



```
example_script.R
Source on Save
1 # Loading two packages into your library
2 # tidyverse and gapminder
3 library(tidyverse)
4 library(gapminder)
5
6 # Modify data
7 gapminder2007 = gapminder %>%
8   filter(year == 2007)
9
10 # Plot data
11 gapminder2007 %>%
12   ggplot(aes(x = gdpPerCap, y = lifeExp)) +
13     geom_point()
14
15 # Statistical test
16 .test(lifeExp ~ gdpPerCap > 20000, data = gapminder2007)
17
18
```

**writing code  
for others**



# Some disclaimers

- ▶ My credentials
- ▶ Reproducibility { rant time }
- ▶ Note: this talk
  - will be (mostly) **R**-based
  - inspired by a couple of talks from this lady →
- ▶ Advertisement time: join the [EMC coding café](#)

POV: you look at code  
you wrote last year



# But enough about me...

You need to repeat an operation a few times (3-10). Chose a character:

- Copy-paste athlete
- For loop master
- Apply family mobster
- Vector-ciraptor

How confident are you writing a function:

- A what now?
- I can try if you really need me to
- definitely
- My functions write their own functions, b\*tch

code **7217 1669**



# The *holy trinity* of code quality

**stability**

aka **reproducible**

**flexibility**

aka **cheap to modify**

**readability**

aka **easy to understand**





# The 10 commandments of good code

# 1 DO NOT REPEAT YOURSELF (DRY)

**Don't** do this

```
# Welcome to my script

## [some code]

# Please input the path to save some output
write.csv(some_output, 'your/user/path/output1.csv')

## [some more code]

write.csv(some_other_output, 'your/user/path/output10.csv')
```

a. use **variables**

```
# Welcome to my script
# Please input the path where to save the output
user_path <- 'your/user/path'

## [some code]

write.csv(some_output, file.path(user_path, 'output1.csv'))

## [some more code]

write.csv(some_other_output, file.path(user_path, 'output10.csv'))
```

**readability**

**flexibility**

**stability**

# 1 DRY level: beginner

## Don't do this

```
# e.g. fitting linear models
fit1 <- lm(out1 ~ exp1 + cov1, data = data)
# [some more code]

fit2 <- lm(out1 ~ exp2 + cov1, data = data)
# [some more code]

fit3 <- lm(out1 ~ exp3 + cov1, data = data)
# [some more code]
```

## b. use a **for loop**

```
fits <- list()

for (exp in c('exp1', 'exp2', 'exp3')) {
  # Construct formula as a string
  my_formula <- paste('out1 ~', exp, '+ cov1')
  # Save linear model output inside a list
  fits[[paste0('fit_', exp)]] <- lm(as.formula(my_formula), data=data)
}
# Access model information (e.g. coefficients)
coef(fits$fit_exp1) # or fits[['fit_exp1']] or fits[1] or fits [[1]]
```

readability

flexibility

stability



# 1 DRY level: intermediate

**Don't** do this

```
# e.g. fitting linear models
fit1 <- lm(out1 ~ exp1 + cov1, data = data)
# [some more code]

fit2 <- lm(out1 ~ exp2 + cov1, data = data)
# [some more code]

fit3 <- lm(out1 ~ exp3 + cov1, data = data)
# [some more code]
```

Unlock **vectorization!** →

c. use a **function**

```
fit_model <- function(exp) {
  # Construct formula as a string
  my_formula <- paste('out1 ~', exp, '+ cov1')
  # Fit the linear model
  fit <- lm(as.formula(my_formula), data=data)
  # [some more code]
  return(fit)
}
```

```
# Fit the model with each exposure
fits <- lapply(c('exp1', 'exp2', 'exp3'), fit_model)
```

```
fits <- list()
for (exp in c('exp1', 'exp2', 'exp3')) {
  fits[[paste0('fit_', exp)]] <- fit_model(exp)
}
```

**readability**

**flexibility**

**stability**



# 1 DRY level: advanced

**Don't** do this

```
# e.g. fitting linear models
fit1 <- lm(out1 ~ exp1 + cov1, data = data)
# [some more code]

fit2 <- lm(out1 ~ exp2 + cov1, data = data)
# [some more code]

fit3 <- lm(out1 ~ exp3 + cov1, data = data)
# [some more code]

fit4 <- lm(out2 ~ exp1 + cov1, data = data)
# [some more code]

fit5 <- lm(out2 ~ exp2 + cov1, data = data)
# [some more code]
```

d. add **arguments** to your **function**

```
fit_model <- function(exp, out) {
  my_formula <- paste(out, '~', exp, '+ cov1')
  fit <- lm(as.formula(my_formula), data=data)
}

fits <- list()
for (out in c('out1','out2')) {
  for (exp in c('exp1','exp2','exp3')) {
    fits[[paste0('fit_', out, exp)]] <- fit_model(exp, out)
  }
}
```

