Programming with R/Advanced R

Dries Debeer & Benjamin Becker

18. and 19. March 2021

FDZ Spring Academy

Who are we?

Dries Debeer

Senior Researcher at itec (imec Research Group at KU Leuven)

scDIFtest, permimp, eatATA

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. Institution and Status
2. Previous knowledge and experience
   - with R
   - with other statistic software
   - with other programming languages
3. Specific interest/motivation for this workshop?
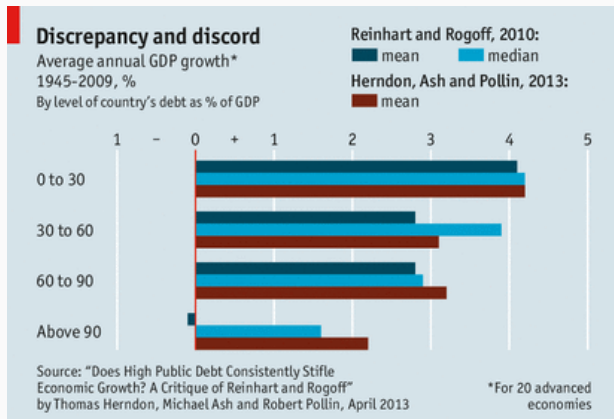
- Being more efficient in your research
    - Save time and nerves
    - Avoid errors and bugs
    - High transfer effect to all projects (with data analyses)
- Successful collaborations (with your future self?)
- Syntaxes as part of paper submissions

Two of your worst enemies

- Past Self
  - Is the biggest mess in existence
  - Is the biggest messy in existence
  - Did not document anything
  - Uses a completely different style of writing code than yourself
  - Is the worst collaborator (does not reply to e-mails)
- Future Self
  - Has the memory of a goldfish
  - Will have zero understanding for your current brilliance

Concept of Technical Debt

- We write (messy) code for data cleaning/analyses
- We decide on data sets/models/graphs/tables/...
- We try to publish it, get a major revision
- We need to rerun some analyses
- Modifying/extending our code is more difficult than it should be

Solutions

- Refactor/rewrite your could before submitting
- **Write better R code**

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

R Objects (Recap)

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

— John Chambers

- What are objects?
- Attomic vectors
- Vector structures
- Subsetting
- Replacement

- Data-structures that can be used in computations
- Collections of data of al kinds that are dynamically created and manipulated
- Can be very small, like a single number: 2.1, or very big, like a complete data set (i.e., `data.frame`) or a random forest output $\rightarrow$ *Everything in R is an object*
- Elementary data structures can be combined in more complex data structures
- Creating new types of *complex* objects is part of programming in R (S3, S4)

Basic object types	
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`

mulitple values in one object $\rightarrow$ `length()` starting from 0

Elements of the same type can be combined into an attomic vector using c().

```
c(3, 4, 5, 6) seq(3, 6) 3:6 character(6) logical(0) vector(
= "double", 4) length(3:6) length(double(0))
```

All elements are of the same type!

An important object type with special behavior is NULL. It is an empty object that can be interpreted as *nothing*. It's length is 0.

```
c(NULL, "a", NULL)
length(NULL)
c(NULL, NULL, NULL)
```

NULL is mostly used as a default argument in functions, in order to create some default behavior.

Coercion/Conversion

Automatic conversion:
NULL $\rightarrow$ logical $\rightarrow$ integer $\rightarrow$ double $\rightarrow$ character

```r
1 + TRUE
c(FALSE, "word", NULL)
```

explicit conversion: as."type"() as.vector(, mode = "type"

```r
as.character(FALSE)
as.logical(0:5)
as.vector(c(1.1, 1.5, 1.9), "integer")
as.null(c(1.1, 1.5, 1.9))
```

- check type using: `is."type"()`

```
is.logical(FALSE)
is.character(c(FALSE, "a string")[1])
is.null(NULL)
```

- check type using: `typeof()`

```
typeof(Inf)
typeof(TRUE + FALSE)
```

In order to compute with objects efficiently, names can be assigned to the objects using the assignment operator <- (or =)

```
my_object <- TRUE
my_object
```

The objects (with references) that are available to a user can be seen in the global environment using ls(). R overrides previous assignments without a message. Removed objects (rm(objectName)) cannot be restored.
→ *May the source code be with you!*

