R Package Development

Maoying Wu

Outline

Next...

References

- R Installation and Administration [R-admin], R Core team
- Writing R Extensions [R-ext], R Core team

Use help.start() to access them from your local installation, or
http://cran.r-project.org/manuals.html from the web.

Terminology

A **package** is loaded from a **library** by the function library(). Thus a library is a directory containing installed packages.

Calling library("foo", lib.loc = "/path/to/bar") loads the package (book) *foo* from the library bar located at /path/to/bar.

Packages
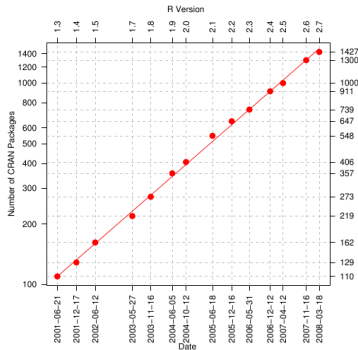
One of the aspects that make
R appealing:

CRAN package repository features
7542 available packages.

R-forge 1965 hosted projects.

Bioconductor 2164 packages in latest
release (version 3.0).

Numbers checked in December 2015

Why packages

Packages provide a mechanism for loading optional code and attached documentation when needed.

Also – packages are a means to

- logically group your own functions

- keep code and documentation together and consistent

- keep code and data together

- keep track of changes in code

- summarise all packages used for analyses (see `sessionInfo()`)

- make a reproducible research compendium (container for code, text, data as a means for distributing, managing and updating)

- optionally test your code

- . . . project managment

even if you do not plan to distribute them.

Package creation workflow

Workflow

1. Prepare your code as .R files
2. Run `package.skeleton()` to place them in a "package layout"
3. Create the .tar.gz file using **R CMD build myRpackage**
4. Check the .tar.gz for errors using **R CMD check myRpackage_1.0.tar.gz**
5. Install the package using **R CMD INSTALL myRpackage_1.0.tar.gz** or `install.packages()`

Notes

- Use steps 1-5 to create the initial version of the package, and steps 3-5 to update the package.
- Text in **bold** is executed **outside R**.

Creating a package - package.skeleton()

A minimal package

We prepared a file read.R that containt the function `readFasta()`.
Function package.skeleton() automates some of the setup for a new package:

```
> package.skeleton(name="myRpackage", code_files=c("read.R", '

Creating directories ...
Error in package.skeleton(name = "myRpackage", code_files =
c("read.R", :  directory './myRpackage' already exists
```

```
> package.skeleton(name="myRpackage", code_files=c("read.R", '
```

produces

```
myRpackage/
|-- DESCRIPTION
|-- NAMESPACE
|-- man
|    -- myRpackage-package.Rd
|    -- readFasta.Rd
|-- R
|    -- read.R
-- Read-and-delete-me

2 directories, 6 files
```

Building the package

Note
To build the package we will first delete the documentation templates (we will revisit this later).

```
# delete documentation templates
$ rm -f myRpackage/man/*
$ R CMD build myRpackage
* checking for file myRpackage/DESCRIPTION ... OK
* preparing myRpackage:
* checking DESCRIPTION meta-information ... OK
* checking for LF line-endings in source and make files
* checking for empty or unneeded directories
* building myRpackage_1.0.tar.gz
```

Installing and using the package

```
$ R CMD INSTALL myRpackage_1.0.tar.gz
* installing to library /ext/home/R/x86_64-pc-linux-gnu-library/3.1
* installing *source* package myRpackage ...
** R
** preparing package for lazy loading
** help
No man pages found in package  myRpackage
*** installing help indices
** building package indices
** testing if installed package can be loaded
* DONE (myRpackage)
```

Use package

We can verify the new package is now available (although still not fully functional!) from R:

```
> library("myRpackage")
>
> readFasta("aDnaSeq.fasta")

An object of class "GenericSeq"
Slot "id":
[1] "example dna sequence"
```

Exercise 1: Create a package

Follow the steps in the previous slides to create the example package containing the readFasta() function.

Exercise 2: Complete the package functionality

Add the GenericSeq class to the package by adding a new file DataClasses.R to your package. Copy this file to the package R/ directory and then rebuild the package following steps 3-5.

Some practical tips

Using devtools

Rebuilding and reloading packages can be a bit cumbersome. Package devtools has many great tools for package development, such as load_all() that mimics the rebuild/reload process.

```
> library(devtools)
> load_all("myRpackage/")
>
> readFasta("aDnaSeq.fasta")
```

... and many more!

Next...