**readability**

**flexibility**

**stability**

# 1 DRY level: brat

## Don't do this

```
# e.g. fitting linear models
fit1 <- lm(out1 ~ exp1 + cov1, data = data)
# [some more code]

fit2 <- lm(out1 ~ exp2 + cov1, data = data)
# [some more code]

fit3 <- lm(out1 ~ exp3 + cov1, data = data)
# [some more code]

fit4 <- lm(out2 ~ exp1 + cov1, data = data)
# [some more code]

fit5 <- lm(out2 ~ exp2 + cov1, data = data)
# [some more code]
```

```
fit_model <- function(exp, out, cov = '+ cov1') {
  my_formula <- paste(out, '~', exp, cov)
  fit <- lm(as.formula(my_formula), data=data)
  print(summary(fit))
  NULL
}

# Define all possible combinations of arguments
exps <- c('exp1', 'exp2', 'exp3')
outs <- c('out1', 'out2')
covs <- c('+ cov1', '+ cov1 + cov2')

all_models <- expand.grid(exp = exps, out = outs, cov = covs)

# Apply the model
do.call(mapply, c(fit_model, all_models))
# OR
# use purrr::pmap()
```

readability

flexibility

stability

## 2 # comment. your. code.

*Explain yourself, what seems obvious now may become confusing once you forgot the context (...for me: pretty darn soon)*

## 3 Don't expect *anyone* to read the comments

Good code should *"read like English"*

→ Pick the right **names** your functions and variables

→ (in R) use **pipng** ( `%>%` , `|>` )

"there are only two hard things in Computer Science: cache invalidation and **naming things.**"

The code should explain **what**, the comment should explain **why**

readability

flexibility

stability



## 4 Use “chapters” and “paragraphs”

Avoid very long scripts (with 500+ lines)

- Divide your pipeline into “chapter” **files**
- ... *number* them (when it makes sense)
- *Tip*: access variables or functions defined inside other files:

```
source("0-functions.R")
```

```
from definitions.backend import *
```

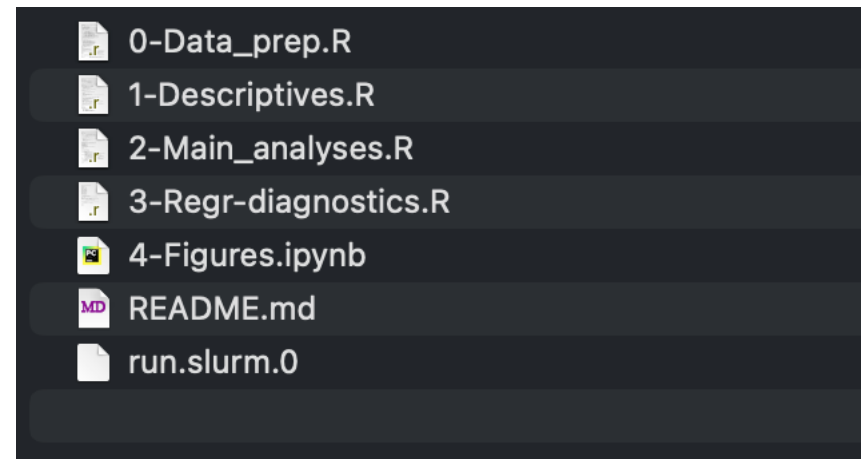
Divide your script into **sections** using

```
# [section title] -----
```

**readability**

**flexibility**

**stability**



A **few small functions** > one *monster* function

## 5 You have to break it (before you can fix it)

Repeat after me: errors should **never** pass silently .

In the face of *ambiguity* (which will probably be there), don't guess

---> Check your **user input** (...often: classes and types)

```
if (is.numeric(variable)) {  
  mean(variable)  
} else if (is.factor(variable)) {  
  warning("Factor means are not meaningful, how about frequencies.")  
  table(variable) |  
} else {  
  stop("You need to input either a numeric variable or a factor.")  
}
```

Don't comment  
& uncomment  
code, *especially*  
if you need to  
do it in multiple  
places!

