Programming with R/Advanced R

Dries Debeer & Benjamin Becker

18. and 19. March 2021

FDZ Spring Academy

Introduction

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

Introduction

Who are you?

1. Specific interests/motivation for this workshop?
2. Previous knowledge and experience?
    - with R
    - with other statistic software
    - with other programming languages

Motivation

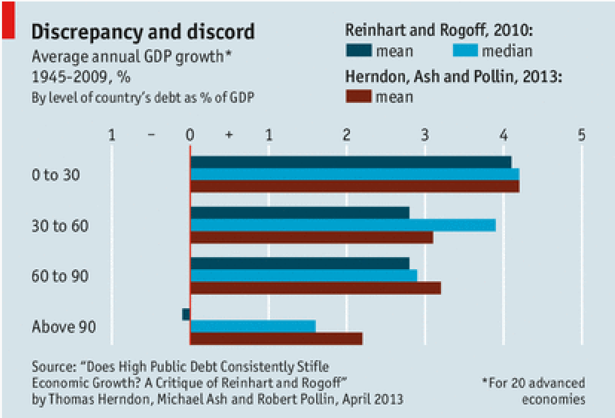
- Being more efficient in your research
  - Save time and nerves
  - Avoid errors and bugs
  - High transfer effect to all projects (which use data)
- Successful collaborations (with your future self?)
- Syntaxes as part of paper submissions

Motivation

Two of your worst enemies

- Past Self
  - Is the biggest messy in existence
  - Did not document anything
  - Uses a completely different style of writing code than yourself
- 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

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'

Motivation

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**

Goals of this workshop

- 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

R Objects (Recap)

- What are objects?
- Attomic vectors
- Vector structures
- subsetting
- replacement

What are objects?

- 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.
- 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).

Attomic Vectors

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 → `length()` starting from 0

Attomic Vectors

An important 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 fuction, in order to create some default behavior.

Coercion/Conversion

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

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

- explicit conversion: `as."type"()`

```
as.character(FALSE)
as.logical(0:5)
```

Attomic vectors - check type

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

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

- check type using: `typeof()`

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

Assignment

In order to compute with objects efficiently, names can be assigned to the objects Using <- (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. $\rightarrow$ *May the sourcecode be with you!*.

Attributes

Attributes can be attached to objects together with an name for that attribute. An attribute is itself an object.

Attributes are easily lost in computations. (One of the reasons to use OOP with classes and methods.)

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

Attributes

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) <- .` → a `matrix` or `matrix` is a vector with a `"dim"` attribute.
- `"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

Attributes

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", "
attr(int, "class") <- "factor"
int
```

More Basic Object Types

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)`
...	...

Vector Structures

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

List

- "vectors" that can contain any type of element
- including lists → recursive
- can have attributes, even `"dim"` (though mostly not useful)

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

Data.frame

- a list of (named) vectors of equal length
- including lists (using `I()`).
- has row names (but ignore these)
- has dimensions (but not a `"dim"`-attribute)

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

Matrix & Array

- 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

```
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)
```

Subsetting - attomic vectors

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

Subsetting - attomic vectors

Using a logical vector:

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

Subsetting - attomic vectors

```
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, `NA`s 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()`)

Subsetting - attomic vectors

```
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]
```

Subsetting - attomic vectors

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`

Subsetting - attomic vectors

```
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", "
my_object[-c("a", "b")]

## Error in -c("a", "b"):  invalid argument to unary
operator
```

Subsetting - attomic vectors

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

```
my_object <- c(a = 1, b = 5, c = 3, c2 = 8)
my_object[[2]]
my_object[["c"]]
```

Subsetting - Matrix & Arrays

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

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

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

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

Subsetting - Lists

- For list, the rules are similar as for atomic vectors.
- `list[selection]` gives a list
- `list[[selection]]` gives the element (which can be a list)
- `list[["element_name"]]` is the same as `list$element_name`

```
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"]])
```

Subsetting - data.frames

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

In addition, the selection rules for matrices can be used: (i.e., with
).

