How to Write an R Package

Martin Mächler
maechler@R-project.org

Seminar für Statistik, ETH Zürich
(and · ∈ {R Core Team} since 1995)

Zurich, “R Programming”, 20./21. Oct. 2016

- The following slides are (“only”) an *Introduction* to R packages.

Additionally, we will work with

- **The** “reference” : the *“Writing R Extensions”* manual[1].
  We will get an overview and consider some sections in detail.
- *Name Space Management for R*, by Luke Tierney, R News June 2003 (5 pages)
- `package.skeleton()` to get started
- Look at many examples, including your own ones.

[1] part of R (as HTML), as PDF also available from CRAN

1. Packages in R - Why and How - Overview

1.1 Why Packaging R ?

R packages provide a way to manage collections of functions or data and their documentation.

- Dynamically loaded and unloaded: the package only occupies memory when it is being used.
- Easily installed and updated: the functions, data and documentation are all installed in the correct places by a single command that can be executed either inside or outside R .
- Customizable by users or administrators: in addition to a site-wide *library*, users can have one or more private libraries of packages.
- Validated: R has commands to check that documentation exists, to spot common errors, and to check that examples actually run

1.1 Why Packaging R ? — (2)

- Most users first see the packages of functions distributed with R or from *CRAN*. The package system allows many more people to contribute to R while still enforcing some standards.
- **Data** packages are useful for teaching: datasets can be made available together with documentation and examples. For example, Doug Bates translated data sets and analysis exercises from an engineering statistics textbook into the `Devore5` package
- Private packages are useful to organise and store frequently used functions or data. One R author has packaged ICD9 codes, for example.

1.2 Structure of R packages

The basic structure of package is a *directory* (aka "folder"), commonly containing

- A `DESCRIPTION` file with descriptions of the package, author, and license conditions in a structured text format that is readable by computers and by people
- A `man/` subdirectory of documentation files
- An `R/` subdirectory of R code
- A `data/` subdirectory of datasets
- A `src/` subdirectory of *C*, *Fortran* or *C++* source

1.2 Structure of R packages — (cont)

Less commonly it contains

- `inst/` for miscellaneous other stuff, notably *package "vignettes"*
- `tests/` for validation tests
- `demo/` for `demo()`-callable demonstrations
- `po/` for message translation "lists" (from English, almost always) to other languages.
- `exec/` for other executables (eg Perl or Java)
- A `configure` script to check for other required software or handle differences between systems.

Apart from `DESCRIPTION` these are all optional, though any useful package will have `man/` and at least one of `R/` and `data/`.
Everything about packages is described in more detail in the *Writing R Extensions* manual distributed with R .

Data formats

The `data()` command loads datasets from packages. These can be

- Rectangular text files, either whitespace or comma-separated
- S source code, produced by the `dump()` function in R or S-PLUS.
- R binary files produced by the `save()` function.

The file type is chosen automatically, based on the file extension.

Documentation - Help files

```
> help(pbirthday, help_type = "pdf")
```

produces a *nice* pdf version of what you typically get by `?pbirthday`.
The R documentation format looks rather like LaTeX.

```
\name{birthday}  % name of the file
\alias{qbirthday} % the functions it documents
\alias{pbirthday}
\title{Probability of coincidences}% <== one-line title of
\description{% short description:
 Computes answers to a generalised \emph{birthday paradox}
  \code{pbirthday} computes the probability of a coincidenc
  \code{qbirthday} computes the smallest number of observat
  to have at least a specified probability of coincidence.
}
\usage{ % how to invoke the function
qbirthday(prob = 0.5, classes = 365, coincident = 2)
pbirthday(n, classes = 365, coincident = 2)
}
........
```


Documentation (2)

The file continues with sections

- `\arguments`, listing the arguments and their meaning
- `\value`, describing the returned value
- `\details`, a longer description of the function, if necessary.
- `\references`, giving places to look for detailed information
- `\seealso`, with links to related documentation
- `\examples`, with ***directly executable*** examples of how to use the functions.
- `\keyword` for indexing

There are other possible sections, and ways of specifying equations, urls, links to other R documentation, and more.

