Good Practices in R Programming

Martin Mächler

maechler@R-project.org
The R Core Team

maechler@stat.math.ethz.ch
Seminar für Statistik
ETH Zurich, Switzerland

useR! – July 1, 2014
Outline

Introduction

Seven Guidelines for Good Practices in R Programming

FAQ 7.31 — generalized: Loss of Accuracy

Specific Hints — to give your friends
Prehistoric – 10 years ago

- May 2004: First UseR! conference in Vienna
- 8 (eight!) keynote talks by R Core members (about exciting new features, such as namespaces)
- R version 1.9.1 a month later in June
This talk is ...

- *not* systematic and comprehensive like a *book* such as
  John Chambers “Programming with Data” (1998),
  Venables + Ripley “S Programming” (2000),
  Uwe Ligges “R Programmierung” (2004) [in German]
  Norm Mattloff’s “The Art of R Programming” (2011)

- *not* for complete newbies

- *not* really for experts either

- *not* about C++ (or C or Fortran or . . .) programming

- *not* always entirely serious 😊
This talk is...

- on R language programming
- my own view, and hence *biased*
- hopefully helping userRs to improve
- ....... somewhat entertaining?
“Good Practices in R Programming”

- “Good”, not “best practice”
- “Programming” using R: use $R$
- “Practice”: What I’ve learned over the years, with examples
What is Programming?

Is Programming
- like driving a car, a skill you learn and then know to do?
- a scientific process to be undertaken with care?
- a creative art?

→ all of them, but not the least an art.
→ Your R ‘programs’ should become works of art . . . 😊

In spite of this,
→ Guidelines (or Rules) for Good Practices in R Programming:
Rule 1: Work with Source files!

R Source files aka ‘R Scripts’ (but more).

- obvious to some,
  not intuitive for useRs used to GUIs.
- **Paradigm** (shift):
  Do not edit *objects* or `fix()` them, but modify (and re-evaluate) their **source**!

In other words (from the ESS manual):

*The source code is real.*

*The objects are realizations of the source code.*
(Rule 1: Work with Source files!)

- Use a *smart* editor or IDE (Interactive Development Environment)
  - syntax-aware: parentheses matching “( .. )” highlighting (differing *fonts* & *colors* syntax dependently)
  - able to *evaluate* R code, by line, whole selection (region), *function*, and the whole file
  - command completion on R objects

such as (available on all platforms):
- Emacs + ESS (*Emacs Speaks Statistics*)
- RStudio
- StatET (R + Eclipse)
- . . . . . and more
Good source code

1. is well readable by humans
2. is as much self-explaining as possible
Rule 2: Keep R source well readable & maintainable

Good, well readable R source code $\rightarrow$ is also well maintainable

1. Do indent lines! (i.e. initial spaces)
2. Do use spaces!
   e.g., around $\leftarrow$, $=$, $<=$, $\ldots$, $+$, $-$, $\ldots$;
   after 
   after 
   before
3. Do wrap long lines!
   (at column 70–80; do not put the editor in fullscreen mode)
well maintainable (*Rule 2 cont.*)

4. Do use comments copiously! (about every 10 lines)
   We recommend
   ‘##’ for the usually indented comments,
   ‘#’ for end-of-line comments, and
   ‘###’ for the (major) “sectioning” or beginning-of-line ones.

5. Sometimes even better (but more laborious): Use Sweave or knitr (or org-mode or another “weave & tangle” system (noweb))

6. E.g., R source in R Markdown (*.Rmd) format.
... well readable code and the assignment operator

Beware: this is very controversial, and I am severely biased! Some (including me, but by far not all!) believe that using \(<\) instead of \(=\) leads to far easier readable code:

- ‘\(=\)’ is also used much in function \(\text{calls}\) (incl. \(\text{list}(a=.., b=..)\)) and definitions (argument defaults)

\(<\) stands out visually
- \(<\) can be marked up (by font/color) quite easily
- something hard to achieve correctly with \(=\) (distinguishing \textit{assignment} from function arguments (both calls \textit{and} formals)

Keyboard shortcut for \(<\) in both Rstudio and ESS (configurable)

\end{really-controversial}
well maintainable (Rule 2 (end))

2 x. Do follow *naming conventions* for function *argument names*, and if available also for new functions and/or classes.

But do *not* impose rigid rules here, since

1. programming is *art* (😊)
2. The S language has a long history with many contributors:
   
   Live with some historical misnomers . . .

