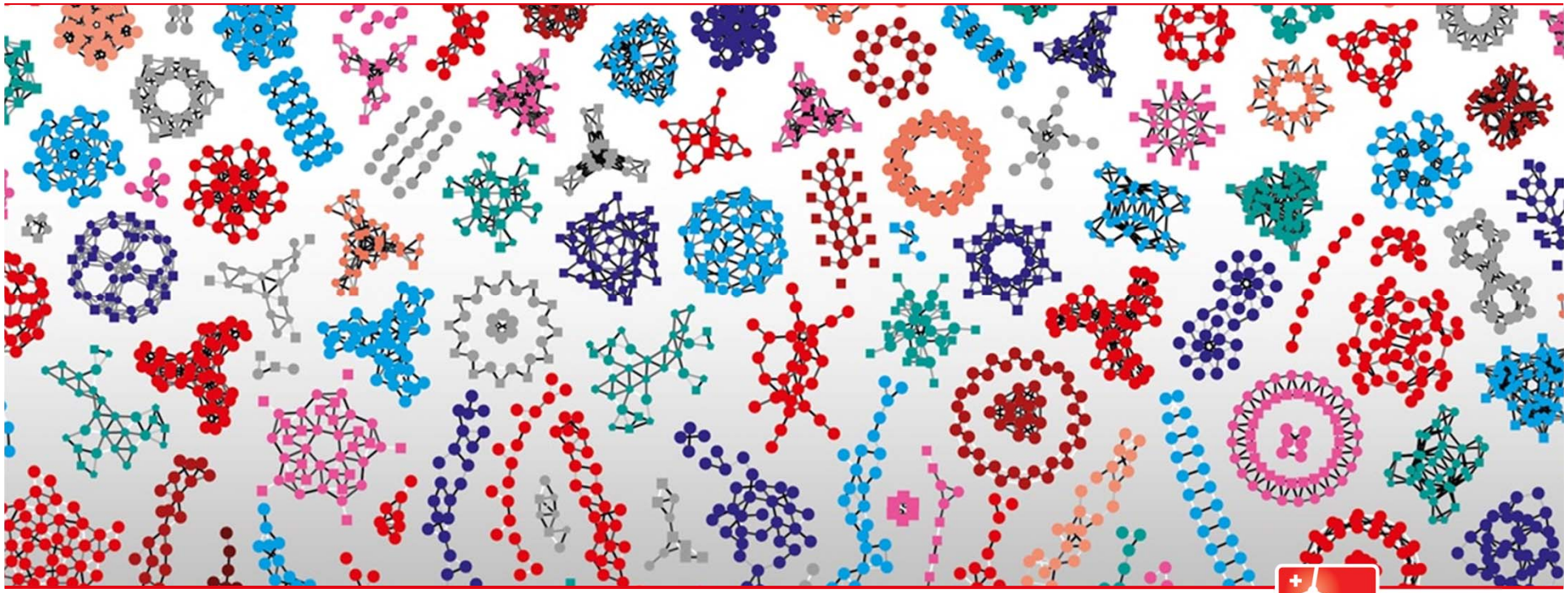


Swiss Institute of Bioinformatics

R packaging
September 2024

Frédéric Schütz (Frederic.Schutz@sib.swiss)

How can we distribute/share R code ?

Copy/paste snippets of code

analyze1.R

```
data <- read.table("data.txt")
# Our routine data analysis
par(mfrow=c(2,2))
hist(data[,1])
plot(data[,2], data[,3])
hist(data[,4])
boxplot(data[,5])
```

analyze2.R

```
data <- read.table("data2.txt")
pa                              )
hi
pl
hi
bo
```

analyze3.R

```
data <- read.table("data3.txt")
par(mfrow=c(2,2))
hist(data[,1])
plot(data[,2], data[,3])
hist(data[,4])
boxplot(data[,5])
```

Source files

analyze1.R

```
data <- read.table("data.txt")
source("common.R")
```

analyze2.R

```
data <- read.table("data2.txt")
source("common.R")
```

common.R

```
par(mfrow=c(2,2))
hist(data[,1])
plot(data[,2], data[,3])
hist(data[,4])
boxplot(data[,5])
```

analyze1.R

```
library("common_analysis")
data <- read.table("data.txt")
plot_data(data)
```

analyze2.R

```
library("common_analysis")
data <- read.table("data2.txt")
plot_data(data)
```

analyze3.R

```
library("common_analysis")
data <- read.table("data3.txt")
plot_data(data)
```

Several ways to distribute R code

- Copy/paste snippets of code
- Source files
- Packages

«*In R, the fundamental unit of shareable code is the package. A package bundles together code, data, documentation, and tests, and is easy to share with others.* »

– Hadley Wickham

http://cran.r-project.org, *September 2024*

CRAN
Mirrors
What's new?
Search
CRAN Team

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Task Views
Other

Documentation
Manuals
FAQs
Contributed

Contributed Packages

Available Packages

Currently, the CRAN package repository features 21232 available packages.