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

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

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)

```
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

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.
- 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

Clean Code

- Code Style
- Efficient R
- Efficient RStudio

"Write code for humans, not for machines!"

Code Style

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/`.

Go easy on your eyes

- 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

```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

```r
final_results <- data.frame(first_variable = sqrt(results$m

# versus

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

Go easy on your mind

- 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

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)
```

Use meaningful names

```r
V <- myFun(m1_B)

# versus

RMSE_age_gender <- get_RMSE(lm_age_gender)
```

Use `verbs` for functions and `nouns` for objects.

Comment, comment, comment

```
## Start every Rscript with a comment that explains what th
##  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"
```

Don’t grow, replace

```
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$x[ro
                      log(data$z[row_nr]) + 5 * sqrt(data
  }
})
```

Vectorize

```r
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.

Never Attach

Forget about `attacht()`! Don't use it, unless you completely understand what happens (`?attach`).

With ‘data.frames‘, 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. 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`, ...

Iteration

Functions I

Building Blocks

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

Readability

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

[1] 20.09062

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

[1] 20.09062

Readability

```
summary(mtcars$mpg)

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

Readability

```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
```

Types of functions

Some useful terms to know:

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

Elements of a function

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

Function definition

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

Arguments

Usually:

- One or two data arguments
- Additional Options

Programming advice: The less arguments, the better!

Default arguments

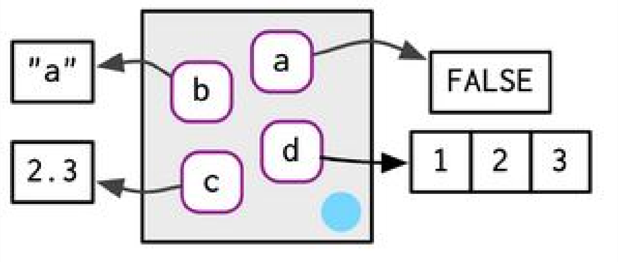
What happens if the user omits an argument?

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

```
[1] 11
```

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

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

¡environment: 0000000014E820D8¿ parent: ¡environment: global¿ bindings: * b: ¡chr¿ * a: ¡dbl¿

Scoping

Where does R find things?

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

Programming advice: Keep it simple, this can create chaos.

Scoping

```r
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

If clauses

Conditional evaluation of code

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

If clauses

Use cases

- Input validation
- Different function behavior dependent on option arguments
- Requires a logical of length 1

If clauses

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

Writing Functions

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 is a good function?

- pure functions
  - no sideeffects
  - no dependency on global environment
  - easier understanding, easier transfer!

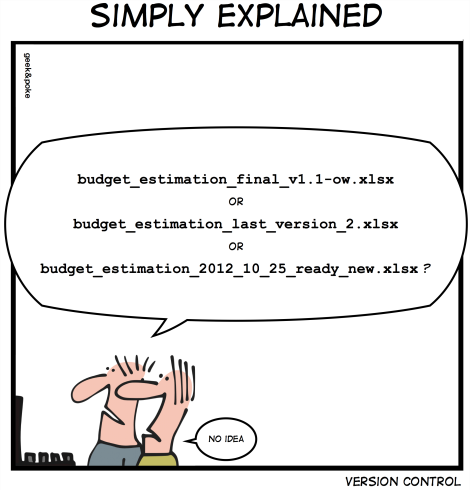
Debugging

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

Object Oriented Programming (S3)

Version Controlling (Git + Github)

Motivation


SIMPLY EXPLAINED
geek&poke
budget_estimation_final_v1.1-ow.xlsx
OR
budget_estimation_last_version_2.xlsx
OR
budget_estimation_2012_10_25_ready_new.xlsx ?
NO IDEA
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…


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Requirements

- Install git
- Install User Interface for git (RStudio, Gitkraken, ...)
- Setup account for Github/Bitbucket/Gitlab/...
- Connect everything

Workflow

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

Workflow

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

Recommendations

- Keep it simple!
  - No branches/forks/pull requests
- Have meaningful commits
- Keep it lean (no big files)

Resources

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?