R Packaging and New Development Features for Building R Packages

author: Matthias Templ date: Olomouc 2015

Packaging: Overview Approaches

- Basically two approaches:
  1. traditional approach including a lot of manual work
  2. new approach using **roxygen2**, package **devtools** and **RStudio**

We concentrate on the second approach since it is just easy with it!

But first some basic concepts about R packages are given...

What is an R package?

- Packages are standardized units for extending R
- Transparent and cross-platform extension base R
- The R distribution itself contains 30 packages.
- Packages must provide a min. of information to the core R system:
  - name and version;
  - license, description, title,
  - author and maintainer.
- A package must be installed, using for example the R command **install.packages()**.
- Before using a package, load it with the **library()** or **require()** command.

Why R packages? (1/2)

- Accessible functions and data
  - Convenient means for code storage and version control
  - Functions, data and other objects can be easily made available for use (loaded) by a single **library(myPackage)** command
  - Facilitates access to native code (C/C++/FORTRAN)
  - Sharing code with others
  - Using a package makes sense even for personal use

Why R packages? (2/2)

- Reliable and maintainable code
  - Facilitates for code development (more disciplined software development),

  - particularly in collaborative projects
  - Better design of the functions
  - Less bugs and easier to fix them
  - More reliable code
  - Maintainable code

Basic terms related to R packages (1/2)

- **Package**: A set of code, example data and documentation in a standard form extending R
- **Library**: Directory containing installed packages
- **Repository:** A formalized web site providing packages for installation
- **Source**: The source version of the package containing the R source code, data, documentation and other components in its original form
- **Binary**: A compiled version of the package suitable for use only on a particular platform (e.g. Windows, Mac OS)

Basic terms related to R packages (2/2)

- **Base package**s: Packages maintained by the R core development team, distributed and installed as a part of the R software
- **Recommended packages**: Packages distributed with the main R software but not necessarily maintained by the R core development team
- **Contributed packages**: All other packages—most of them can be downloaded and installed from the CRAN repository.

CRAN - 6346 add-on packages

CRAN - top 10

Top 100 Pakete von Jan.-Dez 2013: http://bit.ly/JxgNXD

CRAN: How often are my packages downloaded?

Using R packages (1/3)

- Which packages are currently loaded? Search path: use the function **search()**

- What packages are currently installed?
    - **library()** without arguments
    - **installed.packages()** returns a data frame, a row per package.
- Information about a package, e.g. for package MASS
    - **packageDescription("MASS")**
    - **help(package="MASS")**

Using R packages (2/3)

- Load package / use the functions in a package:
    - **library(packagename)** or
    - **require(packagename)**
- List the available packages in a repository:
    - **available.packages()**
- Installing and updating packages:
    - **install.packages("packagename")**
    - **old.packages()**
    - **update.packages()**
- Package vignettes: function **vignette()** to list all available vignettes or to view a vignette.

Using R packages (3/3)

How to find packages? - Ask Google, but do not expect a precise answer. - Ask a question at R-Help or - better - ask at Stack Overflow. - Go to CRAN Task Views, see http://http://cran.r-project.org/web/views/

Use the R package sos. For example try

```
library(sos)
findFn("robust+multivariate")
```

The results will be shown in the web browser.

Useful functions

- Save an R object to a file in binary R format

```
save(..., file="filename.rda")
```

Find the R working directory

```
getwd()
```

```
[1] "/Users/templ/workspace/V12-packaging"
```

With traditional approach: to generate a help (.Rd) file

```
prompt(object)
```

Structure of a package

A package is a directory with a given subdirectory structure. - A **DESCRIPTION** file containing the metadata of the package; Debian Control File format. - A **NAMESPACE** file. - A **man/** subdirectory containing the documentation files. - An **R/** subdirectory containing the R-code. - A **data/** subdirectory containing data sets.

Structure of a package (optional)

Further optional subdirectories could be: - A **src/** subdirectory containing C/C++/FORTRAN code - A **tests/** subdirectory containing validation tests - A **exec/** subdirectory containing other executables, like Perl or Java - A **vignettes** subdirectory containing package vignettes - A **inst/** subdirectory containing other stuff. - Files **NEWS** and ChangeLog - ignored by R but could be helpful for the user.

Creating R packages - the traditional approach

- Step 1: Create the package files.
  - Load all R source code and data set(s) into a clean session and
  - Run **package.skeleton("packagename")**
  - Alternative: create the directory structure yourself (DESCRIPTION, NAMESPACE, ...)
- Step 2: Edit the package files.
  - Fill in the DESCRIPTION file
  - Complete documentation files in *man/*
  - Edit the NAMESPACE file (def: export everything)
- Step 3: Build, check and install the package.