Table of available packages, sorted by date of publication

Table of available packages, sorted by name

CRAN Task Views aim to provide some guidance which packages on CRAN are relevant for tasks related to a certain topic. They provide tools to automatically install all packages from each view. Currently, 44 views are available.

Installation of Packages

Please type `help("INSTALL")` or `help("install.packages")` in R for information on how to install packages from this repository. The manual R Installation and Administration (also contained in the R base sources) explains the process in detail.

Package Check Results

All packages are tested regularly on machines running Debian GNU/Linux, Fedora, macOS (formerly OS X) and Windows.

The results are summarized in the check summary (some timings are also available).

Linking to Packages

Please use the canonical form `<https://CRAN.R-project.org/package=PKG>` to link to the CRAN web page of package *PKG*.

Why create R packages ?

If you want to …

- share code
- organize code
- version code
- make your code easily available, even if only for yourself

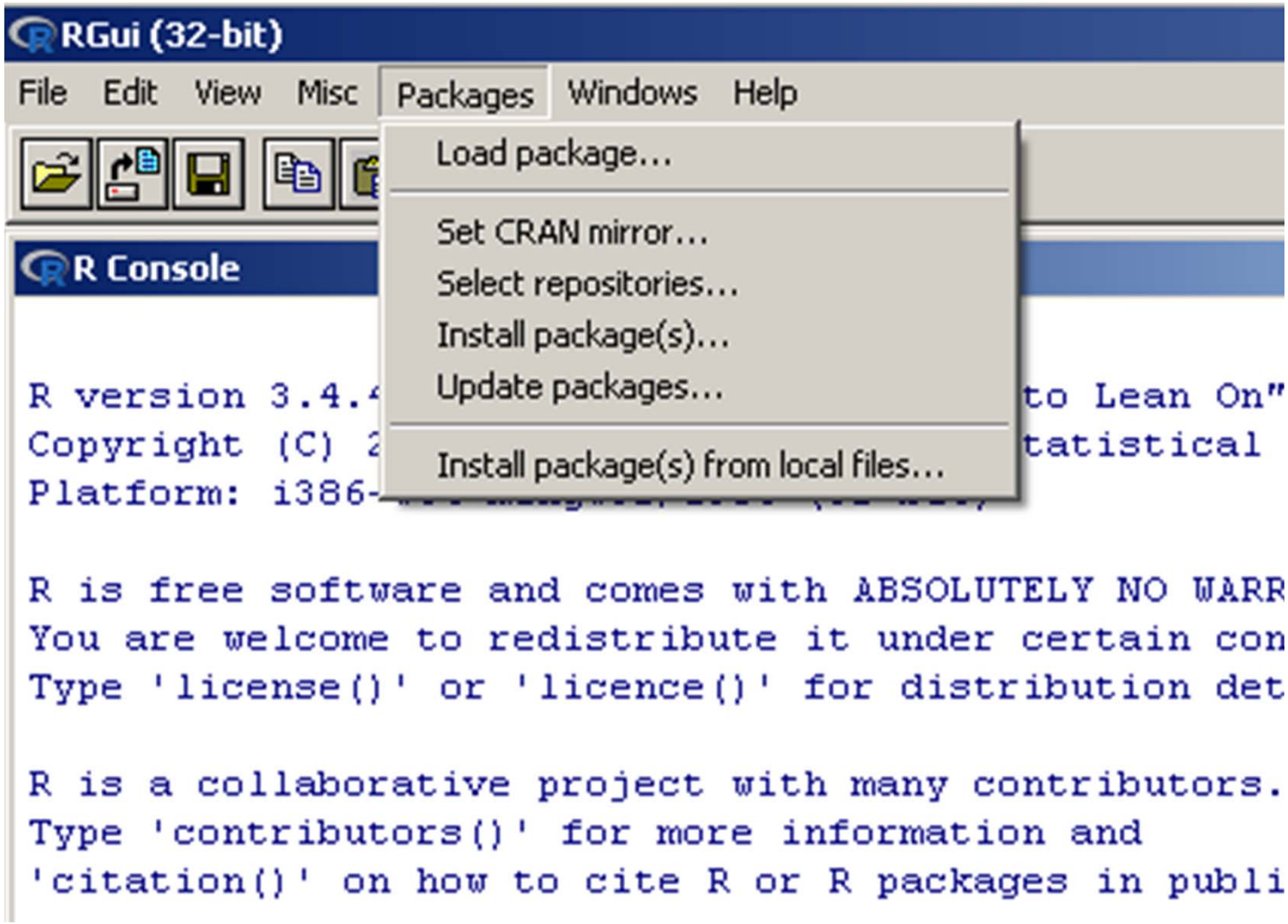
Using R packages

From the console

```
# Package from CRAN
> install.packages("ABPS")
Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)
[…]
* DONE (ABPS)
> library(ABPS)
>
# Local package file
> install.packages("test_0.01.tar.gz")
Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)
inferring 'repos = NULL' from 'pkgs'
[…]
* DONE (test)
>
```

