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?

Flow & Iteration

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

Flow & Iteration

conditional computation

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

iteration

- loops (`repeat`, `while`, `for`)
- functionals (`apply` - family)
- split & apply & combine

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 0.

```
# 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 - ifelse

A vectorized version is `ifelse()`.

```
# all elements are evaluated
ifelse(age >= 18, "can vote", "too young")


# [1] "too young" "too young" "can vote"  "can vote"  "can vote"
```

Conditional Computation - Vectorization

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

```
age <- c(8, 17, 25, 39, 55)
c("too young", "can vote")[1 + (age >= 18)]


# [1] "too young" "too young" "can vote"  "can vote"  "can vote"
```

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"))
```

Iteration - for

`for` statements have the basic form

```
for (element in vector) {
  computation
}
```

For each element in the vector, the computation is executed.

Often, the computation depends on the element in that iteration.

Iteration - for

```
for (index in 1:3){
  cat(" computation -")
}

for (name in c("Alice", "Bob", "Casey")){
  if(name == "Bob") print(" This was Bob -")
  else cat(" Not Bob -")
}
```

Iteration - while

`while` statements have the basic form

```
while (condition){
  computation
}
```

As long as the condition is TRUE, the computation is executed. Often, the computation depends on something that is related to the condition.

Iteration - repeat

`repeat` statements have the basic form

```
repeat {
  computation
}
```

Without a `break` the computation is repeated infinite times

Iteration - next break

- `next` starts next iteration
- `break` ends iteration (of the innermost loop)

```
index <- 0
repeat {
  index <- index + 1
  if (index %in% c(3, 5)) next
  if (index > 6) break
  print(index)
}
```

Iteration - Good practice

- Use `seq`, `seq_len`, or `seq_along`

```
x <- numeric()
for (index in 1:length(x)){
  print(index)
}

# [1] 1
# [1] 0

for (index in seq_along(x)){
  print(index)
}
```

Iteration - Good practice

- Don't grow, but replace

```
x <- letters
result1 <- numeric()           # grow
result2 <- numeric(length(x)) # replace
for (index in seq_along(x)){
  result1 <- c(result1, paste(index, x[index]))  # grow
  result2[index] <- paste(index, x[index])       # replace
}
```

Functionals

A functional is a function that takes another funciton 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`, `mapply`, `vapply`
- `rapply`, `eapply`

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

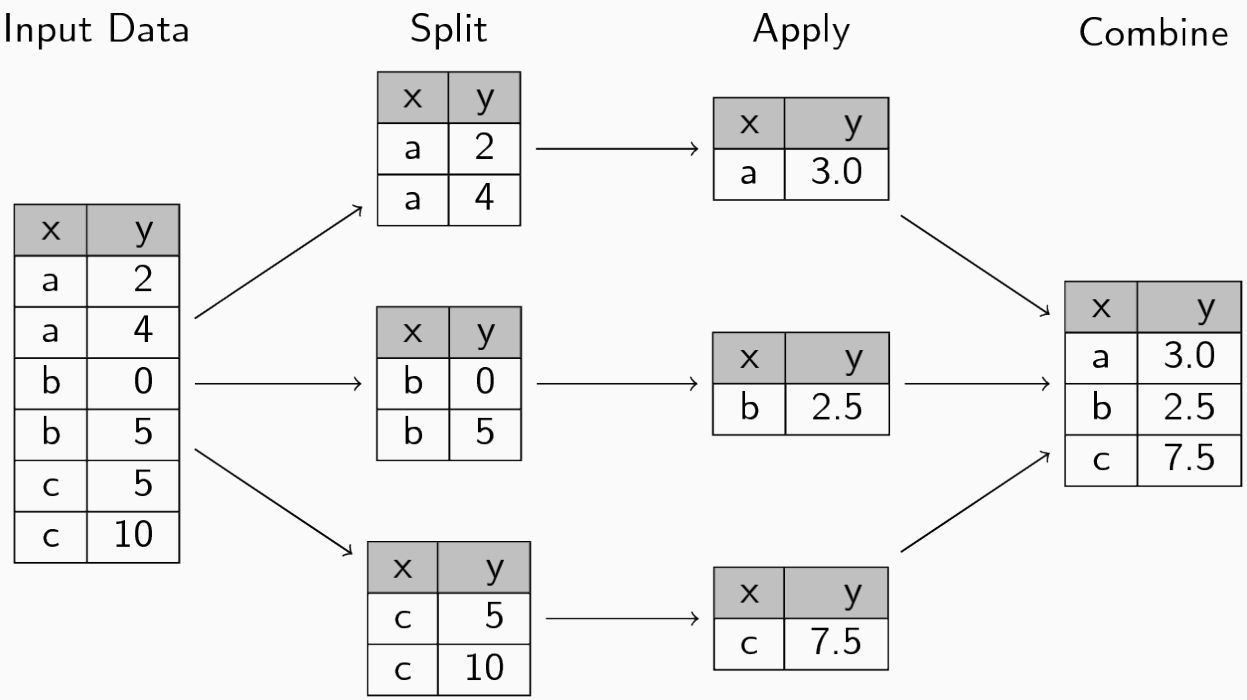
- for objects with dimension (matrix, array, data.frame)
- apply over (a) chosen dimension(s)

```
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(my_matrix, c(1, 2),
      function(value) value^2)    # apply per row and column

#      [,1] [,2] [,3]
# [1,]    1    9   25
```