Package commands

Building packages

`R CMD build myRpackage` – the R package builder builds R package (and vignettes if available).

Checking packages

`R CMD check myRpackage_1.0.tar.gz` or `R CMD check myRpackage` – the R package checker tests whether the package or source work correctly.

- The package is installed (checks missing cross-references and duplicate aliases in help files).
- File names validity, permissions.
- Package DESCRIPTION file is checked for completeness, and some of its entries for correctness.
- `R` and `.Rd` files are checked for syntax errors.
- A check is made for missing documentation entries.
- Codoc checking
- Examples provided by the package's documentation are run.
- If available, package tests are run and vignettes are executed and compiled.

Package commands

Installing packages

`R CMD INSTALL myRpackage_1.0.tar.gz` or
`install.packages("myRpackage_1.0.tar.gz")` — installs the package
in the default library. Other libraries can be specified with the `-l` option or
`lib` argument.

Loading

Use `library()` or `require()`.

On Windows

R is very much Unix centric. To build from source on Windows, you will
need `Rtools`[a]. See the *The Windows toolset* in R-Admin for more details.
It is also possible to build Windows binary packages from Linux[b].

---

[a] http://cran.r-project.org/bin/windows/Rtools/
[b] http://cran.r-project.org/doc/contrib/cross-build.pdf

DESCRIPTION

```
Package: myRpackage ## mandatory (*)
Type: Package ## optional, Package is default type
Title: What the package does (short line) ## *
Version: 1.0 ## *
Date: 2013-05-10 ## release date of the current version
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? ## *
Depends: methods, Biostrings ## for e.g.
Imports: evd ## for e.g.
Suggests: BSgenome.Hsapiens.UCSC.hg19 ## for e.g.
Collate: DataClasses.R read.R ## for e.g.
```

Lazy

Lazy loading

A mechanism used to defer initialization of an object until the point at which it is needed. The individual objects in the package's environment are indirect references to the actual objects until, for example a function is called or an object loaded.

The `LazyLoad` and `LazyData` fields control whether the R objects and the datasets (respectively) use lazy-loading. `LazyLoad` must be set if the *methods* package is used.

`LazyLoad` is now on by default.

> **Example**
>
> R uses *Lazy evaluation*, which delays the evaluation of an expression (here the argument) until its value is actually required [a]:
>
> ---
> [a] example from Hadley Wickham's devtools

```
> f <- function(x) { 10 }
> system.time(f(Sys.sleep(3)))

  user  system elapsed
    0       0       0

> f <- function(x) { force(x); 10 }
> system.time(f(Sys.sleep(3)))

  user  system elapsed
 0.000   0.000   3.004
```

DESCRIPTION

Other important fields

Depends A comma-separated list of package names (optionally with versions) which this package depends on.

Suggests Packages that are not necessarily needed: used only in examples, tests or vignettes, packages loaded in the body of functions (see `require()`).

Imports Packages whose name spaces are imported from (as specified in the `NAMESPACE` file) but which do not need to be attached to the search path.

Collate Controls the collation order for the R code files in a package. If filed is present, all source files must be listed.

URL A list of URLs separated by commas or whitespace.

. . .

NAMESPACE

The NAMESPACE file

Stored in the package directory. Restrict the symbols that are exported and imports functionality from other packages. Only the exported symbols will have to be documented.

Note: NAMESPACE is now required (since R 2.14).

```
export(f, g) ## exports f and g
exportPattern("^[^\\.]")
import(foo) ## imports all symbols from package foo
importFrom(foo, f, g) ## imports f and g from foo
```

It is possible to explicitely use symbol s from package *foo* with foo::s or foo:::s if s is not exported.

Attach and load

Packages are attached to the search path with `library` or `require`.

Attach When a package is attached, then all of its dependencies
(see `Depends` field in its `DESCRIPTION` file) are also
attached. Such packages are part of the evaluation
environment and will be searched.

Load One can also use the `Imports` field in the `NAMESPACE` file.
Imported packages are loaded but are not attached: they do
not appear on the search path and are available only to the
package that imported them.

Package subdirectories

R

Contains source()able R source code to be installed. Files must start with an ASCII (lower or upper case) letter or digit and have one of the extensions .R (recommended), .S, .q, .r, or .s. File order is important if code relies on *earlier* code – order use Collate filed in DESCRIPTION file.

Example

```
## works fine without Collate field
AllGenerics.R       DataClasses.R
methods-ClassA.R    methods-ClassB.R
functions-ClassA.R  ...
```

zzz.R is generally used to define special functions used to initialize (called after a package is loaded and attached) and clean up (just before the package is detached). See help(".onLoad")), ?.First.Lib and ?.Last.Lib for more details.

