

Advanced Programming with R

Dries Debeer & Benjamin Becker

15. October 2021

Zurich University

Introduction

Who are we?

Dries Debeer

Senior Researcher at itec (imec research group at KU Leuven)

scDIFtest, permimp, eatATA, mstDIF

dries.debeer@kuleuven.be

Benjamin Becker

Researcher at IQB (Statistics Department)

eatGADS, eatDB, eatATA, pisaRT

b.becker@iqb.hu-berlin.de

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?

What do we expect from you?

You ...

- are a frequent R user (e.g. for data wrangling/analyses)
- have a rough understanding of R as a programming language
- have already written some R functions yourself
- are interested in learning more about the intricacies of R

Goal of this workshop

A deeper dive into R as a Programming language

- Better understanding of how R works as a programming language
- Better understanding of how larger programming projects can/should be structured
- Practical Git skills
- Practical R package building skills

Agenda

- Scoping & Environments
- Functionals & Split-Apply Paradigm

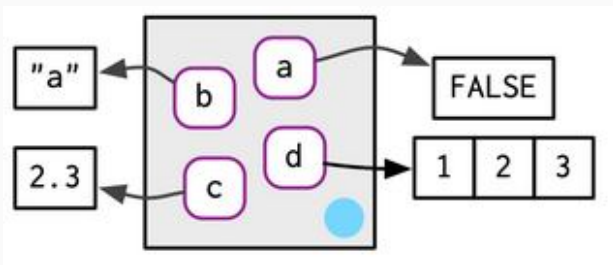
Lunch Break

- Object Orientation (S3)
- Packages & Version Controlling

Scoping & Environments

Environments

Like boxes, containing objects.



A bit simplified: If a function is called, its own environment is created with its parent being the environment from which it was called.

Environments

```
simple_fun <- function(){  
  a <- 1  
  b <- "a"  
  environment()  
}  
a <- simple_fun()  
rlang::env_print(a)  
  
# <environment: 0000000013C6E640>  
# parent: <environment: global>  
# bindings:  
# * b: <chr>  
# * a: <dbl>
```

Where does R find things?

- Argument matching (name, place...)
- Current environment
- Parent environment

Programming advice

Keep it simple, this can create chaos!

Scoping

```
add_things2 <- function(x) {  
  x + 10 + y  
}
```

```
add_things2(2)
```

```
# Error in add_things2(2): object 'y' not found
```

```
y <- 100
```

```
add_things2(2)
```

```
[1] 112
```

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 tidy-verse: `purrr`

- `map` vector / list \rightarrow list
- `map2` multiple vectors / lists \rightarrow vector (matrix)
- ...

Our focus: `lapply` and `Map`

Why?

- Consistent output
- Fast
- No dependencies
- Understanding R basics

lapply

`lapply` takes mainly two 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
```

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"
```

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.

Map

Works very similar to `lapply`, with a few differences:

- Multiple input lists/vectors
- The list input should be named explicitly
- The order of the function and the list-input is switched

```
list1 <- list(mtcars[1:2, 1:3], iris[1:2, c(1, 2, 5)])  
list2 <- list(mtcars[3:4, 1:3], iris[3:4, c(1, 2, 5)])  
  
Map(rbind, x = list1, y = list2)
```

Map

```
# [[1]]  
#           mpg cyl disp  
# Mazda RX4      21.0   6  160  
# Mazda RX4 Wag  21.0   6  160  
# Datsun 710      22.8   4  108  
# Hornet 4 Drive 21.4   6  258  
#  
# [[2]]  
# Sepal.Length Sepal.Width Species  
# 1           5.1           3.5  setosa  
# 2           4.9           3.0  setosa  
# 3           4.7           3.2  setosa  
# 4           4.6           3.1  setosa
```

Map

Again, anonymous functions can be supplied as input:

```
list1 <- list(mtcars[1:2, 1:3], iris[1:2, c(1, 2, 5)])
list2 <- list(mtcars[3:4, 1:3], iris[3:4, c(1, 2, 5)])

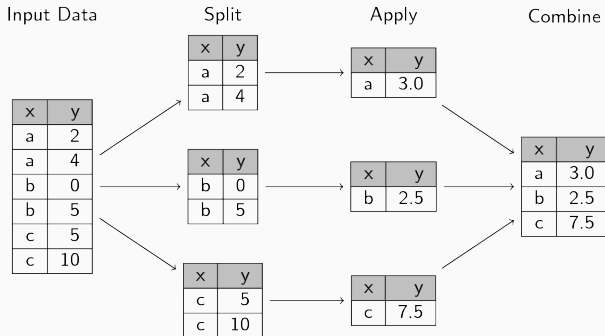
Map(function(x, y) {
  rbind(x, y)
},
x = list1, y = list2)
```

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 (`?sapply`, `?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



Object Oriented Programming (S3)

Why?