Documentation (3)

The documentation files can be converted into HTML, plain text, and (via LaTeX) PDF.
The packaging system can check that all objects are documented, that the `usage` corresponds to the actual definition of the function, and that the `examples` will run. This enforces a minimal level of accuracy on the documentation.

- Emacs (ESS) supports editing of R documentation (as does Rstudio and StatET).
- function `prompt()` and its siblings for producing such pages:

```
> apropos("^prompt")
[1] "prompt"         "promptClass"    "promptData"     "promptImpor
[5] "promptMethods" "promptPackage"
```

NB: The `prompt*()` functions are called from `package.skeleton()`

1.3 Setting up a package

The `package.skeleton()` function partly automates setting up a package with the correct structure and documentation.
The `usage` section from `help(package.skeleton)` looks like

```
package.skeleton(name = "anRpackage", list = character(),
    environment = .GlobalEnv, path = ".", force = FALSE,
    namespace = TRUE, code_files = character())
```

Given a collection of R objects (data or functions) specified by a `list` of names or an `environment`, or nowadays typically rather by a few `code_files` ("*.R - files"), it creates a package called *name* in the directory specified by `path`.
The objects are sorted into data (put in `data/`) or functions (`R/`), skeleton help files are created for them using `prompt()` and a `DESCRIPTION` file, and from R 2.14.0 on, always a `NAMESPACE` file is created. The function then prints out a list of things for you to do next.

1.4 Building a package

`R CMD build` (`Rcmd build` on Windows) will create a compressed package file from your (source) package directory, also called "tarball".

It does this in a reasonably intelligent way, omitting object code, emacs backup files, and other junk. The resulting file is easy to transport across systems and can be `INSTALL`ed without decompressing.

All help, R, and data files now are stored in "data bases", in compressed form. This is particularly useful on older Windows systems where packages with many small files waste a lot of disk space.

Binary and source packages

`CMD build` makes source packages (by default). If you want to distribute a package that contains C or Fortran for Windows users, they may well need a binary package, as compiling under Windows requires downloading exactly the right versions of quite a number of tools.
Binary packages are created by `R CMD INSTALL`ing with the extra option `--build`. This produces a `<pkg>.zip` file which is basically a zip archive of `R CMD INSTALL`ing the package.
(In earlier R versions, binary packages were created by `R CMD build`ing with the extra option `--binary`. This may still work, but do not get into the habit!)

1.5 Checking a package

`R CMD check` (`Rcmd check` in Windows) helps you do QA/QC[2] on packages.

- The directory structure and the format of `DESCRIPTION` (and possibly some sub-directories) are checked.
- The documentation is converted into text, HTML, and LaTeX, and run through `pdflatex` if available.
- The examples are run
- Any tests in the `tests/` subdirectory are run (and possibly compared with previously saved results)
- Undocumented objects, and those whose `usage` and definition disagree are reported.
- .....
- (the current enumeration list in "Writing R Extensions" goes up to number **21** !!)

