statistical computing in r

source

20 May 2015

statistical computing in r

20 May 2015

source

(html best viewed in chrome)

background

- Best practices for scientific computing, PLoS biol, 2014

OPEN ACCESS Freely available online

PLOS | BIOLOGY

Community Page

Best Practices for Scientific Computing

Greg Wilson[1]*, D. A. Aruliah[2], C. Titus Brown[3], Neil P. Chue Hong[4], Matt Davis[5], Richard T. Guy[6¤], Steven H. D. Haddock[7], Kathryn D. Huff[8], Ian M. Mitchell[9], Mark D. Plumbley[10], Ben Waugh[11], Ethan P. White[12], Paul Wilson[13]

1 Mozilla Foundation, Toronto, Ontario, Canada, **2** University of Ontario Institute of Technology, Oshawa, Ontario, Canada, **3** Michigan State University, East Lansing, Michigan, United States of America, **4** Software Sustainability Institute, Edinburgh, United Kingdom, **5** Space Telescope Science Institute, Baltimore, Maryland, United States of America, **6** University of Toronto, Toronto, Ontario, Canada, **7** Monterey Bay Aquarium Research Institute, Moss Landing, California, United States of America, **8** University of California Berkeley, Berkeley, California, United States of America, **9** University of British Columbia, Vancouver, British Columbia, Canada, **10** Queen Mary University of London, London, United Kingdom, **11** University College London, London, United Kingdom, **12** Utah State University, Logan, Utah, United States of America, **13** University of Wisconsin, Madison, Wisconsin, United States of America

1 write programs for people, not computers
2 let the computer do the work
3 make incremental changes
4 don't repeat yourself (or others)
5 plan for mistakes
6 optimize only after it works correctly
7 document design/purpose, not mechanics
8 collaborate

motivation

- computing has become the backbone of science
- nearly all scientific papers have theoretical modeling, data acquisition, cleaning, data analysis, figures and plots, p-values, etc
  - **every** result depends on computing
- do computing well, but in your language
- if we do not do computing well, we duke
- style and organization matter
  - coding for others
  - coding for (**future**) you

motivation

Our studies support the claim that knowledge of programming plans and rules of programming discourse can have a *significant impact on program comprehension*.

It is not merely a matter of aesthetics that programs should be written in a particular style.

Rather there is a *psychological basis for writing programs in a conventional manner*: programmers have strong expectations that other programmers will follow these discourse rules.

If the rules are violated, then the utility afforded by the expectations that programmers have built up over time is effectively nullified.

Soloway and Ehrlich, 1984

un-motivation

- styling and organization can be very personal
  - … and old habits die hard
- overhead of learning new things
- do what works best ~~for you~~
- regardless of preferences, have a coding style and follow it
  - be ***consistent***

1 write code for (other) people, not computers

- clear, concise, **transparent** code
- our brains are designed to recognize patterns
  - with consistent patterns (style/formatting), only content remains
- comment copiously about ***what*** code does *not* **how** it works
- naming things is hard: short, concise words; *verbs* for functions
  - don't use reserved words (function names, others: `?reserved`)

```
fortunes::fortune('dog')
```

Firstly, don't call your matrix 'matrix'. Would you call your dog 'dog'? Anyway, it might clash with the function 'matrix'.

Barry Rowlingson, R-help (October 2004)

(1b) styling and formatting

- style guides (google, hadley)
- indent lines **!**
- spacing around operators (`+`, `-`, `%in%`, `<-`, `,`)
  - improves readability, pinpoint mistakes
- any decent text editor (vim, sublime, emacs/ESS) or IDE (Rstudio) will do this for you
- 80 character width
  - scroll up **+** down, **not** left **+** right *and* up **+** down
  - promotes good code and logic by avoiding wrapping long lines

(1c) styling and formatting - example

```
mylmfit=lm(mpg~wt+disp,mtcars)
summary=summary(mylmfit)
coefficients=summary$coefficients
coefficients=round(coefficients,digits=2)
estimates=coefficients[,1]
pvalues=coefficients[,4];pvalues
```

```
## (Intercept)          wt        disp
##        0.00        0.01        0.06
```

```
pvalues[1]="<0.01"
matrix=cbind(estimates,pvalues)
colnames(matrix)=c("Estimates","p-values")
as.data.frame(matrix)
```

```
##             Estimates p-values
## (Intercept)     34.96    <0.01
## wt              -3.35     0.01
## disp            -0.02     0.06
```

