Seminar III: R/Bioconductor

Leonardo Collado Torres
lcollado@lcg.unam.mx
Bachelor in Genomic Sciences
www.lcg.unam.mx/~lcollado/

August - December, 2009

Building Packages

Intro

Motivation

Installing a package

Package structure

`DESCRIPTION` file

Rd files

Building Packages

Useful tools

Namespaces

Homework

About

- Package structure
- Helpful functions for creating packages
- Documenting your package

An example

- As a quick way to learn how to create a package is to look at one, please download the SpeCond source tarball.
- It's a new package developped by Florence Cavalli from EBI. She kindly provided us with a copy before its official release :)
- As such, it's only available for download from the private site (cursos.lcg).
- After downloading it, `gunzip` it and use `tar -xvf`[1].

[1] In Unix only, in Windows use WinRar or another program.

What a pain. . . , why bother?

- Documenting any script, for anyone, is generally a pain.
- However, your scripts, data, etc. need to meet some requirements so others, or yourself later on, can understand them without much trouble.
- Packages are the most reliable, standard, trustworthy and flexible way of sharing R code with others.
- If you meet the requirements, you'll have access to a large audience :)

On this course

- We want you to learn how to build packages, so you can easily create them later in your career and share the data associated with each of your publications. Its the best way to guarantee and fulfill the reproducibility requirement in science.
- Contribute to the Bioconductor experimental data package repository. It's been under-exploited and its a great way to introduce yourselves on the real world of research. Your package has to be innovative in some aspect(s) :)

You'll ace it :)

- Don't worry, with practice it'll be easy :)
- That's why you've been handing homeworks using `Sweave` and LaTeX
- *Hopefully* easier or more straight forward than documenting Perl scripts with Pod :P

Quick review

- So far we've been using functions such as install.package:

```
> install.packages("lattice")
```

- Or with the biocLite script:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("ShortRead")
```

- Those commands are the ones we use for CRAN and Bioconductor repositories. For the third mayor repository, OmegaHat we would have to use:

```
> install.packages("mypackage", rep = "http://www.omegahat.org/R")
```

- How would you choose the name for your NEW package?
- Just as a note, you might want to check the other packages functions using apropos.

R in batch mode

- For installing a package that is NOT hosted in a repository, we need to use some `Unix`-like commands.[2]
- We'll actually be running R in batch mode. The basic syntaxis is:

  `R CMD` *`operation`*
- For installing a package[3] you need to use:

  `R CMD INSTALL` *`package(s)`*

[2] If you have correctly set your environmental variables (specially PATH) in Windows, it should be fine. To avoid problems use the Solaris servers.

[3] In your current working directory.

Specifying the library

- A library is the folder where your packages get installed.
- As you've noticed by now, you can install packages in different libraries.
- In `Montealban`, there is an admin controlled library and probably everyone of you has his own extra R library.
- You can use the shell option `-l` to specify the library:
  `R CMD INSTALL -l LibDirectory` *`package(s)`*

Building a package

- The previous commands work if you have a directory inside your current working directory for every package you are going to install.
- However, if someone sends you a package, its normally in a tarball.
- Once you extract the files, you can install the package. You should now be able to install the SpeCond package![4]
- These tarballs are not built with `tar`, but with the build command:
  `R CMD build` *`package(s)`*
- The `build` command uses information from specific files inside the R package folder, so its important to meet all the requirements :)

[4] Florence built it on a Mac, so I'm not sure that it works in Windows. It should work in Solaris.

Check and remove

- Once you've built your package, you need to check it:
  `R CMD check` *`package(s)`*
- From what I've read, this command is a real pain. It will check lots of details and prompt you of everything you missed. Nevertheless, its necessary.
- Specially when developing a package, you'll want to uninstall a package. To do so, use:
  `R CMD REMOVE` *`package(s)`*
- There are some functions to avoid installing and uninstalling packages too frequently, but you'll need to do so in several situations.

Overall structure

Before building a package, you need a folder `MyPkg`[5] with the following structure:

- `R` folder
- `data` folder
- `exec` folder
- `inst` folder
- `man` folder
- `po` folder
- `src` folder
- `tests` folder
- A `DESCRIPTION` file
- A `NAMESPACE` file

Most are optional, though Bioconductor has more strict requirements than CRAN. Specially on the documentation side!

[5] Name it to your liking, but it has to be new!

R folder

- This folder contains at least one file: a `.R` file with all the R source code.
- The code is normally made up of functions, but it can also include some objects.
- If you are defining `classes`, `methods`, etc. include a `.R` file for each of these.

data folder

- Here you'll find loadable versions of the data objects; generally data frames.
- The normal extensions for these files are `.rda` and `.Rdata`. Both are fine though I personally prefer the later one. You can also create this kind of file using save.
- In the case of your projects, we'll mainly use this folder :)
- You might prefer to use `csv` files instead of `.Rdata` files. In this case, the name of the file specifies the name of the object, so be careful!
- A third option is to include some `.R` files that only contain assignments that do not use external objects. For example:

data folder