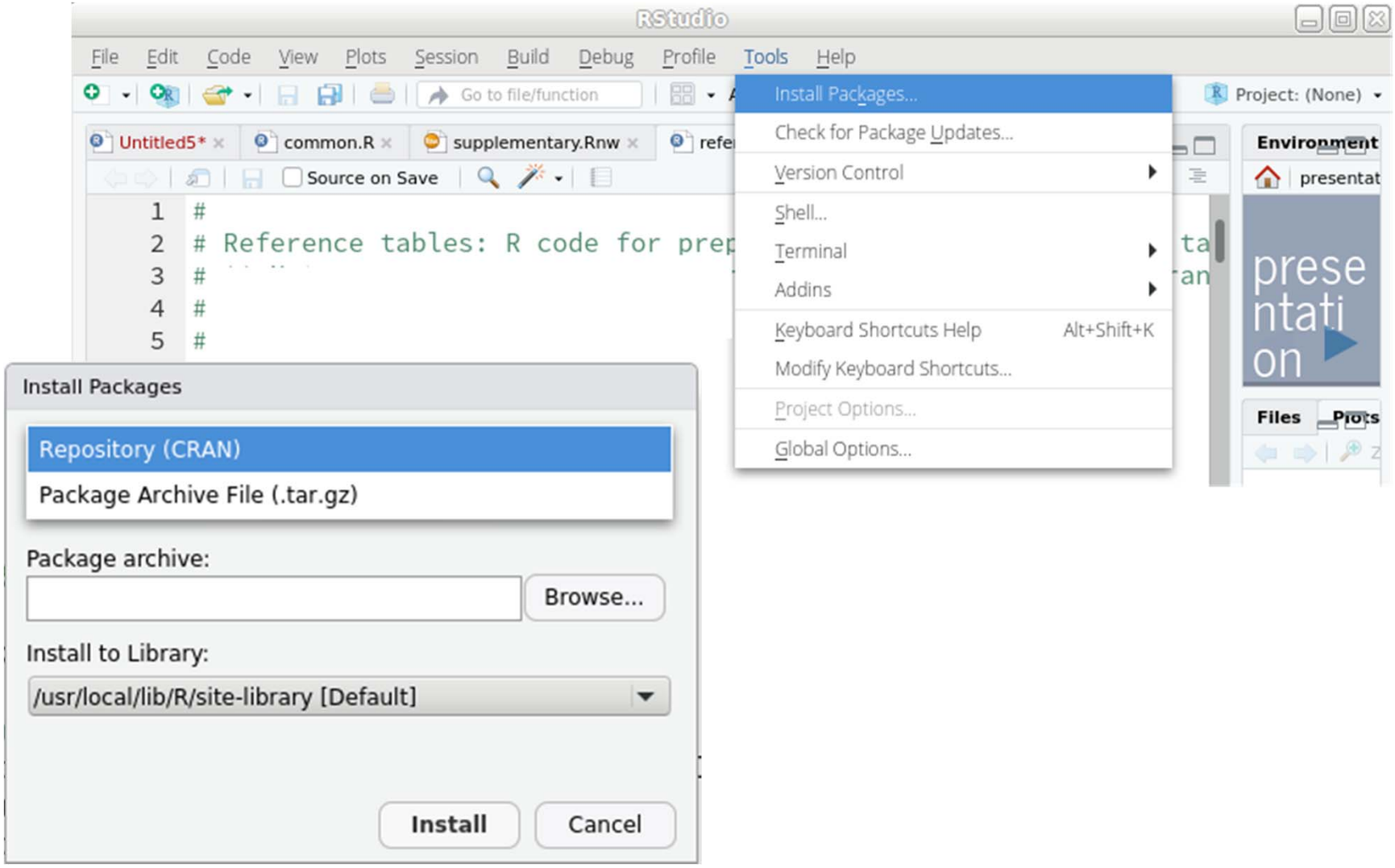
Downloading, installing and loading an R package

From the basic R GUI (Windows)



Downloading, installing and loading an R package

From RStudio



When using `install.packages()` within RStudio, you are calling a **different version** of the function than in base R.