(1d) styling and formatting - example

```
fit <- lm(mpg ~ wt + disp, data = mtcars)
summ <- summary(fit)$coefficients
summ <- round(summ, digits = 2)
cbind.data.frame(Estimate = summ[, 1],
                 `p-value` = format.pval(summ[, 4], eps = .01))
```

```
##             Estimate p-value
## (Intercept)    34.96   <0.01
## wt             -3.35    0.01
## disp           -0.02    0.06
```

2 let the computer do the work

- computer time is cheap; people time (and **frustration**) is expensive
  - number-crunching, run-time, simulations
  - thinking about problem, writing code, errors, manipulating, road-blocks, re-writing, higher priorities, tables and figures, tweaking, writing words, adding analyses, removing observations, etc
- modularize code into reusable tools
- functions do **one** thing and do it well
- small, easy-to-understand pieces that can combine into something more complex

Divide each difficulty into many parts as is feasible and necessary to resolve it.

René Descartes

(2b) modularize code - example

```
f1 <- function(...) {
  fit <- lm(mpg ~ wt + disp, data = mtcars)
  summ <- round(summary(fit)$coefficients, digits = 2)
  cbind.data.frame(Estimate = summ[, 1],
                   `p-value` = format.pval(summ[, 4], eps = .01))
}

f1()
```

```
##             Estimate p-value
## (Intercept)    34.96   <0.01
## wt             -3.35    0.01
## disp           -0.02    0.06
```

```
f1(mpg ~ wt + disp + hp, digits = 3)
```

```
##             Estimate p-value
## (Intercept)    34.96   <0.01
## wt             -3.35    0.01
## disp           -0.02    0.06
```

(2c) modularize code - example

```
f2 <- function(form, dat = mtcars, digits = 2) {
  fit <- lm(form, data = dat)
  summ <- round(summary(fit)$coefficients, digits = digits)
  cbind.data.frame(Estimate = summ[, 1],
                   `p-value` = format.pval(summ[, 4], eps = 10 ** -digits))
}

f2(mpg ~ wt + hp + factor(cyl), digits = 3)
```

```
##              Estimate p-value
## (Intercept)    35.846  <0.001
## wt             -3.181  <0.001
## hp             -0.023   0.064
## factor(cyl)6   -3.359   0.024
## factor(cyl)8   -3.186   0.154
```

3 make incremental changes

without version control:

- freeze the current state to create a daily working-copy
  - if unsatisfied, rollback to previous day (hour, week)
- for major changes, freeze current project and create a new directory
- organize data pulls by date
  - unlimited ***free*** storage (thanks, dana-farber)

with version control (git, github):

- for your packages: **yes**
- for collaboration: **yes**
- for analyses: **no**

4 don't repeat yourself

- write (and test) functions *once* and re-use
  - add frequently-used functons (or data) to a personal package
- repeating similar tasks in the same project
  - "Treatments included RCHOP (n = 10, 50%), R-CVP (n = 5, 30%), and RCHOEP (n = 4, 20%)"
  - "Most common toxicities were anemia (n = 10, 50%), thrombocytopenia (n = 9, 45%), nuetropena (n = 7, 30%)"
  - **mistakes, typos, and errors, oh my!**
- e.g., repeating similar tasks across multiple projects
  - plotting parameters to suit a particular journal
  - wrapper functions to save typing and mistakes

(4b) don't repeat yourself - example

```
print_counts <- function(x) {
  ## x a vector of character strings (or factors)
  ## usage: print_counts(letters[1:5])

  ## helper function: ipr
  ipr <- function(x) {
    ## tests
    ## ipr(1); ipr(1:2); ipr(1:5)
    if (length(x) == 1) x else
    if (length(x) == 2) paste(x, collapse = ' and ') else
      sprintf('%s, and %s', paste(x[-length(x)], collapse = ', '), tail(x, 1))
  }
  tt <- if (!is.table(x)) sort(table(x), decreasing = TRUE) else x
  ipr(sprintf('%s (n = %s, %s%%)', names(tt), tt, round(tt / sum(tt) * 100)))
}

table(tx <- rep(c('RCHOP','R-CVP','RCHOEP'), c(10, 6, 4)))
print_counts(tx)
```

```
##
##  R-CVP RCHOEP  RCHOP
##      6      4     10
## [1] "RCHOP (n = 10, 50%), R-CVP (n = 6, 30%), and RCHOEP (n = 4, 20%)"
```