Attributes can be attached to objects together with a name for that attribute. An attribute is itself also an object. Attributes are easily lost in computations. (One of the reasons to use OOP with classes and methods, see later.)

```r
my_object <- structure(5,
                       my_attribute = "string",
                       other_attribute = FALSE)
attributes(my_object)
attr(my_object, "new") <- c(14, 25)
str(my_object)
str(as.integer(my_object))
```

There are several attributes with a specific use: "names", "dim ""class", "levels"

- "names" is a character vector that contains the names of elements of the vector/object. Names can be printed and set using `names(object) <- `.

- "dim" is an integer vector that specifies how we should interpret the vector (i.e., as a matrix, as an array). The dimensions of a vector can be printed and set using `dim(object) <- ` . $\rightarrow$ a `matrix` or `matrix` is a vector with a "dim" attribute.

There are several attributes with a specific use: "names", "dim
""class", "levels"

- "class" is a character vector that contains class names.
  Classes can be printed and set using `class(object) <-` .
  See **Object Oriented Programming (S3)**

- "levels" is a character vector that contains the names levels
  of a factor. Levels can be printed and set using
  `levels(factor) <-` .

```r
a <- 1:8
str(a)
attr(a, "dim") <- c(2, 4)
str(a)
is.matrix(a)
as.character(c(aa = 1, bb = 2))
is.factor(structure(1:2,
                     levels = c("a", "b"),
                     class = "factor"))
as.character(c(a = 1, b = 2))
```

A factor in R is actually an integer vector with a "class" attribute
set to "factor", and a "levels" attribute set to the level-labels
that correspond to the integer values from 1 to the highest integer
value in the integer vectors.

```
int <- as.integer(c(1, 2, 1, 1, 3, 1, 5, 2))
attr(int, "levels") <- c("Now way!", "Not sure",
                         "Maybe", "OK", "Definitively!")
attr(int, "class") <- "factor"
int
```

More basic object types	
complex	`1 + 2.31i, ... NA`
raw	`as.raw(2), charToRaw("a")`
expression	`expression(1+1, sum(a, b))`
language	a function call, `quote(1 + y)`
closure	`function(x) x - 1, mean`
builtin	`sum, c`
special	`for, return`
environment	an environment
symbol	`quote(x)`
...	...

More basic object types	
list	`list()`, `as.list()`, ...
matrix	an **vector** with "dim" argument: two dimensions
	`matrix() as.matrix()`
	matrix algebra
array	a **vector** with with "dim" argument
data.frame	a **list** with vectors of equal length
	`data.frame`, `as.dataframe`

A list is a "vector" that can contain any type of elements

- the types of elements can differ ↔ attomic vectors
- possible elements including lists → recursive
- can have attributes, even "dim" (though mostly not useful)

```r
my_list <- list(1.23,
                "this",
                a = list(a = c(1:2)),
                b = TRUE)
attr(my_list, "dim") <- c(2, 2)
my_list  # printing fails
```

A matrix or an array is a vector with a `"dim"`-attribute

- mostly usefull for numeric vectors (integer and double)
- matrix algebra! `t(matrix)`, `%*%`, `aperm(array)`)...
- matrix has two dimensions, array has n dimensions

```r
my_matrix <- matrix(2.5, nrow = 3, ncol = 3)
is.array(my_matrix)
dim(my_matrix)
typeof(my_matrix)
as.double(my_matrix)
my_array <- array(1:8, dim = c(2, 2, 2))
is.array(my_array)
dim(my_array)
typeof(my_array)
length(my_array)
```

A data.frame is a list of (named) vectors of equal length.

