R Programming

José R. Valverde
jrvalverde@cnb.csic.es
CNB/CSIC

Packages

Statistics is the art of never having to say you're wrong.

Introduction

Packages

- Packages are the fundamental unit of shared code
  - They bundle together code, data, documentation and tests
  - That can be shared with others, usually through CRAN, the **C**omprehensive **R** **A**rchive **N**etwork.
- CRAN is the public hub where you can find most packages written by others

Installation

- You install a package with
  - install.packages(“packagename”)
- It is often a good idea to use
  - install.packages(“packagename”, dependencies=T)
- To ensure that if one package needs code from other packages, those others are automatically installed as well.
- R will contact CRAN to download and install the package.
  - Usually R will ask you which copy of CRAN is closer

Using a package

- Is very easy:
  - library(“package”)
- Alternatively, you may use
  - require(“package”)
  - This tells R that there is no point in executing any more commands if the package is not installed.

Getting help

- Simply use
  - library(help=”package.name”)
  - help(package = “package.name”)

- When you load a package, all of its components (code, functions, datasets, **documentation**, etc...) become automatically available.

Pre-installed vs. third-party

- By default, when you install R, there is a number of packages that are automatically installed
  - They provide the basics of a fully functional statistical environment
- There are literally thousands (and many thousands indeed) of 'packages' available to extend the functionality of R

CRAN

- This is one of the key advantages of R:
  - Whenever a user develops new functions and wants to share them
  - All that's needed is to make them available in CRAN
  - CRAN is the central exchange point
  - Whenever a user wants to use new functions shared by another person
  - All that's needed is to look for them in CRAN

CRAN on the web

- CRAN is not only accessible through R
- You can also consult it on the web
  - See the packages that are available
  - Read a one-liner description
  - Read the full documentation

  - https://cran.r-project.org/web/packages/

Quick search

- Go to the listing of packages by name
  - Find a package to do “flux balance analysis”
    - Type [CTRL] + [F] (or [cloverleaf] + [F]) or select “Find in Page”
    - Type in the search terms
    - Find all the packages containing the search terms
  - Find a package for NGS data analysis

Beyond CRAN

- Not all packages are available in CRAN
- Some packages/communities have grown so big that they have acquired an identity of their own
  - Rstudio
  - JGR
  - Bioconductor
  - USGS-R, etc...
- Some of these may be accessible through other packages (e.g. Deducer contains JGR)

Do I have it?

```
# check to see if a single package is installed, if not, install it, and use it.
usePackage <- function(p) {
    if (!is.element(p, installed.packages()[,1]))      # if the package is not installed
        install.packages(p, dependencies = TRUE)       # install it
    require(p, character.only = TRUE)                  # load the package
}
# check.packages function: install and load multiple R packages.
# Check to see if packages are installed. Install them if they are not,
# then load them into the R session.
use.packages <- function(pkgs){
    new.pkgs <- pkgs[!(pkgs %in% installed.packages()[, "Package"])]
    if (length(new.pkgs))                              # if there are uninstalled packages
        install.packages(new.pkgs, dependencies = TRUE)       # install them
    sapply(pkgs, require, character.only = TRUE)     # load all the packages
}
```

%in%

- Note the use of %in% in the second function.
- This tells R to give you the values from the first collection of data that are also present in the second argument.
  - This is useful whenever we want to find common items in collections of values
  - e.g. to find common genes in two listings/groups

Example

foreign

- This package provides functionality to read “foreign” data, that is, data produced by other statistical packages.
- It is one of the packages recommended for installation with R
- **`library("foreign")`**
- If that doesn't work, don't worry, it is also available in CRAN

```
install.packages("foreign",
                 dependencies=TRUE)
```

foreign: Give it a try

- Remember that we can open files over the web by giving their URL instead of the file name.
- Reading SPSS data files
  - **read.spss( )** reads SPSS data files

```
dat.spss <- read.spss(
"http://faculty.washington.edu/
tathornt/sisg/hsb2.sav",
to.data.frame=TRUE)
```

foreign: Give it a try (2)

- Reading STATA files
  - **read.dta( )** reads Stata files

```
dat.dta <- read.dta(
"http://faculty.washington.edu/
tathornt/sisg/hsb2.dta")
```

- See the help on the package for a list of all the foreign formats supported
  - You are likely to find several familiar formats!