(4c) don't repeat yourself - example

- wrappers
  - set defaults for another function

```
(x <- c(rnorm(4), NA))
```

```
## [1] -0.53813422 -1.95486073 -0.04045531 -1.20426459          NA
```

```
sum(x)
```

```
## [1] NA
```

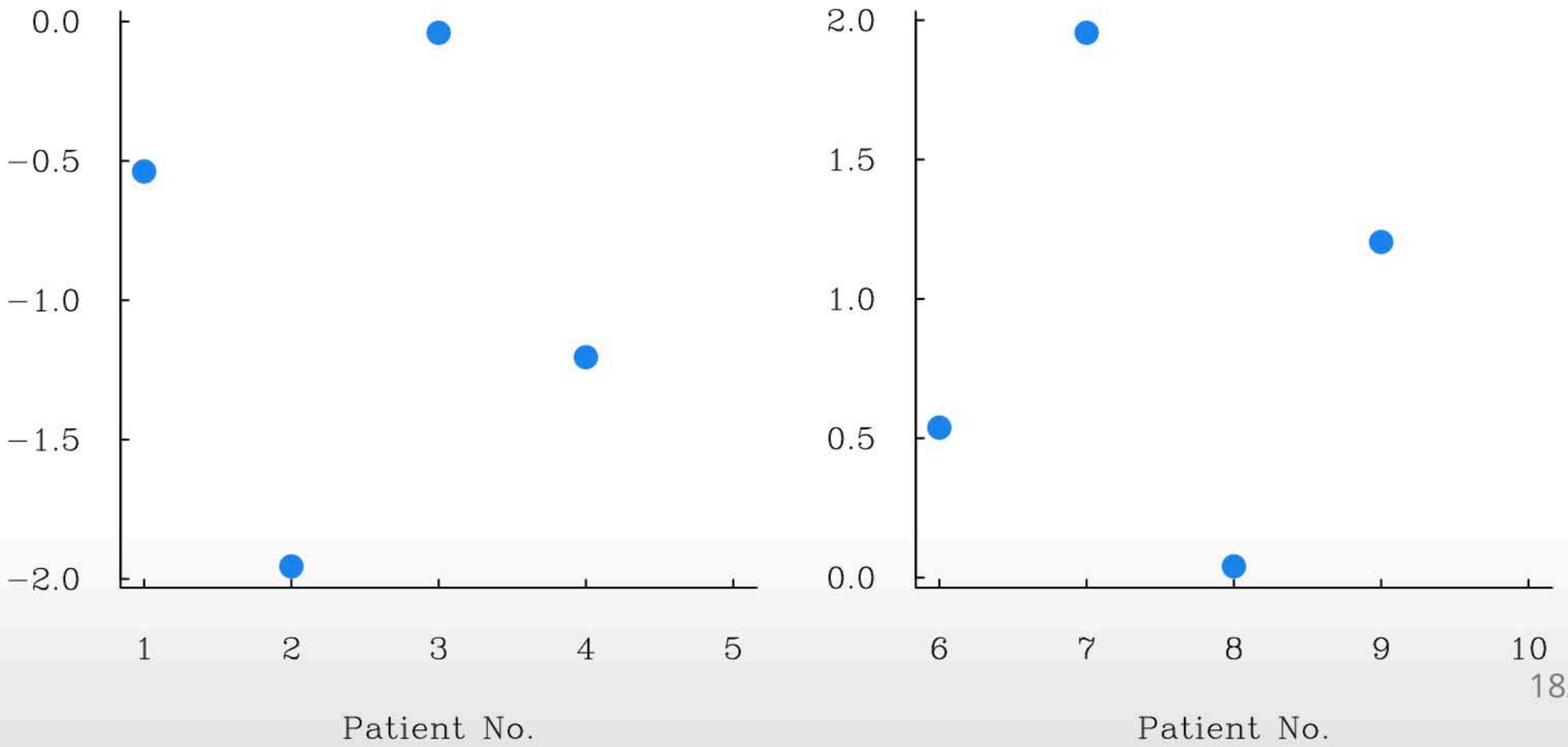
```
sum2 <- function(...) sum(..., na.rm = TRUE)
sum2(x)
```

```
## [1] -3.737715
```