- has dimensions (but not a "dim"-attribute)
- the columns are the vectors
- the vectors can be lists (using I()).
- a data.frame has row names (but ignore these)

Data.frame

```
my_data  <- data.frame(1:4,
                       var1 = c("a", "b", "c", "d"))
attributes(my_data)
dim(my_data)
names(my_data)
colnames(my_data)
rownames(my_data)
typeof(my_data)
data.frame(1:4,
           var1 = I(list(c("a", "b"),
                         list(FALSE, "FALSE"),
                         1:5, c("c", "d"))))
```

A subset of elements from a vector can be accessed using
`object[selection]`, where `selection` is:

- a **logical** vector with the same length of the original vector
  (`TRUE`: select; `FALSE`: don't select)
- an **integer** vector indicating the indeces of the elements to
  select (or exclude)
- a **character** vector with the names of the elements to select

Using a **logical** vector:

- the logical vector should have the same length of the object. If shorter, the logical is repeated; if longer, NAs are added if TRUE. → always use the same length!
- handy when you want to select based on a condition related to the object values

Using a **logical** vector:

```r
my_object <- c(a = 1, b = 5, c = 3, d = 8)
my_object[c(TRUE, FALSE, FALSE, TRUE)]
my_object[TRUE]
my_object[c(FALSE, TRUE)]
my_object[c(FALSE, TRUE, TRUE, FALSE, TRUE)]
my_object[my_object > 4]
my_object[1:4 < 4]
```

Subsetting - attomic vectors

Using an **integer** vector:

- the integer vector can have any length (repeated indices are repeatedly selected)
- positive values mean *select*, negative values mean *drop*
- positive and negative values cannot be combined
- for integers higher than the number of elements in the vector, NAs are added
- using which() a logical vector is transformed in an integer vector with the indices of the elements that were TRUE
- double elements are truncated towards zero (using as.integer()))

Using an **integer** vector:

```r
my_object <- c(a = 1, b = 5, c = 3, d = 8)
my_object[c(1, 5)]
my_object[c(1, 2, 1, 2, 1, 2, 1 ,2, 1, 2, 1, 2)]
my_object[-c(2, 4, 2)]
my_object[my_object > 4]
my_object[which(my_object > 4)]
which(my_object > 4)
my_object[1.999]
```

Using a **character** vector:

- the strings that match with the names of the elements in the vector are returned
- the character vector can have any length (repeated names are repeatedly selected)
- only selection is possible (dropping is not)
- strings that are not matched with names return `NA`

Using a **character** vector:

```
my_object <- c(a = 1, b = 5, c = 3, d = 8)
my_object[c("a", "b")]
my_object[c("a", "b", "a", "b", "a", "b", "a", "b", "a", "b"
my_object[-c("a", "b")] # returns error
```

A **sinlge** element from a vector can be accessed using
`object[[selection]]`, where `selection` is:

- an **integer** value indicating the index of the element to select
- a **character** vector with the names of the elements to select

```r
my_object <- c(a = 1, b = 5, c = 3, c2 = 8)
my_object[[2]]
my_object[["c"]]
my_object[[TRUE]]   # works due to automatic conversion
my_object[[2.2]]    # works due to automatic conversion
```

Because arrays and martrices are attomic vectors (with a "dim"
argument), the rules for attomic vectors apply.

```r
my_matrix <- matrix(c(11, 12, 21, 22), ncol = 2, dimnames =
names(my_matrix) <- letters[1:4]
my_matrix[2]
my_matrix[["b"]]
my_matrix[my_matrix < 20]
```

In addition, selection is possible per dimension:

- multiple selection vectors separated by a comma
- selection vectors can be character (match row or column names), integer (row and column number) or logical
- the first vector selects the rows, the second the columns (and so on)
- automatically the dimensions are dropped. Use `drop = FALSE` to avoid this

```r
my_matrix <- matrix(c(11, 12, 21, 22), ncol = 2, dimnames =
names(my_matrix) <- letters[1:4]
my_matrix[2,]
my_matrix[,2]
my_matrix[-1,"col2"]
my_matrix["row1",c(TRUE, FALSE)]
```

Finally, the selection ellement can also be a matrix (with one column per dimension). Each row in the matrix selects one value.

```r
my_matrix <- matrix(c(11, 12, 21, 22), ncol = 2, dimnames =
names(my_matrix) <- letters[1:4]
my_matrix[matrix(c(c(1, 2), c(1, 2)), ncol = 2)]
selection_matrix <- rbind(c(1, 1), c(1, 2), c(2, 1))
my_matrix[selection_matrix]
```

For list, the rules are similar as for atomic vectors.

