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, mstDIF

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. 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?

Motivation

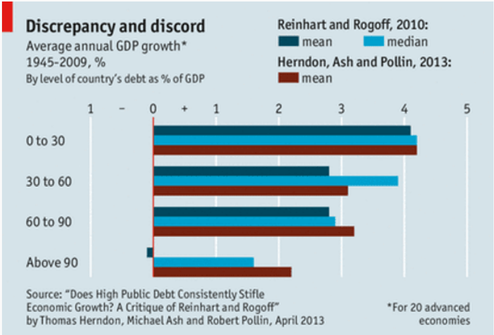
- Being more efficient in your research
  - Save time and nerves
  - Avoid errors and bugs
  - High transfer effect to all projects (with data analyses)
- Successful collaborations (with your future self?)
- Syntaxes as part of paper submissions

Motivation

Two of your worst enemies

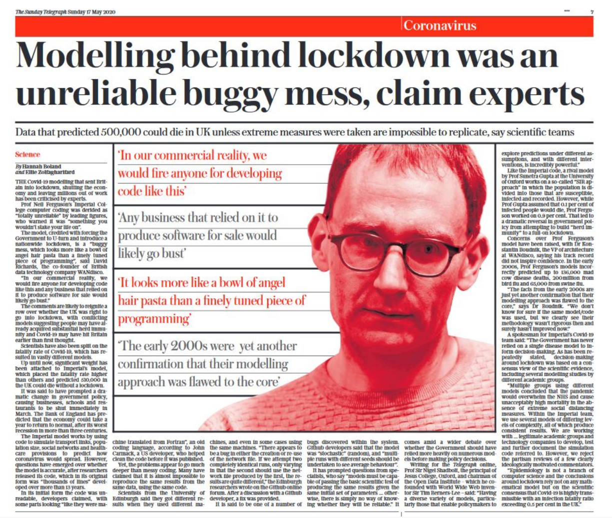
- Past Self
    - Is the biggest mess in existence
    - Did not document anything
    - Uses a completely different style of writing code than yourself
    - Is the worst collaborator (does not reply to e-mails)
- Future Self
    - Has the memory of a goldfish
    - Will have zero understanding for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

Motivation



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

Motivation



The Sunday Telegraph Sunday 17 May 2020 7

Coronavirus

Modelling behind lockdown was an unreliable buggy mess, claim experts

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

Science

By Hannah Boland *and* Ellie Zolfagharifard

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Trade-off

- Being fast vs. writing (or refactoring) perfect code

But also

- 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

Agenda

Day 1

- Recap & Clean Code
- Functions (Introduction)
- Functions (Advanced)

Day 2

- Flow & Iteration
- Object oriented programming: S3
- Version Controlling

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?
- Atomic 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, or very big. $\rightarrow$ *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)

Atomic 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 $\rightarrow$ `length()` starting from `0`

Atomic Vectors

Elements of the same type can be combined into an atomic vector using c.

```
c(3.3, 2.44, 9, 634)
```

[1] 3.30 2.44 9.00 634.00

All elements are of the same type!

Atomic Vectors

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.

```
length(NULL)

# [1] 0
```

`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

```
1 + TRUE

# [1] 2
```

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

```
as.logical(0:5)

# [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

atomic vectors - check type

Check type using: `is."type"()`

```r
is.null(NULL)

# [1] TRUE
```

Check type using: `typeof()`

```r
typeof(TRUE + FALSE)

# [1] "integer"
```

Assignment

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

```
my_object <- TRUE
my_object

# [1] TRUE
```

- 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

Attributes can be attached to objects. An attribute:

- has a name
- is itself also 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_attribute = FALSE)
attributes(my_object)

# $my_attribute
# [1] "string"
#
# $other_attribute
# [1] FALSE
```

Attributes

- `"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 `array` is a vector with a `"dim"` attribute.

Attributes

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) <- .`

Attributes

A factor in R is actually an integer vector with

- a `"class"` attribute set to `"factor"`
- a `"levels"` attribute set to the level-labels that correspond to the integer values from 1 to the highest integer value in the integer vector.

