Programming with R/Advanced R

Dries Debeer & Benjamin Becker

18. and 19. March 2021

FDZ Spring Academy

Agenda

Day 1

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

Day 2

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

Open questions from day 1?

Functions II

Functions I

- Good functions?
- dot dot dot
- on.exit()
- Accessing the call
- Debugging

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]>
```

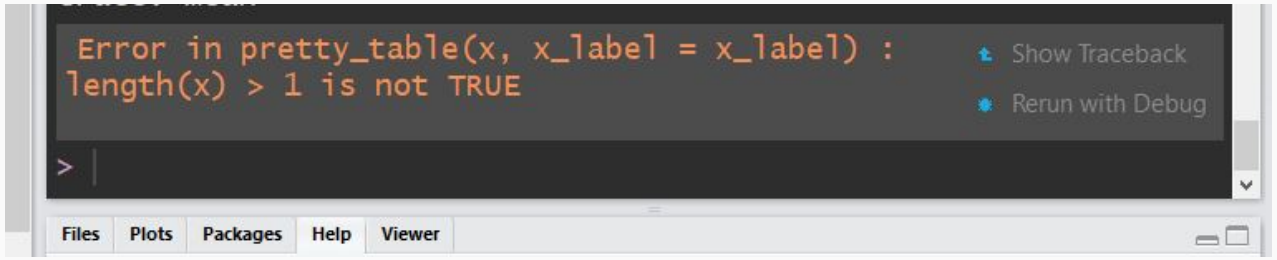
browser()

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) :
length(x) > 1 is not TRUE
                                        Hide Traceback
                                        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")
    })
>
```

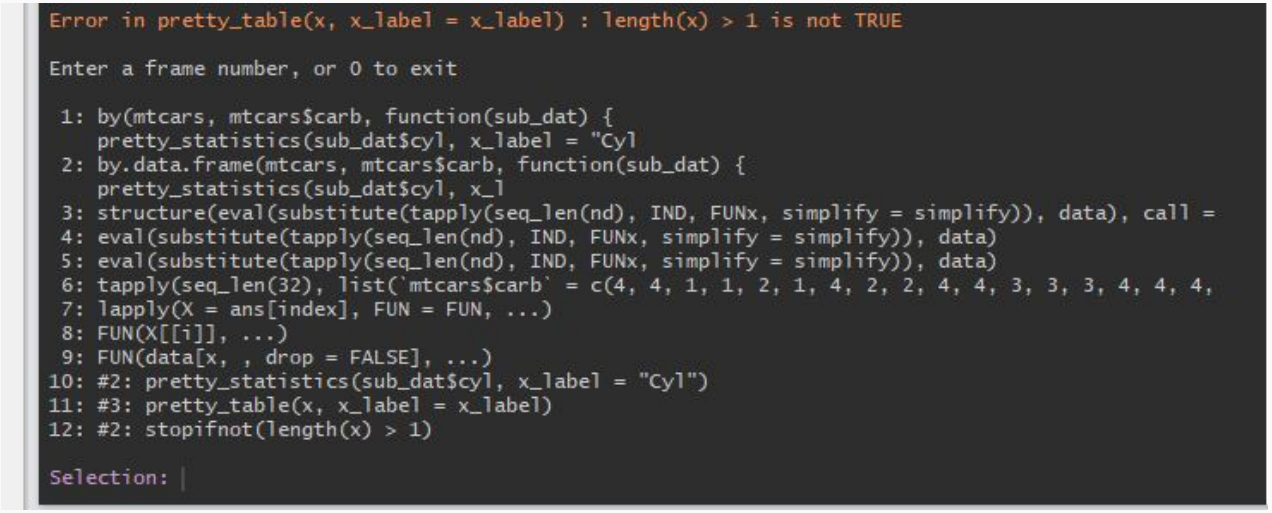
Recover

Being able to chose an environment from the call stack:

```
# on
options(error = recover)

# off
options(error = NULL)
```

Recover

Being able to chosse an enrivonment from a call stack:



Warnings

Turning warnings into errors

```
# on
options(warn = 2)

# off
options(warn = 1)
```

Exercises

Functionals

A functional is a function that takes another function as an argument.
Focus on the `apply`-family. These functions *apply* a function repeatedly.
Can be seens as an abstraction of a for loop, with the following advantages

- requires less code to write
- does not store intermediate results
- no need to replace / grow

Functionals

The most commonly used functionals are:

- `lapply` vector / list $\rightarrow$ list
- `sapply` vector / list $\rightarrow$ vector (matrix)
- `apply` matrix / array / data.frame $\rightarrow$ vector (matrix)
- `tapply`, `by`
- `mapply`, `Map`
- `rapply`, `eapply`, `vapply`

All of which have an argument that should be a function.

lapply

Data.frames are lists

```
lapply(iris, FUN = class)


# $Sepal.Length
# [1] "numeric"
#
# $Sepal.Width
# [1] "numeric"
#
# $Petal.Length
# [1] "numeric"
#
# $Petal.Width
# [1] "numeric"
#
# $Species
# [1] "factor"
```

lapply

- an annonymous function can be used
- any type of element can be used
- other arguments can be passed through

```
lapply(c(min, median, max), FUN = function(fun, x) {
  fun(x)
}, x = 2:8)


# [[1]]
# [1] 2
#
# [[2]]
# [1] 5
#
# [[3]]
# [1] 8
```

sapply

- wrapper around lapply
- if possible, the ouput is combined into a atomic vector or matrix

```
sapply(airquality, FUN = sd)

#     Ozone   Solar.R      Wind      Temp     Month       Day
#        NA        NA 3.523001 9.465270 1.416522 8.864520

sapply(airquality, FUN = quantile, prob = c(.1, .9),
       na.rm = TRUE)

#     Ozone Solar.R  Wind Temp Month Day
# 10%    11    47.5  5.82 64.2     5   4
# 90%    87   288.5 14.90 90.0     9  28
```

apply

- for objects with dimension (matrix, array, data.frame)
- apply over (a) chosen dimension(s)

```
my_matrix <- matrix(1:6, nrow = 2)
apply(my_matrix, 1, max)    # apply per row

# [1] 5 6

apply(my_matrix, 2, max)    # apply per column

# [1] 2 4 6
```

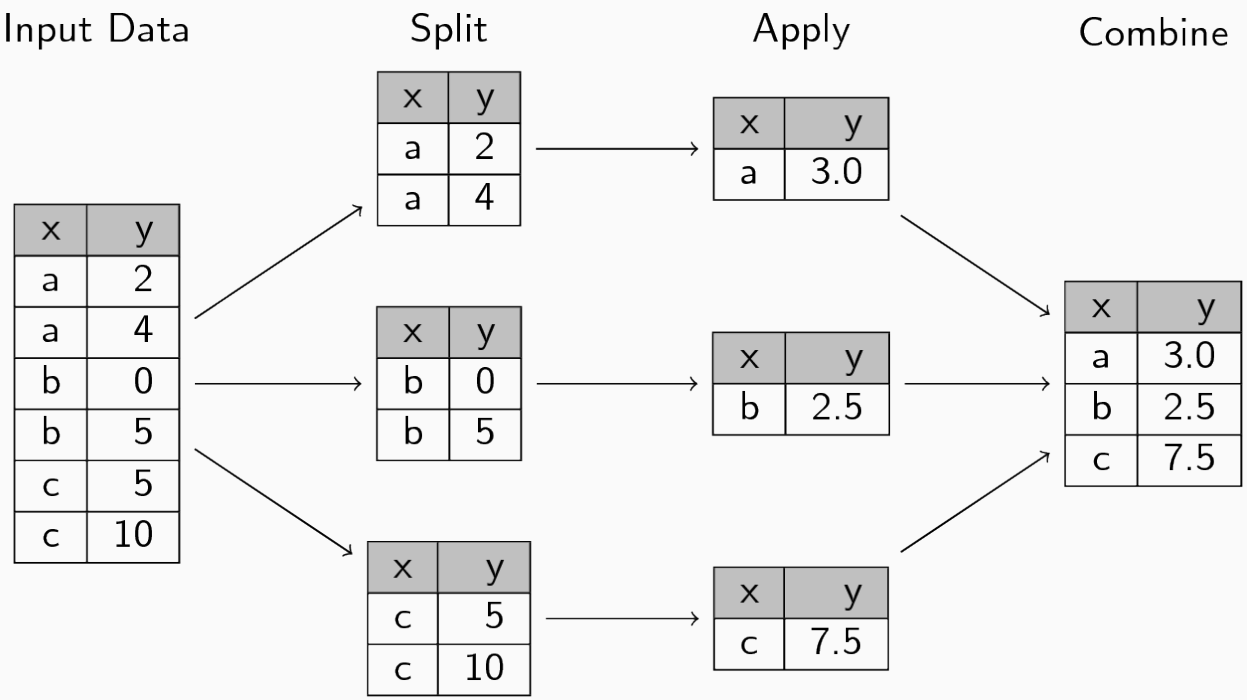
apply

```r
my_matrix <- matrix(1:6, nrow = 2)
apply(my_matrix, c(1, 2),
      function(value) value^2)    # apply per row and column

#      [,1] [,2] [,3]
# [1,]    1    9   25
# [2,]    4   16   36

apply(airquality, 2, median)

#   Ozone Solar.R    Wind    Temp   Month     Day
#      NA      NA     9.7    79.0     7.0    16.0
```