- `list[selection]` gives a list (i.e., a subset of the original list)
- `list[[selection]]` gives the element (which can be a list)
- `list[["element_name"]]` is the same as `list$element_name`

```r
my_list<- list(a = 1, b = 5, c = 3, d = 8)
my_list[c(2, 2)]
my_list[["b"]]
my_list$b
my_list[c(TRUE, TRUE, FALSE)]
is.list(my_list["a"])
is.list(my_list[["a"]])
```

Because data.frames are lists, the rules for lists apply.

```r
my_dat <- data.frame(col1 = c(11, 21),
                     col2 = c(12, 22))
my_dat[1]
my_dat["col1"]
my_dat$col1
my_dat[c(TRUE, FALSE)]
is.data.frame(my_dat["col1"])
is.data.frame(my_dat[["col1"]])
```

In addition, the selection rules for matrices can be used:

- selection per row and column
- selection via a matrix with two columns

```r
my_dat <- data.frame(col1 = c(11, 21),
                     col2 = c(12, 22))
my_dat[,"col1"]
my_dat[,"col1", drop = FALSE]
my_dat[c(TRUE, FALSE), "col2"]
my_dat[c(TRUE, FALSE), "col2", drop = FALSE]
my_dat[matrix(c(c(1, 2), c(1, 2)), ncol = 2)]
```

Element Replacement

A subset of elements from a vector or vector structure can be replaced using `object[selection] <- new_values`:

- the modifications are done in place
- the structure and class of the object stay unchanged
- the length of the new values should correspond with the length of the selection (the number of elements to replace should be a multiple of the number of new values)
- only for lists, the replacement can be `NULL` (which removes the element from the list)

```r
my_dat <- data.frame(col1 = c(11, 21),
                     col2 = c(12, 22))
my_dat[1] <- 33
my_dat["col2"] <- NULL
my_dat[,1] <- NULL
```

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

— John Chambers

- Computing in R happens through function calls. A function is applied to one or more objects, and returns an object after the computation.
- The typical use is:
  `function_name(object1, argument_name = object2)`
- Computations that seem not to be done using functions are actually also functions. Check `` `<-` ``(a, 5) or `` `>` ``(5, 2)
- most functions that seem not to return an object, return it invisibly. Check `` `<-` ``(a, 5).

Clean Code

- Code Style
- R Peculiarities
- Working with RStudio

"Write code for humans, not for machines!"

Invest time in writing readable R-code.

- It will make collaboration easier
- It will make debugging easier
- It will help make your analysis reproducible

There is a complete *tidyverse* style-guide
https://style.tidyverse.org/.

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

White spaces

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

# versus

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

```r
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

```r
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)
```

- use meaningful names: "self-explainable"
- benefit from autocompletion (`<tab>`) => embrace longer names
- always write the formal arguments in function calls (except the first)
- 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

Benefit from auto completion using `tab`

```r
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)
```

```
V <- myFun(m1_B)

# versus

RMSE_age_gender <- get_RMSE(lm_age_gender)
```

Use verbs for functions and nouns for objects.

```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"
data[data %in% c(99, 999)] <- NA  # missings coded 99 or 999
```

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

# grow object
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]))
```

```
# 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])
  }
})
```

Replace preferably by column, not by row (in data.frames)

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

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

Keep your code slim

Try to limit your *package-dependencies*. Only load (i.e., `library()`) the packages that you absolutely need. I you are only using `dplyr`, it does not make sense to load the complete `tidyverse`. **Controversial:** when you are only using a function from a package once or twice, DON'T load the package, but directly access the function using the `::` operator. Less loaded packages mean less changes or name conflicts.

Forget about `attacht()`! Don't use it, unless you completely
understand what happens (see `?attach`). With 'data.frames', use
'with(data.frame, expression)' instead.

```r
# 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. For instance, when you do data manipulations like a
complex restructuring of the data, or a complex recoding of
multiple variable, write some code that allows you the check
whether the obtained results are what you want them to be.

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

Every project should get an RStudio Project! Issues when

- folders names are changed
- folders are moved
- a shared drive is used
- you ZIP and send folder

Don't save workspace to `.RData`.

- Tools < Global Options < Workspace < Save workspace ....
- Save the code instead!
- `saveRDS` and `readRDS` for objects that require long computations

Don't use `rm(list = ls())` at the start of an Rscript.

- Start clean, everytime.
- Keep it clean. No outside code, no outside computing.
- Regularly completely clean the workspace/restart.

