

# Advanced Programming in R

---

Dries Debeer & Benjamin Becker

15. October 2021

Zurich R Courses

# Introduction

---

# 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/plots)
- have a rough understanding of R as a programming language
  - e.g. you 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
- Object Orientation (S3)
- Packages & Version Controlling

# Environments

---



Understanding environments is key to understanding how R behaves, how R finds stuff.

- Scoping
- Closures
- Namespaces

# Environments

An environment = “a bag of objects”.

It differs from a list with respect to four key points:

1. All elements have names
2. The elements do not have an order
3. Environments are not copied when modified
4. Each environment has an **enclosing environment**

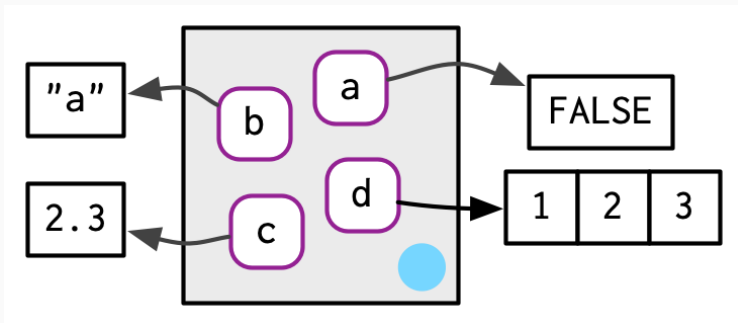
Each environment consists of two parts:

1. a **frame**: a collection of named objects.
  - the objects can be of any type (like in `lists`)
  - the objects have to be named (**unlike** in `lists`)
2. an **enclosing environment**: a reference or link to another environment.
  - the link goes in one direction
  - determines the search path

# Environments - frame

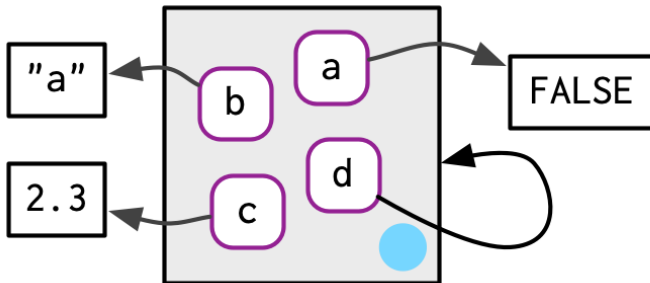
The frame is a “**bag**” of named objects.

- Actually, it is a bag of *bindings* between names and objects
- The combination *environment* + *name* points to a specific object (a location in memory)
- The names in the bag have no order



## Environments - frame

An environment can contain itself!



# Environments - Reference Semantics

How lists work:

```
original <- list(a = 15, b = "original", c = mean)
copy <- original
copy$b <- "new"
c(original$b, copy$b)

> [1] "original" "new"

original$a <- NULL
names(original)

> [1] "b" "c"
```

# Environments - Reference Semantics

How environments work:

```
original <- rlang::env(a = 15, b = "original", c = mean)
copy <- original
copy$b <- "new"
c(original$b, copy$b)
```

```
> [1] "new" "new"
```

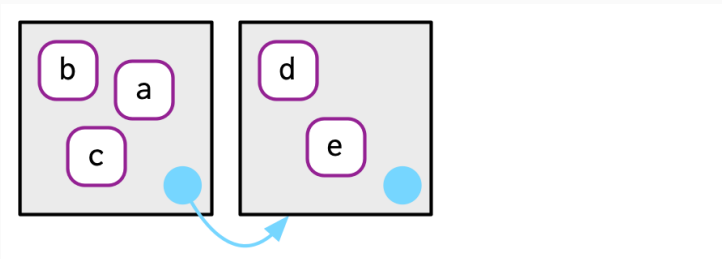
```
original$a <- NULL
names(original)
```

```
> [1] "a" "b" "c"
```

# Environments - enclosing environment

The link is used for lexical scoping:

If the reference to an object is not found in the environment, R starts searching in the enclosing environment



- the link goes in **one** direction
- an environment has only one enclosing environment
- an environment can enclose multiple environments (but it is hard to find out if so, and which)



# Exercises



## Environments - scoping

`?exists` returns a logical if a reference is found in an environment. The search jumps to the enclosing environment until the object is found when `inherits = TRUE` (the default).

