-making. As has been repeatedly stated, decision-making around lockdown was based on a consensus view of the scientific evidence, including several modelling studies by different academic groups.

"Multiple groups using different models concluded that the pandemic would overwhelm the NHS and cause unacceptably high mortality in the absence of extreme social distancing measures. Within the Imperial team, we use several models of differing levels of complexity, all of which produce consistent results. We are working with ... legitimate academic groups and technology companies to develop, test and further document the simulation code referred to. However, we reject the partisan reviews of a few clearly ideologically motivated commentators.

"Epidemiology is not a branch of computer science and the conclusions around lockdown rely not on any mathematical model but on the scientific consensus that Covid-19 is highly transmissible with an infection fatality ratio exceeding 0.5 per cent in the UK."

'In our commercial reality, we would fire anyone for developing code like this'

'Any business that relied on it to produce software for sale would likely go bust'

'It looks more like a bowl of angel hair pasta than a finely tuned piece of programming'

'The early 2000s were yet another confirmation that their modelling approach was flawed to the core'

Goal of this workshop

An introduction to R as a Programming language

- Better practical R skills
- Better understanding of R (and programming)
- Different framing: R as a programming language

Agenda

Day 1

- RStudio setup
- Flow & conditional programming
- Loops & iteration
- Functions (part I)

Day 2

- Functions (part II)
- *Functionals* & split-apply-combine
- Good programming practices

RStudio setup

RStudio setup

1. Save the course content to a directory on your machine
2. Open RStudio
3. Choose `File < New Project ...`
4. Choose `Existing Directory`
5. Browse to the directory on your machine where you saved the course content and select the "R-programming" folder as the `Project working directory`
6. Click `Open in new session`
7. Click `Create Project`

RStudio setup - optional

1. Choose `Tools < Global options`
2. Under `General`
   - DON'T `Restore .RData into workspace at startup`
   - NEVER `Save workspace to .Rdata on exit:`
3. Further personalize RStudio

Flow & conditional programming

Flow & conditional programming

R has specific tools (functions) that help organize the flow of computations.
You can make computations conditional on other objects ("conditional computation")
The most commonly used tools are:

- `if` (+ `else`)
- `ifelse`

Conditional Computation - if

`if` statements have the basic form

```
if(test){
  some_computations
}
```

- `test` should be either `TRUE` or `FALSE` (or code that results in one of both).
- If `test == TRUE`, than `some_computations` is executed, if `test == FALSE`, than not.
- **Important**: `test` should have length 1. If not, only the first element is considered.

Conditional Computation - if

`else` can be added, but it is optional

```
if(test){
  some_computations
} else if (test_2){
  other_computations
} else {
  more_computations
}
```

Typical test functions

Vectorized, elementwise	
`==`	equal to
`!=`	NOT equal to
`>`, `<`	is greater, less than
`>=`, `<=`	is greater, less than or equal to
`&`	AND operator
`|`	OR operator
`xor`	exclusive OR

Typical test functions

Not Vectorized	
`identical()`	identical to
`any()`	at least one `TRUE`
`all()`	all `TRUE`
`&&`	AND operator
`||`	OR operator
`is.character()`, `is.data.frame()`, ...	

Typical test functions

Compare:

```
c(TRUE, TRUE) & c(FALSE, TRUE)

> [1] FALSE  TRUE

c(TRUE, TRUE) && c(FALSE, FALSE)

> [1] FALSE
```

Conditional Computation - if

The test should have length 1!

```
# only the first element is evaluated
age <- c(8, 17, 39, 55)
if (age >= 18) {
  "can vote"
} else {
    "too young"
}

> Warning in if (age >= 18) {:  the condition has length > 1 and
only the first element will be used

> [1] "too young"
```

Conditional Computation - if

Typical uses

```
if(any(is.na(x))){
  stop("computation impossible due to NA values")
}

if(!is.integer(vector)){
  warning("'vector' is automatically converted to integer.
          This may affect the results")
  vector <- as.integer(vector)
}

if(is.null(argument)){
  # default computations
} else if (argument == specific_value) {
  # other computations
}
```

Conditional Computation - if

Programming advice

- *if* is almost always used inside of functions or loops
- If possible, avoid using *else*
- Use meaningful initialization, early return(), stop(), etc. instead

Conditional Computation - if

Solution using *if* and *else*

```
age <- 17
if (age >= 18) {
  vote <- "can vote"
} else {
  vote <- "too young"
}
vote

> [1] "too young"
```