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



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, it how an object behaves.
- 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"

head(iris)

#   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
# 1          5.1         3.5          1.4         0.2  setosa
# 2          4.9         3.0          1.4         0.2  setosa
# 3          4.7         3.2          1.3         0.2  setosa
# 4          4.6         3.1          1.5         0.2  setosa
# 5          5.0         3.6          1.4         0.2  setosa
# 6          5.4         3.9          1.7         0.4  setosa
```

S3 - class

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

# [1] "no data.frame"

head(iris)

# $Sepal.Length
#   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3
#  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4
#  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0
#  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
#  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0
#  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3
# [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6
# [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9
# [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

S3 - class

Good practice: *constuctor 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 intensivevvalidation.

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
```

! Check the generic to see the arguments that should be included.

Good practices (enforced for packages on CRAN):

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

Inspect the generic!

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 classes

```
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: use inheritance hierachically (subclass and

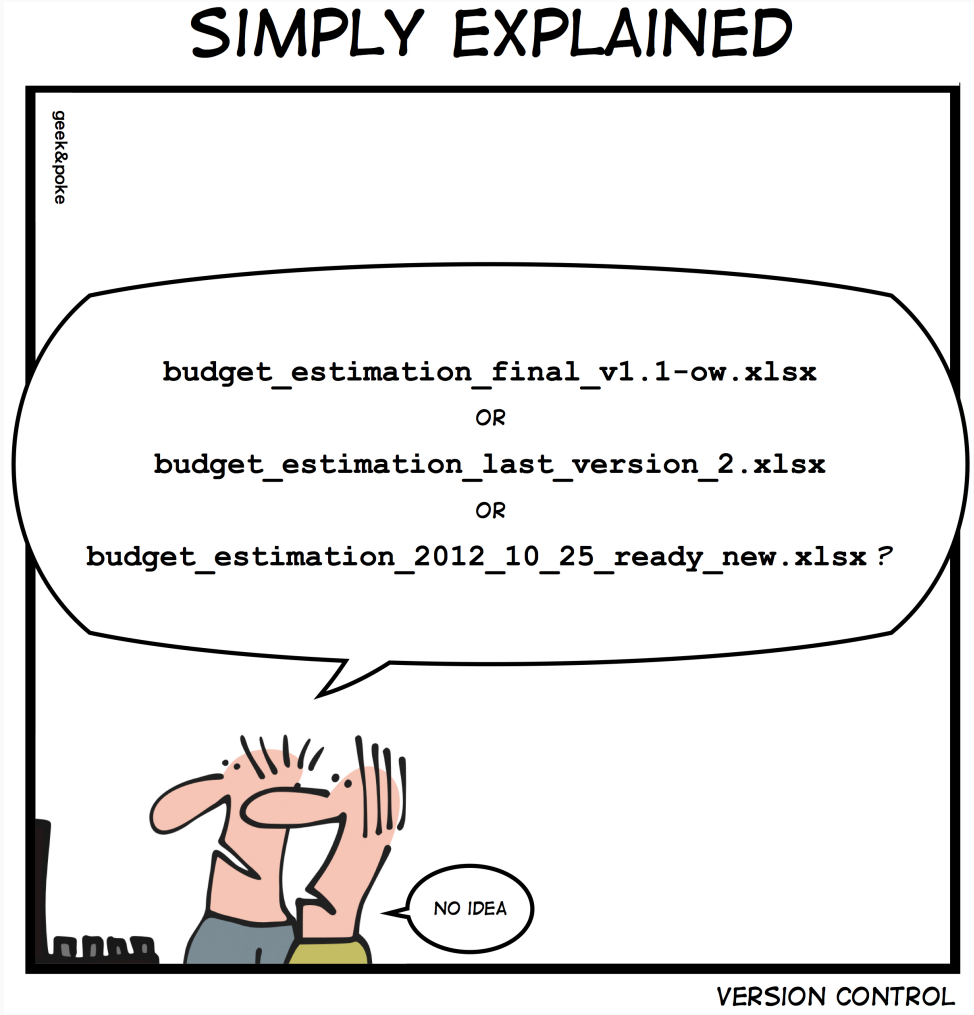
S3 - Inheritance and Dispatch

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"
```