[2] QA := Quality Assurance; QC := Quality Control

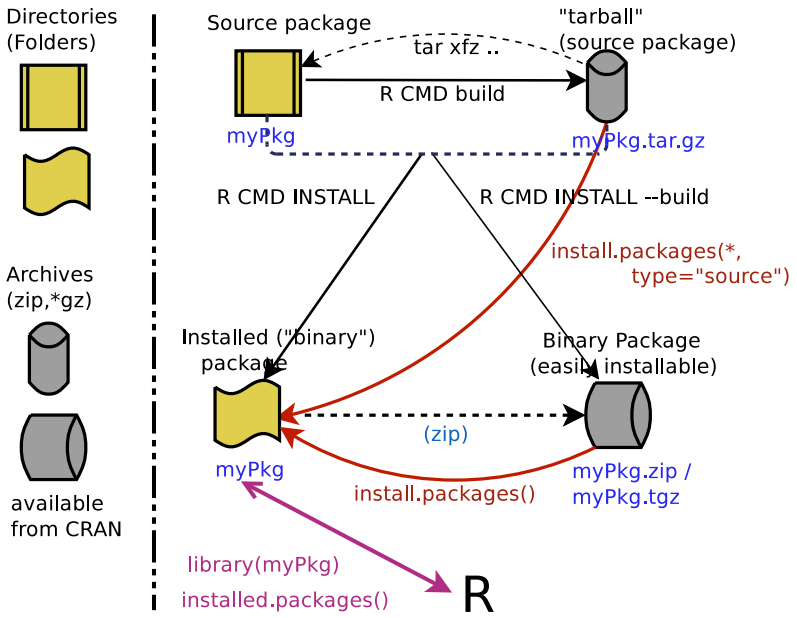
1.6 Distributing packages

If you have a package that does something useful and is well-tested and documented, you might want other people to use it too. Contributed packages have been very important to the success of R (and before that of S).
Packages can be submitted to *CRAN*

- The *CRAN* maintainers will make sure that the package passes `CMD check` (and will keep improving `CMD check` to find more things for you to fix in future versions :-)).
- Other users will complain if it doesn't work on more esoteric systems and no-one will tell you how helpful it has been.
- But it will be appreciated. Really.

2. What Packages in R and How?

2.1 The many "kinds" of R packages:



2.2 Packages: Where you get your R objects from

- In R, by default you "see" only a basic set of functions, e.g., `c`, `read.table`, `mean`, `plot`, ..., ...
- They are found in your "search path" of packages

```
> search() # the first is "your workspace"
[1] ".GlobalEnv"        "package:graphics"  "package:grDevices
[4] "package:datasets"  "package:stats"     "package:utils"
[7] "package:methods"   "Autoloads"         "package:base"
> ls(pos=1) # == ls()  ~= "your workspace" - learned in
[1] "Mlibrary" "pkg"      "tpkgs"
> str(ls(pos=2)) # content of the  2nd  search() entry
 chr [1:87] "abline" "arrows" "assocplot" "axis" "Axis" ...
> str(ls(pos=9)) # content of the  9th  search() entry
 chr [1:1216] "-" "-.Date" "-.POSIXt" ":" "::" ":::" "!" ...
```


- The default list of R objects (functions, some data sets) is actually not so small: Let's call `ls()` on each `search()` entry:

```
> ls.srch <- sapply(grep("^package:", search(), value =
+                        # "package:<name>" entries
+                   ls, all.names = TRUE)
> fn.srch <- sapply(ls.srch, function(nm) {
+     nm[ sapply(lapply(nm, get), is.function) ] })
> rbind(cbind(ls    = (N1 <- sapply(ls.srch, length)),
+             funs = (N2 <- sapply(fn.srch, length))),
+       TOTAL = c(sum(N1), sum(N2)))
                    ls funs
package:graphics    88   88
package:grDevices  108  105
package:datasets   104    0
package:stats      453  452
package:utils      209  206
package:methods    381  226
package:base      1321 1278
TOTAL             2664 2355
```

i.e., 2355 functions in R version 3.3.1

- Till now, we have used functions from packages "base", "stats", "utils", "graphics", and "grDevices" without a need to be aware of that.
- `find("⟨name⟩")` can be used:

```
> c(find("print"), find("find"))
[1] "package:base"  "package:utils"
> ## sophisticated version of  rbind(find("mean"), find
> cbind(sapply(c("mean", "quantile", "read.csv", "plot"
+              find))
         [,1]
mean     "package:base"
quantile "package:stats"
read.csv "package:utils"
plot     "package:graphics"
```


- R already comes with $14 + 15 = 29$ packages pre-installed, namely the "standard (or "base") packages

```
base, compiler, datasets, graphics, grDevices, grid,
methods, parallel, splines, stats, stats4, tcltk, tools,
utils
```

and the "recommended" packages

```
boot, class, cluster, codetools, foreign, KernSmooth,
lattice, MASS, Matrix, mgcv, nlme, nnet, rpart, spatial,
survival
```