Conditional Computation - if

Solution using meaningful initialization

```
age <- 17
vote <- "too young"
if (age >= 18) {
  vote <- "can vote"
}
vote

> [1] "too young"
```

Conditional Computation - ifelse

A vectorized version is `ifelse()`.

```
# all elements are evaluated
age <- c(8, 17, 39, 55)
ifelse(age >= 18,
       yes = "can vote",
       no = "too young")

> [1] "too young" "too young" "can vote"  "can vote"
```

Conditional Computation - ifelse

Go-to tool for conditional recoding

```
age_estimated <- c(10, 20, 35, 60)
age_self_rep <- c(NA, 17, 39, NA)

# Use available information, prioritize self report
ifelse(!is.na(age_self_rep),
       yes = age_self_rep,
       no = age_estimated)

> [1] 10 17 39 60
```

Exercises

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`)
- *functionals* (`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 - for

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
```

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 - while

Sample five random values from a normal distribution, the distance between the minimum and maximum should be at least 4.

```
max_dif <- 0
while (max_dif <= 4){
  cat("|")
  values <- rnorm(5)
  max_dif <- max(values) - min(values)
}

> |||||||||||||||||||||||||||||||||||||

max_dif

> [1] 4.476298
```

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)

```
index <- 0
repeat {
  index <- index + 1
  if (index %in% c(3, 5)) next
  if (index > 6) break
  print(index)
}

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

Iteration - Good practice

Programming advice

Use `seq()`, `seq_len()`, or `seq_along()`.

```
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 - *Functionals*

There are specific functions that allow efficient and clean iteration.
We focus on the `apply`-family. These functions *apply* a function repeatedly.
Can be seen as an abstraction of a for loop, with the following advantages

- requires less code to write
- does not store intermediate results
- no need to replace / grow

(also see the `purrr` package)

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
```

lapply

data.frames are lists with the columns as elements:

```
lapply(iris, FUN = class)


> $Sepal.Length
> [1] "numeric"
>
> $Sepal.Width
> [1] "numeric"
>
> $Petal.Length
> [1] "numeric"
>
> $Petal.Width
> [1] "numeric"
>
> $Species
> [1] "factor"
```

lapply

- any type of element can be used
- other arguments can be passed through

```
means <- lapply(mtcars[1:7], FUN = mean)
str(means)


> List of 7
>  $ mpg : num 20.1
>  $ cyl : num 6.19
>  $ disp: num 231
>  $ hp  : num 147
>  $ drat: num 3.6
>  $ wt  : num 3.22
>  $ qsec: num 17.8
```

Functionals

More on how these *functionals* can be used tomorrow.

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

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

Stepping Stone

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

```
                                    # Name
countNA <- function(x) {            # Arguments/Formals
  out <- sum(is.na(x))              # Body
  out                               # 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

Arguments

Examples for zero arguments

- getwd()
- Sys.time()

Arguments

Examples for one argument

- dim()
- names()

Arguments

Examples for multiple arguments

- mean()
- median()
- lm()

Arguments

Programming advice

Less arguments = better!

Arguments

Often arguments have to by objects of a specific type.

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

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

Single return object

Pure functions return a single object.

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

Figure 2: A pure function.

Single 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
```

Single return object

Multiple return objects can be combined in a list!

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

Single Return Object

The return object is 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
```

Default arguments

What happens if the user omits an argument?

```
return_early <- function(x, early) {
  x2 <- x*2
  if(early) (return(x2))
  out <- x + x2 # not executed
  out
}
return_early(2)

> Error in return_early(2):  argument "early" is missing, with no
default
```

Default arguments

Default arguments are made for such instances!

```
return_early <- function(x, early = FALSE) {
  x2 <- x*2
  if(early) (return(x2))
  out <- x + x2 # not executed
  out
}
return_early(2)

> [1] 6
```

Default arguments

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

Try `?lm`

Programming advice

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

Exercises

Functions II

Reasons

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 transferability to ...
  - other use cases
  - other projects
  - other persons
- They keep your work space clean

Readability

Writing a function:

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

Not writing a function:

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

Readability

Writing a function:

```
summary(mtcars$mpg)

>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
>   10.40   15.43   19.20   20.09   22.80   33.90
```

Readability

Not writing a function:

```
round(c("Min." = min(mtcars$mpg),
  "1st Qu." = as.numeric(quantile(mtcars$mpg)[2]),
  "Median" = median(mtcars$mpg),
  "Mean" = mean(mtcars$mpg),
  "3rd Qu." = as.numeric(quantile(mtcars$mpg)[4]),
  "Max." = max(mtcars$mpg)), 2)