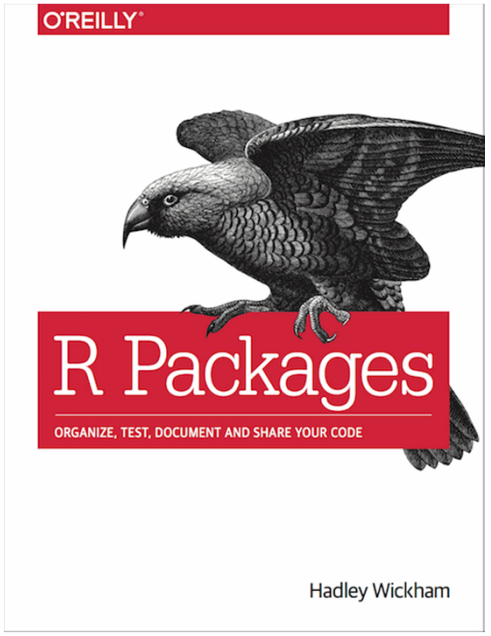
To install a local package, you must specify explicitly that it does not come from a repository:

```
install.packages("test_0.01.tar.gz",
                 repos=NULL)
```

Creating R packages

References

Hadley Wickham, "*R packages*".
Print: O'Reilly, April 2015
Up-to-date version online at:
`http://r-pkgs.org/`

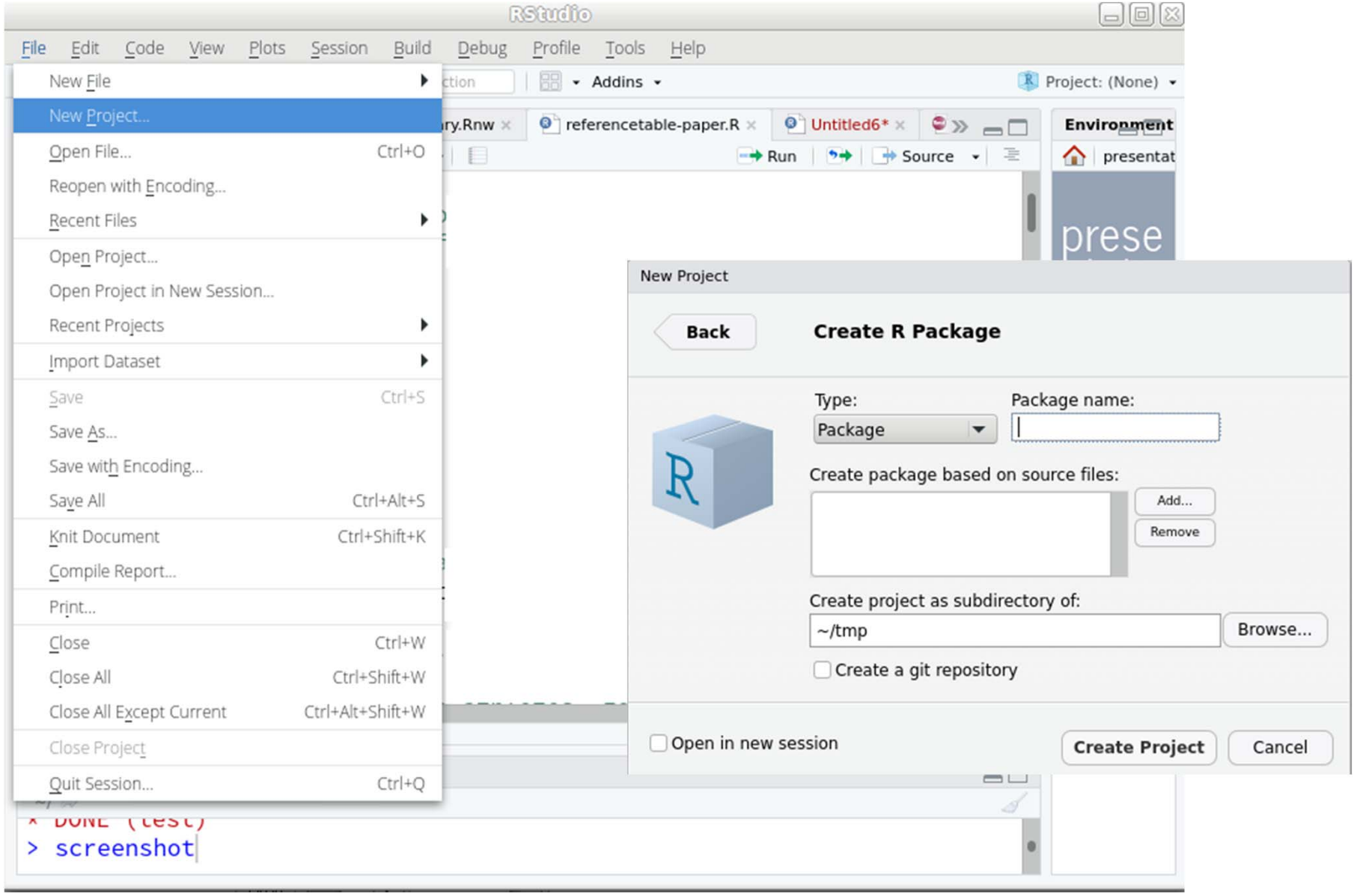


"*Writing R Extensions*"
`https://cran.r-project.org/doc/manuals/r-release/R-exts.html`

Prerequisites

R contains the bare minimum in order to build packages.