- User-friendly: same function for different objects (`summary()`)
- Coder-friendly: implementation can depends on object
- Coder-friendly: easier to maintain, extend

Object Oriented Programming

Basics

- a *class* is a definition
- an object is an *instance* of a class.
- a *method* is a procedure that behaves differently depending of the class it is applied to.
- *inheritance*: classes are organized in hierarchy.
“is-a-type-of”-relation.

Object Oriented Programming in R

- S3
- S4
- RC
- R6
- ...

Compromise between interactive use, functional programming and object-oriented programming.

- “naming conventions”
- used in base R
- flexible: new classes, new methods

- `class-attribute`
- generics and methods
- inheritance and dispatch

S3 - class

A new class is made by adding an arbitrary class attribute to an object.

```
object <- 1:10  
class(object) <- "myClass"  
attributes(object)  
  
# $class  
# [1] "myClass"
```

Very flexible! Use it wisely!

S3 - class

```
class(iris)
```

```
# [1] "data.frame"
```

```
iris[1:4, 1:3]
```

```
#   Sepal.Length Sepal.Width Petal.Length
# 1           5.1           3.5           1.4
# 2           4.9           3.0           1.4
# 3           4.7           3.2           1.3
# 4           4.6           3.1           1.5
```

Changing the class changes the behavior!

```
class(iris) <- "no data.frame"  
class(iris)
```

```
# [1] "no data.frame"
```

```
iris[1:4, 1:3]
```

```
# Error in iris[1:4, 1:3]: incorrect number of dimensions
```

Good practice: *constructor function*

- defines the structure of the class
- should be used to create objects of that class

```
myClass <- function(element1, element2){  
  # validation of the elements  
  structure(list(element1),  
            attribute1 = element2,  
            class = "myClass")  
}
```

Write a separate validation-function for computationally intensive validation.

S3 - Generics and Methods

- *methods* for a class belong to *generics*
- when a generic is called for a specific class, the corresponding method for that class will be used. common generics are:
`print()`, `plot()`, `summary()`, `anova()`
- same function, but different computation depending on object-class

```
body(print)
```

```
# UseMethod("print")
```


S3 - Generics and Methods

Creating a new method (for an existing generic):

```
print.myClass <- function(x, ...){  
  cat("This is a myClass-print:\n")  
  cat(round(c(nValues = length(x),  
            mean = mean(x),  
            SD = sd(x)), 3), ...)  
}  
  
my_object <- 1:5  
class(my_object) <- "myClass"  
print(my_object)  
  
# This is a myClass-print:  
# 5 3 1.581
```

Good practices (enforced for packages on CRAN):

- A method must have all the arguments of the generic, including `...` if the generic does.
- A method must have arguments in exactly the same order as the generic.
- If the generic specifies defaults, all methods should use the same defaults.

S3 - Generics and Methods

Inspect the generic!

```
formalArgs(print)
```

```
# [1] "x"    "..."
```

```
formalArgs(summary)
```

```
# [1] "object" "..."
```

S3 - Generics and Methods

Creating a new generic:

```
center <- function(x, ...)  
  UseMethod("center")  
  
center.myClass <- function(x, ...){  
  print("centering myClass:\n")  
  return(x - mean(x))  
}  
  
center(my_object)  
  
# [1] "centering myClass:\n"  
# This is a myClass-print:  
# 5 0 1.581
```

S3 - Inheritance and Dispatch

Objects can have more than one class

```
my_model <- glm(as.factor(books) ~ pared, data = pisa,  
               family = "binomial")  
  
class(my_model)  
  
# [1] "glm" "lm"  
  
class(my_model) == "lm"  
  
# [1] FALSE TRUE  
  
inherits(my_model, "lm")  
  
# [1] TRUE
```

Good practice: hierarchical inheritance (subclass and superclass)