(4d) don't repeat yourself - example

```
my_plot <- function(...)
  plot(..., las = 1, bty = 'l', tcl = .2, xlab = 'Patient No.', ylab = '',
       pch = 19, col = 'dodgerblue2', cex = 1.5, family = 'HersheySerif')

par(mfrow = c(1, 2), mar = c(5, 3, .5, 1), bg = 'transparent')
my_plot(1:5, x)
my_plot(6:10, -x)
```



5 plan for mistakes

- record your work in **self-contained**, reproducible script(s)
- restart with a clean session at regular intervals
  - `source()` your code
- clean out anything you are no longer using or forgot to define
  - added benefit of having tested your code
  - **(**knitr does this each time you compile**)**
- name scripts and analyses with **numbered**, descriptive words:
  - `00-get_data.R`
  - `01-munge_data.R`
  - `02-describe_data.R`
  - `03-fancy_models.R`
  - `99-appendix.R`
  - `99-1-appendix.R`
  - `99-2-appendix.R`
  - etc

6 optimization (a deviation)

- optimizing your time >>> writing fastest, most-efficient code
  - cost of your time vs cost of computing time
  - e.g., `for` loops vs vectorization, `*apply`
- reproducibility: past, present
- automation: future
- `package`: everything in a workable environment
- most of the work should be data munging and exploration
  - i.e., error-checking
  - **do not** waste time on mundane, obnoxious tasks like ms word or excel
  - copy/pasting fewer than ***one*** time
  - automation of output - create dynamic documents (knitr, sweave)

7 document (everything)

- functions

```
dmy <- function(d, m, y, origin = c(1, 1, 1900)) {
  ## parse day/month/year column data into standard date format
  ## examples:
    # dmy(NA, 5, 2005)
    # dmy(NA, 5, 13, origin = c(1, 1, 2000))
  ## arguments:
    # d, m, y: day, month, year as single integers, NAs, or vectors
    # origin: a vector of length three giving the origins for d, m, and y
  f <- function(a, b) {
    suppressWarnings(a <- as.numeric(a))
    ifelse(is.na(a), b, a)
  }
  y <- ifelse(nchar(y) <= 2, f(y, 0) + origin[3], f(y, 0))
  as.Date(sprintf('%04s-%02s-%02s', y, f(m, origin[2]), f(d, origin[1])))
}

dmy(NA, 5, 11:15, origin = c(15, 1, 2000))
```

```
## [1] "2011-05-15" "2012-05-15" "2013-05-15" "2014-05-15" "2015-05-15"
```

(7b) document (everything)

- functions in packages (with roxygen2)

```
#' Date parse
#'
#' Parse day/month/year column data into standard date format
#'
#' For two-digit years, the origin year should be specified; otherwise, the
#' default of 1900 will be used. For NA year, month, or day, origin is used
#' for defaults, i.e., origin = c(15, 6, 2000) will convert missing days to
#' day 15, missing months to June, and missing years to 2000.
#'
#' @param d,m,y day, month, year as single integers or vectors
#' @param origin vector of length 3 with origins for d, m, and y, respectively;
#' see details
#'
#' @return A vector of \code{\link{Date}}-formatted strings.
#'
#' @examples
#' dmy(25, 7, 87)
#' dmy(NA, NA, 2000:2005)
#' @export

dmy <- function(d, m, y, origin) {
  ## do stuff
}
```

(7c) document (everything)

- data in packages (with roxygen2)