Highly recommended: RStudio



Highly recommended: the devtools package

devtools: Tools to Make Developing R Packages Easier

Collection of package development tools.

Version:	1.13.5
Depends:	R (≥ 3.0.2)
Imports:	httr (≥ 0.4), utils, tools, methods, memoise (≥ 1.0.0), whisker, digest, rstudioapi (≥ 0.2.0), jsonlite, stats, git2r (≥ 0.11.0), withr
Suggests:	curl (≥ 0.9), crayon, testthat (≥ 1.0.2), BiocInstaller, Rcpp (≥ 0.10.0), MASS, rmarkdown, knitr, hunspell (≥ 2.0), lintr (≥ 0.2.1), bitops, roxygen2 (≥ 5.0.0), evaluate, rversions, covr, gmailr (> 0.7.0)
Published:	2018-02-18
Author:	Hadley Wickham [aut], Jim Hester [aut, cre], Winston Chang [aut], RStudio [cph], R Core team [ctb] (Some namespace and vignette code extracted from base R)
Maintainer:	Jim Hester <james.hester at rstudio.com>
BugReports:	https://github.com/hadley/devtools/issues
License:	GPL-2 | GPL-3 [expanded from: GPL (≥ 2)]
URL:	https://github.com/hadley/devtools
NeedsCompilation:	yes
Materials:	README NEWS
CRAN checks:	devtools results

Downloads:

Reference manual:	devtools.pdf
Vignettes:	Devtools dependencies
Package source:	devtools_1.13.5.tar.gz
Windows binaries:	r-prerel: devtools_1.13.5.zip, r-release: devtools_1.13.5.zip, r-oldrel: devtools_1.13.5.zip
OS X binaries:	r-prerel: devtools_1.13.5.tgz, r-release: devtools_1.13.5.tgz

You may also need:

- LaTeX for building manuals in PDF
- GNU software tools, if your package is compiled (for example, if it contains C code)

https://support.rstudio.com/hc/en-us/articles/200486498-Package-Development-Prerequisites

- **Linux:** depends on your distribution
  for example, Ubuntu/Debian:
  `sudo apt install r-base-dev texlive-full`

- **Mac**:
  Install "Command Line Tools for XCode"
  Install MacTeX

- **Windows**
  Install the Rtools:
  https://stat.ethz.ch/CRAN/bin/windows/Rtools/
  Install Miktex:
  http://miktex.org/download

https://support.rstudio.com/hc/en-us/articles/200486498-Package-Development-Prerequisites

How to create a basic R package ?

What you need to create a basic "test" package by hand

`test` (directory), containing

`DESCRIPTION` (file)

The DESCRIPTION file contains two lines:

```
Package: test
Version: 1.0
```

This is the bare minimum required to create an R package

What you need to create a basic "test" package by hand

`test` (directory), containing

`DESCRIPTION` (file)

The DESCRIPTION file contains two lines:

```
Package: test
Version: 1.0
```

This is the bare minimum
required to create
an R package
… which is completely useless.

*The most basic **useful** package*

`test` (directory), containing

- `DESCRIPTION` (file)
- `NAMESPACE` (file)
- `R` (directory), containing
  - `any_file.R` (files), containing R functions

The `DESCRIPTION` file contains two lines:

```
Package: test
Version: 1.0
```

The `NAMESPACE` file indicates which functions are made available to users of the package.

The name of the package is provided by the DESCRIPTION file.

The name of the directory containing the package file should have the same name.

It makes your life easier, but you could choose any other name if you wanted

(but you really, really, should not)

The R directory

Content of the R directory

myfunctions_1.R

```
myfunction <- function(arguments) {
…
}

plotdata <- function(data, add=TRUE) {
…
}
```

otherfunctions.R

```
analyze_data <- function(arguments) {
…
}

pipeline <- function(data, add=TRUE) {
…
}
```

- The R directory contains .R files which are executed when building the package, thus creating the function objects

- The .R files must contain the assignment of functions:

```
myfunction <- function(…) {…}
```

- You are free to organize the files as you wish (one or more functions per file)

The NAMESPACE file

- Any function you define in the R directory is by default internal to the package only

- To make it available to the user, you need to **export** it explicitly.

You can export a function by adding a line

```
export(myfunction)
```

in the file NAMESPACE for each function you want to export.