Version Controlling (Git + Github)

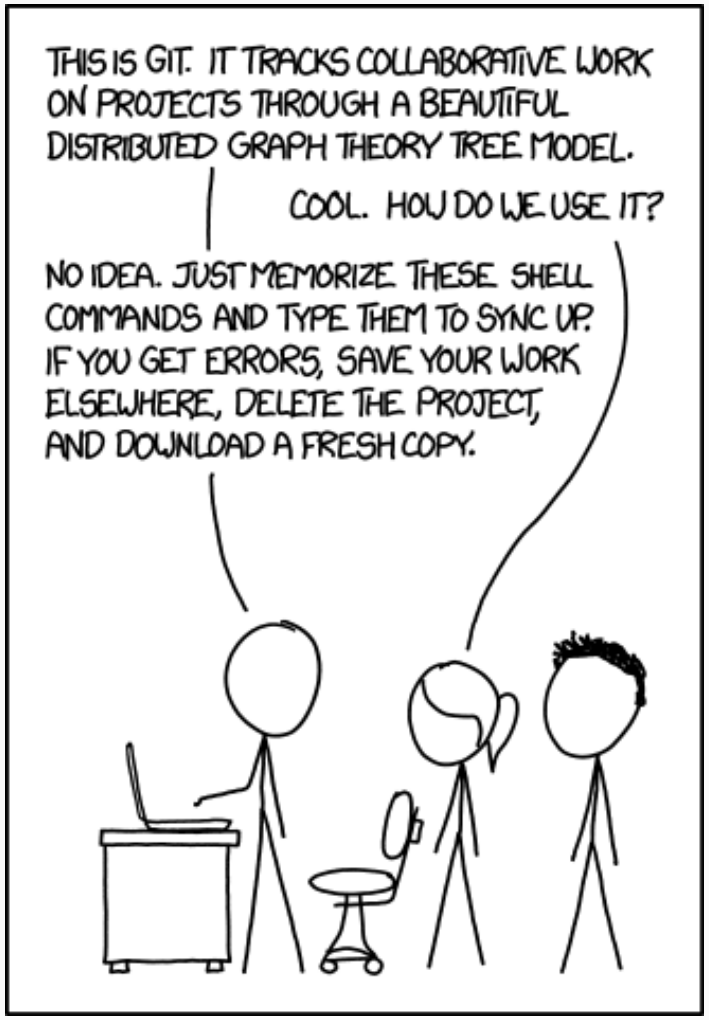
Motivation


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)