Split & Apply & Combine

- split a data.frame or vector (`?split`)
- apply computations on each split (`lapply`)
- combine the results (`sapply`, `?do.call`)

Split & Apply & Combine



Exercises

Object Oriented Programming (S3)

Object Oriented Programming

Why?

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

Object Oriented Programming

Basics

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

Object Oriented Programming in R

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

S3

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

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

S3

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

S3 - class

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

```
object <- 1:10
class(object) <- "myClass"
attributes(object)

# $class
# [1] "myClass"
```

Very flexible! Use it wisely!

S3 - class

```
class(iris)


# [1] "data.frame"


iris[1:4, 1:3]


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

S3 - class

Changing the class changes the behavior!

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

# [1] "no data.frame"

iris[1:4, 1:3]

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

S3 - class

Good practice: ***constructor function***

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

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

Write a separate validation-function for computationally intensive validation.

S3 - Generics and Methods

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

```
body(print)

# UseMethod("print")
```

S3 - Generics and Methods

Creating a new method (for an existing generic):

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

# This is a myClass-print:
# 5 3 1.581
```

S3 - Generics and Methods

Good practices (enforced for packages on CRAN):

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

S3 - Generics and Methods

Inspect the generic!

```
formalArgs(print)

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

formalArgs(summary)

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

S3 - Generics and Methods

Creating a new generic:

```
center <- function(x, ...)
  UseMethod("center")

center.myClass <- function(x, ...){
  print("centering myClass:\n")
  return(x - mean(x))
}

center(my_object)

# [1] "centering myClass:\n"
# This is a myClass-print:
# 5 0 1.581
```

S3 - Inheritance and Dispatch

Objects can have more than one class

```
my_model <- glm(as.factor(books) ~ pared, data = pisa,
                family = "binomial")
class(my_model)

# [1] "glm" "lm"

class(my_model) == "lm"

# [1] FALSE  TRUE

inherits(my_model, "lm")