Package subdirectories

man

Manuals for the objects (package, functions, generics, methods, classes and data sets) in the package in R documentation (Rd) format. The filenames must start with an ASCII (lower or upper case) letter or digit and have the extension .Rd or .rd and should be URL compatible. If you use a NAMESPACE, only exported symbols need to be documented. Without NAMESPACE, internal use only objects should be documented in pkg-internal.Rd.

Package subdirectories

data

Contains data files, made available via *lazy-loading* or for loading using
data(). Data types that are allowed are

- **R code** self-sufficient plain R code (.R or .r),
- **Tables** possibly compressed tables (.tab, .txt, or .csv, see ?data for the file formats)
- **Objects** created using save() (.RData or .rda).

Example

There is a DnaSeq object in sequences/data.

Package subdirectories

inst

Content is copied recursively to the installation directory, for example

CITATION file (see `citation()` function),

doc directory for additional documents (see vignettes[a], later).

extdata directory for other data files, not belonging in `data`.

tests code for unit tests (see later).

---

[a]It is now also possible to use the `./vignettes` directory for these.

Example

In our *sequences* package, there is a `fasta` sequence in
`sequences/inst/extdata` used to illustrate the `readFasta` function.

Package subdirectories

tests

Contains additional package-specific test code. We will talk about unit tests later.

demo

R scripts runned via demo() that demonstrate some of the functionality of the package. Execution of these scripts is not checked.

src

Contains sources and headers for the compiled code, plus optionally a file Makevars or Makefile.

Adding C/C++ code

Using package Rcpp**:**

Package Rcpp makes it easy to call C/C++ code from R. It also has a function:

```
> Rcpp.package.skeleton("myRpackage")
```

that will create a fully functioning R package with C/C++ integration.

Or manually:

1. Add your C/C++ files to directory src/
2. Create R wrapper functions for your C/C++ code
3. (Optionally) provide Makefile and Makevars files in src/
4. Add useDynLib(myRpackage) to NAMESPACE file
5. (For Rcpp) also add LinkingTo: Rcpp to DESCRIPTION

Next...

Rd format

R documentation format

R objects are documented in files written in *R documentation* (Rd) format, a simple markup language much of which closely resembles LaTeX, which can be processed into a variety of formats, including LaTeX, HTML, pdf and plain text.

man/ directory

The documentation files are in the separate man/ package directory.

Using roxygen2

Package roxyxgen2 makes it easier to write documentation using special **in-source** markup (similarly to Doxygen).

Roxygen2

Example

```
# Reads sequences data in fasta and create \code{DnaSeq}
# and \code{RnaSeq} instances.
#
# This funtion reads DNA and RNA fasta files and generates
# valid \code{"DnaSeq"} and \code{"RnaSeq"} instances.
#
# @title Read fasta files.
# @param infile  the name of the fasta file which the data are to be read from.
# @return an instance of \code{DnaSeq} or \code{RnaSeq}.
# @seealso \code{\linkS4class{GenericSeq}}, \code{\linkS4class{DnaSeq}} and \code{\linkS4class{RnaSeq}}.
# @examples
# f <- dir(system.file("extdata",package="sequences"),pattern="fasta",full.names=TRUE)
# f
#
# aa <- readFasta(f)
# aa
# @author Laurent Gatto \email{lg390@@cam.ac.uk}
# @export
readFasta <- function(infile){
  lines <- readLines(infile)
  header <- grep("^>", lines)
  if (length(header)>1) {
    warning("Reading first sequence only.")
    lines <- lines[header[1]:(header[2]-1)]
    header <- header[1]
  }
  ##### (code cut for space reasons) #####
  if (validObject(newseq))
    return(newseq)
}
```

Using roxygen2 to write documentation

Exercise 3: Document readFasta

Use the code from the previous slide to document your own readFasta() function inside your package. When you are finished do the following:

1. To generate the .Rd file for your package run:

```
library(roxygen2)
roxygenise("myRpackage/")
```

2. Rebuild/reload your package

3. Verify that ?readFasta gives the expected help page.

4. Look at man/readFasta.Rd and try to understand the structure.

roxygen2 syntax

For more information on roxygen2: http://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html.

Rd format

An Rd file constists of

   Header provides basic information about the name of the file, the topics documented, a title, a short textual description and R usage information – mandatory.

   Body gives further information defined within *sections* (for example, on the function's arguments and return value, as in the example)

   Footer with keyword information – optional.