>     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
>    10.40   15.43   19.20   20.09   22.80   33.90
```

Side Effects

Functions can have **“side effects”**:

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

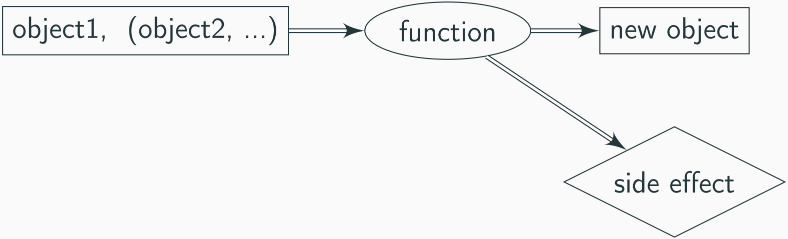


Figure 3: A function with side effect.

Side Effects

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.
```

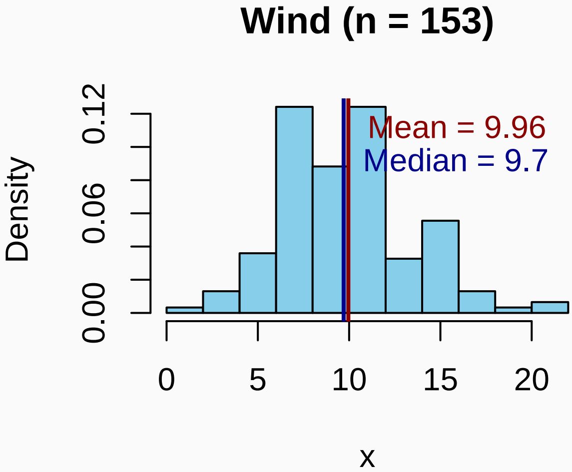
Side effects

Graphics output: Standard plot, ggplot2, lattice

```
hist2 <- function(x, title){
  info <- get_info(x)
  mean_median <- as.numeric(info[c("mean", "median")])
  hist(x, col = "skyblue", freq = FALSE,
       main = paste0(title, " (n = ", info$n_obs, ")"))
  abline(v = mean_median, lwd = 2,
         col = c("darkred", "darkblue"))
  text(mean_median, y = c(.11, .09),
       labels = paste(c("Mean", "Median"),
                      round(mean_median, 2),
                      sep = " = "),
       col = c("darkred", "darkblue"), pos = 4)
}
hist2(airquality$Wind, "Wind")
```

Side effects

Graphics output



Output

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

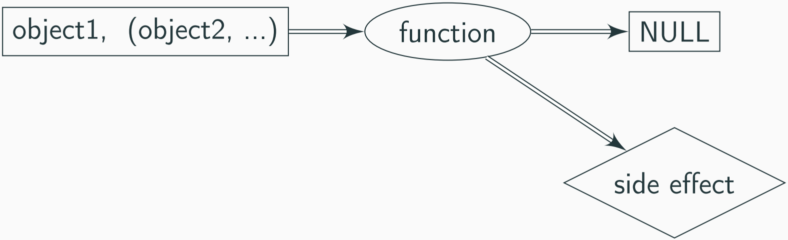


Figure 4: A side effect function.

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!

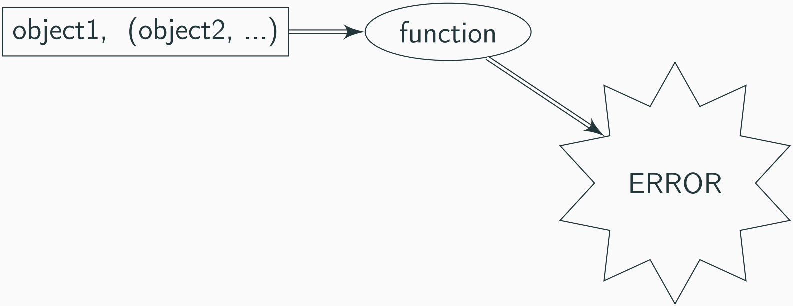


Figure 5: Computation with Error.

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) is not 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!