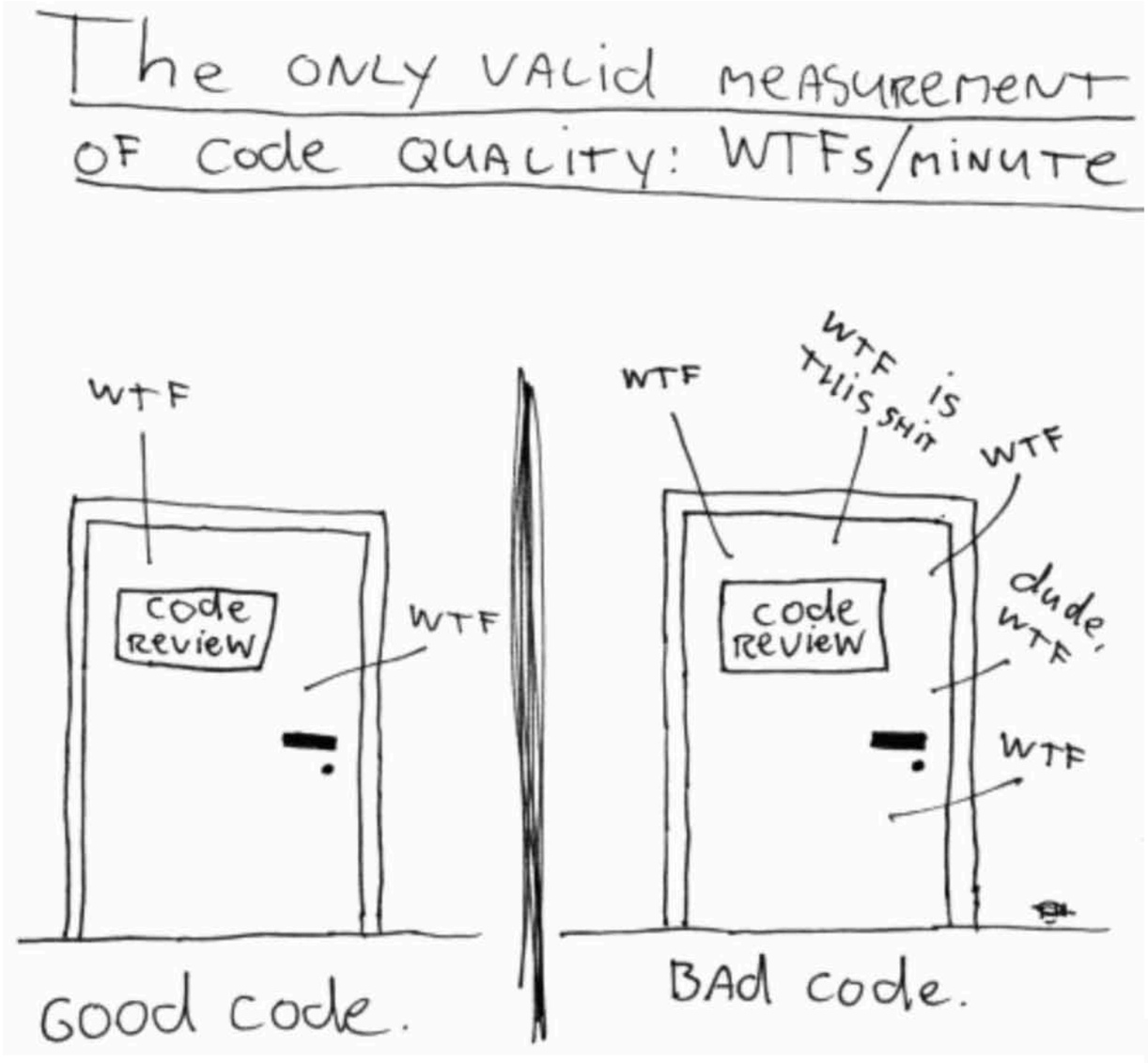
```
#' Patient demographic data for protocol 15-000
#'
#' Baseline demographic and lab results data for xx subjects enrolled on study.
#'
#' @seealso \code{\link{trainr}}, \code{\link{tox}}, \code{\link{surv}}
#'
#' @format An object of class \code{data.frame} containing xx observations and
#' yy variables:
#'
#' \tabular{rll}{
#' \tab \code{id} \tab patient unique id \cr
#' \tab \code{site} \tab enrollment site \cr
#' \tab \code{\dots} \tab ... \cr
#' \tab \code{wt} \tab weight (kg) \cr
#' \tab \code{ht} \tab height (cm) \cr
#' \tab \code{hgb} \tab hemoglobin (g/dL) \cr
#' \tab \code{\dots} \tab ... \cr
#' }
"demo"

?demo
summary(demo)
```