`?get` returns an object based on a name and an environment. The search jumps to the enclosing environment until the object is found when `inherits = TRUE` (the default).

`?assign` assigns an object to a name in an environment.

If the reference to an object is not found in the environment, R starts searching in the enclosing environment

Some important environments:

- the global environment (`.GlobalEnv` or `globalenv()`)
- the empty environment (`emptyenv()`)
- the `package`-environment
- the `namespace`-environment
- the `imports`-environment

## The global environment

- the user's work space
- first item on the search path
- `.GlobalEnv` or `globalenv()`

## The empty environment

- An “empty bag”
- last item on the search path
- has no enclosing environment
- `emptyenv()`

# Search path

Where does R find objects *during interactive use*?

`?search()`

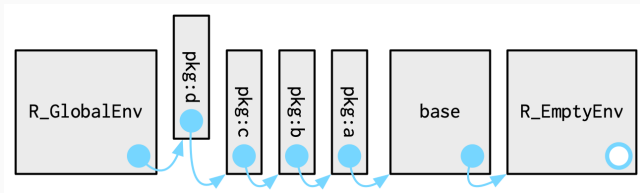
```
search()
```

```
> [1] ".GlobalEnv"          "package:knitr"       "package:stats"
> [4] "package:graphics"    "package:grDevices"   "package:utils"
> [7] "package:datasets"    "package:methods"     "Autoloads"
> [10] "package:base"
```

`package:<name>` environments have bindings to the exported objects (functions) of the package.

# Search path

Where does R find objects *during interactive use*?



Newly attached packages get a specific place in the search path. They become the enclosing environment of the global environment, and they are enclosed by the previously attached package.

# Environments and functions

Four environments are related to functions:

1. the environment that **binds** the function
2. the environment in which the body of the function is **executed**
3. the environment from which the function is **called**
4. the **enclosing** environment



# Environments and functions

1. the environment that binds the function

?find (can be misleading)

?pryr::where

```
add_10 <- function(x) x + 10  
find("add_10")
```

```
> [1] ".GlobalEnv"
```

```
find("sd")
```

```
> [1] "package:stats"
```

# Environments and functions

## 1. the environment that binds the function

```
sd <- sd  
find("add_10")  
  
> [1] ".GlobalEnv"  
  
find("sd")  
  
> [1] ".GlobalEnv"      "package:stats"
```

One name can be used in different environments (for different objects).

Functions (objects) can have references in more than one environment!

# Environments and functions

2. the environment in which the body is executed  
`environment(NULL)`

```
get_executing_env <- function() environment()  
get_executing_env()
```

```
> <environment: 0x00000000152cd578>
```

```
get_executing_env() # always new
```

```
> <environment: 0x0000000015332488>
```

The execution environment disappears when the body is executed. Unless it becomes the enclosing environment of another function.

# Environments and functions

3. the environment from which the function is called  
`?parent.frame`

```
print_calling_env <- function() parent.frame()  
print_calling_env()
```

```
> <environment: R_GlobalEnv>
```

```
other_fun <- function() print_calling_env()  
other_fun()
```

```
> <environment: 0x0000000013af5008>
```

## 3. the environment from which the function is called

This environment is often a execution environment of another function.

- To get the calling function use `?sys.call`.
- To get the complete call stack use `?sys.calls`.