R subdirectory

myfunctions_1.R

```
myfunction <- function(arguments) {…}
plotdata <- function(data, add=TRUE) {…}
```

otherfunctions.R

```
analyze_data <- function(arguments) {…}
pipeline <- function(data, add=TRUE) {…}

internalfunction <- function(data) {…}
```

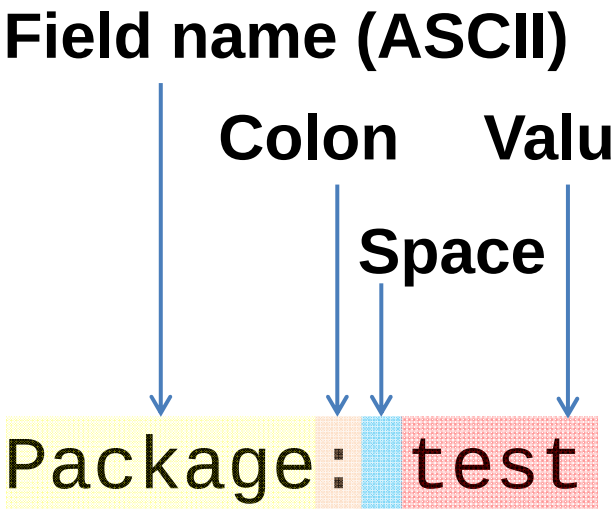
NAMESPACE

```
export(myfunction)
export(plotdata)
export(analyze_data)
export(pipeline)
```

The DESCRIPTION file

The `DESCRIPTION` file contains the metadata regarding the package.

The presence of this file is what makes a directory recognized as an R package.




```
Title: What the Package Does (one line)
Version: 0.1
Description: What the package does (one
 paragraph). The description can span
 several lines if needed.
```

Indentations (spaces or tabs) indicate multi-line fields

This file format is called "Debian Control File" (DCF)

See the help page for the `read.dcf()` function for more information.

Field names are case-sensitive.

All those used by R are capitalized.

Some fields expect a boolean value.

You can use either "yes", "true", "no", "false" (or a capitalized version of these)

Example

`devtools: Tools to Make Developing R Packages Easier`

Collection of package development tools.

Version:	1.13.5
Depends:	R (≥ 3.0.2)
Imports:	httr (≥ 0.4), utils, tools, methods, memoise (≥ 1.0.0), whisker, digest, rstudioapi (≥ 0.2.0), jsonlite, stats, git2r (≥ 0.11.0), withr
Suggests:	curl (≥ 0.9), crayon, testthat (≥ 1.0.2), BiocInstaller, Rcpp (≥ 0.10.0), MASS, rmarkdown, knitr, hunspell (≥ 2.0), lintr (≥ 0.2.1), bitops, roxygen2 (≥ 5.0.0), evaluate, rversions, covr, gmailr (> 0.7.0)
Published:	2018-02-18
Author:	Hadley Wickham [aut], Jim Hester [aut, cre], Winston Chang [aut], RStudio [cph], R Core team [ctb] (Some namespace and vignette code extracted from base R)
Maintainer:	Jim Hester <james.hester at rstudio.com>
BugReports:	https://github.com/hadley/devtools/issues
License:	GPL-2 | GPL-3 [expanded from: GPL (≥ 2)]
URL:	https://github.com/hadley/devtools
NeedsCompilation:	yes
Materials:	README NEWS
CRAN checks:	devtools results

`Downloads:`

Reference manual:	devtools.pdf
Vignettes:	Devtools dependencies
Package source:	devtools_1.13.5.tar.gz
Windows binaries:	r-prerel: devtools_1.13.5.zip, r-release: devtools_1.13.5.zip, r-oldrel: devtools_1.13.5.zip
OS X binaries:	r-prerel: devtools_1.13.5.tgz, r-release: devtools_1.13.5.tgz

The 7 mandatory fields of a `DESCRIPTION` file

The 7 "mandatory" fields in the DESCRIPTION

You need these in order to *create* the package

```
Package: test
Version: 0.0.1
```

```
Title: What the Package Does (one line, title case)
Description: What the package does (one paragraph),
  over one or more lines.
Author: John Smith
Maintainer: John Smith <john@smith.com>
License: What license is it under?
```

Later *checks* of the package will *fail* without these

Each package must have at least one author and one maintainer (which may be the same person)

`Author`: people who have made significant contributions to the package

`Maintainer`: the person responsible for the package, the main contact

At least the maintainer should have an email address.

You can specify the authors in a more flexible way using the Authors@R field.

This field contains an executable R command, containing 4 fields for each author, e.g. :

```
Authors@R: person("First",
                  "Last",
                  email = "first.last@example.com",
                  role = c("aut", "cre"))
```

The Author and Maintainer fields will be automatically generated from this field.

The most important roles:

- `cre`: the creator or maintainer, the person responsible for the package, main contact
- `aut`: authors (people who have made significant contributions to the package)
- `ctb`: contributors (people who have made smaller contributions)

(note: there are many other possible roles available, see `https://www.loc.gov/marc/relators/relaterm.html`)

You can specify several authors using a vector:

```
Authors@R: c(
   person("Joe", "Developer", role = c("aut", "cre"),
          email = "Joe.Developer@some.domain.net"),
   person("A.", "User", role = "ctb",
          email = "A.User@whereever.net")
)
```

(example taken from the R manual)

The package *must* specify under which conditions it can be reused and distributed.

… even if you do not intend to distribute it.

This is done by specifying a **License**.