```
.rs.restartR()
```

Keep it clean

- one folder per project!
- work on different projects in different RStudio instances!
- each with own R console/working directory/…

Organize your project folder

- R-folder with R scripts
- Data-folder with data
- split long scripts in meaningful chunks
- ...
- use relative paths (alternative: here-package)

```r
# read data
this_data <- read.csv("Data\\the-correct-file.csv")

# source Rscript
source("R\\01_first-script-to-source.R")
```

Use keyboard shortcuts

- Can make working in RStudio more efficient
- Completely tunable: Tools < Modify Keyboard Shortcuts...
- Useful shortcuts (defaults):
  - jump to editor: `ctrl + 1`
  - jump to console: `ctrl + 2`
  - jump to ...: `ctrl + 3-9`
  - jump to next tab: `ctrl + tab`
  - jump to previous tab: `ctrl + shift + tab`

Use keyboard shortcuts More useful shortcuts (defaults):

- run selection/selected line: ctrl + enter
- save current file: ctrl + s
- close current file: ctrl + w
- restart R: ctrl + shift + F10
- Show help (for function at cursor) F1
- Show source code (for function at cursor) F2

More on this

Flow & Iteration

R has specific tools (functions) that help organizing the flow of computations. You can either make computations conditional on other objects ("conditional computation") or repeat a similar computation multiple times typically with changing options ("iteration"). The most commonly used tools are:

- `if` for conditional computation
- `for` for iteration

conditional computation "`if`"

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

Loops "`for`"

- `repeat`
- `while`
- `for`

Functionals "`lapply`"

Conditional Computation - if

if statements have the basic form

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

test should be either TRUE or FALSE (or code that results 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.

else can be added, but it is optional

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

Typical uses

```
if(any(is.na(x))){
  stop("computation impossible due to NA values")
}
if(is.null(default_argument)){
  <default computations>
} else if (default_argument == specific value) {
  ...
}
```

The *test* should have length 0. A vectorized version is ifelse()
note that `if`(test, ifTRUE, ifFALSE) does exactly the same
as if (test) ifTRUE else ifFALSE

```
age <- 8:88
if (age >= 18) {
  "can vote"
} else {
    "too young"
}
'if'(age >= 18, "can vote", "too young")
ifelse(age >= 18, "can vote", "too young")
}
```

Vectorization can bring you a long way. But it may be less readable

```r
age <- 8:88
c("too young", "can vote")[1 + (age >= 18)]
```

Conditional Computation - switch

switch() is often a more elegant solution than using else if ()
multiple times.

```
method <- "method 5"
switch(method,
        "method 1" = <computations>,
        "method 2" = <computations>,
        "method 3" = <computations>,
        "method 4" = <computations>,
        "method 5" = <computations>,
        "method 6" = <computations>,
        "method 7" = <computations>,
        "method 8" = <computations>,
        stop("Not an existing method"))
```

Functions I

Functions are the building blocks of R code. As frequent users of functions we know that they 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.

Reasons:

- Readability
  - Shorter
  - Easier understanding
  - Removes distractions, like references in a paper
- Transferability
  - Other use cases
  - Other projects
  - Other persons

```r
mean(mtcars$mpg)
```

[1] 20.09062

```r
# vs.
sum(mtcars$mpg)/dim(mtcars)[1]
```

[1] 20.09062

```
summary(mtcars$mpg)

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

```r
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
```

Some useful terms to know:

- Anonymouse functions
- Primitive functions
- Exported functions
- Not exported functions

- Name
- Arguments/Formals (input)
- Body (what happens inside)
- Output

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

Usually:

- One or two data arguments
- Additional Options

Programming advice: The less arguments, the better!