```
> assistants <- c("Alejandro", "Victor",
+     "Jose")
```

- Is this the only place where you can include such assignments?

exec folder

- Here you include only executable scripts.
- Is there an `exec` folder on the `SpeCond` package?

inst folder

- All the contents in this folder will be copied, recursively, to the directory where the package gets installed.
- Normally, it includes a subfolder named `doc`. A must in Bioconductor packages.
- Any idea why?

inst folder

- That's where you store the vignette(s)!
- Vignettes have to be on `.Rnw` format. You are experts on it by now, but it always helps to check another vignette.
- What difference do you note on the code chunks between SpeCond's vignette and this class `.Rnw` file?

man folder

- In your opinion, is this an optional folder?

man folder

- The R, the `inst/doc` and the man folder are the mandatory folders for any given Bioconductor package.
- Here you keep the `.Rd` help files.
- If you have plataform specific code (either `unix` or `windows`), you need to create such sub-folders both on the `man` and `R` folders.

po folder

- This is probably THE most optional folder of all.
- It contains translation files.

src folder

- If you are linking R to a compiled library, mostly from C, you'll store it inside this folder.

tests folder

- Here you save some `.R` files with test code.
- Its useful to keep here code that at some point caused a bug in your package. This is called regression testing.
- The `R CMD check` command will still run all the examples from the manual as tests. Nevertheless, you might want to include some advanced tests here.

Basic

- This file contains the general info for your package.
- The syntax is rather special. A single space can break it :(
- You need be extra careful with this file. It is used several times later on!
- For Bioconductor packages, you need to include the biocViews line!

Basic info

- `Package`
  The official package name. Make sure its unique!
- `Type`
  Package
- `Title`
  A one line (short).
- `Version`
  For Bioconductor, you need to use an `x.y.z` style. For example, `0.99.1` Note that x will be 0 until your package is made public. Please check this for more info.
- `Date`
  In YYYY-MM-DD format.

Basic info

- `Author`
  The list of people who wrote this package.
- `Maintainer`
  The one in charge of keeping the package functional. Its a one line format with the name followed by the valid email. There can't be two maintainers per package. Bioconductor package maintainers have these responsibilities:
  1. *Subscribe to the bioc-devel mailing list.*
  2. *Respond to bug reports and questions from users regarding your package.*

Basic info

3. *Maintain your package and its capabilities as R and other Bioconductor packages evolve. Typically this involves some work every six months, when a new release is being prepared. We can assist by answering questions, but it is your responsibility to look for and make any needed changes. If you are unwilling or unable to do this your package will be removed from the upcoming release. Users will still be able to find the older versions.*

- `Description`
  Another *one* line explaination of your package. It can actually go over one line when you print it, but type it only in one so the `DESCRIPTION` file won't break.

Basic info

- `License`
  This specifies how you'll share your work. For example, if a company can use your package freely or if they have to acquire a license. The quick solution is to just use the same one as `R`. Check it with:

  ```
  > license()
  ```

Advanced info

- `Depends`
  The packages (in the same order) that your package depends on. Remember that a user will have access to the functions of the depended packages. Its generally a good practice to include the R version at the beginning[6]:
  `Depends: R(>= 2.10), SomePkg`
  Note that we did not specify a minimum package version for `SomePkg`.
- `Imports`
  Very similar to `Depends`. The end level user will not have access to the these packages, but you might use them for your own package functions. If you are importing a package on the `NAMESPACE`, you'll need to include it here.

Advanced info

- `Suggests`
  Another list of packages that the user might want to load but are not used by the package per se. For example, a plotting package. If you load a package in an example or a test, you'll need to include it here.
- `biocViews`
  The Bioconductor categories that fit your package. This line is used to construct the Bioconductor site. Check them at the biocViews site.
  How many biocViews does the `limma` package use?
- `LazyLoad`
  Basically, set it to `Yes` :)
  Turns on the lazyloading of your package R objects, which is normally faster for the end-user.

Advanced info

- `LazyData`
  Similar to `LazyLoad`. Say you have a data frame named df. Once the user types `df` the first time, it will attach the data set. Otherwise the user will need to use the data function. Anyhow, saves time!
- `Collate`
  The order in which you want your `.R` files in the `R` folder to be loaded. Useful if a 2nd function uses a 1st function and you don't want R to crash.
- `Packaged`
  A line more precise than `Date`.

[6] Be VERY careful with the spacing here.

Search path

Depends, imports, suggests. . . its a pain, why go through it?

- Basically, to avoid mistakes where you use `SomeFunction` that exists in more than 1 package.
- Its also a matter of speed. It reduces the search path. Its a list of the packages where R checks if `SomeFunction` exists.

```
> search()

[1] ".GlobalEnv"
[2] "package:stats"
[3] "package:graphics"
[4] "package:grDevices"
[5] "package:utils"
[6] "package:datasets"
[7] "package:methods"
[8] "Autoloads"
[9] "package:base"
```

Search path

- You can also find the currently attached packages using:

```
> temp <- .packages()
> temp
```

- Do they return the same list?
- Of course, you can also use sessionInfo.
- If you run R interactively or in batch mode, does R load the same set of packages?

Help files

- Now we need to document our functions, objects, methods, etc.
- This is done with `.Rd` files, which use a format similar to TEXand LATEX
- Open the `fitPrior.Rd` file so it'll be easier to understand :)

Format

`Rd` files follow the TEXsyntax by using: `backslash tag text`

- `name`: The name of your function or object.
- `alias`: Normally, the same as the name. This changes if its a method...
- `title`
- `description`
- `usage`
- `arguments` with `item` tags
- `value`
- `details`
- `references`
- `author`

Format

- `examples` Used by the check command.
- `keyword`

Functions

- The most useful function while creating a package is package.skeleton:

```
> package.skeleton("NAME", ListOfObjects,
+      path = "PATH")
```

- Create a data frame with some data, then create a package with it.
- Check the files it created :)

Prompt

- Another useful function is prompt

```
> `?`(prompt)
```

Saves work, but doesn't do it all

- These functions save a lot of time, but you still need to type, then copy and paste lots of information.
- Be careful with the names and the like, so you don't have inconsistencies between files.

Basics

- Following the principle of importing and depending, you can use the `NAMESPACE` to specify which functions you'll use.
- Avoids conflicts :)

Importing and exporting

- It follows a special format, similar to R code but it isn't R code exactly.
- You use `importFrom` to specify which functions you are importing. For example, the function `prompt` from the package `utilities`:
  `importFrom(utilities, prompt)`
- You can also import methods using `import()`
- Then you have to specify which functions you'll be exporting. Meaning that the end level user will be able to use.
- You do so using:
  `export(function1)`

Importing and exporting

- Alternatively, you can use `exportPattern("pattern")` For example, you could name your internal functions with a dot, and then export those that begin with a letter.
- For explorting classes or methods you need to use different functions: `exportClass()` and `exportMethods()`
- Take a look at the `SpeCond` `NAMESPACE` file.

Extra

- For more information on how to create a package check the `R Extensions Manual`. Its around 150 pages long :P

Homework

- With your team, hand in a tarball with the layout for your package.
- Just include a data frame in your package and a simple function on the `R` folder.
- That means that you'll decide who will maintain it :)

SessionInfo

```
> sessionInfo()

R version 2.10.0 Under development (unstable) (2009-07-21 r48968)
i386-pc-mingw32

locale:
[1] LC_COLLATE=English_United States.1252
[2] LC_CTYPE=English_United States.1252
[3] LC_MONETARY=English_United States.1252
[4] LC_NUMERIC=C
[5] LC_TIME=English_United States.1252

attached base packages:
[1] stats     graphics  grDevices
[4] utils     datasets  methods
[7] base
```