S3 - Inheritance and Dispatch

When a method is not available for a (sub)class, the next available method (i.e., for the superclass) will be used.

```
"variable.names.glm" %in% methods(class = "glm")
```

```
# [1] FALSE
```

```
"variable.names.lm" %in% methods(class = "lm")
```

```
# [1] TRUE
```

```
variable.names(my_model)
```

```
# [1] "(Intercept)" "pared"
```

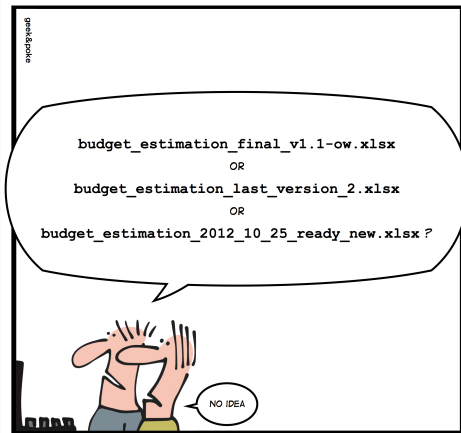
Exercises



Version Controlling (Git + Github)

- Motivation
- Setup
- Work flow
- Recommendations
- Resources

SIMPLY EXPLAINED



VERSION CONTROL

Motivation

- Implementation of long term change history
 - No ridiculous file names
 - No archive subfolder
 - Always perfect overview of file history and changes
- Collaborations
 - What has changed?
 - Who has changed it?
 - Documentation of changes
 - Parallel working possible (merging)

But...



See the workshop preparation materials

Creating a repository

- Create an **online repository** (e.g. on Github)
 - Use an R specific `.gitignore`
 - Initialize with a short readme (`.md`)
- Clone the repository to your local machine
- An R-Project is added automatically to the existing repository

- Plain text file
- Which files should not be tracked by git?
→ These then only exist locally in their current version!
- Options
 - Single files
 - Folder
 - Specific data types
 - Combinations of the above
- Use cases
 - Large files (Data, images, ...)
 - Auxiliary files (e.g. created during latex compilation)

Working with an existing repository

- Before working: Synch your local repo (**Pull** or **clone**)
- Perform changes in your local repository →
Create/modify/delete files
- **Stage** your changes
- **Commit** your changes (aka new version)
- **Push** your changes (online repository is updated)

Conflicts between different updated versions

- Common when working collaboratively
- Own discrepancies between different machines → Git communicates these and indicates conflicts
- Select the desired changes
- Stage selection, commit and push

Multiple parallel versions of a project within one repository

- e.g. one stable and one development branch
- Only certain modifications should be made in the stable branch
- Careful: RStudio GUI has limited support for this

Recommendations

- Keep it simple!
 - If not necessary, no branches/forks/pull requests
- Have meaningful commits
- Keep it lean (no big files)

Git (+ R) Resources

- Small Intro
(<https://r-bio.github.io/intro-git-rstudio/>)
- Happy Git with R (<https://happygitwithr.com/>)
- R Packages and Git (<https://r-pkgs.org/git.html>)
- Git Book (<http://git-scm.com/book/en/v2>)

R Packages

Motivation

- It is incredibly easy!
- It makes your code easier accessible (for others and yourself)
- It gives a great framework for documenting your code
- There are great tools for testing your code
- It is a form of scientific output (packages can be cited)

Use `usethis` and/or `RStudio` to set up everything.

- Create a Github Repository and clone it locally
- Create an R project (-> R package) in the existing repository
- Use `usethis` if you want to create more specific architecture

Your actual code lives in the `R` folder. Organize your functions logically and avoid very long scripts.

Using other packages

If you want to use another package in your source code, add it to the Imports-Field in your description file and use its functions using `package::function`.

Use `roxygen2` to document your code and to manage your namespace.

Use `testthat` for automated testing. This has multiple advantages:

- Good test coverage serves as a quality attribute of a good and stable package
- modifying existing code becomes much easier
- Your code becomes more robust
- It helps you sleep at night

Writing R packages

- Extending R
(<https://r-bio.github.io/intro-git-rstudio/>)
- R Packages (<https://r-pkgs.org/>)

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

- Advanced R Ed. 1 (<http://adv-r.had.co.nz/>)
- Advanced R Ed. 2 (<https://adv-r.hadley.nz/>)
- Notes on Functionals (<https://www.stat.umn.edu/geyer/8054/notes/functional.html>)
- R language definition (<https://cloud.r-project.org/doc/manuals/r-release/R-lang.html>)
- 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?