Vignettes

Vignettes

- Many packages contain “vignettes” in addition to the usual textual help.
- Vignettes give an overview of the package, usually including examples and tutorials
- You can read these “vignettes” for an installed (and loaded) package using
- **BrowseVignettes("package.name")**
- If the package contains any, R will open a web browser and show them

Example

- Package **car** is another package recommended to be installed with base R. You surely have it already

```
library(car)
```

```
browseVignettes("car")
```

```
help(package="car")
```

vignette( )

- This function can also be used to display vignettes.
- Imagine you find package “vegan” in CRAN and that you decide that you want to use it.

```
install.packages("vegan")

library("vegan")

vignette(package="vegan")

vignette(topic="intro-vegan")
```

Shortcuts

- ?name
  - help(name)
- ??name
  - Search for “name” in the available help pages.
- library()
  - With no arguments will list all packages installed in your computer and ready for loading

Masking

Name clashes, code crashes

- With so many packages, and so little imagination, we will often run into trouble.
  - Two contributors may have decided to use the same name for one of their functions
    - Possibly doing different things (or doing things differently)
  - You will see a message when this happens after loading a new package that contains a function with the same name as a function that you already have
    - In that case only the new version is visible
    - The old version is **masked**

Example

- **`library("plyr")`**
- **`library("reshape")`**
- You get a message indicating that “rename” and “round_any” have been masked.
  - This means that when we loaded package “plyr”, it added -among others- two functions with these names
  - And when we loaded “reshape” it also added two functions with the same names.

How does R decide

- Remember variable assignments
  - X = 10 ; X = 20 ; X
- Remember how did we define a function
  - func < - function(....)
- A function is just another variable. It will contain the result of the last assignment
  - Therefore, only the last definition of a function will be valid.

How can you decide

- That's not the whole story. Remember that the function was defined in a package...
  - Remember variables in a dataset
    - cherry$Girth
    - attach(cherry) ; Girth, cherry$Girth
    - deattach(cherry) ; Girth ; cherry$Girth
  - With packages we have something similar: elements of a package can be referred to using '**::**'
    - We can always refer to a function in a package as **package.name::function( )**

Explanation

- In effect, library works as if we loaded the package and attached it afterwards:
  - Exit R and start over again

```
rename()
?rename                 # get help on "rename()"
plyr::rename()
?plyr::rename           # get help
reshape::rename()
?reshape::rename        # get help
```

- This works like $ for variables in dataframes. Note that we have not loaded any package with "**library()**" yet.

Explanation (2)

```
library("plyr")    # works as attach()
rename()            # rename from plyr
?rename
library("reshape")
rename()           # reshape's rename
?rename
plyr::rename()
?plyr::rename
```

- Corollary: pay attention to warnings about read masking and be careful if you need to use a masked function

Housekeeping

- We can keep track of everything that we have loaded with **sessionInfo( )**

```
sessionInfo( )
```

- The info may contain warnings about package versions being outdated. You can use **update.packages( )** to update them.
- You may use also **new.packages( )** to see what's new in CRAN.

Namespace

- The collection of all the variable names (functions, vectors, data-frames, lists, etc...) known to R is called the “namespace”
- Each package has its own “namespace” (the names it defines)
- Your session has its own “namespace” (the objects you load from packages or scripts and the names that you define).

Undoing package loads

- You may remove a package after loading it. This will remove all the variables (functions, dataframes, etc...) created when loading the package.
  - detach(“package”)
- This will usually not be needed, but may sometimes be useful.
  - We are not getting into more detail now.

Creating your own package

Why

- Writing your own package is useful
  - Even if you do not plan to share the code (yet)
- Organising your code (scripts) as a package will make your life easier
  - Packages impose a number of conventions that will make your life easier.
  - You do not need to think on how to organise your analysis, just follow the common trend
  - Standardised conventions lead to standard tools
- **Reproducible research !!!**

Conventions

- Within your working directory/folder
  - ./R – this folder will contain all .R scripts
  - ./data – this folder will contain all data (.Rdata, .rda, .csv. .tab, .txt files...)
  - DESCRIPTION – is a text file with a description of the folder contents
  - NAMESPACE – a text file which we'll ignore now
  - You may have other folders for documentation in various formats.