8 collaborate

- github
  - share, download/install packages (even if not on cran)
  - improve code; learn from others
- share useful code
  - fancy, pretty plots and graphics (or games)
  - complex data cleaning (regex)
- review code for others

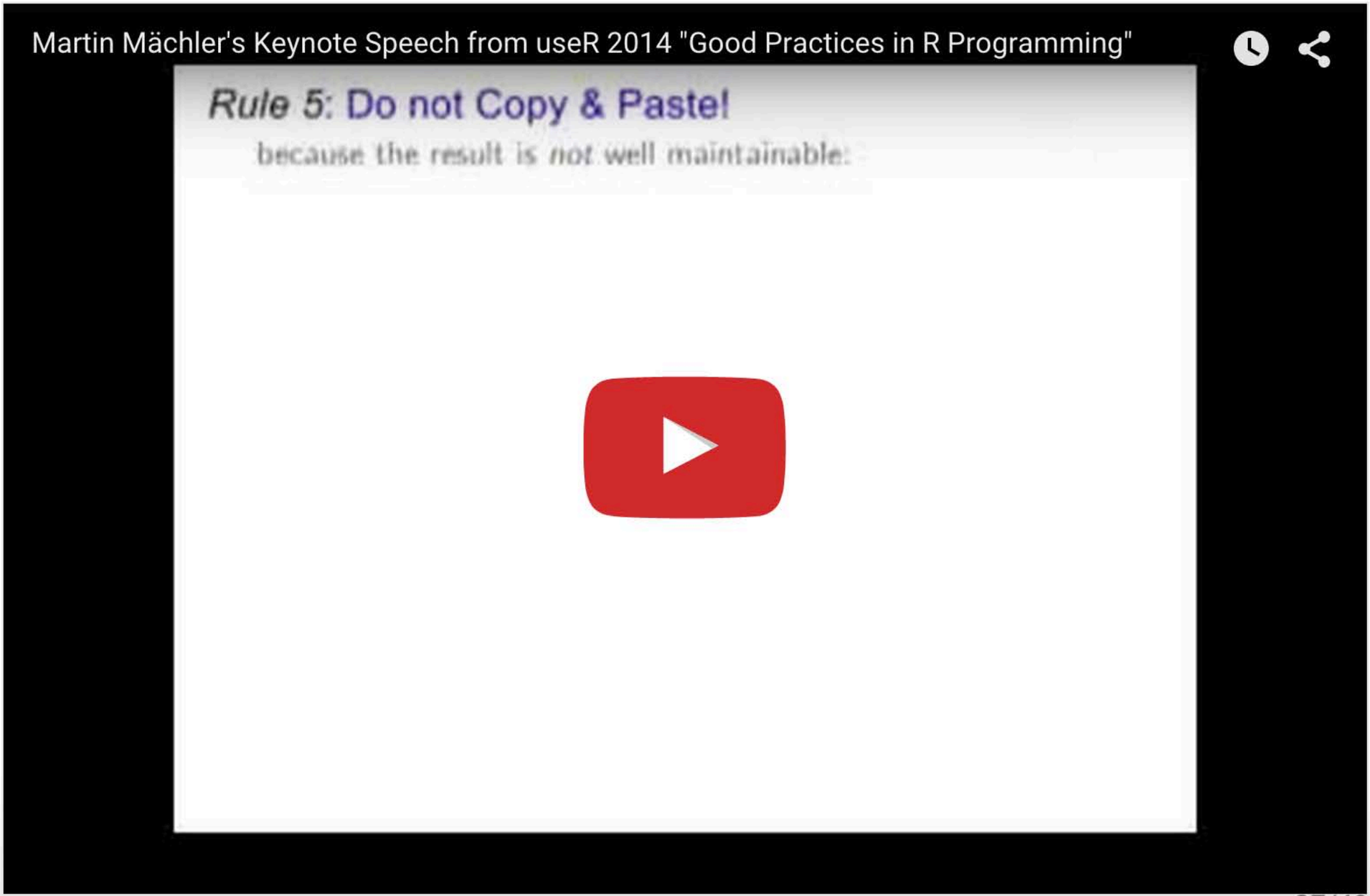
at the end of the day…



resources

- general
  - **the R Inferno**
  - advanced r [hadley]
  - Martin Mächler, R-core, slides + **youtube**
- intro to r
  - **general coding (interactive)**
  - ucla stats
  - (even) more slides
- r packages/knitr
  - r packages book [hadley]
  - **how-to + code [hilary parker]**
  - knitr [yihui]
- version control
  - git
  - git/github in rstudio