- Additional functions (and datasets) are obtained by (possibly first *installing* and then) loading additional "packages".
- `> library(MASS)` or `require(MASS)`
- How to find a command and the corresponding package?
  `> help.search("...")` [3], (see Intro)
- On the internet: CRAN (http://cran.r-project.org, see ‣ Resources on the internet (slide 15) is a huge repository[4] of R packages, written by many experts.
- More search possibilities
  http://www.r-project.org/search.html (*before* using Google!)
- CRAN Task Views help find packages by application area
- What does a package do?
  `> help(package = class)` or (⟷)
  `> library(help = class)` .
  Example (of small recommended) package:

```
> help(package = class)
```

[3] can take l..o..n..g.. (only the first time it's called in an R session !)
[4] actually a distributed Network with a server and many mirrors,


```
> help(package = class)
                Information für Paket 'class'

Description:

Package:            class
Priority:           recommended
Version:            7.3-3
Date:               2010-12-06
Depends:            R (>= 2.5.0), stats, utils
Imports:            MASS
Author:             Brian Ripley <ripley@stats.ox.ac.uk>.
Maintainer:         Brian Ripley <ripley@stats.ox.ac.uk>
Description:        Various functions for classification.
Title:              Functions for Classification
License:            GPL-2 | GPL-3
URL:                http://www.stats.ox.ac.uk/pub/MASS4/
LazyLoad:           yes
Packaged:           2010-12-06 11:46:04 UTC; ripley
Repository:         CRAN
Date/Publication:   2010-12-09 11:56:32
Built:              R 2.12.0; x86_64-unknown-linux-gnu; 2010-12-1(
```

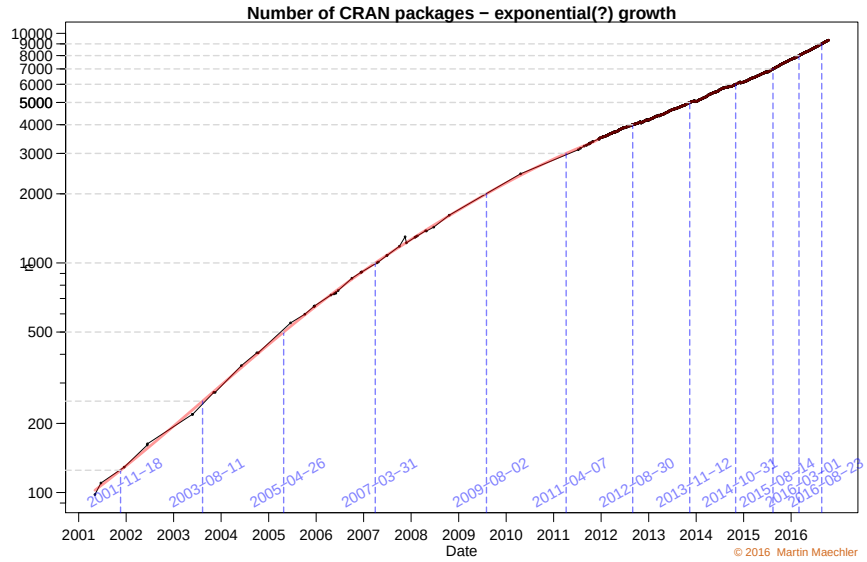
```
Index:

SOM                     Self-Organizing Maps: Online Algorithm
batchSOM                Self-Organizing Maps: Batch Algorithm
condense                Condense training set for k-NN classifier
knn                     k-Nearest Neighbour Classification
knn.cv                  k-Nearest Neighbour Cross-Validatory
                        Classification
knn1                    1-nearest neighbour classification
lvq1                    Learning Vector Quantization 1
lvq2                    Learning Vector Quantization 2.1
lvq3                    Learning Vector Quantization 3
lvqinit                 Initialize a LVQ Codebook
lvqtest                 Classify Test Set from LVQ Codebook
multiedit               Multiedit for k-NN Classifier
olvq1                   Optimized Learning Vector Quantization 1
reduce.nn               Reduce Training Set for a k-NN Classifier
somgrid                 Plot SOM Fits
```


3. CRAN - Where to Get and Put Packages

Intermezzo: Browse CRAN

Number of CRAN (source) packages: *Exponential* growth for about 15 years; number 9000 hit on August 23, 2016