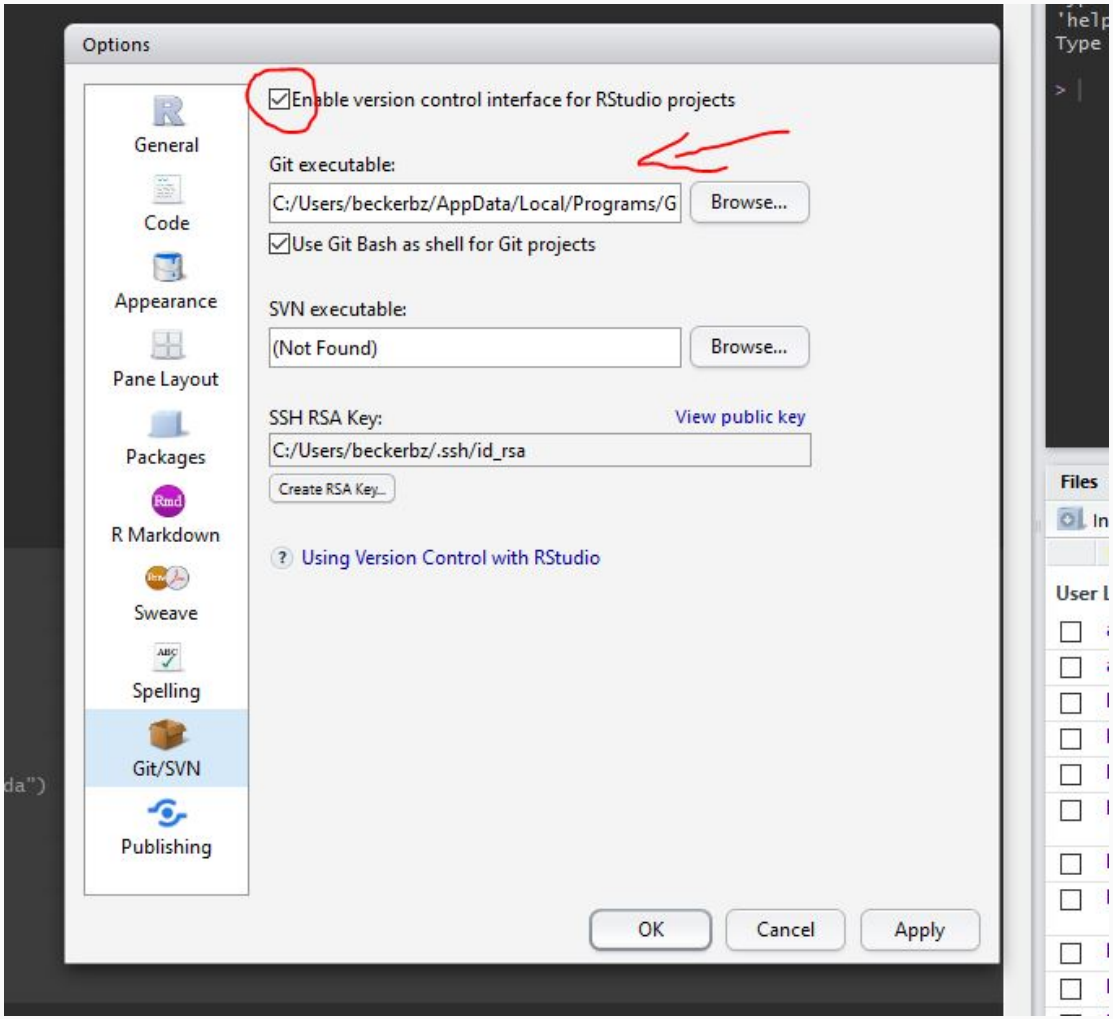
But...


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.

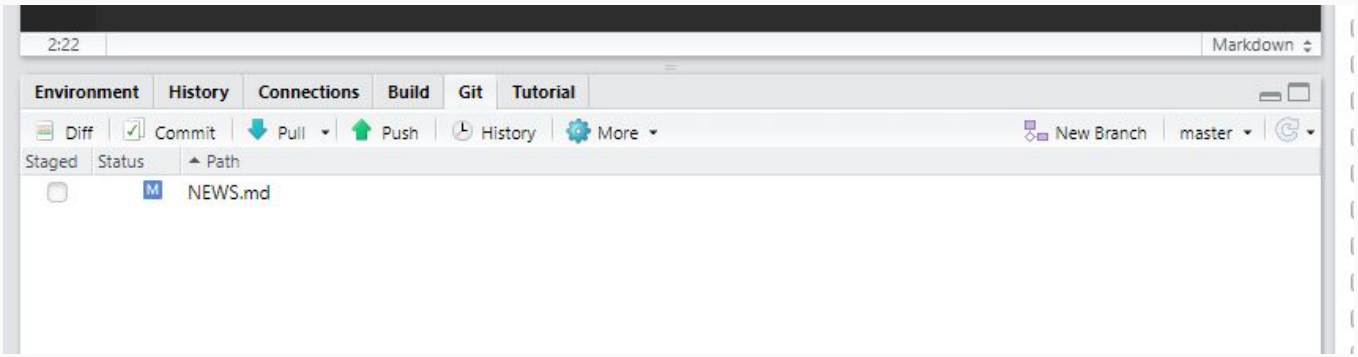
Requirements

- Install git
- (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

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?