Lesson 3

Loïc Dutrieux, Jan Verbesselt, Ben DeVries, Sytze de Bruin

October 11, 2013

1 Today's learning objectives

At the end of the lecture, you should be able to

- Find help for R related issues
- Produce a reproducible example
- Adopt some good scripting/programming habits
- Use control flow for efficient function writing
- Create a R package
- Use version control to develop, maintain, and share with others your R packages

2 Introduction

NA NA

3 Finding help

NA NA Warning, these mailing list have heavy mail traffic, use your mail client efficiently and set filters, otherwise it will quickly bother you.

NA

- no-one gets offended by your question,
- people who are able to answer the question are actually willing to do so,
- you get the best quality answer

So, when posting to the mail list:

- Be courteous,
- Provide a brief description of the problem and why you are trying to do that.
- Provide a reproducible example that illustrate the problem, reproducing the eventual error

- Sign with your name and your affiliation
- Do not expect an immediate answer (although well presented question often get answered fairly quickly)

4 Creating a reproducible example

Indispensable when asking a question to the online community, being able to write a reproducible example has many advantages. NA

But what is a reproducible example exactly?

Well, one could define a reproducible example by a piece of code that can be executed by anyone who has R, independently of the data present on his machine or any preloaded variables. The computation time should not exceed a few seconds and if the code automatically downloads data, the data volume should be as small as possible. So basically, if you can quickly start a R session on your neighbour's computer while he is on a break, copy-paste the code without making any adjustments and see almost immediately what you want to demonstrate; congratulation, you have created a reproducible example.

Let's illustrate this by an example. I want to perform values replacements of one raster layer, based on the values of another raster Layer.

```
> # Create two rastersLayer objects of similar extent
> library(raster)
> r <- s <- raster(ncol=50, nrow=50)
> # fill the raster with values
> r[] <- 1:ncell(r)
> s[] <- 2 * (1:ncell(s))
> s[200:400] <- 150
> s[50:150] <- 151
> # perform the replacement
> r[s %in% c(150, 151)] <- NA
> # Visualize the result TODO(dutri001) embed the figure in the pdf
> plot(r)
```

NA Some popular datasets are: cars, meuse.grid$_l l$, $Rlogo, etcTheautocompletionmenuofthedatafunctionu$

```
> # This demonstration of datasets still need to be writen
> data(cars)
> class(cars)

[1] "data.frame"

> head(cars)

  speed dist
1     4    2
2     4   10
```

```
3     7    4
4     7   22
5     8   16
6     9   10
```

```
> plot(cars)
> ## Add another example with meuse, from sp
```

5 Good scripting/programming habits

Increasing your scripting/programming efficiency goes through adopting good scripting habits. Following a couple of guidelines will ensure that your work:

- Can be understood and used by others
- Can be understood and reused by you in the future
- Can be debugged with minimal efforts
- Can be re-used across different projects
- Is easily accessible by others

In order to achieve these objectives, you should try to follow a few good practices. The list below is not exhaustive, but already constitutes a good basis that will help you getting more efficient now and in the future when working on R projects.

- Write short functions.
- Make your functions generic and flexible, using control flow
- Comment your code
- Document your functions
- Build a package
- Keep a TODO list
- Use version control to develop/maintain your package

Function writing

Short, generic and flexible, so that it will be easier to re-use them for a slightly different processing/analysis chain. More flexibility in your function can be achieve thanks to so easy tricks provided by control flow. The next section develops further this concept and provides examples of how control flow can help your functions becoming more flexible.

Control flow

Control flow refers to the use of conditions in your code that redirect the flow to different directions depending on variables values of classes. Make use of that in your code, as this will make your functions more flexible and generic.

Object classes and Control flow

You have seen in a previous lesson already that every variable in your R working environment belongs to a class. You can take advantage of that, using control flow, to make your functions more flexible.

A quick reminder on classes

```
> # 5 different objects belonging to 5 different classes
> a <- 12
> class(a)

[1] "numeric"

> b <- "I have a class too"
> class(b)

[1] "character"

> library(raster)
> c <- raster(ncol=10, nrow=10)
> class(c)

[1] "RasterLayer"
attr(,"package")
[1] "raster"

> d <- stack(c, c)
> class(d)

[1] "RasterStack"
attr(,"package")
[1] "raster"

> e <- brick(d)
> class(e)

[1] "RasterBrick"
attr(,"package")
[1] "raster"
```

Controling the class of input variables of a function

This section needs to be writen, containing examples.

Use of try and debugging your functions

try The try function may help you writing functions that do not crash whenever they encounter an unknown of any kind. Anything (subfuntion, piece of code) that is wrapped into try will not interupt the bigger function that contains try. So for instance, this is useful if you want to apply a function sequentially but independently over a large set of raster files, and you already know that some of the files are corrupted and might return an error. By wrapping your function into try you allow the overall process to continue until its end, regardless of the success of individual layers. An example of the use of try

```
> # I need to write this example
> # Something with a list of rasterLayers, with some layers containing nothing but NAs
>
```

Also try returns an object of different class when it fails. You can take advantage of that at a later stage of your processing chain to make your function more adaptive. See the example below

```
> # Also need to write this one
> # The prefast package does something like that
>
```

Package building

Also part of good programming practices, building a R package is a great way to stay organized, keep track of what you are doing and be able to use it quickly and properly at any time.

Why building a package?

- Easy to share with others
- Dependencies are automatically imported and functions are sourced
- Documentation is attached to the functions and cannot be lost (or forgotten)

For these reasons, if you build a package you'll still be able to run the functions you wrote next year. Which is often not the case for stand alone functions that are poorly documented and depend on a millions other functions ... that you cannot find anymore.

Structure of a package

TODO(dutri001) Finish that part

NA The default basic structure of a package is as follows: 2 subdirectories (R and man), NAMESPACE and DESCRIPTION files. The R directory includes the package functions of the package, usually each in a separate functionName.R file; Each one of these function should have an associated functionName.Rd file, stored in the man subdirectory. These .Rd files are the functions documentation. NAMESPACE and DESCRIPTION files contain general information about the package (package metadata, dependencies, version number, etc). That is quite a minimalistic package structure; more elaborate packages may include extra subdirectories, such as data, demo or vignette. /R/ /man/ /NAMESPACE /DESCRIPTION

The package.skeleton() function will help you get the package structure from a list of sourced functions. prompt() creates a tailored documentation (.Rd) file from an existing function. The project functionalities of the R Studio IDE can greatly assist you in creating a package from scratch, including Good practices: http://mages.github.io/R_package_development/#1

Version control

Not done yet

NA What is it?

How is it useful? Version control is very well suited to large software development projects and for that reason it is an indispensable part of such projects. Although we are not software developpers, version control systems present a set of advantages from which we, and the scientists who use scripting in general, can benefit. Among others, some arguments in favour of using version control are that:

- It facilitates collaboration with others
- Allows you to keep your code archived in a safe place (the cloud)
- Allows you to go back to previous version of your code
- Allows you to have experimental branches without breaking your code
- Keep different versions of your code without having to worry about filenames and archiving organization

What systems exist? Git, Subversion (svn), mercurial (hg) Online repositories: GitHub Bitbucket Sourceforge Rforge

Semantics of version control Commits Push Clone (git) branch (git): Creates a branch checkout (svn): equivalent of git clone

How to To be able to perform the following tutorial, you must have git installed on your machine. Please refer to the system setup vignette to know how to install and setup git and allow it to be embedded into R Studio. Tutorial on how to start a R package with version control, hosted on GitHub. Setup NA NA NA NA NA NA NA

NA