Building Packages: Self-Study Exercises

Chao-Jen Wong

Fred Hutchinson Cancer Research Center

17-18 February, 2011

1 Introduction

This document covers topics of *R* source packages, including creating a package, using *R* tools, namespace and *R* documentation (Rd).

Throughout the workshop, you will develop tools for facilitating the manipulation and processing of GWAS data during the lab hour following the lecture. The exercises in this document lead you to build, extend and polish a package named *StudentGWAS* containing tools and functions, classes and methods that you have developed from each session.

2 Creating a package

When developing software in R, a package is a better way to organize your code and to share the software with others. This section introduces package structure and construction of a simple package of limited capability.

An *R* source package consists of a top-level directory containing `DESCRIPTION` and `NAMESPACE` files and subdirectories `R, data, inst, man, src, demo` and `tests`. The subdirectories may contain optional files such as `ChangeLog`, `NEWS`, `LICENSE`, `COPYING`, `CITATION`, `configure` and `cleanup`.

To create a package, you can manually create the structure and place the essential files, *R* codes, documentation and data sets in appropriate places. Alternatively, the *R* function `package.skeleton()` creates a package skeleton and initial documentation files. Consult the `package.skeleton()` help page for more details.

Exercise 1
Create a package named StudentGWAS containing a simple object `.fapply`.

- *Make sure that the* `.fapply` *function you have created during the* Efficient R Programming *session is accessible on your R session.*
- *Create the package with the following comment:*

```
package.skeleton(name="StudentGWAS", list=c(".fapply"),
                 namespace=TRUE)
```

- *What got created?*

2.1 The DESCRIPTION file

The `DESCRIPTION` file contains basic information about the package. The 'Package', 'Version', 'License', 'Description', 'Title', 'Author', and 'Maintainer' fields are mandatory. The 'Depends', 'Imports', 'Suggests' and 'biocViews' fields are recommended for packages contributed to Bioconductor. All other fields such as 'LazyLoad', 'LazyData', 'URL' and etc. are optional.

The 'Package' field gives the name of the package. The 'Version' field gives the version of the package. The conventional version format for Bioconductor packages is the form of x.y.z, where x, y and z are integers.

Exercise 2
Edit the DESCRIPTION file in the StudentGWAS package and fill out the mandatory fields. Specify the 'License' field as Artistic-2.0.

2.2 The NAMESPACE file

R supports namespaces for packages. The way to specify a namespace for a package is to place the `NAMESPACE` file in the top directory of the package. In the `NAMESPACE` file, specify which variables in the package are exported and which variables are imported from other packages. It is strongly recommended to use namespaces for various reasons. Most importantly, it helps to avoid naming collisions and to clarify what is public and private and the relationship of your package with other packages.

The default `NAMESPACE` file of your `StudentGWAS` package is

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

which means that all visible objects (whose names starts with a letter) are exported. Note that the `.fapply` object at this point would be invisible (unexported). When developing a more complicated package, you might want to make some objects private and some public.

Adding a namespace affects the search hierarchical order. Having a namespace for a package guarantees it comes first in the search, followed by the imports, then the base namespace, and then the normal search path.

Exercise 3
Export objects can be specified using the `export` *directive in the* `NAMESPACE` *file. Upon to this point, you have the StudentGWAS package containing the object* `.fapply` *in the R subdirectory. To make this object public, use a directive of the form*

```
export(.fapply)
```

to specify that `.fapply` *is to be exported.*

Exercise 4
At the end of this lab, please download the first snapshot of the `StudentGWAS` *package from* http://bioconductor.org/course-packages/src/contrib/StudentGWAS_0.1.0.tar.gz

3 *R* tools

R provides a set of tools to manage packages. These tools are usually accessed from a command shell. Listed below are the shell commands for building, checking and installing packages.

The shell command

```
$ R CMD INSTALL pkg
```

carries out package installation.

The shell command

```
$ R CMD INSTALL --binary pkg
```

installs and produce a binary source archive (*.zip) for use on Windows only.

The shell command

```
$ R CMD build pkg
```

packages up the package source and builds an archive file.

To check whether the package works correctly, use the shell command

```
$ R CMD check pkg (or pkg.tar.gz)
```

This command runs a set of checks for issues such as *R* files syntax errors, package dependencies, completeness and correctness of documentation, consistency of method definition, and etc. See the *Writing R Extensions* manual for more details.

Note that those commands takes a variety of shell-style options. For example,

```
$ R CMD check --no-codoc
```

tells *R* not to check and execute the examples in the documentation.