---> **Throw errors** and/or **catch** them using `tryCatch()`

readability

flexibility

stability

# BUT... beware of **onion functions**

```
get_some_data <- function(config, outfile) {
  if (config_ok(config)) {
    if (can_write(outfile)) {
      if (can_open_network_connection(config)) {
        data <- parse_something_from_network()
        if(makes_sense(data)) {
          data <- beautify(data)
          write_it(data, outfile)
          return(TRUE)
        } else {
          return(FALSE)
        }
      } else {
        stop("Can't access network")
      }
    } else {
      ## uhm. What was this else for again?
    }
  } else {
    ## maybe, some bad news about ... the config?
  }
}
```

```
get_some_data <- function(config, outfile) {
  if (config_bad(config)) {
    stop("Bad config")
  }

  if (!can_write(outfile)) {
    stop("Can't write outfile")
  }

  if (!can_open_network_connection(config)) {
    stop("Can't access network")
  }

  data <- parse_something_from_network()
  if(!makes_sense(data)) {
    return(FALSE)
  }

  data <- beautify(data)
  write_it(data, outfile)
  TRUE
}
```



## 6 Flat is better than nested

Not all **ifs** need an **else**...

→ Use quick `stop()` and `return()` ("*guard clauses*")  
e.g. `stopifnot(is.numeric(x) || is.logical(x))`

Use control-flow *alternatives*:

```
ifelse(age_yrs < 2, "baby",
       ifelse(age_yrs < 13, "kid",
              ifelse(age_yrs < 20, "teen",
                     "adult")
       )
)
```

```
library(dplyr)

tibble(
  age_yrs = c(0, 4, 15, 24, 55),
  age_cat = case_when(
    age_yrs < 2 ~ "baby",
    age_yrs < 13 ~ "kid",
    age_yrs < 20 ~ "teen",
    TRUE ~ "adult"
  )
)
```

**Less indentation**  
> more indentation.

**readability**

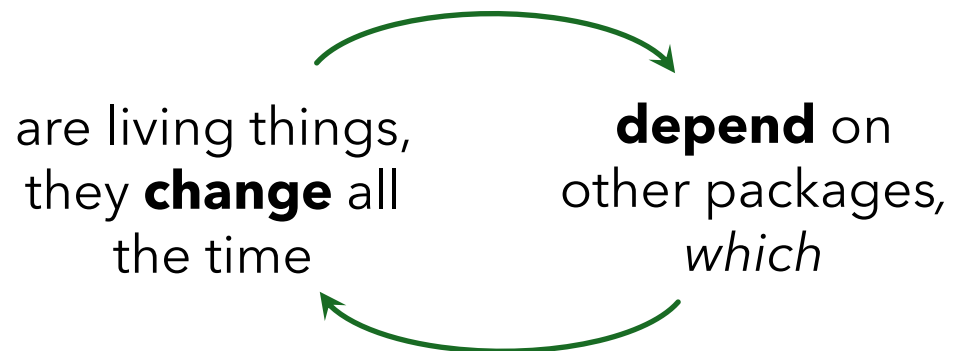
**flexibility**

**stability**

# 7 Code independence

Avoid adding dependencies  
(unless you *really* need them)

Packages:



Good practice:: **contextualize functions!**

```
# Read file  
foreign::read.spss(my_spss_file)
```

**Don't** do this

```
# Upload packages  
library('foreign')  
library('table1')  
library('tidyverse')  
library('dplyr')  
library('gt')  
library('plyr')  
library('car')  
library('nnet')  
library('pscl')  
library('broom')  
library('xtable')  
library('car')  
library('MASS')  
library('emmeans')  
library('mice')
```

readability

flexibility

**stability**

## 8 Version control (a): use **environments**

A *programming environment* = “the infrastructure (including the compiler and the tools) where the code can run”

Basically: the **version** of R and the packages

...and, sometimes, the OS

Environment managers: `conda`, `venv`, `pyenv`

From R:

```
# 1. Activate the environment
renv::activate()
# 2. Create a "lockfile": a "snapshot" of the current project
renv::snapshot()
# 3. Reproduce it any place else
renv::restore()
```