3. the environment from which the function is called

`?sys.calls`

```
print_calling_funs <- function() sys.calls()
print_calling_funs()
other_fun <- function() print_calling_funs()
other_fun()
```

## 4. the enclosing environment

`environment(function)`

```
get_enclosing_env <- function() parent.env(environment())  
get_enclosing_env()
```

```
> <environment: R_GlobalEnv>
```

# Environments and functions

## 4. the enclosing environment environment(function)

```
make_adder <- function(add = 0) {  
  print(environment())  
  return(function(x) x + add)}  
add_5 <- make_adder(5)  
  
> <environment: 0x00000000152db580>  
  
environment(add_5)  
  
> <environment: 0x00000000152db580>
```

The execution environment disappears when the body is executed. Unless it becomes the enclosing environment of another function



# Environments and functions

## 4. the enclosing environment

When a function is created, the binding environment and the enclosing environment are typically equal. But the enclosing environment can be changed (using `environment <-`)

```
strange_mean <- function(x, ...) mean(x, ...)
strange_mean(1:3)
```

```
> [1] 2
```

```
env1 <- rlang::env(mean = function(x, ...) "Strange!")
environment(strange_mean) <- env1
strange_mean(1:3)
```

```
> [1] "Strange!"
```

# Environments and functions

Four environments are related to functions:

1. the environment that **binds** the function
2. the environment in which the body of the function is **executed**
3. the environment from which the function is **called**
4. the **enclosing** environment

# Where does R find stuff?

From *inside functions*?

1. the execution environment
2. the enclosing environment
3. the enclosing environment of that environment
4. ...
5. the empty environment

## Where does R find stuff?

From inside functions *defined in the global environment*?

1. the executing environment
2. the global environment (i.e., the enclosing environment)
3. the **package:<name>** environment of the last attached package
4. ...
5. the empty environment

# Where does R find stuff?

From inside functions *defined in packages*?

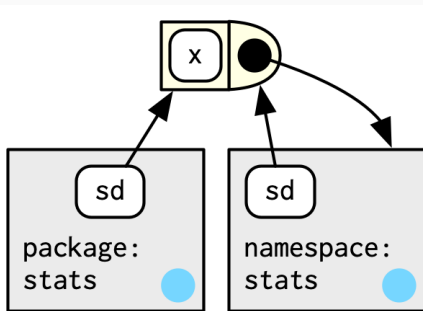
1. the executing environment
2. the `namespace:<name>` environment
3. the `imports:<name>` environment
4. the `namespace:base` environment
5. the global environment (i.e., the enclosing environment)
6. the `package:<name>` environment of the last attached package
7. ...
8. the empty environment

# Where does R find stuff?

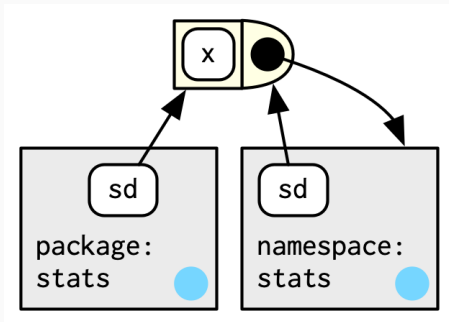
Exported functions from packages have two binding environments!

- the `namespace:<name>` environment
- the `package:<name>` environment

Both `::` and `:::` work.



## Where does R find stuff?



But, only one enclosing environment: the **namespace:<name>** environment!

Thus, from inside a (package-) function, R first looks for imported and exported functions of that package!

## Where does R find stuff?

Unexported functions from packages have one binding environment: the `namespace:<name>` environment

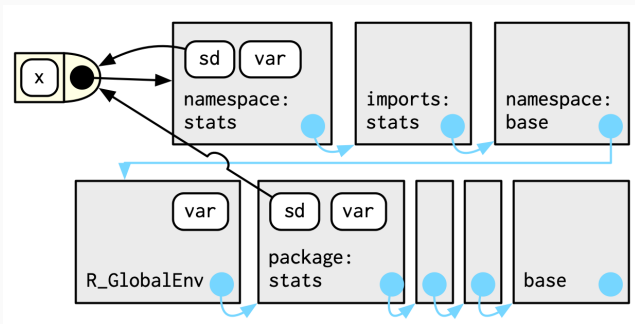
Which is also the enclosing environment

Only `:::` works.