# [1] TRUE
```

Good practice: hierachical inheritance (subclass and superclass)

S3 - Inheritance and Dispatch

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

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

# [1] FALSE

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

# [1] TRUE

variable.names(my_model)

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

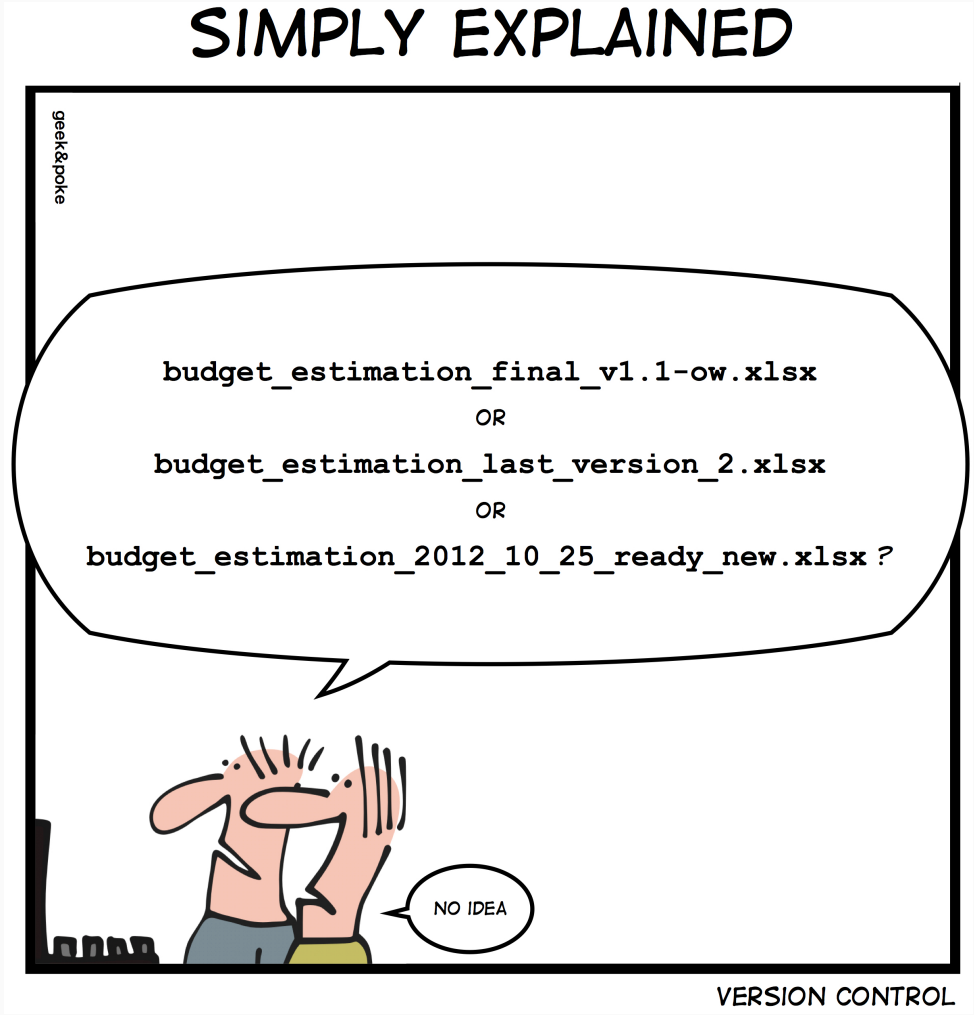
Exercises

Version Controlling (Git + Github)

Version Controlling

- Motivation
- Setup
- Work flow
- Recommendations
- Resources

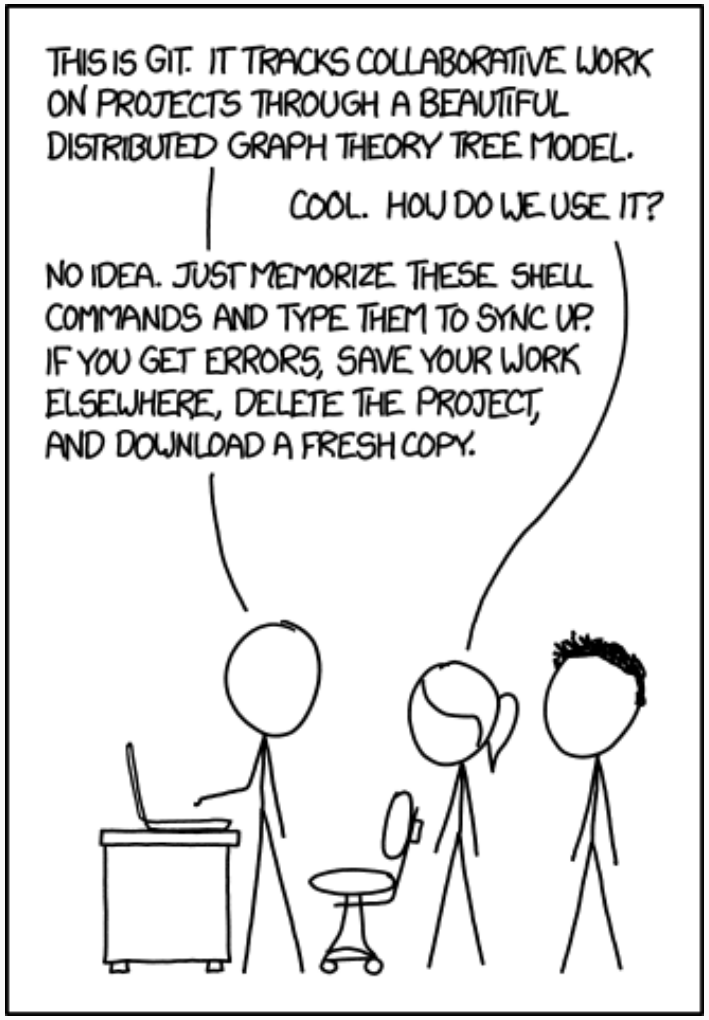
Motivation


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)

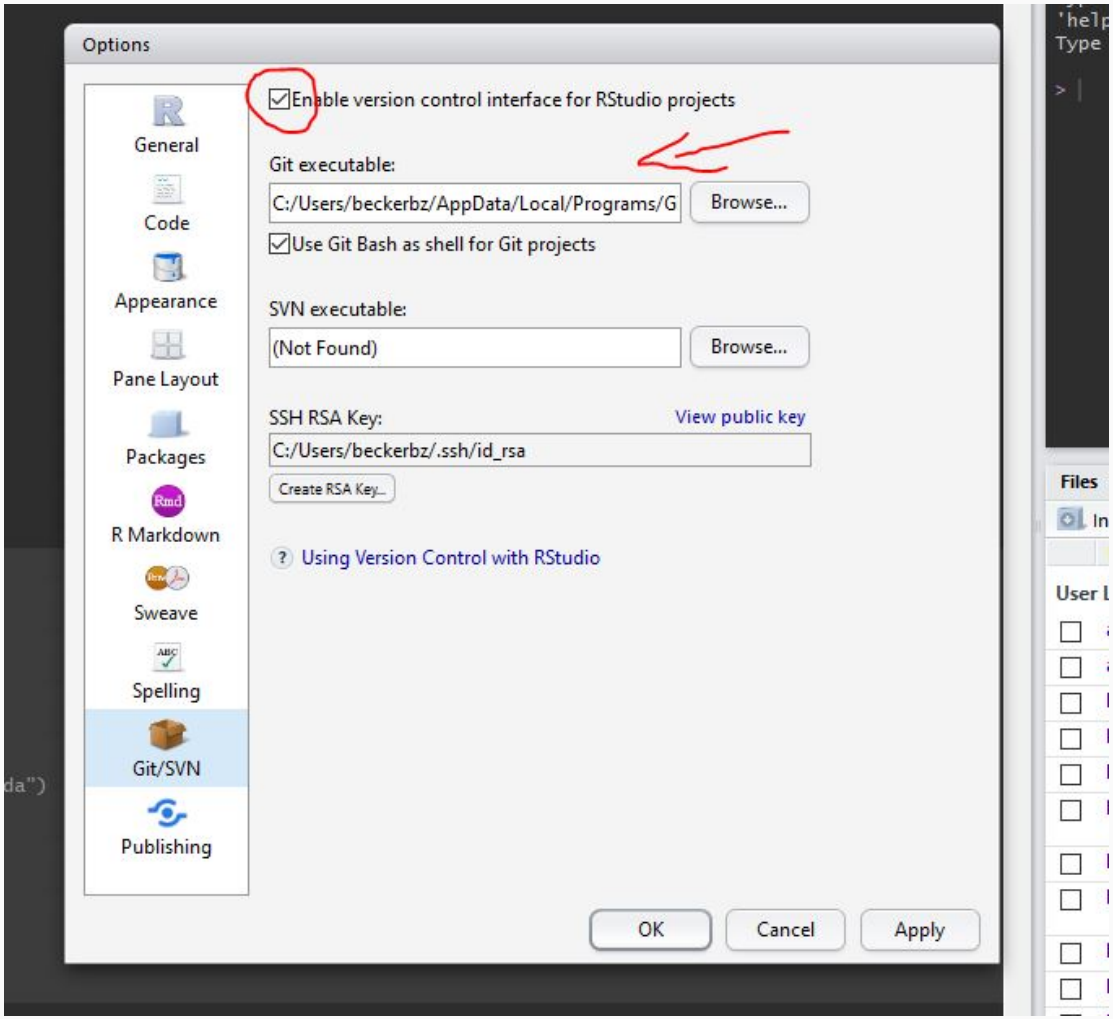
But...