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	
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`

List

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

- the types of elements can differ $\leftrightarrow$ atomic vectors
- possible elements including lists $\rightarrow$ recursive
- can have attributes

```
my_list <- list("this",
                a = list(a = c(1:2)))
my_list

# [[1]]
# [1] "this"
#
# $a
# $a$a
# [1] 1 2
```

Matrix & Array

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
- `cbind(vector1, vector2)`
- `rbind(vector1, vector2)`
- `matrix(vector, ncol = 4, nrow = 2)`
- `array(vector), dim = c())`

Data.frame

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)

Subsetting - Atomic 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 - Atomic 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 - Atomic vectors

Using a **logical** vector:

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

# b d
# 5 8
```

Subsetting - Atomic 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 - Atomic vectors

Using an **integer** vector:

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

# a b a b a b a b a b a b
# 1 5 1 5 1 5 1 5 1 5 1 5
```

Subsetting - Atomic 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 - Atomic vectors

Using a **character** vector:

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

# a b
# 1 5
```

Subsetting - Atomic 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]]

# [1] 5
```

Subsetting - Matrix & Arrays

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

Subsetting - Matrix & Arrays

In addition, selection is possible per dimension:

- separated by a comma `[, ]`
- selection via character (match row or column names), integer (row and column number) or logical vectors
- the first vector selects the rows, the second the columns (and so on)
- dimensions are dropped, unless `drop = FALSE`

```
my_matrix <- matrix(c(11, 12, 21, 22), ncol = 2,
                    dimnames = list(paste0("row", 1:2),
                                    paste0("col", 1:2)))
my_matrix[,2]

# row1 row2
#   21   22
```

Subsetting - Matrix & Arrays

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 = list(paste0("row", 1:2),
                                    paste0("col", 1:2)))
selection_matrix <- rbind(c(1, 1), c(1, 2), c(2, 1))
my_matrix[selection_matrix]

# [1] 11 21 12
```

Subsetting - Lists

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`

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

# [1] TRUE

is.list(my_list[["a"]])

# [1] FALSE
```

Subsetting - data.frames

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

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

#   col1
# 1   11
# 2   21
```

Subsetting - data.frames

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

- selection per row and column (note the `drop` argument)
- selection via a matrix with two columns

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

#   col1
# 1   11
# 2   21
```

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)

Element Replacement

```
my_dat <- data.frame(col1 = c(11, 21),
                     col2 = c(12, 22))
my_dat[1, 2] <- 33
```

“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
- R Peculiarities
- Working with 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$mean_squared_error), second_variable =
paste0(results$condition, results$class, sep = ":"),
third_variable = results$bias)

# versus

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

Go easy on your mind

- use meaningful names: “self-explainable”
- always write the formal arguments in function calls (except the first)
- benefit from autocompletion (`<tab>`) => embrace longer names
- use `TRUE` and `FALSE` not `T` and `F`
- comment, comment, comment
  - not what (should be clear from the code)
  - but why
  - explain the reasoning, not the code

Use meaningful names

```r
V <- myFun(m1_B)

# versus

RMSE_age_gender <- get_RMSE(lm_age_gender)
```

Use verbs for functions and nouns for objects.

Write formal arguments

Benefit from auto completion using tab

```
m1_B <- lm(outcome ~ age*gender,
           exp1, condition_1, freq)

# versus

lm_age_gender <- lm(outcome ~ age*gender,
                    data = exp1,
                    subset = condition_1,
                    weigths = freq)
```

Comment, comment, comment

```r
## Start every Rscript with a comment that explains
##  what the code in the script does, why it does
##  this, and to which project it belongs.
##  Your future self will be very thankful!
##
## Mention which packages you are using in this Rscript.

## Use sections to separate chunks ----------------------

## Maybe even subsections ================================

## Recode variables so that missings are coded as "NA"
dat[dat %in% c(99, 999)] <- NA  # missings coded 99 or 999
```

Keep your code slim

Try to limit your *package-dependencies*.
Only load `library()` the packages that you absolutely need. If you are only using `dplyr`, it does not make sense to load the complete `tidyverse`.
Controversial: when possible, use the `::` operator(and consider not loading the package). `<package>::<function>`

- explicit dependencies
- less name conflicts

Never Attach

Forget about attacht()!