A license is a contract, which allows an author to specify what a user is allowed to do (or not do) with software that he created.

R recognizes shortcuts for many common software licenses, such as:

- GNU General Public Licence
  `GPL-2`, `GPL-3`, `GPL (>=2)`
- BSD or MIT licenses
  `MIT`, `BSD_2_clause`
- Creative Commons licenses
  `CC BY-SA 4.0`, `CC BY 4.0`
- No restrictions
  `Unlimited`, `CC0`

https://svn.r-project.org/R/trunk/share/licenses/license.db

R is centred on the concept of **free software**, which

- can be used by anyone without any limitation
- can be studied (e.g. you can read the code)
- can be modified
- can be redistributed (including the modifications)

R makes it easy to distribute your code as free software.

R itself is distributed under the GPL

If you want to submit your package to the CRAN repository, it must be distributed under a recognized free software license.

You can use a custom license by creating a file called LICENSE that contains the terms of your licenses and by specifying

```
License: file LICENSE
```

in the DESCRIPTION file.

You can use the license file to indicate that a package is proprietary (e.g. internal to your group) and should not be distributed.

However, you should **not** try to go further and write an actual license that you will distribute – use a standard license or ask a lawyer first.

At least two (often 3) integers, separated by either "." or "-".

The canonical form is "1.0-1".

This is used to check for dependencies (for example, for specifying minimum versions of software to use)

A version field does not represent a decimal number – the parts must be considered separately:

Version 0.9 < 0.75

Because 9 < 75

The 7 mandatory fields

```
Package: test
Version: 0.0.1
Title: What the Package Does (one line, title case)
Description: What the package does (one paragraph),
  over one or more lines.
Author: John Smith
Maintainer: John Smith <john@smith.com>
License: What license is it under?
```

The field names and the package name can only contain ASCII characters:

A-Z, a-z, 0-9, plus a few signs

Ideally, the *whole* file should be written using only ASCII characters.

It is not always possible – e.g. if an author has accents in his/her name, we need more than ASCII.

In this case, the file must specify an encoding, e.g.

```
Encoding: UTF-8
```

Encoding ?

How to make an R package out of this directory ?

The R software can be used as a batch utility:

```
R CMD command …
```

Examples:

```
R CMD INSTALL package
R CMD REMOVE package
R CMD BATCH script.R
R CMD build directory
```

Go to the directory which contains the package directory (one directory up from the DESCRIPTION file)

Call the R batch command:

```
R CMD build DIRECTORY
```

This is relatively easy to do when using Linux or Mac

Under Windows, it may be harder to call the program. For example, the command may look like:

```
C:\PROGRA~1\R\R-3.4.4\bin\R.exe
                    CMD build directory
```

Example of package build

```
schutz@laptop:~/packages$ R CMD build test
* checking for file 'test/DESCRIPTION' ... OK
* preparing 'test':
* checking DESCRIPTION meta-information ... OK
* checking for LF line-endings in source and
  make files and shell scripts
* checking for empty or unneeded directories
* building 'test_0.01.tar.gz'

schutz@laptop:~/packages$ ls -l test_0.01.tar.gz
[...] 311 test_0.01.tar.gz
```

- From the command line
  `R CMD INSTALL package.tar.gz`
- From R
  `install.packages("package.tar.gz")`
  `The argumet,`
  `repos=NULL, type="source")`
- `(repos/source are now infered)`
- `Packages > Install package(s) from local files`
- From R Studio:
  `Tools > "Install packages"`

Exercise

- Create the minimum R package, build it, install it and load it
- Do the same with a package that is actually useful: add an R function, and make sure that you can use it after loading the package.

- If you need help finding R functions to package, look at the Moodle website.

Congratulations !

More about the `DESCRIPTION` file:

how to specify dependencies

Dependencies allow you to specify which other packages your package need in order to do its job.

You can specify hard dependencies using the Imports keyword in the `DESCRIPTION` file:

```
Imports: kernlab, ggplot2
```

Packages specified this way will be installed at the same time as your package.

Packages indicated as "suggestions" can be used by your package, but they are not absolutely required to use it.

```
Suggests: kernlab, ggplot2
```

Your package should always check for the presence of the suggested package before relying on it.

Dependencies and suggestions can be versioned.

```
Imports: ggplot2 (>=2.2.0)
Suggests: ggvis (>=0.2)
```

Always depend on a given version **or higher**.

Dependency on a particular version of R can be specified using the Depends keyword:

```
Depends: R (>= 3.4.0)
```

Do **not** specify a version: Depends: R (==3.4.0)

"Depends" can also be used to specify packages; however, they will be automatically loaded and attached. "Imports" is usually the best choice.

Example: the devtools package

devtools: Tools to Make Developing R Packages Easier

Collection of package development tools.