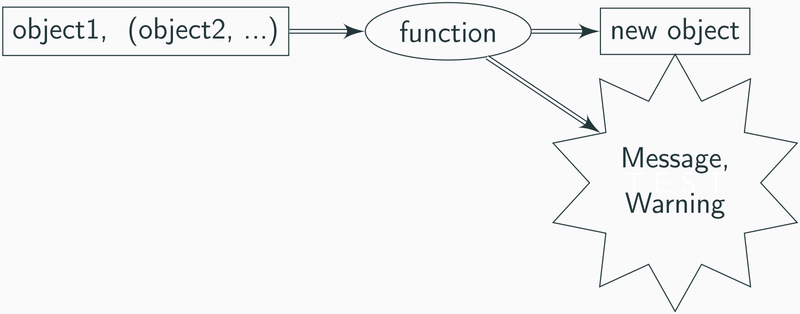


Figure 6: A message or warning.

Output

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
```

Functional programming

The return object should only depend on the arguments of the function, ***not*** on the context!

BAD:

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

> [1] 60
```

Functional programming

The return object should only depend on the arguments of the function, ***not*** on the context!

GOOD:

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


> [1] 60
```

Functional programming

The function should not change the context.

BAD

```
a <- 55
change_a <- function(new_a){
  a <<- new_a
  return(invisible(NULL))
}
change_a(5)
a

> [1] 5
```

... 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(...){
  lapply(list(...), is.character)
}
is_character(a = "Awesome", b = 5)


> $a
> [1] TRUE
>
> $b
> [1] FALSE
```

... 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, ...){
  apply(x, 2, quantile, ...)
}
get_quantiles(airquality, na.rm = TRUE,
              probs = c(.25, .5, .27))

>      Ozone Solar.R Wind Temp Month Day
> 25% 18.00  115.75  7.4   72     6   8
> 50% 31.50  205.00  9.7   79     7  16
> 27% 18.05  127.00  7.4   73     6   9
```

WARNING! Watch out with spelling mistakes, arguments can get lost!

```
get_quantiles <- function(x, ...){
  apply(x, 2, quantile, ...)
}
get_quantiles(airquality, na.rm = TRUE,
              prosb = c(.25, .5, .27))

>        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 the function

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

After creating the function

- Test it
- Add input validation
- Document

What makes a good function?

Pure functions!

- no side effects
- no dependency on global environment
- only input via arguments (functional programming)

Results in easier understanding and higher portability.

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

`apply`-family

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`

`apply`-family

A popular alternative from the tidyverse: `purrr`

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

`apply`-family

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 '...'.

```
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
```

lapply

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

Atomic 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"
```

lapply

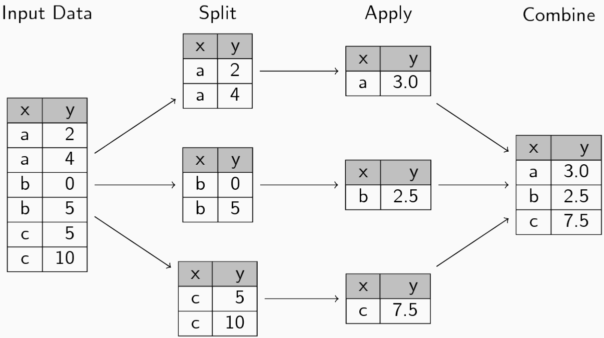
Limitation of `lapply`:
Only a single list/vector can be supplied as input. `Map` is a generalization of `lapply`! It is usually needed less often but a very powerful tool.

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!”

Code Style

Invest time in writing readable R-code.

- It will make collaborations easier
- It will make debugging easier
- It will help make your analyses 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

```r
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

```r
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,
                    weigths = freq)
```

Comment, comment, comment

```r
## 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))
```

Testing R code

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

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

Speed

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))
```

Speed

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 elapsed
>    1.73    0.00    1.73
```

Speed

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 elapsed
>    0.29    0.01    0.31
```

Speed

Vectorize!

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


>    user  system elapsed
>       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)
}
```

Speed

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 cld
>       mean(x)  1900  2000  2285   2100  2200    9900   100   a
>  get_mean1(x) 11000 11300 33302  12200 12500 2147900   100   a
>  get_mean2(x)   600   600  8662    700   700  794900   100   a
```

Speed

Programming advice

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

Wrap Up

General Advice

- 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

Literature Recommendations

R Resources

- Avanced R Ed. 1 (http://adv-r.had.co.nz/)
- Avanced R Ed. 2 (https://adv-r.hadley.nz/)
- R Inferno (https://www.burns-stat.com/pages/Tutor/R_inferno.pdf)
- R Packages (https://r-pkgs.org/)
- Clean Code (https://enos.itcollege.ee/~jpoial/oop/naited/Clean%20Code.pdf)

Thank you for your attention!

Thank you for your attention!

Questions? Remarks?