Don't use it, unless you completely understand what happens (see ?attach).

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))
```

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`, ...

Working with RStudio

"Every project should get an RStudio Project!"

Don't use `setwd("pathtomylocal_folder")`

Issues when:

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

Working with RStudio

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

Working with RStudio

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 (or restart the session).

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

Working with RStudio

Keep it clean

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

Working with RStudio

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)

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

# source Rscript
source("R\0_first-script-to-source.R")
```

Working with RStudio

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`

Working with RStudio

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 HERE.

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) {
  x + 10
}
add_things_def()

# Error in add_things_def():  argument "x" is missing,
with no default
```

Default arguments

What happens if the user omits an argument?

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

```
[1] 11
```

Lazy Evaluation

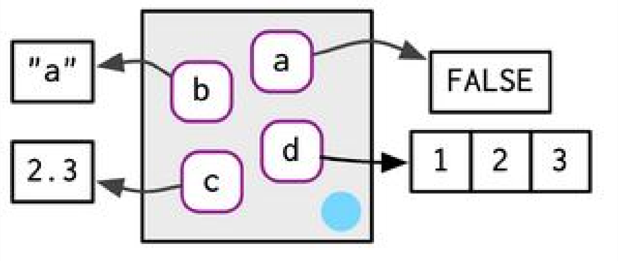
Sometimes missing arguments are irrelevant!

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

```
[1] 12
```

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: 00000000144D4C88>
# 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

```
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

Output

What does a function return?

- (Standard) The last evaluated object
- Object defined by return()
- An error via stop()
- Additional: Warnings + Messages

Output

The last evaluated object

```
add_things_standard <- function(x = 1) {
  x2 <- x*2
  out <- x + x2
  out
}
add_things_standard(2)
```

[1] 6

Output

return()

```
add_things_return <- function(x = 1) {
  x2 <- x*2
  return(x)
  out <- x + x2
  out
}
add_things_return(2)
```

[1] 2

Output

Error: stop()

```
add_things_stop <- function(x = 1) {
  x2 <- x*2
  stop("My own error message")
  out <- x + x2
  out
}
add_things_stop(2)

# Error in add_things_stop(2):  My own error message
```

Output

Warnings: warning()

```
add_things_warning <- function(x = 1) {
  x2 <- x*2
  warning("My own warning message")
  out <- x + x2
  out
}
add_things_warning(2)

# Warning in add_things_warning(2):  My own warning
message
```

[1] 6

Output

Messages: message()

```
add_things_message <- function(x = 1) {
  x2 <- x*2
  message("My own message")
  out <- x + x2
  out
}
add_things_message(2)

# My own message
```

[1] 6

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.

Conditional Computation - if

else can be added, but it is optional

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

Conditional Computation - if

Typical uses

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

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

if(is.null(default_argument)){
  <default computations>
} else if (default_argument == specific value) {
  ...
}
```

Conditional Computation - if

The *test* should have length 1.

```r
# only the first element is evaluated
age <- c(8, 17, 25, 39, 55)
if (age >= 18) {
  "can vote"
} else {
    "too young"
}

# [1] "too young"
```

Conditional Computation - if

Tipp:

- If possible, avoid using else()
- Use early return(), stop(), etc. instead

Conditional Computation - if

Abbreviation for if(!test) and stop():

```
mean2 <- function(x, na.rm = FALSE) {
  stopifnot(is.numeric(x))
  sum(x)/length(x)
}
mean2("a")

# Error in mean2("a"):  is.numeric(x) is not TRUE
```

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 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.

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

on.exit()

Performing an action when the function terminates