What happens if the user omits an argument?

```
add_things_def <- function(x) {
  x + 10
}
add_things_def()

## Error in add_things_def():  argument "x" is
missing, with no default
```
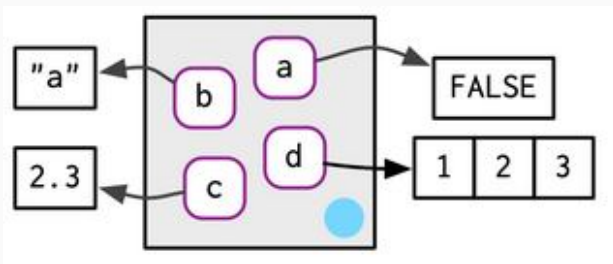
What happens if the user omits an argument?

```
add_things_def <- function(x = 1) {
  x + 10
}
add_things_def()
```

[1] 11

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.

```r
simple_fun <- function(){
  a <- 1
  b <- "a"
  environment()
}
a <- simple_fun()
rlang::env_print(a)

## <environment: 0000000013ED9DC0>
## 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.

```
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

Conditional evaluation of code

- Requires a logical of length 1
- Almost never useful outside of functions
- if() ... else ... can almost always be substituted by if() ... return()

Also: stopifnot()

Use cases

- Different behavior within loops
- Input validation
- Different function behavior dependent on option arguments

```r
mean2 <- function(x, na.rm = FALSE) {
  if (na.rm){
    x <- x[!is.na(x)]
  }
  sum(x)/length(x)
}
```

Before creating the function

- What should my function do?
- Input (Arguments)
- Output

After creating the function

- Test it
- Add input validation
- Document it

Functions II

What makes a good function?

Pure functions!

- no side effects
- the only output is returned
- no dependency on global environment
- only input via arguments

Results in easier understanding and higher portability.

Sometimes missings arguments are irrelevant

```r
add_things3 <- function(x, y) {
  x + 10
}
add_things3(2)
```

[1] 12

How can functions receive flexible numbers of inputs?

Examples:

- sum()
- save()
- ...

via dot dot dot (...)

```
add_all_things2 <- function(...) {
  l <- list(...)
  do.call(sum, l)
}
add_all_things2(2, 3, 5, 10)
```

[1] 20

Performing an action when the function terminates

```r
add_things <- function(x, y) {
  on.exit(cat("Sum of", x, "and", y))
  x <- x + 20
  x+y
}
out <- add_things(1, 2)
```

Sum of 21 and 2

```r
out
```

[1] 23

Accessing the function call

```
showArgs <- function(x, y) {
  match.call()
}
showArgs(1, 2)
```

showArgs(x = 1, y = 2)

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

Object Oriented Programming (S3)

Version Controlling (Git + Github)

- 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)

- Install git
- (optional) Install/Setup GUI for git (RStudio, Gitkraken, …)
- Setup account for Github/Bitbucket/Gitlab/…
- Connect everything
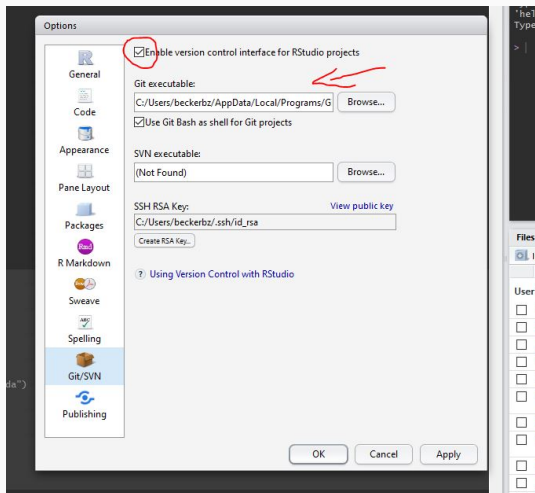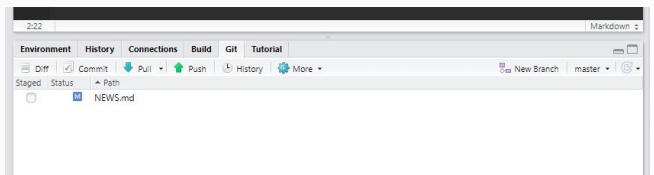
Creating a repository

- Create an online repository (e.g. on Github)
  - Use an R specific .gitignore
  - Initialize with a short readme
- Clone the repository to your local machine
- (optional) Place an R project in the existing repository

Working with a repository

- Before working: Synch your local repo (**Pull**)
- Perform changes in your local repository
- **Stage** your changes
- **Commit** your changes (aka new version)
- **Push** your changes

- Keep it simple!
  - 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)

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?