resources


Martin Mächler's Keynote Speech from useR 2014 "Good Practices in R Programming"
Rule 5: Do not Copy & Paste!
because the result is not well maintainable:

demo

putting these suggestions to practical use…

specific tips

- learn to use functionals well
  - take *functions* as arguments, apply functions to the pieces
  - `lappy`, `tapply`, `aggregate`, `mapply`/`Map`, `Reduce`, `Filter`, etc

```
## divide each column of a data frame by a different value of zz
zz <- 2:4
(out1 <- out2 <- out3 <- out4 <- data.frame(x = rep(4, 2), y = 9, z = 16))
```

```
##   x y  z
## 1 4 9 16
## 2 4 9 16
```

```
## using a for loop is clumsy and litters the workspace with
## ii, the index, and any other variables created in the loop
for (ii in 1:3)
  out1[, ii] <- out1[, ii] / zz[ii]
out1
```

```
##   x y z
## 1 2 3 4
## 2 2 3 4
```

(b) functionals

```
## better, cleaner but still clumsy
out2[] <- lapply(1:3, function(ii) out2[, ii] / zz[ii])
out2
```

```
##   x y z
## 1 2 3 4
## 2 2 3 4
```

```
## even better
out3[] <- Map(`/`, out3, zz)
out3
```

```
##   x y z
## 1 2 3 4
## 2 2 3 4
```

```
## many other ways of getting the same result
as.data.frame(as.matrix(out4) / (col(out4) + 1))
```

```
##   x y z
## 1 2 3 4
## 2 2 3 4
```

growing objects

```
n <- 1e4

system.time({
  set.seed(1)
  out1 <- NULL
  for (ii in seq_len(n))
    out1 <- cbind(out1, rnorm(100))
})
```

```
##    user  system elapsed
##  79.804  13.670  93.755
```

```
system.time({
  set.seed(1)
  out2 <- matrix(NA, nrow = 100, ncol = n)
  for (ii in seq_len(n))
    out2[, ii] <- rnorm(100)
})
```

```
##    user  system elapsed
##   0.136   0.016   0.152
```

for loops, functionals & vectorization

```
system.time({
  set.seed(1)
  out3 <- sapply(seq_len(n), function(x) rnorm(100))
})
```

```
##    user  system elapsed
##   0.244   0.013   0.257
```

```
system.time({
  set.seed(1)
  out4 <- matrix(rnorm(100 * n), nrow = 100, ncol = n)
})
```

```
##    user  system elapsed
##   0.084   0.002   0.088
```

```
l <- list(out1, out2, out3, out4)
all(sapply(seq_along(l)[-1], function(x) identical(l[x - 1], l[x])))
```

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

Never use …

- `subset()`
  - non-standard evaluation
  - not intended for programmatic use
  - `$`, `[`, `[[` are faster and more powerful

(b) subset()

```
dd <- data.frame(a = 1, x = 1:5)
f1 <- function(...) {
  y <- 3
  ## do stuff
  subset(dd, ...)
}
f1(x > y)
```

```
##   a x
## 4 1 4
## 5 1 5
```

```
## calculate y and use it to subset
y <- 1
f1(x > y)
```

```
##   a x
## 4 1 4
## 5 1 5
```

(c) subset()

```
dd <- data.frame(a = 1, x = 1:5)
subset(dd, select = a:x)
```

```
##   a x
## 1 1 1
## 2 1 2
## 3 1 3
## 4 1 4
## 5 1 5
```

```
subset(dd, select = -x)
```

```
##   a
## 1 1
## 2 1
## 3 1
## 4 1
## 5 1
```

```
a <- b <- c('a','x')
subset(dd, select = b)
```

```
##   a x
## 1 1 1
## 2 1 2
## 3 1 3
## 4 1 4
## 5 1 5
```

```
subset(dd, select = a)
```

```
##   a
## 1 1
## 2 1
## 3 1
## 4 1
## 5 1
```

(d) attach()

- `subset()`
- `attach()`
  - side-effects, clutter environments
  - `detach` is used 110% less often than `attach`

```
x <- 0
d1 <- data.frame(x = 1 , y = 2)
d2 <- data.frame(x = 2, y = 1)
attach(d1)
attach(d2)
search()
x
y
detach(d2)
y
x
rm(x)
x
rm(x)
detach(d1)
```

(e) `setwd()`

- `subset()`
- `attach()`
- `setwd()`
  - use relative paths (avoid absolute paths)
  - re-organizing directories breaks absolute paths
  - opening a project or script in R automatically sets the working directory to the directory of that script

… just remove these words from your vocabulary

Do use ...