2 . . . Modularity, Clarity: “*refine and polish your code*” (V&R): More on “well maintainable” in the following rules
Rule 3: Do read the documentation

and read it again and again ......
(and—only then—submit bug reports 😊)

1. Books (see above), ...
2. The manuals “An Introduction to R” (early),
   “Writing R Extensions” (when you’re mutating from
   useR to programmeR)
3. R package vignettes
4. The help pages! and try their **examples**
5. Use `help.search()` (and read its help page to find out about fuzzy matching and the `agrep` argument!)
Rule 4: Do learn from the masters


Read others’ source — Learning by examples
Obi-Wan Kenobi . . . :
“Use the source, Luke!”

> install.packages("fortunes")
> fortune(250)

As Obi-Wan Kenobi may have said in Star Wars: "Use the source, Luke!"

-- Barry Rowlingson (answering a question on the documentation of some implementation details)
R-devel (January 2010)
Reading Source for '??' → Find Easter egg

> Anybody ? there ???
  ?
  ,

Contacting Delphi...the oracle is unavailable.
We apologize for any inconvenience.
Read the source – of packages

- Note: The R source of an R package (in source state) is inside ⟨pkg⟩/R/*.R, and not what you get when you display the function in R(by typing its name).

- R FAQ 7.40 How do I access the source code for a function?

- Download the source package, ⟨pkg⟩⟨n.m⟩.tar.gz typically from CRAN, unpack it and
  - read it,
  - experiment with it, and
  - learn from it,

- Or browse the package source code on R-forge or github, or ...


Rule 5: Do not Copy & Paste!

because the result is not well maintainable:
Changes in one part do not propagate to the copy!

1. write functions instead
2. break a long function into several smaller ones, if possible
   → write local or (package) global helper functions
   → use many small helper functions (nicely hidden in NAMESPACE).

“Use functions”, e.g., use

1. \textbf{mat}[\texttt{complicated, compcomp}] \leftarrow\
   \textbf{if} (A) \ A.\ expr \ \textbf{else} \ B.\ expr

instead of

1. \textbf{if} (A) \ \textbf{mat}[\texttt{complicated, compcomp}] \leftarrow \ A.\ expr
2. \textbf{else} \ \textbf{mat}[\texttt{complicated, compcomp}] \leftarrow \ B.\ expr
Use Functions

Everything you do in R is calling functions anyway: In R,

Everything that exists is an object;
Everything that happens is a function call.

(John Chambers — this morning, first two of three principles)

Quiz:

When \texttt{if(\*)} \ldots is regarded as function with three arguments, the last being optional with a default, What is the default?

\begin{verbatim}
  if (C) A
  if (C) A else B
\end{verbatim}

Answer: NULL: \texttt{if(FALSE) A} returns NULL invisibly
Rule 6: Strive for clarity and simplicity

first! ... and second ... and again, e.g.,
think about naming of intermediate results with “self-explainable” variable names
but use short names (plus comments) for formulae

Venables & Ripley:
“Refine and polish your code in the same way you would polish your English prose”

(prose: using as “dictionary” your reference material)
→ modularity (“granularity”)

Optimization: much much later, see below
Rule 7: **Test** your code!

1. Carefully write (small) testing examples, for each function ("modularity", "unit testing")
   
2. Next step: Start a 'package' via `package.skeleton()`. This allows (via `R CMD check`)
   
   ▶ auto-testing (all the help pages examples).
   
   use `example(your_function)`

   ▶ specific testing (in a `.tests/` subdirectory, with or without strict comparison to previous results)

   ▶ documenting your functions (and data, classes, methods):
   
   takes time, but almost always leads you to improve your code!
Test your code! (*Rule 7 cont.*)

3. Use software tools for testing:
   - Those of `R CMD check` are in the standard R package `tools`, and `codetools` (by Luke Tierney)
   - Unit testing by packages, RUnit, testthat, etc.
After Testing, **maybe** Optimizing

Citing from V&R’s “S Programming” (p.172):

Jackson “Principles of Program Design” (on ‘code optimization’):

- **Rule 1** Don’t do it.
- **Rule 2** (for experts only) Don’t do it yet—not until you have a perfectly clear and unoptimized solution. ‘to the right problem by an efficient method’.

*Premature optimization is the root of all evil* – Donald Knuth
1. Really do clean up and test your code and think twice before you even start contemplating optimizing the code . . .