```
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

```
out
```

[1] 23

Accessing the function call

Accessing the function call

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

showArgs(x = 1, y = 2)

Debugging

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

Also:

- trace() & untrace()
- debug() & undebug(), debugonce()

browser()

Inspecting a function interactively

```
some_function <- function(x, y) {
  z <- x + y
  browser()
  z
}
some_function(x = 1, y = 5)
```

browser()

```
> some_function <- function(x, y) {
+   z <- x + y
+   browser()
+   z
+ }
> some_function(x = 1, y = 5)
Called from: some_function(x = 1, y = 5)
Browse[1]>
```

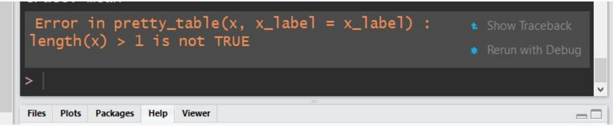
browser()

Navigating within a browser:

ls()	Show existing objects in the current environment
c	Exit the browser and continue execution
Q	Exit the browser, return to top level
where	Show call stack

traceback()

Understanding the call stack:



traceback()

Understanding the call stack:

```
Error in pretty_table(x, x_label = x_label) :          Hide Traceback
length(x) > 1 is not TRUE
                                                       Rerun with Debug
13. stop(simpleError(msg, call = if (p <- sys.parent(1L)) sys.c
    all(p)))
12. stopifnot(length(x) > 1)
11. pretty_table(x, x_label = x_label)
10. pretty_statistics(sub_dat$cyl, x_label = "cyl")
 9. FUN(data[x, , drop = FALSE], ...)
 8. FUN(X[[i]], ...)
 7. lapply(X = ans[index], FUN = FUN, ...)
 6. tapply(seq_len(32L), list(`mtcars$carb` = c(4, 4, 1, 1, 2,
    1,
    4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, 2, 2, 4, 2, 1,
     2,
    2, 4, 6, 8, 2)), function (x)
    FUN(data[x, , drop = FALSE], ...), simplify = TRUE)
 5. eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = s
    implify)),
        data)
 4. eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = s
    implify)),
        data)
 3. structure(eval(substitute(tapply(seq_len(nd), IND, FUNx, si
    mplify = simplify)),
        data), call = match.call(), class = "by")
 2. by.data.frame(mtcars, mtcars$carb, function(sub_dat) {
        pretty_statistics(sub_dat$cyl, x_label = "cyl")
    })
 1. by(mtcars, mtcars$carb, function(sub_dat) {
        pretty_statistics(sub_dat$cyl, x_label = "cyl")
    })
>
```

Recover

Being able to chosse an enrivonment from a call stack:

```
# on
options(error = recover)

# off
options(error = NULL)
```

Recover

Being able to chosse an enrivonment from a call stack:

```
Error in pretty_table(x, x_label = x_label) : length(x) > 1 is not TRUE

Enter a frame number, or 0 to exit

 1: by(mtcars, mtcars$carb, function(sub_dat) {
    pretty_statistics(sub_dat$cyl, x_label = "Cyl
 2: by.data.frame(mtcars, mtcars$carb, function(sub_dat) {
    pretty_statistics(sub_dat$cyl, x_l
 3: structure(eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = simplify)), data), call =
 4: eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = simplify)), data)
 5: eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = simplify)), data)
 6: tapply(seq_len(32), list(`mtcars$carb` = c(4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4,
 7: lapply(X = ans[index], FUN = FUN, ...)
 8: FUN(X[[i]], ...)
 9: FUN(data[x, , drop = FALSE], ...)
10: #2: pretty_statistics(sub_dat$cyl, x_label = "Cyl")
11: #3: pretty_table(x, x_label = x_label)
12: #2: stopifnot(length(x) > 1)

Selection:
```

Warnings

Turning warnings into errors

```r
# on
options(warn = 2)

# off
options(warn = 1)
```

That's it for today!

That's it for today!

Questions? Remarks?