Every (exported) object must be documented. Package documentation is optional.

```
% File src/library/base/man/load.Rd
\name{load}
\alias{load}
\title{Reload Saved Datasets}
\description{
  Reload the datasets written to a file with the function
  \code{save}.
}
\usage{
load(file, envir = parent.frame())
}
\arguments{
  \item{file}{a connection or a character string giving the
    name of the file to load.}
  \item{envir}{the environment where the data should be
    loaded.}
}
\seealso{
  \code{\link{save}}.
}
\examples{
## save all data
save(list = ls(), file= "all.RData")

## restore the saved values to the current environment
load("all.RData")

## restore the saved values to the workspace
load("all.RData", .GlobalEnv)
}
\keyword{file}
```

Documentation

General comments

- Different objects are documented with different types of Rd files, as defined by the \docType{} tag.
- Different object documentation require or are advised to contain different sections.
- One .Rd file can document several objects by defining multiple \alias{}'es.

Documentation

Guidelines for Rd files

These are suggested guidelines for the system help files (in `.Rd` format) that are intended for core developers but may also be useful for package writers. (see http://developer.r-project.org/Rds.html)

There are many different sections and marking text (for mathematical notation, tables, cross-references, ...), that will look very familiar to LaTeX users. All are described in *Writing R documentation files* (section 2) of the R-ext manual.

Fortunately, the `prompt(object)` *et. al.* functions will inspect the `object` to be documented and create a specific documentation skeleton for us to be completed.

Package documentation

Provides an short and optional overview of a package.

Example

```
promptPackage("sequences")
```

Demonstration

Let's look at the `sequences-package.Rd` that documents our package.

Data sets documentation

Example

```
\name{rivers}
\docType{data}
\alias{rivers}
\title{Lengths of Major North American Rivers}
\description{
  This data set gives the lengths (in miles) of 141 \dQuote{major}
  rivers in North America, as compiled by the US Geological
  Survey.
}
\usage{rivers}
\format{A vector containing 141 observations.}
\source{World Almanac and Book of Facts, 1975, page 406.}
\references{
  McNeil, D. R. (1977) \emph{Interactive Data Analysis}.
  New York: Wiley.
}
\keyword{datasets}
```

Example

prompt(myDataFrame) or promptData(myDataObject)

Demonstration

Let's look at the document of the dnaseq object.

Function documentation

Many markup command, including \usage{fun(arg1, arg2, ...)}, \arguments{...}, \section{Warning}{...} and \examples{...}, which are executed!

Example

```
prompt(object=myFunction) or prompt(name="myFunction")
```

Demonstration

We have written one functions for our package so far, readFasta. It's documentation is available in man/readFasta.Rd.

Documenting S4 classes and methods

Documentation is 'similar' than for functions. Note that `aliases` are of the form `MyClass-class` or `MyGeneric,signature_list-method`. Additionnal aliases should be added to refer to `MyGeneric`, `MyGeneric-method`, ...and the manuals are accessed with `class?topic` and `method?topic`. Overall documentation for methods should be aliased with `MyGeneric-methods`

See `help("Documentation", package = "methods")` for more details.

Example

```
promptClass("MyClass") and promptMethods("myMethod")
```

Document GenericSeq

Exercise 4: Document GenericSeq using `promptClass()`

Load GenericSeq **and** all of its methods into your R session and then run
`promptClass("GenericSeq")`. Edit the resulting .Rd file and place it
into `man/`.

Compare: `promptClass()` **vs roxygen2**

Roxygen makes it easy to document individual methods, `promptClass()`
makes it easy to document the whole class with all the methods in a single
file.

Vignettes

Package vignette

These *executable* documents are in Sweave format (`.Rnw` extension), which is an extended LaTeX document that includes code chunks. These are executed and the output (variable, but also tables and graphs) are displayed in the document. These dynamic reports, are updated automatically if data or analysis change.

The package vignettes are compiled at build time and are the prefered place for more extensive package documentation and use-cases.

Reference: http://www.stat.uni-muenchen.de/~leisch/Sweave/

The `knitr`[a] package is an alternative to the `Sweave` package that provides syntax highlighting, caching, ... and markdown out of the box.