2. do **measure**, not guess:

   *In 2001, when R was at version 1.1.x,*

   From: Thomas Lumley (tlumley@u.washington.edu)
   
   To : R-help

   There are two fundamental principles of optimisation
   1) Don’t do it unless you need it
   2) Measure, don’t guess, about speed.

   The simple way to answer questions about which way is slower/more memory intensive is to try it and see. *Between Rprof(), unix.time() and gc(), you have all the information you need. ...........

   In 2014: Have packages rbenchmark, microbenchmark, pbdPROF, and more.
Seven Guidelines ("Rules") – still relevant

1. Work with Source files
2. Keep R source code well readable and maintainable
3. Do read the documentation
4. Do learn from the masters — Read R (package) sources
5. Do not Copy & Paste! — Modularize into (small) Functions
6. Strive for clarity and simplicity
7. Test your code — and test, and test!
New Guidelines:

8. Maintain R code in Packages (extension of “Test!”)
9. Source code management, e.g., subversion(svn) or github(git)
10. Rscript or R CMD BATCH ⟨mysource⟩.R should “always” work!

→ Reproducible Data Analysis and Research
  ▶ Do not use .RData no, really, not ever! . . .
  ▶ Rather, use save() explicitly only for expensive parts.
  ▶ Consider attach("myStuff.rda") instead of load("myStuff.rda")
  ▶ Use the following outline:

```r
savefile ← "<myThings>.rda"
2  if(file.exists(savefile)) attach(savefile) else
6    save(o1, o2, ..., o.n, file = savefile)
```
Why doesn’t R think these numbers are equal?

The only numbers that can be represented exactly in R’s numeric type are integers and fractions whose denominator is a power of 2. Other numbers have to be rounded to (typically) 53 binary digits accuracy. As a result, two floating point numbers will not reliably be equal unless they have been computed by the same algorithm, and not always even then. For example

```r
> a <- sqrt(2)
> a * a == 2  # mathematically, yes, ...
[1] FALSE
> a * a - 2
[1] 4.440892e-16
```

Why doesn’t R think these numbers are equal?

To quote from “The Elements of Programming Style” by Kernighan and Plauger:

10.0 times 0.1 is hardly ever 1.0.

Actually, it is in R, (always / typically (?)), nowadays.
FAQ 7.31 ++ : The ”log” in the dpq-functions

All “dpq” distribution functions in R, i.e. density cumulative probability and quantile functions, have a log or log.p argument (FALSE / TRUE).

Why ?

→ Compute Likelihoods via d<foo>(*, log = TRUE)
→ Probalistic Networks, MC(MC): $P = P_1 \cdot P_2 \cdots \cdots P_n$ quickly underflows to zero.

Solution: Work in “log space”: $\log P = \sum_j \log P_j$, where $\log P_j$ are computed via R’s d<foo>(* , log=TRUE) or p<foo>(* , log.p=TRUE), rather than taking logs
FAQ 7.31 ... Why R needs even more functions

1. log1p() (since R 1.0.0), expm1() (since R 1.5.0)
Why \( \log(1 + x) \) is not good enough, but \( \log1p(x) \) is

\( 1 + x \) cannot be numerically accurate when \( |x| \ll 1 \). In double precision (53 bits \( \approx 16 \) digits) accuracy, \( 1 + x \) “sees” only 2–3 digits of \( x \) when \( x = 10^{-14} \),

\[
> u <- 1 + (e <- 4e-13/9) \quad \text{## then} \quad u - 1 == e \quad \text{mathematically}:
\]

\[
> \text{rbind('u-1' = u - 1, e)}
\]

\[
[,1]
\]

\begin{tabular}{l l}
  u-1 & 4.440892e-14 \\
e   & 4.444444e-14 \\
\end{tabular}

And the consequence for \( \log(1 + x) \),

\[
> \text{curve(abs(1 - log(1+x) / log1p(x)), 1e-17, .2, log = 'xy', main = '..', eaxis(1); eaxis(2)}
\]

\begin{center}
| relative error of \( \log(1+x) \) |
\end{center}
Why \( \log 1p(x) \) beats \( \log(1 + x) \)

Solution: Expand \( \log(1 + x) \) around \( x = 0 \). Well known

\[
\log(1 + x) = x - x^2/2 + x^3/3 \pm \ldots = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{x^n}{n},
\]

for \( |x| < 1 \).