Getting started

```
install.packages("devtools",
                 dependencies=T)
library("devtools")
install.packages("roxygen2")
library(roxygen2)
```

Create a package in a subdirectory

- use "getwd()" to learn where are you now
- use "setwd(somewhere) to go where you want it
- create(“packagename”)
- e.g.

```
getwd()
#setwd("~/testR/")
create("my.test")
```

- Check your working directory and see what changed.

What happened?

- You should now have a folder named “my.test”
- Inside that folder there will be a file named “DESCRIPTION”, a folder named “R” and some other files and possibly folders.
- The “R” folder is for you to save all your scripts into it.
- The DESCRIPTION file is for you to describe the contents of this “package”.

DESCRIPTION: General data

- **Package:** my.test
- **Title:** What the package does (one line)
- **Version:** 0.1
- **Authors@R**: person(“First”, “Last”, email=”god@heaven.sky”, roles=c(“aut”, “cre”)
- **Description**: What it does described in more detail

  (one paragraph)
- **Depends:** R (>= 3.1.0)
- **License**: GNU-GPL
- **LazyData:** true

DESCRIPTION: Dependencies

- Which packages do you need for the analysis?
- **Imports:** vegan, indicspec
- **Suggests:** CCA
- You can specify version numbers for the packages.
- You will normally leave this empty and, as you carry out your analysis and start loading packages, you will come back to this file and update these lists.

Start coding

- It is time now for you to start writing the code, the scripts, needed to carry out your analyses.
- Store the scripts in the “R” folder, using file names ending in “.R”
- Your scripts will contain commands, functions... to deal with data.
- Create a folder named “data” and store the data files there.

A sample script

- Write any silly-stupid function and save it as “my.test.R” in the “R” folder, for example:

```
my.test <- function(like=FALSE) {
    if (like == TRUE)
        print("R is great")
    else
        print("I hate R")
}
```

Document your functions

- There are many ways, but following a standard is always useful: Roxygen provides suitable guidelines. In your script, add before your function

```
#' A test function
#'
#' This function tells whether we
#' like R or not
#' @keywords my.test
#' @export
#' @examples
#' my.test( TRUE )
```

Explanation

- All those lines are comments, hence R will ignore them.
- Comments followed by a single quote (') will be analyzed by another tool (Roxygen) which will interpret them
  - The first line is a one-line description of the function
  - The next paragraph is a longer description
  - @something are keywords to describe specific properties of the function
- See the contents of “my.test” and “my.test/man”

Generate the documentation

```
setwd("./my.test")

document( )
```

- This changes our working directory to the project directory and then builds the documentation automatically
  - It will call Roxygen
  - Roxygen will analyze all files in R
  - Roxygen will generate the documentation in a new folder called “man”

Install your newly created package

```
setwd("..")

install("my.test")
```

- Now you have a shiny new package installed in your R system. You can load this package with “library()”, obtain help on the function with “help()”, etc...

```
library(my.test)

help(my.test)
```

Prepare for sharing

- Sooner or later you will want to share your work (e.g. you will publish your analyses and will need to provide all data and scripts as supplementary materials)
- A commonly accepted approach is to share all materials on Github
  - You will need a free account on Github
- **`install_github('my.test', 'github_username')`**

Iterate

- while (finished=FALSE) repeat_all_the_steps( )
- From now on, it's easy
  - Repeat the basic steps until you finish your work
  - Edit the script file(s)
  - Add data into the 'data' folder
  - Rebuild the documentation after any changes
- Until you are satisfied.

A bit more on documentation

- Some useful keywords to use in your function documentation are
  - @param name description
  - @ return description
- Always document the parameters and results of every function, explaining their names and their meaning.
- (Tip: use R markdown to write vignettes)
- There is a lot more to know, but we'll stop here.

Another example

```
#' 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
}
```

Thank you

Questions?