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.

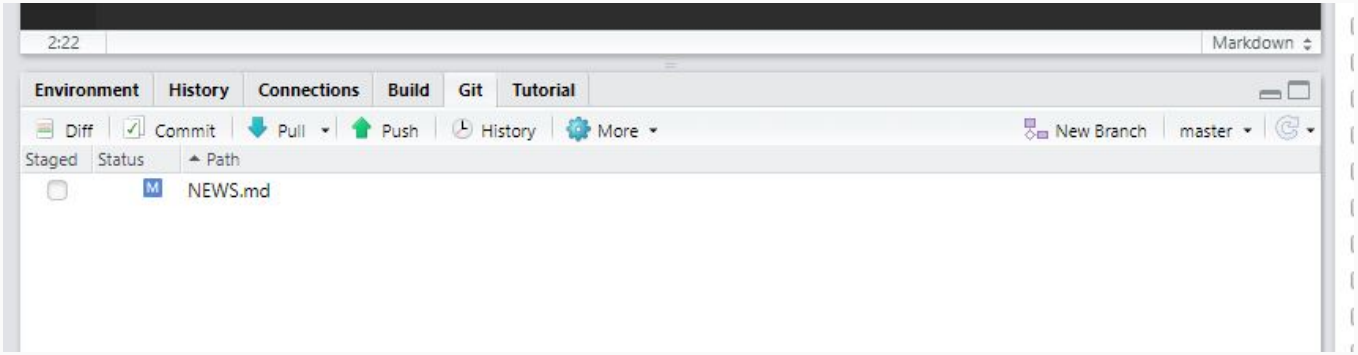
Requirements

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

RStudio


Options
General
Code
Appearance
Pane Layout
Packages
R Markdown
Sweave
Spelling
Git/SVN
Publishing
Enable version control interface for RStudio projects
Git executable:
C:/Users/beckerbz/AppData/Local/Programs/G
Browse...
Use Git Bash as shell for Git projects
SVN executable:
(Not Found)
Browse...
SSH RSA Key:
View public key
C:/Users/beckerbz/.ssh/id_rsa
Create RSA Key...
Using Version Control with RStudio
OK
Cancel
Apply
Files

RStudio


2:22
Markdown
Environment
History
Connections
Build
Git
Tutorial
Diff
Commit
Pull
Push
History
More
New Branch
master
Staged
Status
Path
M
NEWS.md

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

Work flow

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)

Wrap Up

General Advice

- Investing time in learning R pays off
- It's a steady learning curve
- Learn from masters
- Rewrite important code - the first attempt is usually not the best approach

General R Advice

- Document well
- Use a consistent style
- Write functions
- Split long functions in smaller ones
- Write wrappers
- Use Iteration (don't copy paste)
- Use matrix operations and vectorized functions instead of loops
- Use git

Literature Recommendations

R Resources

- 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?