Fast version of this expansion: typically used in \( \log 1p() \).
FAQ 7.31 ... Why R needs even more functions –2–

2. \texttt{cospi()}, \texttt{sinpi()}, \texttt{tanpi()} (from R 3.2.0), e.g.,
\texttt{cospi(x)} := \cos(\pi \cdot x), accurately, e.g., for \( x = \frac{1}{2} \):

\begin{verbatim}
> cos(pi/2) ## mathematically == 0
[1] 6.123234e-17
> cospi(1/2)
[1] 0
\end{verbatim}

3. \texttt{log1mexp()} ...(my research; in R's Rmathlib C code, named differ.)
Simple (semi-artificial!) Example: logit(exp(-L))

Logistic regression: Computing “logit()”s, $\log \frac{p}{1-p}$ accurately for very small $p$, i.e., $p = \exp(-L)$, or

$$
\log \frac{p}{1-p} = \log p - \log(1 - p) = -L - \log(1 - \exp(-L)),
$$

and hence $-\log(1 - \exp(-L))$ is needed, e.g., when $p$ is really really close to 0, say $p = 10^{-1000}$, as then we can only compute logit$(p)$, if we specify $L := -\log(p) \leftrightarrow p = \exp(-L)$.

> curve(-log(1 - exp(-x)), 0, 10)

seems fine. — — However, ...
However, further out to 50 (and on a log scale), we observe

which shows early underflow.
What did happen? Look at

```r
> x <- -40:-35
> -log(1 - exp(x))

[1] 0.000000e+00 0.000000e+00 0.000000e+00 1.110223e-16 2.220446e-16

> log(-log(1 - exp(x)))# --> -Inf values

[1] -Inf -Inf -Inf -36.73680 -36.04365 -34.94504

> ## ok, how about more accuracy
> x. <- mpfr(x, 120)
> log(-log(1 - exp(x.)))# aha... looks perfect now

6 'mpfr' numbers of precision 120 bits

[2] -38.99999999999999999423372196756935807
[3] -37.99999999999999998430451715981029611
[5] -35.99999999999999999884024061830552087239
[6] -34.9999999999999999684744214015307532692
```
Visually, and with "high accuracy" mpfr-numbers:
> x <- seq(-40, -20, by = .5)
> plot(x, x, type="n", ylab="", ann=FALSE)
> lines(x, log(-log(1 - exp(x))), type = "o", col = "purple", lwd=3, cex = .6)
> x. <- mpfr(x, 120)
> lines(x, log(-log(1 - exp(x.))), col=2, lwd=1.5)

The "real" solution uses a piecewise implementation of \( \log_1(-\exp(-x)) \) for \( x > 0 \), see e.g., R package copula.
Specific Hints, Tips:

1. **Subsetting ("[.."]"):**
   - 1.1 Matrices, arrays (& data.frames):
     Instead of `x[ind ,]`, use `x[ind, , drop = FALSE]`!
   - 1.2 tricky because of NAs
     Inside "[..]", often use `%in%` (wrapper of `match()`) or `which()`.

2. Not `x == NA` but `is.na(x)`

3. Use '1:n' only when you *know* that n is positive:
   Instead of `1:length(obj)`, use `seq_along(obj)`
Specific Hints – 2:

4. Do not *grow* objects:
   If you cannot avoid a *for* loop, replace

   ```r
   rmat <- NULL
   for(i in 1:n) {
       rmat <- rbind(rmat, long.computation(i, ....))
   }
   ```

   by

   ```r
   rmat <- matrix(0., n, k)
   for(i in 1:n) {
       rmat[i, ] <- long.computation(i, ....)
   }
   ```

   and almost always, *column* by *column* instead of *row* by *row* (creating the *transpose*):

   ```r
   tmat <- matrix(0., k, n)
   for(i in 1:n) {
       tmat[, i ] <- long.computation(i, ....)
   }
   ```
Specific Hints, Tips (cont.)

5. Use `lapply()`, `sapply()`; sometimes preferably `vapply()` or `mapply()` (Apply to multiple arguments), or sometimes the `replicate()` wrapper:
   
   ```r
   sample <- replicate(1000, median(rt(100, df = 3)))
   hist(sample)
   ```

6. Use `with(<d.frame>, .......)` and do not attach data frames.

7. Use `TRUE` and `FALSE`, not `T` and `F`!

8. Know the difference between `|` vs `||` and `&` vs `&&` and inside `if( .... )` almost always use `||` and `&&`!

9. Use `which.max()`, ..., `findInterval()`

10. Learn about ‘Regular Expressions’: `?regexp` etc
What *Style* is your R programming?
*Perform the art, enjoy and be productive!*

Martin.Maechler@R-project.org