Creating R packages - the traditional approach

- ./mypackage/Read-and-delete-me contains information how to continue:
  - Edit the help file skeletons in *man*, possibly combining help files for multiple functions.
  - Edit the exports in *NAMESPACE*, and add necessary imports.

    - Put any C/C++/Fortran code in **src**
    - If you have compiled code, add a useDynLib() directive to **NAMESPACE**.
    - Run **R CMD build to build** the package tarball.
    - Run **R CMD check** to check the package tarball.
- Read "Writing R Extensions" for more information.

The DESCRIPTION file

The content of the default **DESCRIPTION** file looks like this:

```
Package: pcapack
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2013-09-15
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What license is it under?
Update the information
Choose license: GPL-2
Add dependencies with Depends: directive
```

Step 2: The DESCRIPTION file

- Update the information
- Choose license: GPL-2 or MIT
- Add dependencies with *Depends*

Step 2: The NAMESPACE file

- The **NAMESPACE** file describes which functions in the package are visible to the others.
- The content of the default **NAMESPACE** file looks like that (everything is exported):

```
exportPattern("^[[:alpha:]]+")
```

Step 2: an Example of a NAMESPACE File

```
useDynLib(rrcov)
importFrom(stats4, plot, summary)
importFrom(stats, screeplot, biplot, predict)
importFrom(methods, show)
importFrom(mvtnorm, rmvnorm)
import(robustbase, pcaPP)
```

```
export(CovClassic, Cov, ..., repmat)
##S3 generics
S3method(T2.test, default)
S3method(T2.test, formula)
S3method(Wilks.test, default)
S3method(Wilks.test, formula)
## S4 generics
export(isClassic, ... )
```

Add data

Any time a new data object, say moredata, can be added to a package: - Load the data into R (from CSV, Excel, a database, Web, etc.) - Save the data into a binary R object (into *data* folder): * **save(moredata, file="moredata.rda")**

(- For traditional approach: Create a help file using **prompt(moredata)** and copy the **.rda** and **.Rd** files into the **data/** and **man/** directories respectively.)

Step 2: Add a function

Any time a new function, say newfun, can be added to a package: - We assume that we have already written the code of the function in and **.R** file, say **newfun.R** - Save it in the *R* folder and include documentation

(note: traditional approach takes much more time)

Step 3: Install, check and build on Windows

Under Linux all tools are available, for Windows: - RTools: (REQUIRED) Install from http://cran.r-project.org/bin/windows/Rtools/. RTools is a collection of unix-like tools that can be run from the DOS command prompt; contains the compilers used for compiling Fortran and C code. - $LaTeX$-compiler: (OPTIONAL) Install e.g. Miktex from miktex.org - necessary for building the PDF manual during the checking of the package. - set the PATH variable - http://robjhyndman.com/hyndsight/ building-r-packages-for-windows/

Step 3: Install, check and build

To Install, Check and Build a package the following commands are used: * **R CMD command packagename**

where * **R CMD INSTALL packagename** will install the package from its folder * **R CMD build packagename** will build a source package (tarball or .tar.gz) * **R CMD check packagename** will check the package for consistency

A Note on R CMD check ...

Checks the package for consistency; mandatory for submission to CRAN - Check directory structure and DESCRIPTION file - Documentation is converted and run through $LaTeX$ (if available) - The examples are run - The tests (if available) are run - Undocumented objects or inconsistency between documentation and code are reported

Example Check I

```
R CMD check pcapack
* using R Under development (unstable) (2013-08-19 r63623)
* using platform: i386-w64-mingw32 (32-bit)
* using session charset: ISO8859-1
* checking for file 'pcapack/DESCRIPTION' ... OK
* checking extension type ... Package
* this is package 'pcapack' version '1.0'
* checking package namespace information ... OK
* checking package dependencies ... OK
* checking if this is a source package ... OK
* checking if there is a namespace ... OK
* checking for executable files ... OK
* checking for hidden files and directories ... OK
* checking for portable file names ... OK
* checking whether package 'pcapack' can be installed ... OK
* checking installed package size ... OK
```

Example Check II

```
* checking package directory ... OK
* checking DESCRIPTION meta-information ... OK
* checking top-level files ... OK
* checking for left-over files ... OK
* checking index information ... OK
* checking package subdirectories ... OK
* checking R files for non-ASCII characters ... OK
* checking R files for syntax errors ... OK
* checking whether the package can be loaded ... OK
* checking whether the package can be loaded with stated dependencies ... OK
* checking whether the package can be unloaded cleanly ... OK
* checking whether the namespace can be loaded with stated dependencies ... OK
* checking whether the namespace can be unloaded cleanly ... OK
* checking dependencies in R code ... OK
```

Example Check III

```
* checking S3 generic/method consistency ... OK
* checking replacement functions ... OK
* checking foreign function calls ... OK
* checking R code for possible problems ... OK
* checking Rd files ... OK
* checking Rd metadata ... OK
* checking Rd cross-references ... OK
* checking for missing documentation entries ... OK
* checking for code/documentation mismatches ... OK
* checking Rd \usage sections ... OK
* checking Rd contents ... OK
* checking for unstated dependencies in examples ... OK
* checking examples ... OK
* checking PDF version of manual ... OK
```

A Note on R CMD build ...