To obtain information for a particular tool, use

```
$R CMD operation --help
```

where *operation* is an *R* shell tool.

For Windows users, you should install `Rtools` from http://www.murdoch-sutherland.com/Rtools/ to get Unix-like command line tools, Perl, MinGW gcc compilers and some other files needed to build *R* itself.

Exercise 5
Use R shell tools to build, check and install the StudentGWAS package.

4 R documentation (Rd)

The help pages for R objects are written in *R* documentation (Rd) format. Rd is a simple markup language resembling LaTeX. An Rd file can be translated into a number of forms, including LaTeX, HTML, PDF and plain text.

Below is a simplified version of the Rd file that documents the `sd` function.

```
\name{sd}
\alias{sd}

\title{Standard Deviation}

\description{
This function computes the standard deviation of the values in x. If
na.rm is TRUE then missing values are removed before computation
proceeds. If x is a matrix or a data frame, a vector of the standard
deviation of the columns is returned.
}

\usage{
sd(x, na.rm=FALSE)
}

\details{
Like \code{var} this uses denominator n-1.
The standard deviation of a zero-length vector (after removal of \code{NA}s if
\code{na.rm} = \code{TRUE}) is not defined and gives an error. The
standard deviation of a length-one vector is \code{NA}.
}

\examples{
sd(1:2)^2
}
```

There are several types of Rd documentation depending on the type of objects being documented. These types are:

Functions The Rd document for functions must specify sections for describing the usage, argument, value, and example, as shown above. Multiple functions that are related and similar can be described in one Rd file. The function `prompt()` can be used to construct an outline of Rd files for documenting the functions. The `check()` function checks the conformity of the documentation and the actual functions.

Description of Package A package should have an Rd file describing the general purpose and contents of the package. The alias section of the Rd file should contain `pkg-package`, e.g. `utils-package` for the *utils* package

such that the help page of the package can be invoked in R session as `package?pkg`.

```
\name{pkg-package}
\alias{pkg-package}
\docType{package}
.
.
.
\keyword{ package }
```

The help page of a package is invoked in an R session as `package?pkg`. The function `promptPackage()` helps creating a skeleton of the Rd file for package description.

S4 classes and methods The function `promptClass()` generates an outline of the class documentation, and `promptMethods()` generate an outline of methods for a generic function. The syntax for specifying documentation for this type of topic can be more tedious then other topics. (See the *Writing R Extensions* manual for more information.) The help packages for classes ($\mathcal{C}$) and methods ($\mathcal{M}$) can be invoked in R as `class?`$\mathcal{C}$ and `method?`$\mathcal{M}$, respectively.

Data sets The Rd file documenting data sets does not need to include sections as `argument` and `value`. The function `promptData()` can be use to generate a shell of documentation for a data set.

Exercise 6
In the previous lab, we have created three functions, `getSnps`, `getSubjects` and `getKEGGSnps`. Here, we construct an Rd file named `SQLiteFunctions.Rd` documenting one or more of three functions.

- *Use `prompt` to construct a prototype Rd file ready for manual editing.*

  ```
  prompt(getSnps, filename=''SQLiteFunctions.Rd'' force.function=TRUE)
  ```

- *Edit the Rd file and move it to an appropriate place in the StudentGWAS package.*

The shell commend `R CMD check` checks the syntax and executes the examples in the Rd files and translate the files into a variety of formats, including LaTeX, HTML and plain text. A couple shell commands can help to carry out translation of a single Rd file into a particular format.:

- If you have LaTeX installed, you can use the shell commend

```
$ R CMD Rd2dvi --pdf foo.Rd
```

to convert an Rd file to PDF format.

- To convert an Rd file to HTML, use the shell command

```
$R CMD Rdconv --type='html' --output=foo.html foo.Rd
```

The utility function `checkRd` can check adequacy and accuracy of the documentation. The *tools* package also provides a set of functions for converting Rd into different formats. Consult the help page of `tools::Rd2HTML` for details.

See *Guidelines for Rd files* (http://developer.r-project.org/Rds.html for detailed guidelines for writing Rd files. For detailed syntax and technical discussion, see *Parsing Rd files* (http://developer.r-project.org/parseRd.pdf).

Exercise 7
At the end of this lab, please download the second snapshot of the `StudentGWAS` *package from* *http://bioconductor.org/course-packages/src/contrib/StudentGWAS_0.2.0.tar.gz*