Version:	1.13.5
Depends:	R (≥ 3.0.2)
Imports:	httr (≥ 0.4), utils, tools, methods, memoise (≥ 1.0.0), whisker, digest, rstudioapi (≥ 0.2.0), jsonlite, stats, git2r (≥ 0.11.0), withr
Suggests:	curl (≥ 0.9), crayon, testthat (≥ 1.0.2), BiocInstaller, Rcpp (≥ 0.10.0), MASS, rmarkdown, knitr, hunspell (≥ 2.0), lintr (≥ 0.2.1), bitops, roxygen2 (≥ 5.0.0), evaluate, rversions, covr, gmailr (> 0.7.0)
Published:	2018-02-18
Author:	Hadley Wickham [aut], Jim Hester [aut, cre], Winston Chang [aut], RStudio [cph], R Core team [ctb] (Some namespace and vignette code extracted from base R)
Maintainer:	Jim Hester <james.hester at rstudio.com>
BugReports:	https://github.com/hadley/devtools/issues
License:	GPL-2 | GPL-3 [expanded from: GPL (≥ 2)]
URL:	https://github.com/hadley/devtools
NeedsCompilation:	yes
Materials:	README NEWS
CRAN checks:	devtools results

Downloads:

Reference manual:	devtools.pdf
Vignettes:	Devtools dependencies
Package source:	devtools_1.13.5.tar.gz
Windows binaries:	r-prerel: devtools_1.13.5.zip, r-release: devtools_1.13.5.zip, r-oldrel: devtools_1.13.5.zip
OS X binaries:	r-prerel: devtools_1.13.5.tgz, r-release: devtools_1.13.5.tgz

You may even create an "empty" R package, which contains only the dependencies for your code

Using better tools

It is possible to build packages using the command line.

However, this can be cumbersome – especially if you use Windows.

RStudio allows you to manage packages (in the form of projects) in a more convenient way.

Build again the same package as before, starting from an empty RStudio project.

Checking a package

R provides a **large** number of checks that can be applied to a package.

These will yields errors, warnings or notes about your package.

You should not distribute your package if it does not come clean from a check.

From the command line:

```
R CMD check sourcepackage.tar.gz
```

From Rstudio:

```
Build > Check Package
```

If you plan on uploading your package to CRAN, R can perform additional tests on it.

```
R CMD check --as-cran sourcepackage.tar.gz
```

You **must not** submit any package to CRAN if it is does not come clean after this command.

Check the package that you have created before; try to understand any issue that the check command may report.

Adding documentation

Every function in a package should be documented.

Documentation is stored as .Rd files in the `man/` subdirectory of your package.

The file format is similar to LaTeX.

Example of .Rd file for an add() command

```
\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)
}
```

The `roxygen2` and `devtools` packages offer an easier way to write documention.

The documentation is written in the form of **literate programming**: together with your code.

In practice, you add comments (in a specific format) to your R code.

They are then transformed into .Rd files

Example: Roxygen2 documentation

```
#' 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
}
```

From the "R packages" book

To create the .Rd files: use the command

```
devtools::documents()
```

(or use the relevant shortcuts in Rstudio)

- All lines start with

  `#'`

  in order to differentiate them from code and regular comments

- The lines should form a single block before a function
- Lines should not be over 80 characters long

- The first sentence (on the first) line will the title of the help page
- The second paragraph will contain the introduction
- The next paragraphs will contain the details

You can create arbitrary sections using the @section tag:

```
#' @section Note:
#' The function has the same name, but a
#' different output, as the one provided in
#' the previous version of the package.
```

Most functions will have at least 3 sections defined by given tags:

- `@param NAME description`
  to define the function parameters:
- `@return` to define the value(s) returned
- `@examples` to provide executable R code as an example of use of the function

Example

```
#' A function for calculating XYZ
#'
#' The \code{XYZ} function computes XYZ.
#'
#' @param X a vector or matrix containing the X data
#' @param Y a vector containing the Y data (not needed
#'          if the X parameter is a matrix)
#'
#' @return a vector containing the XYZ score(s).
#'
#' @examples
#' XYZ(1,1)
```

roxygen2 can also manage your NAMESPACE file for you:

```
#'  … documentation
#' @export
skewness <- function(a, na.rm=FALSE) {
   …
}
```

Principle: it is easier to manage exports close to the function.

Adding data to a package

Several kinds of data in a package

- Data that will be called by the user using the `data()` command
- Internal data (e.g. precomputed data tables used by your method)
- Raw data used as example

- The dataset should be provided in the data/ subdirectory
- They should be in .Rdata format, which you can obtain using the `save()` command.
- Typically:

```
mydata <- read.csv("mydata.csv")
save(mydata, "data/mydata.Rdata")
```

Datasets are exported by default, so that you do not need to export them explicitely.

You can document data using roxygen2, by adding the documentation into the R/ directory (for example, in a data.R file)

Specify the name of the dataset at the end of the documentation (after the block):

```
#' Example of mydata
#' …
#' End of description
"mydata"
```