R will create a compressed package file (omitting unnecessary files).

```
R CMD build pcapack
* checking for file pcapack/DESCRIPTION ... OK
* preparing pcapack:
* checking DESCRIPTION meta-information ... OK
* checking for LF line-endings in source and make files
* checking for empty or unneeded directories
* building pcapack_1.0.tar.gz
```

Including native compiled code

Including C/FORTRAN/C++ code in a package. - There are many resources on the web, but the definitive guide is Writing R extensions - Store the C/C++/FORTRAN code into the src/ directory - Update the NAMESPACE file - Use the argument PACKAGE in the call to .C or .FORTRAN (see ?.C) - If using C++, consider using the package **Rcpp**. See the tutorial of Hadley Wickham at http://adv-r.had.co.nz/Rcpp.html. - or use a newer version of **Rcpp** called **Rcpp11**

Submitting to CRAN

1. Read the CRAN Repository Policy from http://cran.r-project.org/web/packages/policies.html.
2. Install the newest developer version of R from CRAN
3. Run **R CMD check --as-cran pcapack**. Packages must pass without warnings to be admitted to the CRAN.
4. Check with htt://http://win-builder.r-project.org/
5. Run **R CMD build pcapack** to make the tar.gz file.
6. Upload and follow instructions at http://bit.ly/1cw8qSS

Almost Ready for a DEMO...

- Building packages with an IDE, e.g. RStudio or Eclipse
- Building packages with **roxygen2** and Hadley Wickham’s package **devtools**
- Writing package vignettes
- Collaborative package development, e.g. github
- Automatic tests
- There is still a lot to learn about the NAMESPACE file
- For the future: read 5 times the manual *Writing R Extensions*

package devtools I

- makes life easy, especially packaging
- to publish packages (CRAN)
- installation of non-CRAN packages (local, github, bitbucket, ...)

```
library(devtools)
install_github("robCompositions","matthias-da")
```

- used when changing code
  - **load_all('pathToPackage')**: restart, re-install and re-load

package devtools II

- **test('pathToPackage')** runs tests placed in the *inst/test/* directory.
- **document('pathToPackage')** converts inline roxygen document blocks to R's standard Rd files in the *man/* directory
- **check()**, **check_docs()**, **run_examples()**, **build_win()**

Modern Approach: STEPS

1. create a project
2. specify that this project is about an R package

- tick roxygen2 documentation - create R folder - put the R functions to this folder - include roxygen2 documentation within the R functions - run **devtools::load_all()** - update DESCRIPTION File manually - build the package

roxygen2 documentation. This:

```
#' Add together two numbers
#'
```

```
#' @param x A number
#' @param y A number
#' @return The sum of \code{x} and \code{y}
#' @examples
#' add(1, 1)
#' add(10, 1)
add <- function(x, y) {
  x + y
}
```

roxygen2 documentation. Gives:

```
\name{add}
\alias{add}
\title{Add together two numbers}
\usage{
add(x, y)
}
\arguments{
  \item{x}{A number}

  \item{y}{A number}
}
\value{
The sum of \code{x} and \code{y}
}
\description{
Add together two numbers
}
\examples{
add(1, 1)
add(10, 1)
}
```

Let's build a package: DEMO

For the newer approach to build a package, keep in mind those steps:

1. create a **project** in RStudio
2. under *Project options* specify that this project is about an **R package** and tick **roxygen2** documentation

- create **R folder** and put the R functions theirein - include roxygen2 documentation within the R files - run **load_all()** (from devtools package) - update the DESCRIPTION File manually

(more steps if C++ code is integrated, vignettes, S4 class code, etc,...)

Summary

- Packages are standardized units for extending R.
- A package contains documented functions, data and other objects.
- install.packages() to install an add-on package.
- Loaded a package into the system by the library() command before using it
- A package is built in few steps:
  - R project / R package
  - R folder with R files containing roxygen2 documentation;
  - run **load_all()**
  - Check, build and install using R CMD ...