- TRUE & FALSE **not** T & F

```
(T <- rnorm(5))
TRUE <- 1
```

- within()
  - to refer to variables without indexing ($, [, [[) in a local environment
  - modify or create new variables in a data frame

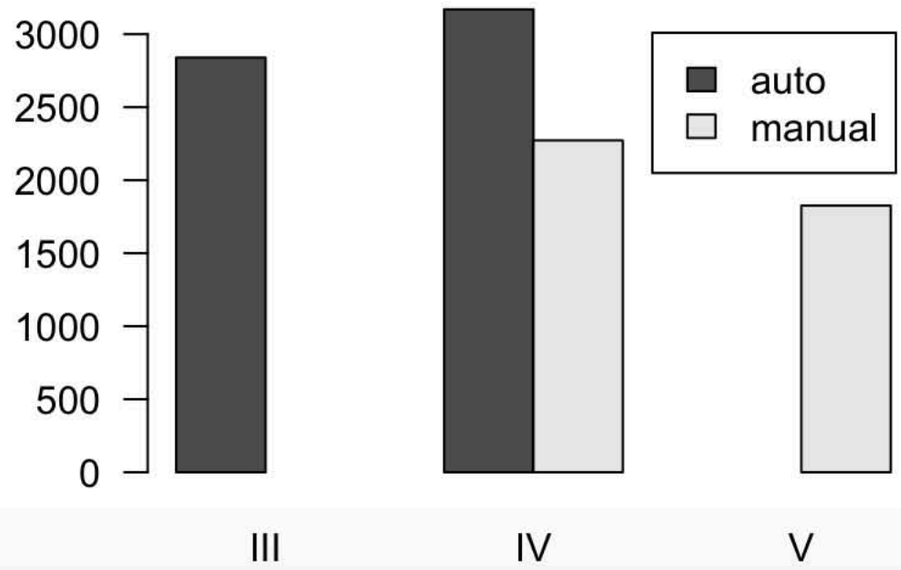
```
(dat <- within(mtcars, {
  gear <- factor(gear, levels = 3:5, labels = as.roman(3:5))
  wt <- wt * 1000
  disp <- drat <- qsec <- hp <- cyl <- NULL
}))[1:5, ]
```

```
##                    mpg   wt vs am gear carb
## Mazda RX4         21.0 2620  0  1   IV    4
## Mazda RX4 Wag     21.0 2875  0  1   IV    4
## Datsun 710        22.8 2320  1  1   IV    1
## Hornet 4 Drive    21.4 3215  1  0  III    1
## Hornet Sportabout 18.7 3440  0  0  III    2
```

(b) with()

- with()
  - to refer to variables without indexing ($, [, [[) in a local environment

```
par(bg = 'transparent', mar = c(3,4,2,2), las = 1)
with(dat[dat$mpg > 20, ],
     barplot(tapply(wt, list(am, gear), mean), beside = TRUE,
             legend.text = c('auto','manual')))
```



(c) data=, str()

- data =

```
(fit <- lm(mpg ~ gear, data = dat))
```

```
##
## Call:
## lm(formula = mpg ~ gear, data = dat)
##
## Coefficients:
## (Intercept)       gearIV        gearV
##      16.107        8.427        5.273
```

- str()

```
str(fit, list.len = 3, give.attr = FALSE)
```

```
## List of 13
##  $ coefficients : Named num [1:3] 16.11 8.43 5.27
##  $ residuals    : Named num [1:32] -3.53 -3.53 -1.73 5.29 2.59 ...
##  $ effects      : Named num [1:32] -113.65 -19.47 10.21 5.49 2.79 ...
##   [list output truncated]
```