# Where does R find stuff?

Where does R find functions from packages?



You can do crazy things with environments.

If you want to make use of reference semantic (i.e., modify objects in place), use R6

## Programming advice

Keep it simple, avoid chaos!

# 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**

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

Our focus: `lapply` and `Map`

Why?

- Consistent output
- Fast
- No dependencies
- We want to understand 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
```

Again, anonymous functions can be supplied as input:

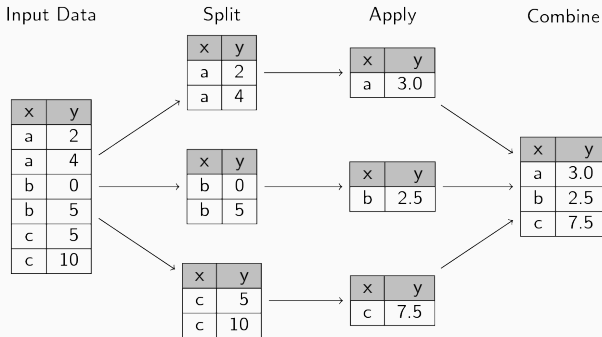
```
list1 <- list(mtcars[1:2, 1:3], iris[1:2, c(1, 2, 5)])  
list2 <- list(mtcars[3:4, 1:3], iris[3:4, c(1, 2, 5)])  
  
Map(function(x, y) {  
  rbind(x, y)  
},  
x = list1, y = list2)
```

## 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 (`?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 depend 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 specific implementation of procedure, and is associated with classes.
- **inheritance**: classes are organized in hierarchy.  
“is-a-type-of”-relation.
- **method-dispatch**: the process of finding the correct method to apply for a specific object.

## Encapsulated OOP

- methods belong to objects or classes
- method calls look like `object.method(arg1, arg2)`
- most popular
- python, R6



## Functional OOP

- methods belong to **generics** (or generic functions)
- method calls look like **generic(object, arg1, arg2)**
- looks like a normal function
- internally build on functions
- S3

- 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, new generics

- `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!

```
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
```

## S3 - class

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.

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

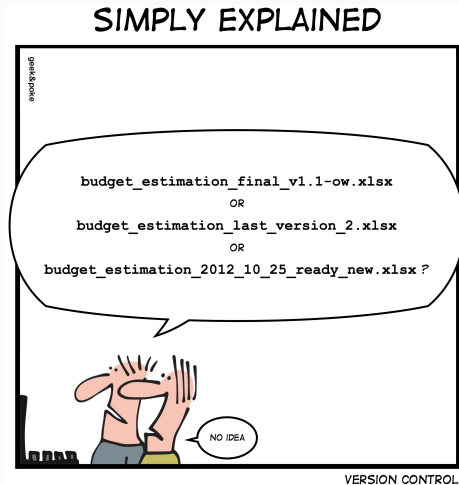




# Version Controlling (Git + Github)

---

- Motivation
- Setup
- Work flow
- Recommendations
- Resources



- 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 via RStudio
- An R-Project is added automatically to the existing repository

## Excursion: gitignore

- 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
- Discrepancies between your own different local repos → 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
- **Note:** 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 provides a great framework for documenting your code
- It provides 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 regular Github Repository
- Clone the repository regularly
- Use `usethis::use_package(getwd())` to create the minimal package structure
- Use `usethis` if you want to add more specific architecture



Your actual code lives in the **R** folder.

- Write small functions which do one specific thing
- Organize your functions logically
- 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

- Writing R Extensions (<https://cloud.r-project.org/doc/manuals/r-release/R-exts.html>)
- R Packages (<https://r-pkgs.org/>)

## Wrap Up

---

## General Advice

- Try to always learn new things about R and / or programming
- Try to automate as much as possible
- If you are re-using code regularly, write a package!
- Learn from masters
- Rewrite important code - the first attempt is usually not the best approach

# 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](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?