Browse CRAN — CRAN Task Views

- allow to browse packages *by topic*
- tools to automatically *install* all packages for areas of interest.
- Currently, 34 views are available:

```
> require("ctv")
> av <- available.views()
> unname(abbreviate( ## <<- compacter for the slide
+     sapply(av, `[[`, "name"),    min = 19, dot=TRUE))
 [1] "Bayesian"             "ChemPhys"             "ClinicalTrials"
 [4] "Cluster"              "DifferentialEquatns." "Distributions"
 [7] "Econometrics"         "Environmetrics"       "ExperimentalDesign"
[10] "ExtremeValueTheory"   "Finance"              "Genetics"
[13] "Graphics"             "HighPerformncCmptng." "MachineLearning"
[16] "MedicalImaging"       "MetaAnalysis"         "Multivariate"
[19] "NaturalLanggPrcssng." "NumericalMathematcs." "OfficialStatistics"
[22] "Optimization"         "Pharmacokinetics"     "Phylogenetics"
[25] "Psychometrics"        "ReproducibleReserch." "Robust"
[28] "SocialSciences"       "Spatial"              "SpatioTemporal"
[31] "Survival"             "TimeSeries"           "WebTechnologies"
[34] "gR"
```

Browse CRAN

Many CRAN mirrors; "of course" we use the Swiss mirror (= http://stat.ethz.ch/CRAN):

- The CRAN Task Views web page: https://stat.ethz.ch/CRAN/web/views/
- Package *developers* may like — or hate — https://stat.ethz.ch/CRAN/web/checks/check_summary.html
- Other "summaries": "Metacran"(= http://www.r-pkg.org/), http://Crantastic.org, "MRAN" from Revolution, ...

Installing packages from CRAN

- Via the "Packages" menu (in GUIs for R such as RStudio)
- Directly via `install.packages()`[5].
  Syntax:

  `install.packages(`*pkgs*`, `*lib*`, `*repos* = *getOption*("*repos*"), ...)`

  pkgs: character vector names of packages whose current versions should be downloaded from the repositories.
  lib: character vector giving the library directories where to install the packages. If missing, defaults to `.libPaths()[1]`.
  repos: character with base URL(s) of the repositories to use, typically from a CRAN mirror. You can choose it interactively via `chooseCRANmirror()` or explicitly by `options(repos= c(CRAN="http://..."))`.
  ...: many more (*optional*) arguments.

[5] which is called anyway from the menu functions

Installing packages – Examples

- Install once, then use it via `require()` or `library)`:

```
> chooseCRANmirror()
> install.packages("sfsmisc")
> ## For use:
> require(sfsmisc) # to ``load and attach'' it
```

-
```
> install.packages("sp", # using default 'lib'
+         repos = "http://cran.CH.r-project.org")
```

- or into a non-default *library* of packages

```
> install.packages("sp", lib = "my_R_folder/library",
+         repos = "http://cran.CH.r-project.org")
> ## and now load it from that library (location):
> library(sp, lib = "my_R_folder/library")
```

- Note: If `lib` is not a writable directory, R offers to create a personal library tree (the first element of `Sys.getenv("R_LIBS_USER")`) and install there.

Finding functionality in CRAN packages

...instead of re-inventing the wheel

- `help.search(foo)` (⟷ `??foo`[6], or "Search" in R-`help.start()` Web browser, finds things in all *installed* packages
- `RSiteSearch()` searches search.r-project.org
- R Project → search mentions the above, and more, including http://www.rseek.org
- CRAN Task Views (see above)
- R-forge - for R package developers https://r-forge.r-project.org also has search functionality
- "Metacran"(= http://www.r-pkg.org/)
- ...Google

[6] → very nice in ESS!

Not re-inventing the wheel . . .

- Asking on R-help, the “general R” mailing list,
  or on `R-package-devel` devoted to help package writing *and checking* problems, see
  `https://www.r-project.org/mail.html` .
  $\longrightarrow$ many readers are helpful, and some are experts :-)
- “Stack Overflow”, “tagged ‘r’”:
  `http://stackoverflow.com/questions/tagged/r` (notably for precise technical questions)
- The R-devel mailing list if you are really advanced