---

[a] http://yihui.name/knitr/

Vignettes

Demonstration

Let's have a look at the *sequences* package vignette in
`sequences/vignettes`.

sessionInfo()

Prints version information about R and attached or loaded packages.

```
sessionInfo()

## R version 3.2.2 (2015-08-14)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 14.04.3 LTS
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=zh_CN.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=zh_CN.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=zh_CN.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=zh_CN.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] knitr_1.11
##
## loaded via a namespace (and not attached):
## [1] magrittr_1.5  formatR_1.2.1 tools_3.2.2    stringi_1.0-1 highr_0.5.1
## [6] stringr_1.0.0 evaluate_0.8
```

sessionInfo() in vignettes

```
toLatex(sessionInfo())
```

- R version 3.2.2 (2015-08-14), x86_64-pc-linux-gnu

- Locale: LC_CTYPE=en_US.UTF-8, LC_NUMERIC=C, LC_TIME=zh_CN.UTF-8,
  LC_COLLATE=en_US.UTF-8, LC_MONETARY=zh_CN.UTF-8,
  LC_MESSAGES=en_US.UTF-8, LC_PAPER=zh_CN.UTF-8, LC_NAME=C,
  LC_ADDRESS=C, LC_TELEPHONE=C, LC_MEASUREMENT=zh_CN.UTF-8,
  LC_IDENTIFICATION=C

- Base packages: base, datasets, graphics, grDevices, methods, stats, utils

- Other packages: knitr 1.11

- Loaded via a namespace (and not attached): evaluate 0.8, formatR 1.2.1,
  highr 0.5.1, magrittr 1.5, stringi 1.0-1, stringr 1.0.0, tools 3.2.2

Next...

How to test the code in your package?

Or how to make sure that changes in your code do not break existing functionality?

- Implicitely, documentation examples and a vignette do some tests.
- Using R 's built-in testing, that runs some code and compares the output to a saved template.
- Specific packages for unit testing: *RUnit*[a] or *testthat*[b].

---

[a] http://cran.r-project.org/web/packages/RUnit/index.html
[b] http://cran.r-project.org/web/packages/testthat/index.html

Using an `.Rout.save` **file**

In package/tests/

Create

- `mytest.R` with code to be tested
- `mytest.Rout.save` with the reference output

When `checking` your package `R` will

1. execute the code in `mytest.R`
2. save the output to `mytest.Rout`
3. compare `mytest.Rout` to `mytest.Rout.save`
4. report any differences

Using testthat

Test individual expression

expect_that(object_or_expression, condition) with conditions

equals expect_that(1+2,equals(3)) or expect_equal(1+2,3)

gives_warning expect_that(warning("a"), gives_warning())

is_a expect_that(1, is_a("numeric")) or
expect_is(1,"numeric")

is_true expect_that(2 == 2, is_true()) or
expect_true(2==2)

matches expect_that("Testing is fun", matches("fun")) or
expect_match("Testing is fun", "f.n")

takes_less_than expect_that(Sys.sleep(1),takes_less_than(3))

...

Using testthat

Example

```r
library(sequences)

## Error in library(sequences):  there is no package called
'sequences'

library(testthat)
a <- new("DnaSeq",sequence="ACGTaa")

## Error in getClass(Class, where =
topenv(parent.frame())):  "DnaSeq" is not a defined class

test_that("ok test", {
  expect_equal(length(a),6)
  expect_true(seq(a)=="ACGTAA")
  expect_is(a,"DnaSeq")
})
```

Next...

Submission

> **CRAN** Read the CRAN Repository Policy[a]. Upload your `--as-cran` `checked` myPackage_x.y.z.tar.gz to `ftp://cran.R-project.org/incoming` or using `http://CRAN.R-project.org/submit.html`. Your package will be installable with `install.packages("myRpackage")`.
>
> **R-forge** Log in, register a project and wait for acceptance. Then commit you code to the svn repository. Your package will be installable with `install.packages` using `repos="http://R-Forge.R-project.org"`.
>
> ---
> [a]`http://cran.r-project.org/web/packages/policies.html`

Submission

Bioconductor Make sure to satisfy submission criteria (pass `check`, have a vignette, use S4 if OO, have a `NAMESPACE`, make use of appropriate existing infrastructure, include a NEWS file, must **not** already be on CRAN, . . .) and submit by email. Your package will then be reviewed before acceptance. A svn account will then be created. Package will be installable with `biocLite("myPackage")`.

Other popular un-official repositories are github, bitbucket, . . . and packages can be installed with `devtools::install_github`, `devtools::install_bitbucket`.

References

Further reading

- R Installation and Administration, R Core team.
- Writing R Extensions, R Core team.
- Robert Gentleman, R Programming for Bioinformatics, 2008.
- Building packages for Bioconductor: http://www.bioconductor.org/developers/how-to/buildingPackagesForBioc/
- R package development, by Hadley Wickham: http://r-pkgs.had.co.nz/