readability

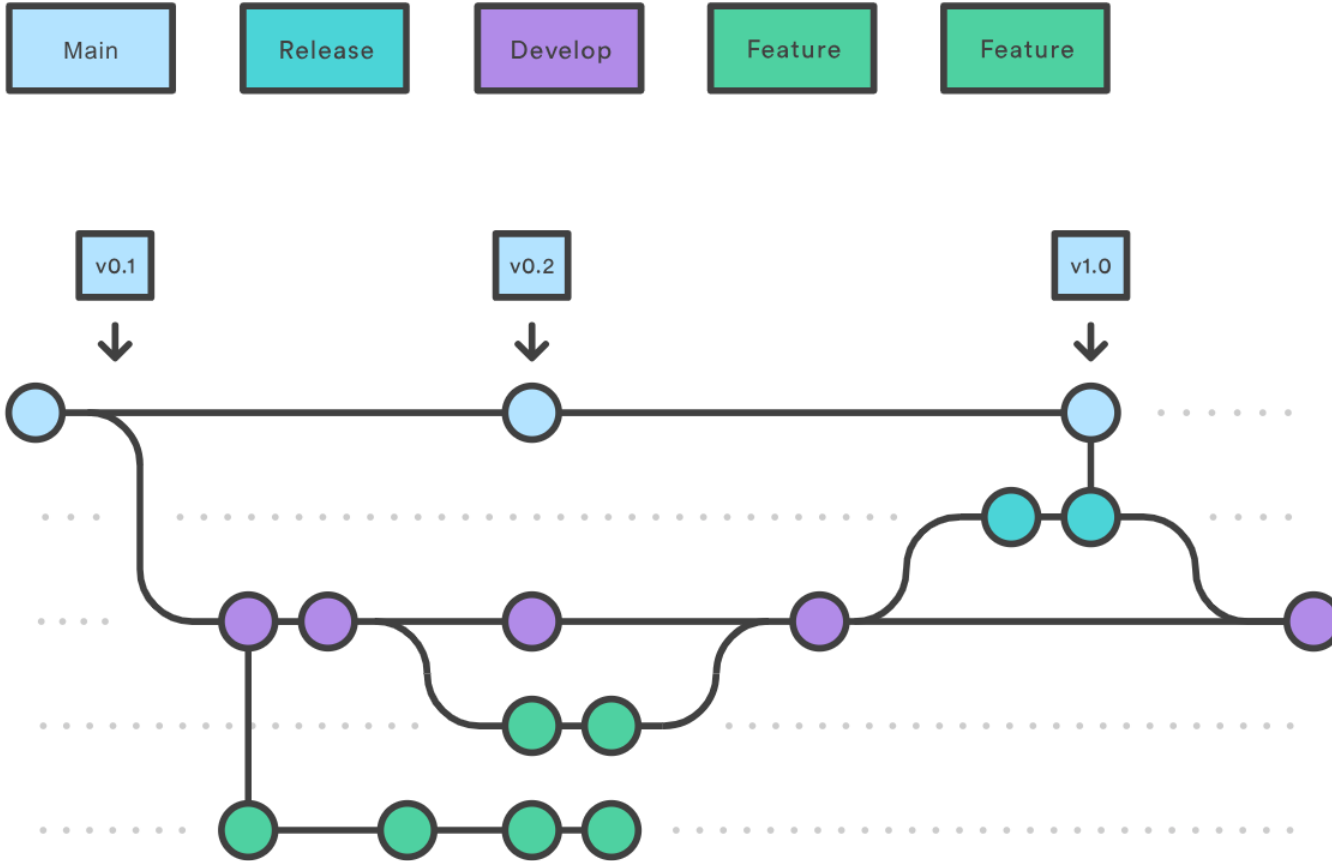
flexibility

stability



# 9

## Version control (b): use **git**



[note] Upcoming talk:

### **Introduction to git and github**

PhD meeting  
5 Feb 2024, 14:00

readability

flexibility

**stability**

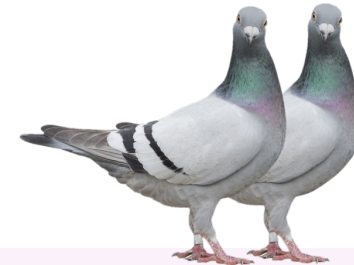
## 8 Version control (a & b): use **projects**

9 I often see scripts starting like this:

```
setwd('path/to/project/folder/')
```

Use a **project** instead...

**RStudio** makes this very easy! [\[demo\]](#)



Once you create a project, it's much easier to manage your files, your dependencies (with renv), track code versions (with git) and (eventually) give it somebody else.

readability

flexibility

**stability**

# 10

## Peer review, baby 🕶️



- As an author, **ask people to review your code!**
  - > Best medicine against *code blindness*
  - > If you can (*and you can*) have at least one co-author review the code
- As a reviewer, **ask to see the code**: that's the real *scientific product*

readability

flexibility

stability

# In summary...

- 1 **Don't repeat yourself**: use variables, functions and loops
  - 2 **#comment** your code
  - 3 Pick names so you can **read\_code\_like("English")**
  - 4 Keep your files (and functions) **short & organised**
  - 5 Throw and catch **errors**
  - 6 Avoid (a lot of) **indentation**
  - 7 Avoid package **dependencies**
  - 8 Use **environments**
  - 9 Use **git**
  - 10 **Peer review** the code
- } control those **versions**

readability

flexibility

stability



## **Bonus:** some extra **don'ts**

- X forget to `set.seed()`
- X refer to columns in a data frame using their numbers
- X make functions dependent on global environment variables

**Thanks for listening!**

**(happy\_coding\_everyone;)**